Investigating Tools and Techniques for Improving Software Performance on Multiprocessor Computer Systems

A thesis submitted in fulfilment of the
requirements for the degree of

Master of Science

of

Rhodes University

by

Waide Barrington Tristram

Grahamstown, South Africa
March 2011

Abstract

The availability of modern commodity multicore processors and multiprocessor computer systems has resulted in the widespread adoption of parallel computers in a variety of environments, ranging from the home to workstation and server environments in particular. Unfortunately, parallel programming is harder and requires more expertise than the traditional sequential programming model. The variety of tools and parallel programming models available to the programmer further complicates the issue. The primary goal of this research was to identify and describe a selection of parallel programming tools and techniques to aid novice parallel programmers in the process of developing efficient parallel C/C++ programs for the Linux platform. This was achieved by highlighting and describing the key concepts and hardware factors that affect parallel programming, providing a brief survey of commonly available software development tools and parallel programming models and libraries, and presenting structured approaches to software performance tuning and parallel programming. Finally, the performance of several parallel programming models and libraries was investigated, along with the programming effort required to implement solutions using the respective models.

A quantitative research methodology was applied to the investigation of the performance and programming effort associated with the selected parallel programming models and libraries, which included automatic parallelisation by the compiler, Boost Threads, Cilk Plus, OpenMP, POSIX threads (Pthreads), and Threading Building Blocks (TBB). Additionally, the performance of the GNU C/C++ and Intel C/C++ compilers was examined. The results revealed that the choice of parallel programming model or library is dependent on the type of problem being solved and that there is no overall best choice for all classes of problem. However, the results also indicate that parallel programming models with higher levels of abstraction require less programming effort and provide similar performance compared to explicit threading models.

The principle conclusion was that the problem analysis and parallel design are an important factor in the selection of the parallel programming model and tools, but that models with higher levels of abstractions, such as OpenMP and Threading Building Blocks, are favoured.

ACM Computing Classification System Classification

Thesis classification under the ACM Computing Classification System (1998 version, valid through 2011) [2]:

D.1.3 [Concurrent Programming]: Parallel programming

D.2.5 [Testing and Debugging]: Debugging aids

D.2.8 [Metrics]: Performance measures, Product metrics

D.3.3 [Language Constructs and Features]: Concurrent programming structures

General-Terms: Design, Experimentation, Measurement, Performance

Acknowledgements

I acknowledge the financial and technical support of Telkom SA, Business Connexion, Comverse SA, Verso Technologies, Stortech, Tellabs, Amatole Telecom Services, Mars Technologies, Bright Ideas Projects 39, and THRIP through the Telkom Centre of Excellence in the Department of Computer Science at Rhodes University. In particular, I extend thanks to Telkom for the generous Centre of Excellence bursary allocated to me during the course of my research.

I am also deeply grateful to my supervisor, Dr Karen Bradshaw, for her exceptional support, advice, and patience. I could not have asked for a better supervisor to guide me through my research.

A special thanks goes to my partner, Lisa, for her love, support and patience, particularly during the final stages of my thesis write-up. Had it not been for her exceptional organisational skills and forward planning, I would have had a significantly harder time finalising this thesis.

Many thanks to my family and friends for their constant support and encouragement.

Contents

List of Figures

List of Tables

Listings

Chapter 1

Introduction

With the advent of commercial multicore processors, modern computer systems have taken on parallel capabilities. It is now common for modern commodity computer systems to ship with two or more processor cores and the chip manufacturers are not stopping there [19, 124]. CPUs with between two and twelve physical processor cores are readily available from manufacturers such as AMD, IBM, Intel, and Oracle Sun, with sixteen-core variants on the near horizon. The Sun SPARC T-series, IBM POWER7, and Intel Hyper-Threading enabled range of processors take multicore parallel processing a step further by supporting multiple threads of execution per core. Intel's Tera-scale research program has even produced 48-core and 80-core experimental CPUs for research purposes, demonstrating the potential for highly parallel manycore architectures [62, 94, 132]. These multicore and multiprocessor computer systems are seeing use almost everywhere, particularly in the server and workstation markets, as well as in scientific computing where hundreds of multicore CPUs are harnessed to form massively parallel supercomputers. This trend towards increasing the number of processor cores instead of improving clock speeds can be attributed to the physical limitations of modern processor designs [54, 53, 124, 136]. Other processor improvements, such as instruction-level parallelism, are also reaching their limits and it is becoming harder for CPU designers to increase the performance of the individual processor cores. As such, multicore designs represent both the present and future of the CPU and signal the need for programmers to start thinking in parallel as opposed to the traditional sequential programming model [54, 136].

This has serious implications for the design and development of applications, since writing and debugging parallel software is a difficult task that requires more knowledge and expertise than sequential programming [53, 136, 141]. So, following the commitment of chip manufacturers to the development of multicore CPUs, the key limiting factors for software performance are the

parallel programming abilities of software programmers and the availability of easy-to-use, efficient parallel programming models and parallel-aware software development tools. However, well over a hundred different parallel programming models and libraries have been developed over the years, most of which failed to gain any traction in the programming community, leaving only a handful still in widespread use. Therefore, programmers and library developers should focus on using and extending the current range of successful parallel programming models instead of attempting to develop new models that may only see limited use. The real problem lies in helping programmers to think in parallel and providing methodical approaches to identifying and exploiting the potential parallelism in a program using an appropriate parallel API or library [21, 102, 103, 141]. Further assistance is required when it comes to identifying and debugging errors and performance issues in the resulting parallel code. This is where parallel-aware software development tools can come to the aid of the programmer [124]. A wide range of both free and commercial debugging and performance analysis tools are available, so the main problem lies in choosing the appropriate tools and learning how to use them effectively. Unfortunately, simply understanding the tools and methodologies for parallel programming is not enough to ensure that the resulting parallel program has good efficiency. The programmer also needs to understand the underlying platforms and architectures as they define a number of factors that affect performance, such as the importance of using cache to reduce memory access latency. The end result is that parallel programming requires more effort than sequential programming, so, where possible, parallel programming environments should attempt to reduce the effort required to develop correct efficient parallel programs [136].

There are several studies into the performance of parallel programs using various parallel programming models [6, 86, 87, 91, 14]. However, these studies tend to focus on the performance aspect and little attention is given to the quantitative analysis and comparison of the programming effort associated with implementing parallel programs using each of the selected parallel programming models. Kegel *et al.* [86, 87] present a subjective analysis of programming effort in respect to the parallel programming model, but very little work has gone into the quantitative analysis of performance versus programming effort.

1.1 Problem Statement and Research Goals

The problem that we attempt to address in this thesis, either in full or in part, is the identification, selection, and usage of appropriate libraries, language extensions, tools, and approaches for the development of efficient parallel programs in C and C++ for the Linux platform from the standpoint of a novice parallel programmer, while minimising programmer effort. Our definition

of a novice parallel programmer primarily includes intermediate and advanced level sequential programmers starting to use parallel constructs. That being said, it is becoming increasingly important for those new to programming to be introduced to parallel programming concepts and the associated shift in the way of thinking about problems and their solutions.

This problem has a number of facets, which include the analysis of the hardware characteristics of multiprocessor systems, the identification of suitable parallel programming libraries and APIs, the identification of suitable parallel programming software development tools, implementing parallelism in several programs using each of the selected parallel programming models, measuring the performance of the implementations, measuring the code metrics associated with the implementations as an estimate of programming effort, and comparing the resulting measurements between the different parallel programming models.

Therefore, the research goals of this thesis are fourfold:

1. Describe the key concepts of parallel programming and provide an overview of the organisational and architectural characteristics of commodity multiprocessor computer systems as they relate to performance.
2. Provide a short survey of current parallel programming models and parallel-aware software development tools for C and C++.
3. Present and describe techniques for parallel programming and performance optimisation.
4. Evaluate and compare the performance and programming effort of several parallel program implementations using the different parallel programming models.

1.2 Thesis Organisation

The relevant chapters of the thesis are organised as described below:

Chapter 2 introduces and discusses several important concepts related to multiprocessor computers and parallel programming.

Chapter 3 presents a short survey of common parallel programming models and libraries, as well as a variety of programming tools.

Chapter 4 describes a methodical approach to performance tuning, lists and describes a number of performance optimisations, and describes an approach to parallel programming using patterns.

Chapter 5 defines the research methodology for the empirical investigation and comparison of several parallel programming models.

Chapter 6 then describes the implementation of optimisations and parallelism in selected programs using the parallel programming models.

Chapter 7 presents, compares, and discusses the performance results and code metrics for the various implementations.

Chapter 8 summarises this thesis and presents the conclusions drawn from the research.

Appendix A lists a selection of relevant compiler options, while **Appendix B** provides the code listings for the original sequential programs.

Chapter 2

Background Work

2.1 Introduction

The field of concurrent and parallel computing has developed extensively over the last few decades, evolving from the early parallel supercomputers and mainframes to the modern massively parallel supercomputers and commodity parallel processors commonly found in desktops, servers, laptops, and even gaming consoles. The software driving these systems has also progressed substantially, resulting in a myriad of concurrent and parallel programming models and systems. Understanding these systems and the underlying hardware is critical, as so aptly stated by Herlihy and Shavit in the quote below.

> You cannot program a multiprocessor effectively unless you know what a multiprocessor *is*. [56, p. 469]

In this chapter, we present the main concepts associated with parallel computing as they pertain to later discussions. We also present a brief introduction to computer organisation, focusing on processor and memory organisations typically found in modern commodity computer hardware, since the programmer can often leverage the characteristics of the hardware to improve the performance of the target program. This is particularly evident with memory and cache systems, where the understanding and effective use of the caching mechanism can improve performance significantly.

Furthermore, we describe the performance characteristics of parallel programs and how these metrics and models can be used to determine the efficiency of parallel algorithms and programs on increasingly parallel computer systems. In addition to these performance metrics,

we describe a number of code metrics that measure or model the complexity of program code. These code metrics are used in the evaluation as a way of defining the effort expended by the programmer.

2.2 Parallel Programming Terms and Concepts

To avoid any ambiguity or misunderstanding, we first present our definitions for some of the key terms and concepts as used in this thesis.

2.2.1 Serial Computing

Serial or *sequential computing* essentially follows the von Neumann architecture, whereby a single *central processing unit (CPU)* executes a program stored in memory that specifies a sequence of read and write operations on that memory [11, 35, 115]. The problem is broken into a sequence of instructions or operations, which perform the computations necessary to solve the problem. These operations execute one after another (in other words, sequentially), with only one instruction being executed at any one time [11].

Modern CPUs and operating systems distort this description somewhat, introducing an element of concurrency to otherwise sequential programs. Hardware level improvements in modern CPUs allow for instruction-level parallelism as described in Section 2.4.1. *Multitasking* operating systems interleave the execution of multiple programs on a computer with a single processor using mechanisms such as *time-slicing*, thereby giving the illusion of parallelism [99, 103]. Serial programs that are written to be executed on a single processor, can typically run on parallel computers without modification. However, this does not mean that the program will execute in parallel, or that the program will execute exclusively on one processor as the operating system may schedule execution over different processors during the program's lifetime.

2.2.2 Parallel Computing

Parallel computing or *parallel processing* refers to the simultaneous execution of computational tasks over multiple processors to solve the specified problem [11]. This is similar to concurrency, but there is a subtle distinction between concurrency and parallel execution. Concurrency refers to the potential for simultaneous execution of independent computations within

a program. The *exploitable concurrency* is the degree to which it is possible to structure the code such that it harnesses or exploits the available concurrency [102].

A programmer harnesses this exploitable concurrency by breaking the problem down into discrete concurrent parts, and then developing and implementing a parallel algorithm using an appropriate parallel programming language or environment. When the resulting parallel program is executed on a parallel system, it is said to be executing in parallel if the instructions of concurrent computations are being executed on multiple different processors at the same time, as opposed to interleaving the execution of these concurrent instructions [11, 102]. The parallel system in question could be a single computer with multiple CPUs or processor cores, or it could be a cluster of computers connected by a network (*distributed processing*). It could even be a hybrid system where there is a cluster of multiprocessor computers [11, 99]. Parallel processing harnesses the exploitable concurrency in various problems to solve these problems in a shorter time or to solve bigger problem sets using more processors [11].

2.2.3 Tasks

For our purposes, a *task* is defined as a chunk of work or sequence of instructions that needs to be processed or executed. The task itself is defined by the programmer and corresponds to the algorithm being implemented [102, 125]. It is also important to note that tasks are not explicitly associated with any particular *thread* or *unit of execution (UE)* until runtime.

2.2.4 Processes and Threads

A *process* is a collection of resources and state, controlled by the operating system (OS). These resources include the input/output (I/O) descriptors or handles, security attributes, context information, and memory resources, including the runtime or call stack, heap, executable code and program data [99, 102, 131]. The state transitions for an executing process can be seen in Figure 2.1 [99, 131].

A *thread* can be seen as a distinct execution path within a process, with its own *instruction pointer* and *stack* and the ability to access shared memory between other threads within the process [102, 125].

In a multitasking operating system, the CPU scheduler allocates and de-allocates processor execution time for threads and processes using a *context switch* [99]. A thread or process can also

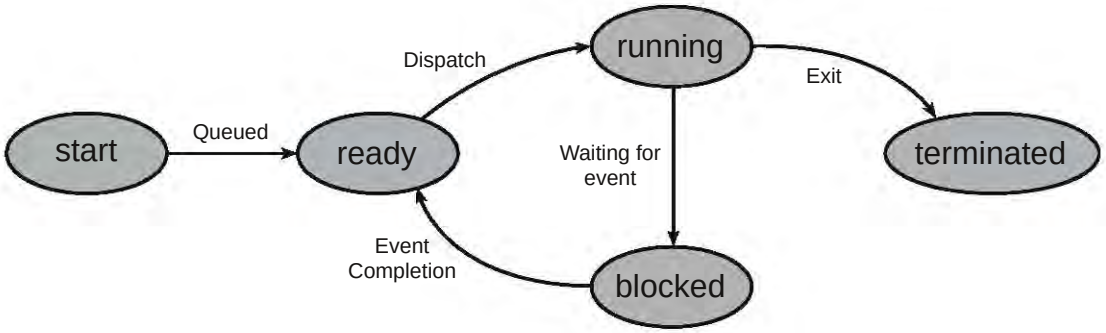


Figure 2.1: State transition diagram for process execution [99, 131].

be referred to generically as a unit of execution (UE). A process is considered a "heavyweight" UE, whereas threads are "lightweight" since they share the context of their parent process and thus have a lower context switching time [102].

2.2.5 Locks

A *lock* or *mutual exclusion lock (mutex)* is a special shared variable or abstract data type that simplistically has two states: locked and unlocked. It also has methods for locking and unlocking the object. When a thread calls the lock method for a particular mutex, the state of the mutex is checked. If the mutex is currently locked (the lock has already been acquired by another thread), the calling thread will typically *spin* or *block* until the lock becomes available, otherwise it will immediately acquire the lock and continue execution. The unlock method simply releases the lock on the mutex and makes it available to other threads [20, 56, 124].

A *semaphore* is slightly different to a mutex in that it is used to constrain access to a shared resource instead of preventing simulataneous execution by multiple threads. The semaphore keeps track of resource accesses using an integer counter that is decremented on access to the resource and incremented once a thread is done using the resource. If the counter reaches zero, threads wanting to gain access to the resource will have to spin or block until the counter is greater than zero [20, 124]. An extended description of the various types of locks can be found in Section 2.5.

2.2.6 Critical Sections, Barriers and Synchronisation

A *critical section* (CS) is a section of code that can only be executed by a single thread at any one time. This section is usually controlled by locking a critical section specific mutex and then

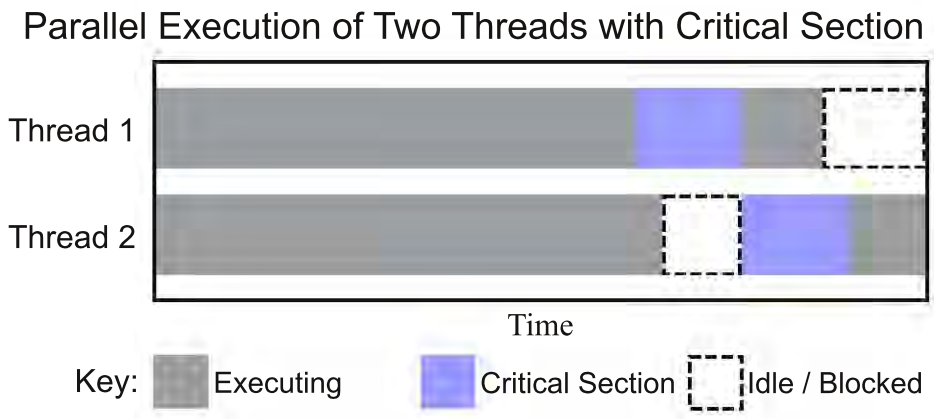


Figure 2.2: Parallel execution of two threads encountering a critical section.

unlocking the mutex at the end of the section. This prevents other threads from entering the critical section as they will block or spin while the mutex is locked. The use of mutexes and semaphores to protect critical sections forms the basis of mutual exclusion and synchronisation in parallel programming [20, 56, 99, 125].

The pseudocode example in Listing 2.1 shows a parallel `max` routine that makes use of a critical section to protect access to a global variable. Figure 2.2 shows the execution of threads encountering a critical section and demonstrates how threads block when they reach a critical section that is currently being traversed by another thread.

```
1  SET globalmax to 0
2  INIT maxmutex
3
4  SUBROUTINE parallel_max ARGS sublist
5    SET localmax to 0
6    FOR each element in the sublist
7      IF element > localmax THEN
8        STORE element in localmax
9      END IF
10   END FOR
11
12   /* start of critical section */
13   LOCK maxmutex
14   IF localmax > globalmax THEN
15     STORE localmax in globalmax
16   END IF
17   UNLOCK maxmutex
18   /* end of critical section */
19 END SUBROUTINE
```

Listing 2.1: Pseudocode example of a critical section.

Another important concept is that of a *barrier*, which is similar to a critical section except that instead of only letting one thread execute a particular section of code, the barrier forces

threads that reach it to wait or block until all threads in the group have reached that point. Once all the threads have reached the barrier, the blocked threads are notified that they may leave the barrier and continue executing [56, 105]. This is typically used in applications where progress on a new computation phase can continue only if the previous phase has been fully completed. Barriers essentially enforce synchronisation points for asynchronously executing groups of threads [56, 105].

2.2.7 Race Conditions

A *data race condition* typically occurs when two or more threads attempt to update the same shared variable. The race condition becomes apparent when one thread overwrites the changes made by another thread, thus losing the original change and affecting the correctness of the program [20, 102, 125].

Consider the following scenario. Thread *A* reads the value of variable `x` with the intent of adding 5 to the existing value and saving the result back to `x`. Thread *B*, however, also wishes to modify `x` by taking the current value and subtracting 3 before saving the result back to `x`. If Thread *B* happens to read the value of `x` after Thread *A* reads it, but before Thread *A* stores the new value, a race condition occurs as the modification made by Thread *A* will be overwritten when Thread *B* stores its result [125]. The effect of this can be seen in Table 2.1. This is typically resolved by placing the code that modifies `x` within a critical section [20, 125].

Table 2.1: Race Condition between two executing threads [125].

Thread A	Thread B	`x`
Read `x`: 7	-	7
-	Read `x`: 7	7
-	Subtract 3	7
Add 5	-	7
Update `x`: 7 + 5	-	12
-	Update `x`: 7 - 3	4

Race conditions arise due to variable thread scheduling where the OS is responsible for the allocation and timing of thread execution as opposed to the programmer [102, 125]. This can lead to unpredictable and often hard to reproduce errors when modifying shared variables. Since the error may not present itself during testing, it often goes unnoticed until it affects the program in its production environment. Even if the error does get noticed, debugging is frustrated by the fact that thread execution is nondeterministic between program runs, which makes reproducing the problem particularly difficult. Errors of this nature are prevented by carefully implementing

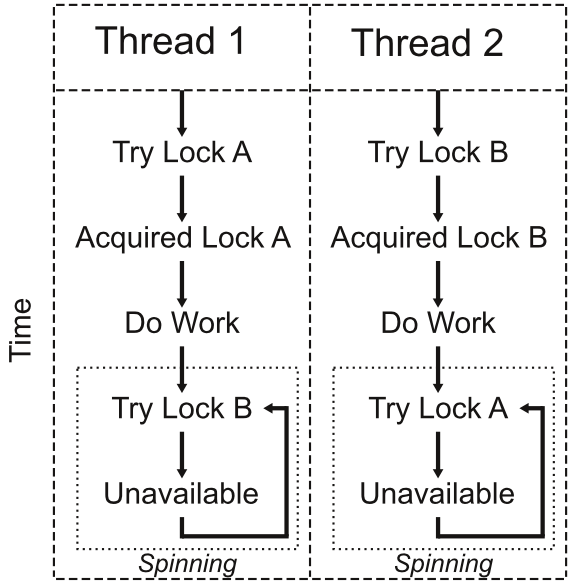


Figure 2.3: Deadlock between two executing threads. Thread 1 acquires Lock A and Thread 2 acquires Lock B. Thread 1 then attempts to acquire Lock B and Thread 2 attempts to acquire Lock A, however, both locks have already been taken, resulting in a deadlock.

and controlling access to shared variables, using mutual exclusion and synchronisation where necessary [20, 102, 125].

2.2.8 Deadlock

Deadlock occurs when at least two threads block indefinitely because they are each waiting for locks to be released that are currently being held by the other. This deadlocked state between threads results in the program never running to completion unless the deadlock is forcibly broken [20, 22, 102, 125].

If none of the threads gives up its currently held lock to allow another thread to continue, the threads will remain blocked forever as illustrated in Figure 2.3. When implementing mutual exclusion using locks, the programmer must take care to avoid implementing the locks such that deadlock can occur. One way that this can be done is by ensuring that locks are acquired in a specific order and by releasing currently held locks if all the necessary locks cannot be acquired at once and then trying again [20, 22].

Deadlock-freedom is the property whereby if some thread tries to acquire a lock, then some thread will succeed in acquiring the lock [56].

2.2.9 Starvation and Livelock

Starvation is a condition whereby a thread is unable to progress because it is unable to gain access to required shared resources, which are protected by a critical section. This is typically caused by other threads monopolising access to the shared resource for extended periods of time [96, 109]. Should the thread in question regain access to the required shared resources, it will continue executing normally. *Starvation-freedom* is the property of a program whereby if a thread reaches a critical section, that thread will eventually execute its critical section [20, 7]. Starvation-freedom implies deadlock-freedom [56].

Livelock is a form of starvation whereby threads are unable to progress and perform useful work because they, or a required resource, are too busy servicing interrupts and requests from other threads [96, 109]. This is unlike deadlock, where the offending threads are mutually blocking each other, in that the program can recover if the interrupts or requests decrease to a point where the thread can progress with useful work. *Livelock-freedom* is a weaker property, compared to starvation-freedom, and states that if a thread reaches a critical section, then some thread eventually executes its critical section [20, 7].

2.3 Parallel Performance

As with most performance improvement efforts, the ultimate goal is to improve speedup and scalability. These performance characteristics can be predicted using models or measured in practice. Some common performance measures are defined below.

- **CPU time** is the sum of the time spent actively executing the instructions of a particular computer program for each processor [54].
- **Wall clock time** or *elapsed time* is the real-world execution time of a program, including overheads and input/output delays [54].
- **Latency** refers to time delay between the issuing of a request and the response to the request [54].
- **Throughput** is a measure of the amount of data transferred or requests serviced per unit of time [54].

While measuring the actual performance of the program in question provides the best indication of speedup and scalability issues, it is often useful to predict and anticipate these metrics with the aid of the models given below. *Amdahl's Law* and *Gustafson's Law* allow one to anticipate the speedup for future systems for which neither processor counts nor workload sizes are currently available, thus preventing direct measurement. These analytical models do not predict all aspects of parallel performance, but they do provide a base for expected performance. However, profiling is the only way to accurately identify the sections of code that will provide the greatest gain from optimisation.

2.3.1 Speedup

When talking about speedup, the total running time to complete execution of the program (wall clock time) is typically taken as the primary metric. This total time is denoted by $T_{total}(P)$, where P represents the number of processors. The total time can often be decomposed into a number of *serial terms*, representing the portions of the program that cannot be run in parallel, and a compute portion, which can be distributed over as many processors as are available [102]. The serial terms usually consist of initialisation and finalisation routines that are unable to split their workload between additional processors. However, the compute portion of the program is able to execute its workload in parallel and is represented by the time to complete the compute portion on one processor over the number of available processors [102]. This relationship between the serial terms and the parallel compute section, resulting in the total time, can be seen in (2.1) [102].

$$T_{total}(P) = T_{initialisation} + \frac{T_{compute}(1)}{P} + T_{finalisation} \tag{2.1}$$

It must be noted that (2.1) only reflects the simplified case where there are no overheads such as thread startup and scheduling or locking, which affect the performance of the parallel compute section. However, this concept of serial and parallel sections of code is important and is re-examined in Section 2.3.2.

The *relative speedup* of a program, denoted by S and shown in (2.2), is another useful measure for parallel programming. It determines how much faster a particular program will run when additional processors are made available. It is calculated by dividing the total time for executing the program with one processor by the total time for P processors. While *perfect linear speedup*, where the speedup is equal to P, is the desired outcome, this is seldom possible as the serial terms are unable to execute faster with additional processors [102]. However, in some rare cases,

superlinear speedup can be achieved through efficient cache utilisation between processor cores on the same CPU, which overcomes the lack of speedup for the serial terms [21].

$$S(P) = \frac{T_{total}(1)}{T_{total}(P)} \tag{2.2}$$

Related to the concept of relative speedup is the measure for *efficiency*, denoted by E and shown in (2.3). Efficiency is simply the speedup of the program over the number of processors [102]. This measure is useful in that it provides an easier to understand measure for how effectively additional processors are utilised.

$$E(P) = \frac{T_{total}(1)}{PT_{total}(P)} \tag{2.3}$$

2.3.2 Amdahl's Law

As previously described, parallel programs consist of both serial and parallel sections of code. A problem arises when one adds increasingly more processors in an attempt to improve the speedup of the program. As the processor count increases, so does the speedup, but not at the same rate. The problem lies with the well-known Amdahl's law, defined in (2.4).

$$S_{\text{AMDAHL}}(P) = \frac{1}{(1-f) + \frac{f}{P}}, \tag{2.4}$$

where S_{AMDAHL} represents the speedup, f the proportion of the program that executes in parallel, and P the number of processors [56, 102, 125, 134]. In essence, this law states that speedup is hampered by the sequential portion of the program. As an example, given $f = 0.9$ and $P = 10$, the maximum speedup is only around 5.26x, which is clearly not ideal. The speedup value predicted by Amdahl's Law is generally an upper bound on the speedup as overheads in the parallel algorithm are not factored into the calculation. However, as mentioned in Section 2.3.1, there are cases where effective caching of data and other factors such as better resource utilisation, can result in a speedup greater than that predicted by Amdahl's Law [21, 102].

Furthermore, we can calculate the maximum attainable speedup given an infinite number of processors and an ideal parallel algorithm using (2.5) [102].

$$S_{\text{MAX}} = \frac{1}{1-f} \tag{2.5}$$

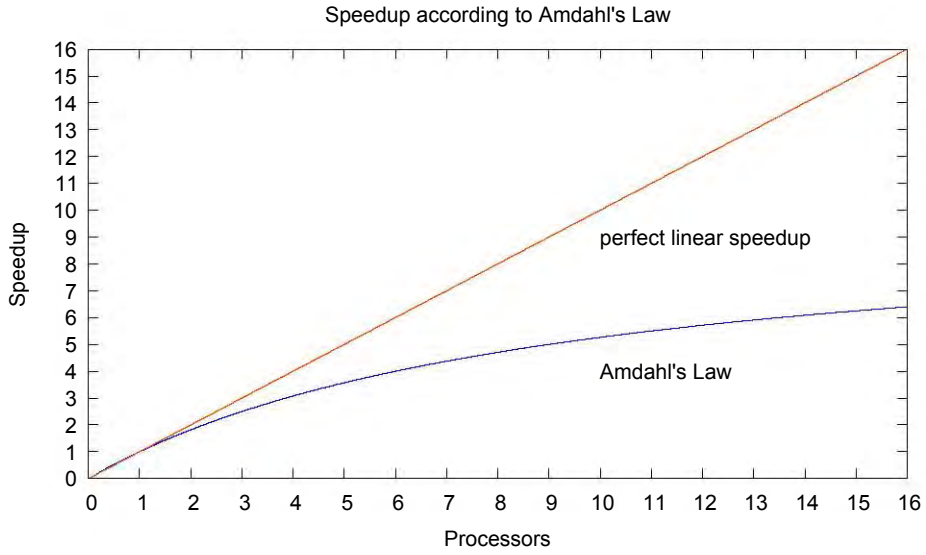


Figure 2.4: Speedup according to Amdahl's Law for $f = 0.9$

It is quite clear from Figure 2.4, which depicts the graph for (2.4) with increasing values of P against perfect linear speedup given $f = 0.9$, that there are diminishing returns to adding more processors in an attempt to solve a problem faster. Amdahl's Law represents *fixed-size speedup* and it is often considered a pessimistic outlook on parallelism [125, 134].

It is therefore crucial to make every effort to reduce the sequential portion in programs to help increase speedup [56]. However, code that works well with four processors may not scale with eight or even sixteen processor cores, so careful attention must also be paid to the implementation of the parallel algorithm and managing the associated overheads [19].

2.3.3 Gustafson's Law

The problem with Amdahl's Law is that it focuses primarily on the speedup achieved when attempting to reduce the total runtime of a fixed workload by increasing the number of processors (fixed-size speedup). While this is the goal when implementing parallelism in many of the cases, John Gustafson re-evaluated Amdahl's Law and took another approach to achieving scalability with increasing hardware capabilities [102, 125, 134].

Gustafson considered the problem in a more positive light. He noted that if the proportion of the program that is sequential cannot be easily parallelised, better scaling can be achieved by performing more work in the parallel sections. The time spent in the serial portions of a program typically increase less than the parallel sections as the problem size or workload is increased,

thus reducing the serial fraction of the program. This is often referred to as *fixed-time scaling*. The reduction in the serial fraction with increasing problem size results in improved speedup and scaling as per Amdahl's Law [48, 102, 125]. Modern society's hunger for more information and the ever increasing volumes of data and information being produced by researchers and society alike play quite aptly into Gustafson's forward-thinking and optimistic approach to scaling [102]. However, not all problem sizes can be increased, so this approach may not always be applicable.

Returning to our earlier equations for total runtime and speedup, we can reformulate some of these terms and produce Gustafson's Law in terms of performance on a P-processor system [102]. First we define the *scaled serial fraction*, denoted by γ_{scaled} in (2.6), as the serial terms over the total time for P processors [102].

$$\gamma_{scaled} = \frac{T_{initialisation} + T_{finalisation}}{T_{total}(P)} \tag{2.6}$$

We derive the computation time on P processors in terms of the total time on P processors and the scaled serial fraction in (2.7).

$$\begin{aligned} T_{compute}(P) &= T_{total}(P) - (T_{initialisation} + T_{finalisation}) \\ &= T_{total}(P) - \gamma_{scaled} T_{total}(P) \end{aligned} \tag{2.7}$$

We then define the total time for one processor in terms of the parallel computation time and the serial terms for P processors, substituting in the equations from (2.6) and (2.7), to produce (2.8) [102]. Following on from this, we rewrite the equation for speedup as expressed in (2.2), taking into account the the scaled serial portion and simplifying, which results in Gustafson's Law as defined in (2.9) [102].

$$\begin{aligned} T_{total}(1) &= T_{initialisation} + PT_{compute}(P) + T_{finalisation} \\ &= \gamma_{scaled} T_{total}(P) + P(T_{total}(P) - \gamma_{scaled} T_{total}(P)) \\ &= \gamma_{scaled} T_{total}(P) + P(1 - \gamma_{scaled}) T_{total}(P) \end{aligned} \tag{2.8}$$

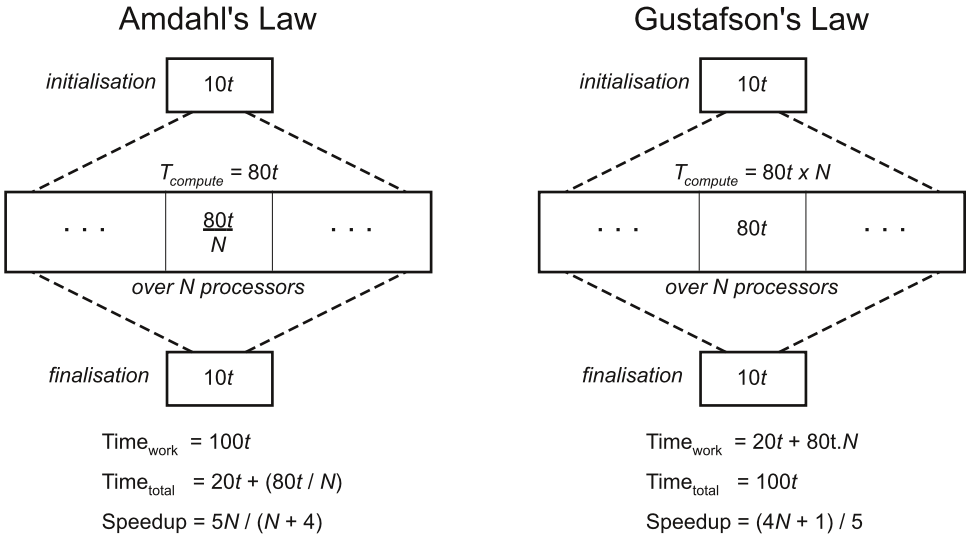


Figure 2.5: Scaling according to Amdahl's Law and Gustafson's Law [125].

$$
\begin{aligned}
S_{\text{GUSTAFSON}}(P) = \frac{T_{total}(1)}{T_{total}(P)} &= \frac{\gamma_{scaled} T_{total}(P) + P(1-\gamma_{scaled}) T_{total}(P)}{T_{total}(P)} \\
&= \gamma_{scaled} + P(1-\gamma_{scaled}) \\
&= P + (1-P)\gamma_{scaled}
\end{aligned}
\tag{2.9}
$$

An interesting observation can be made about the above equation. If the time taken by the serial terms remains constant and we keep the size of the computation for each processor the same, eventually we reach a point where the speedup increases at the same rate as the number of processors, which is known as *linear* or *order of n scaling* [125, 102].

The illustration in Figure 2.5 highlights the key difference between Amdahl's Law (fixed-time scaling) and Gustafson's observations (fixed-size scaling), showing that scaling can be improved by increasing the problem size. This ultimately results in far more effective use of the available processing power compared to a static workload [125]. Depicted in the illustration is a program with serial terms, which perform initialisation and finalisation, and a parallel computation section executing concurrently on N processors. Time taken for each section of work is represented in the arbitrary unit t, and summaries of total times and speedup (highlighting the speedup limitations of Amdahl's Law) are presented below each diagram.

Sun *et al.* [134] note that the *memory-wall problem*, referring to the performance gap between memory and processors, is likely to play a greater role in determining speedup in the future.

2.3.4 Sequential Algorithms versus Parallel Algorithms

While Amdahl's Law and Gustafson's Law provide a means for us to predict speedup for programs running on a parallel system, they are based on the flawed assumption that the program must execute roughly the same number and type of instructions for both the serial and parallel versions. However, there are cases where simply adapting the original serial algorithm to work in parallel is inefficient and a more natural parallel implementation exists. In cases such as this, it is often worthwhile re-examining the problem and designing a more suitable parallel algorithm [125].

2.4 Computer Organisation

At a high level, a computer is made up of a number of components, namely, the central processing unit (CPU), memory (both primary and secondary storage memory), input and output (I/O) devices or components, and the interconnection structure between these components. The CPU executes the instructions that perform the operations necessary for the computer to function and carry out the given tasks. Computer memory is responsible for storing the instructions and data necessary for the operation of the computer. There are typically two kinds of memory: primary memories such as the RAM used by main memory and various caches; and the secondary or storage memory, which is used to store data on a more permanent basis [131]. I/O devices facilitate interaction between the computer and the user, through devices such as the keyboard and graphical display, or other devices and computers through components such as network interface cards (NIC) or serial ports. Finally, the interconnection network, typically in the form of a hierarchy of buses, is responsible for carrying address, control, and data signals between the various components in the system [131].

The performance of a computer depends on a multitude of factors, both hardware and software. Arguably, some of the greatest contributing factors are those pertaining to computer organisation for a given architecture. Organisational design choices such as the memory types, sizes, and hierarchy play an important role, especially given the widening gap between processor and memory performance [54, 131]. Our research focuses on software targeting commodity hardware and processors based on the x86 and x86-64 (64-bit) architectures, as these are the most common architectures due to their cost-effectiveness compared to larger specialised systems [27, 54]. As such, the computer organisation issues discussed in this section typically center around modern commodity hardware.

While at first, it may seem that these organisational factors only affect hardware designers and operating system programmers, many of them have consequences for application programmers in terms of how their programs are structured and coded. A good example of this is the effect on performance of good or bad cache utilisation, which can often be influenced by the programmer. As such, the key concepts regarding memory and processor organisation are presented below with particular emphasis on those factors affecting the programmer, or which can be influenced by the programmer in some manner.

2.4.1 Processor Organisation

Processor organisation describes the characteristics of the processor and how one or more processors are connected to each other and to the rest of the system. In this section we briefly discuss the characteristics of modern CPUs and how they affect performance. We then go on to classify computer systems according to *Flynn's Taxonomy*, and describe common processor organisations for commodity hardware.

CPU Characteristics

The CPU or processor is central to the operation of a computer system. It is responsible for controlling the operations of the computer and executing instructions, thus enabling the computer to perform its allocated tasks [131]. The major components of a CPU in the von Neumann architecture, which paved the way for software in the form of stored programs, are described below and illustrated in Figure 2.6.

- **Control Unit (CU):** The control unit is responsible for fetching and interpreting instructions, and then issuing the necessary control signals to execute the instructions. Obtaining instructions and data from memory and then issuing control signals to general-purpose logic and arithmetic components to carry out those instructions gives these general-purpose processors the ability to execute instructions written by the programmer in the form of *software* programs. This is in contrast to *hardwired* programs where the logic components are designed and connected, at the hardware level, to take the input data and process it in a specific way. For such a system, changing the operation of the program requires that the hardware be rewired [131]. The architecture or *instruction set* of the processor determines the addressing modes and the types and formats of the instructions interpreted by

the control unit. Architectures fall into two broad categories: reduced instruction set computer (RISC) architectures, which are characterised by a large number of general-purpose registers and simple instruction sets with optimised instruction pipelines; or complex instruction set computer (CISC), architectures, which are characterised by a greater number of more complex and specialised instructions and fewer general-purpose registers [131].

The processing required to complete the *fetch cycle* and *execute cycle* for a single instruction, is known as an *instruction cycle*. The *interrupt cycle* must also be considered in this process. Devices and other components within the system can make *interrupt requests*, which are essentially events that cause the sequence of instructions for the current program to be suspended in order to service the appropriate interrupt handler instructions. After the interrupt has been dealt with, execution of the previously running program is automatically resumed [54, 131].

- **Registers:** The CPU contains a number of internal registers, which are a special type of memory or storage used to hold data that is necessary for the operation of the processor. The architecture of the processor determines the types and sizes of the registers that are implemented. For instance, a *stack-based* architecture arranges the registers to be accessed as a stack using `Push` and `Pop` instructions, whereas a *load-store* type of architecture will have a set of general-purpose registers that are accessed explicitly using `Load` and `Store` instructions [54].

  As an example, a hypothetical *accumulator* architecture may have the following general-purpose registers: a *program counter (PC)*, which tracks the address of the next instruction; an *instruction register (IR)*, which holds the operation code or *opcode* for the current instruction; a *memory address register (MAR)*, which holds the memory address to be read or written to; a *memory buffer register (MBR)* where the resulting data of the memory read is stored or where the data to be written is temporarily buffered; and an *accumulator (AC)*, which holds the temporary results of operations performed by the ALU [131].

- **Arithmetic and Logic Unit (ALU):** The ALU performs the actual calculations and logic operations required to execute instructions. General-purpose arithmetic and logic components are found in the ALU and are configured to accept control signals from the control unit. The control signals specify how these components are combined so as to perform the desired operation. Some typical examples of arithmetic operations implemented in the ALU are the *add*, *subtract*, *multiply*, and *divide* operators. Integer and floating point data is treated differently, so there may be different implementations of these operators depending on the data type of the operands. Logic operations will typically include the *equal to*, *greater than*, and *less than* comparison operators, amongst others [131].

While the architecture prescribes which instructions need to be implemented, the manner in which these instructions are implemented is an organisational issue. It is up to the CPU designers to decide how the arithmetic and logic components are utilised to perform the required instructions. For example, the multiply operator can be implemented as a special multiply unit or the same effect can be achieved by using the add operator repeatedly [131]. These kinds of organisational design choices lead to CPUs within the same architecture family having different physical and performance characteristics.

- **CPU Interconnection:** An interconnection structure is needed to enable communication in the form of control, address, and data signals between the various components within the CPU as well as to the rest of the system. There are a variety of interconnection topologies including direct point-to-point, bus, hypercube, crossbar switch, mesh, ring, and torus. The bus interconnect is the most common structure for connecting the main system components. Hierarchies of buses are also typical in modern computer systems, with a local *front-side bus* on the CPU, which is connected to the system bus and lower buses, such as the expansion buses [115, 131]. Early processors used point-to-point links and buses internally, but crossbar switches, meshes, and ring networks have become common as well [54, 131]. The AMD Opteron range of CPUs support multiple HyperTransport links between CPU nodes in a multiprocessor system, which is based on the hypercube topology [27]. The torus, ring, and hypercube topologies are typically used to connect the nodes in supercomputers and clusters of computers [54, 124].

  The performance of a bus or other interconnect is characterised by the *bandwidth* and *latency* of the interconnect, which are affected by the *bus width* and *frequency* of the connections. With regard to buses, bandwidth is typically understood to be the maximum rate of data or information transfer and can be measured in bits per second. Latency is the time it takes for a packet of information to complete its transmission between the source and destination [54, 115]. Bus width is the number of bits that can be transported over the bus in parallel, while frequency (clock frequency) is the number of cycles per second. The frequency and bus width are multiplied to produce the theoretical maximum bandwidth and in some cases, this can also be multiplied by the number of transfers per cycle for buses where multiple transfers per cycle are possible [131].

At the individual processor level, performance is determined by a number of factors, such as the *clock speed* or *clock frequency* and the number of *cycles per instruction* (CPI), although CPI is not directly comparable with other processors and instruction sets [54]. Instruction-level parallelism, as described in the next section, lowers the CPI by optimising the instruction pipeline and is critical to improving processor performance. The clock speed of a processor,

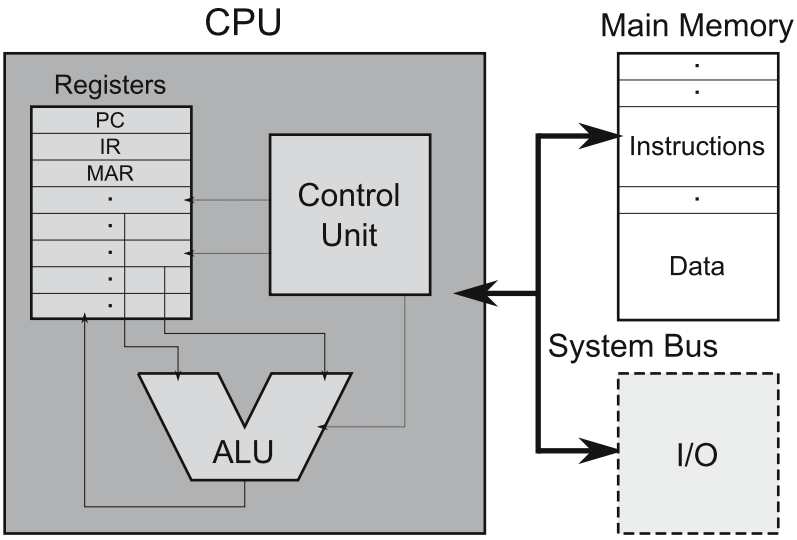


Figure 2.6: Components of a computer and the central processing unit in the von Neumann architecture [115, 54].

which is governed by a clock running at a constant rate, indicates the number of cycles executed per second by the processor. The clock speed is typically described by its rate (e.g., 2.4GHz, which is 2.4 billion cycles per second) [54]. Therefore, an increase in the clock speed of a processor results in more instructions executed per second and thus greater performance.

Historically, increases in processor performance have been brought about by increases in clock speed, along with incremental organisational improvements. This has been made possible by improvements in the fabrication process of integrated circuits, which have enabled the manufacturing of ever smaller electronic components, such as transistors. The reduced size of the components in turn, has allowed for these components to be packed more densely on a chip. This forms the basis of Moore's Law, derived from the observation by Gordon Moore, the co-founder of Intel, that the number of transistors that could be placed on a chip would double every year (this has recently slowed to doubling every 18 months) [53, 124, 131]. The so-called "scaling laws", observed by Robert Dennard and his colleagues at IBM, also derive from the *process shrink* in the manufacturing of chips. Dennard's first observation was that the voltage and current, and thus the power consumption of a transistor, is proportional to its area. This means that as these transistors get smaller and are placed more densely on a chip, the power density of the chip remains constant. The second observation was that the switching delay of a transistor is proportional to its size, and therefore the frequency or speed of the transistor increases as the transistor gets smaller [53].

However, as the size of transistors decreases and the voltages are reduced, they begin to leak more current. Therefore, the voltage supplied to the transistors needs to be decreased at a lower rate than the decrease in transistor size to manage this leakage of current. This breaks the proportional power scaling and results in an increase in power density, which means that the chip produces more heat that needs to be dissipated in some way. Consequently, the frequency cannot be increased as easily because the voltage cannot be decreased and increasing the frequency without decreasing the voltage results in more heat. This places practical limits on the frequencies attainable by chips such as CPUs, based on how effectively the chips can be cooled [53, 124, 131]. It is for this reason that the manufacturers of modern commodity CPUs have turned to *multicore* designs. Since the number of transistors that can be packed on a chip is still increasing, it is possible to keep the processor frequency relatively constant and use the additional transistors to implement multiple processor cores on a single CPU package. This improves the performance of the CPU by adding parallel processing capabilities or *thread-level parallelism* [53, 54, 131].

Instruction-Level Parallelism

One particularly important organisational performance improvement is that of *pipelining*. As previously described, the instruction cycle consists of a fetch cycle and an execute cycle, which can be further broken down into a number of stages: fetching the instruction; decoding the instruction; calculating the addresses for operands and fetching them from memory; executing the instruction with the given operands; and finally storing the result back in memory. Without pipelining, the processor would have to wait for an instruction to pass through all of these stages before proceeding with the next instruction. However, with pipelining, it is possible to have multiple instructions in flight simultaneously, each at a different stage in the instruction cycle. This allows the processor to keep all stages of the instruction cycle busy, greatly improving performance [54, 82, 131]. Several conditions can affect the performance of pipelining due to *pipeline conflicts*, namely, resource conflicts, data dependencies, and conditional branch statements [54, 115]. Superpipelining extends this concept by adding a greater number of small stages, creating a deeper pipeline, which allows more instructions to be in the pipeline simultaneously [54, 82, 131].

Another important organisational improvement is the introduction of *superscalar* processors. In a superscalar processor, there are multiple instruction pipelines, which allows for independent instructions to be executed concurrently on a single processor [54, 82]. The goal of such processors is to enable the issuing of multiple instructions per clock cycle, resulting in a CPI value lower than one. These are also known as *multiple-issue processors* [54]. Both statically

and dynamically scheduled superscalar processors are available, with the Intel Itanium range being an example of statically scheduled superscalar processors, and the Intel Pentium range exhibiting dynamically scheduled superscalar features. Static scheduling relies on the compiler to schedule code for the processor and is typically characterised by *in-order* issuing and execution of instructions. Speculative dynamically scheduled superscalar processors, like the Intel Pentium 4, allow for *out-of-order* issuing and execution of instructions. Out-of-order execution with speculation means that the processor's scheduler is able to fetch, issue, and execute independent instructions from outside the normal, sequential instruction order, thus allowing for better utilisation of the multiple instruction pipelines [54]. This can be combined with pipelining or superpipelining, resulting in a *superpipelined superscalar processor*. Both pipelining and superscalar processors exhibit an *instruction-level parallelism* [131].

Modern commodity CPUs typically implement a number of extensions to the instruction set, such as Multimedia Extensions (MMX) and the numerous versions of Streaming SIMD Extensions (SSE) [54, 131]. The instructions added by MMX and SSE are primarily floating-point operations for performing calculations on single-precision and double-precision floating point data. SSE also adds instructions for cache prefetching and streaming store instructions that bypass the cache [27, 54]. These extensions may also add additional registers and ALU components to support the new instructions, as in the case of SSE, which adds 128-bit registers and support for performing four 32-bit or two 64-bit floating-point operations in parallel [54]. These SSE instructions provide vector operation capabilities much like SIMD based architectures, which are discussed briefly in the following section.

While these organisational improvements are at the hardware level, it may be possible to exploit this knowledge at the software level in some cases by writing code that makes it easier for the processor's instruction scheduler to perform its job more effectively. Some of these techniques are discussed in Chapter 4.

Flynn's Taxonomy

Since the inception of computer systems, numerous serial and parallel architectures and organisations have been proposed. Each system differs from the others in some way, whether it be the target application or the interconnect between processors [102]. However, these differing architectures can easily be characterised by the number of instruction and data streams using Flynn's Taxonomy as described below and illustrated in Figure 2.7 [32, 102].

- **Single Instruction, Single Data (SISD)** refers to a machine in which a single instruction

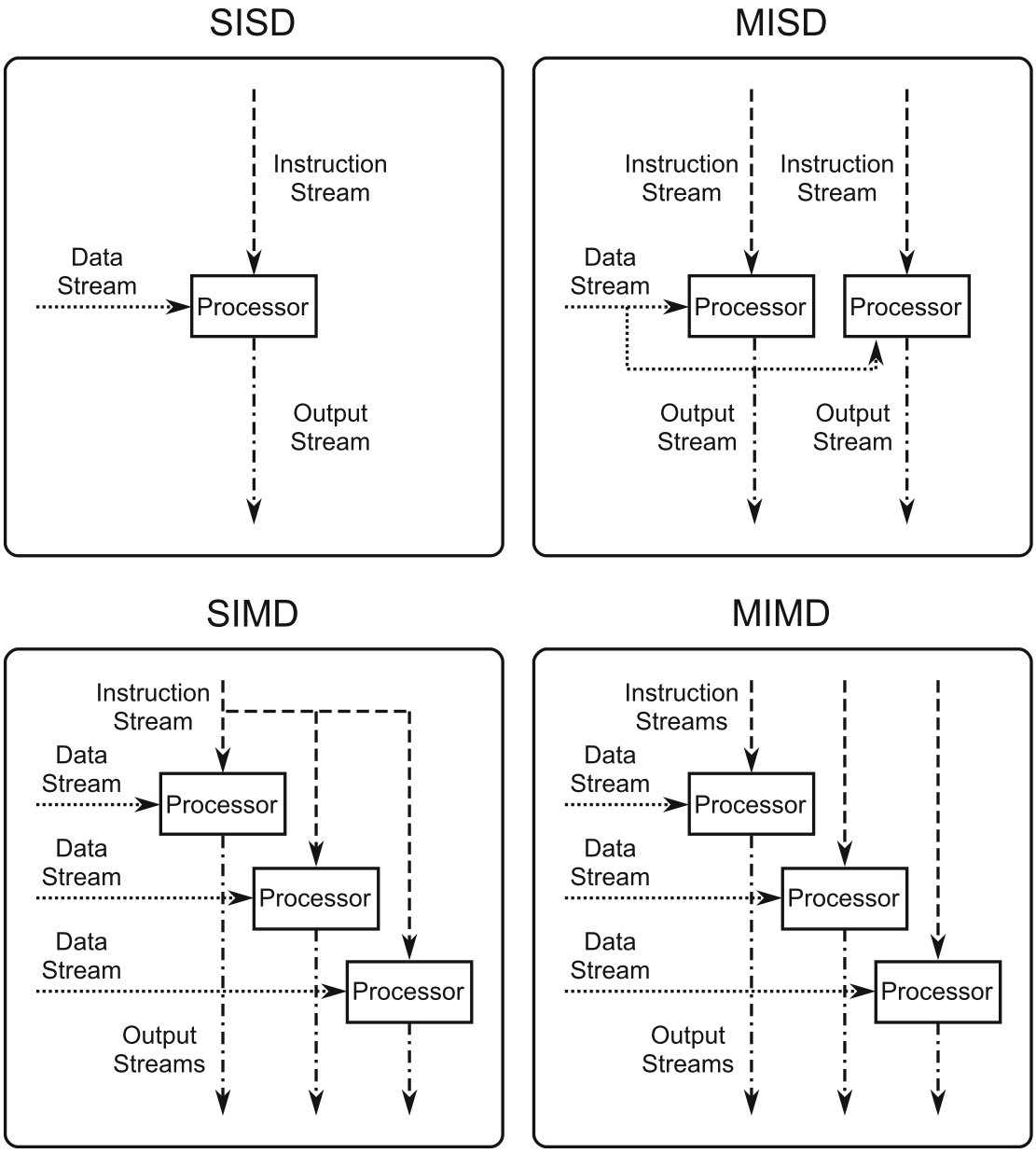


Figure 2.7: Flynn's Taxonomy [32].

stream acts upon a single data stream. Uniprocessor (UP) computer systems fall into this category [32, 102].

- **Single Instruction, Multiple Data (SIMD)** refers to a machine in which a single instruction stream acts upon multiple data streams simultaneously [102]. This is one type of parallel architecture with the most notable examples being vector processors and the modern Graphics Processing Unit (GPU). This kind of architecture is best suited to fine-grained parallelism as seen in many digital signal processing and multimedia applications [32, 102, 131].
- **Multiple Instruction, Single Data (MISD)** is a very uncommon type of machine in which multiple instruction streams all act upon a single data stream [32, 102].
- **Multiple Instruction, Multiple Data (MIMD)** is the architecture most relevant to our interests as most parallel computers fit into this category, including our modern multi-core processors. MIMD is characterised by multiple instruction streams, each acting on its own independent data stream. It is the most general of the architectures [32, 102]. The MIMD architecture, specifically the Shared-Memory Multiprocessor, is the most relevant to our research. As such, a detailed discussion of this architecture can be found in Section 2.4.1.

Uniprocessors

Uniprocessor systems are computer systems with a single logical processor and fall under the SISD category of Flynn's Taxonomy as defined above. They are the oldest type of computer system and represent the von Neumann model of sequential computing as described in Section 2.2.1 [35, 99]. Single processor computers have been and are likely still the predominant type of computer system available, receiving many architectural and organisation improvements over the years. However, the prevalence of multicore and multiprocessor computers systems is on the rise, with most new computer systems shipping with at least a dual-core CPU. This is due to the so-called "power wall" encountered by CPU designers, preventing them from increasing the speed of single-core CPUs much beyond current levels [11, 53, 131].

Distributed Systems

Distributed systems and cluster computing systems involve groups or clusters of interconnected, yet individual computers, each with its own processors, memory, and other resources. The interconnection network varies with the nature of the distributed system being employed, ranging from very fast links such as InfiniBand to slower links such as Ethernet [99, 131]. These

computers collaborate over the network to solve problems in parallel and typically employ *message passing* as the means for data distribution, interprocess communication, and synchronisation. Message passing requires the programmer to program all interprocess communication and distribution of data explicitly by specifying the required messages. *Message Passing Interface (MPI)* is a well-known *application programming interface (API)* for message passing, which provides various routines for sending and receiving messages, as well as controlling processes [11, 99, 102].

Render farms are a good example of cluster computing systems, acting as large compute clusters to render the *computer graphics (CG)* scenes used in movies. The rendering of large animated movies and CG scenes in modern blockbuster movies is a highly concurrent task as the scenes can be easily divided and distributed between multiple computers, each of which can then render its scenes independently of the others [102].

Symmetric Multiprocessors

Symmetric Multiprocessor (SMP) computer systems refer to standalone computers, controlled by a single SMP aware OS, with two or more similar processors that have the same capabilities, and that are connected to a single main memory via a shared interconnect such as a bus. Most commodity multiprocessor systems employ SMP based designs due to their simplicity, however, NUMA (described in Section 2.4.3) based systems are becoming more popular as processor counts increase [131, 56, 102].

SMP systems enable parallel execution of a single program using *multithreaded programming*. This is where a program's instruction stream is split into separate, concurrent instruction streams, known as threads, that are able to execute in parallel over multiple processors instead of interleaving execution on a single processor. This requires the software programmer to develop explicitly multithreaded code to exploit the parallelism of an SMP system [131]. SMP based systems are typically easier to program than clusters and other distributed systems as the programmer is presented with a single global address space, much like uniprocessor systems [54, 102].

The operating system also has to support SMP in that it must be able to schedule and balance threads over multiple processors, otherwise the additional processors sit idle while one processor does all the work. SMP computer systems are also beneficial for programs that are not multithreaded as the OS can schedule separate processes and interrupts to be run on different processors, improving the responsiveness of the system [54, 102].

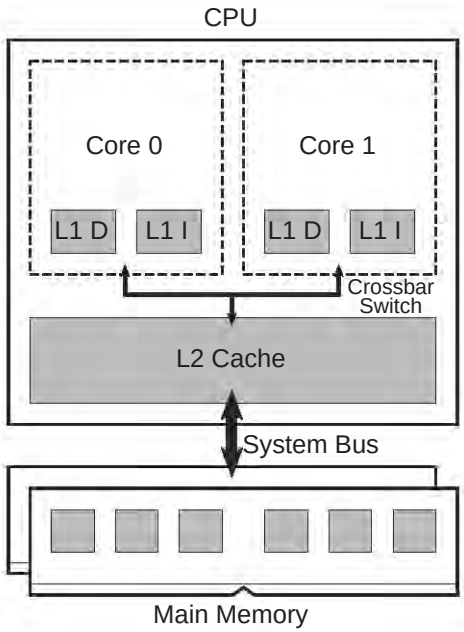


Figure 2.8: Dual-core chip multiprocessor. Each CPU core has its own small L1 data (L1 D) and instruction (L1 I) caches. Each core also has access to a large, shared L2 cache (combined data and instruction) via an internal crossbar switch. Main memory is accessible via a much slower system bus.

Early workstation and server SMP systems used motherboards with more than one CPU socket on which additional single-core CPUs were installed. These CPUs communicate with each other, I/O devices, and main memory via the shared system bus. More recently however, CPU manufacturers like AMD, IBM, Intel, and Sun Microsystems have focused on producing CPUs with increasing numbers of processor cores on a single chip. These multicore CPUs, or *chip multiprocessors (CMP)* as they are sometimes called, are essentially SMP architectures at the core level where two or more identical, self-contained (i.e., with its own ALU, CU, registers, and cache memory) processor cores are connected via an internal interconnection network. This internal interconnect then connects these cores to the system bus and any shared cache [131, 54, 124]. A simple, hypothetical dual-core CMP with a shared cache is illustrated in Figure 2.8.

As with single-core CPUs, multiple CMPs can be connected together on a motherboard with multiple CPU sockets, further increasing the parallel processing capabilities of the system. Alternatively, independent SMP/CMP systems can be connected in a cluster or distributed system. The latter case is typically referred to as a hybrid system [102]. Depending on the layout of the memory banks, the former case may exhibit NUMA characteristics and thus no longer conform to the SMP architecture. This is typically true for cases where each CMP is connected to its own

memory bank, which is then shared with the other CMPs over the interconnection network [54].

Simultaneous multithreading (SMT) is another type of SMP organisation seen in some CPUs. SMT is implemented by duplicating execution units within a single processor or CPU core, thus making a single physical processor appear as two logical processors. This allows the processor to execute two threads simultaneously for only a minor increase in CPU components. Certain models in the Intel Pentium 4 CPU range (and more recently, the Intel Core i7 range and associated Xeon models) implement SMT, which they refer to as *hyperthreading* [124, 131].

The interconnection network, particularly the bus and switch based interconnects, is the biggest limitation to the scalability of processors in an SMP system. As processor counts increase, the interconnect traffic reaches a point where it saturates the available bandwidth on the shared interconnection network, or the cost of connecting the additional processors becomes too high. Faster and wider interconnects alleviate this to some extent, but there is also a limit to the complexity and power requirements of such interconnects. The response to this problem comes in the form of NUMA based systems, which are discussed in Section 2.4.3 [115].

An important factor in the performance of an SMP system is the nature and implementation of the cache coherency mechanisms in relation to the type of interconnect used. Some of the relevant aspects and mechanisms for cache coherency as they pertain to SMP are discussed in Section 2.4.2.

2.4.2 Computer Memory

A computer system's memory is used to store the data and instructions necessary for the computer to operate and run the user's software applications. Some of the primary characteristics of memory are:

- **Capacity and addressable units** – The size or capacity of memory is typically measured in bytes or *words*. A word is the unit of organisation of memory. The word length is dependant on the processor architecture, but it is usually equal to the instruction length of the processor. The *addressable unit* is the granularity to which the processor can reference or address locations in memory. A word is the typical addressable unit, but some architectures allow byte level addressing. The total number of addressable units depends on the address length [131].
- **Unit of transfer** – The *unit of transfer* is the number of bits that can be read from or written to a particular type of memory at once. A unit of transfer that spans multiple words is referred to as a *block*. Block sizes vary with memory type [131].

- **Access method** – *Sequential access* refers to types of memory that are organised into records that must be accessed in a specific linear order. *Direct access* involves accessing areas of memory directly by referring to the unique address of the desired block which is based on the physical location of the block in memory. The memory at the target location is then searched to find the addressed block. *Random access* allows for directly referencing and accessing arbitrary memory locations, resulting in a constant access time. *Associative access* is similar to random access, except that it addresses and accesses the desired memory location by simultaneously checking the contents of all words in memory for a match to a specific tag [131].

- **Performance** – Memory performance is critical to overall system performance. The *access time* or *latency* is the time it takes from the moment an access request (a read or write operation) for a memory location is made to the moment that the request is fulfilled. Access time is sometimes measured as the number of elapsed processor cycles. The *transfer rate* refers to the speed or rate at which data is transferred to or from memory [131].

- **Volatility** – *Volatile* memory requires a charge to retain its current value, meaning that it cannot hold data while the computer is off. Many semiconductor based memories are volatile. *Non-volatile* memories, such as magnetic or optical storage, do not require power to retain data, making them suitable for long-term storage even while the computer system is powered down [131].

There are many different types of memory, varying in speed, capacity, access time, and cost. These memory types are organised in a *memory hierarchy* with the faster, yet smaller and more expensive memories at the top, and the larger, yet slower and cheaper memories at the bottom. At the top of the hierarchy, we find registers, which are very small, yet very fast specialised memory stores built into the CPU. Next in the hierarchy is *cache memory*, which is also found on the CPU and is itself organised in a hierarchy of cache levels. Cache memory has a very low access time, usually in the 0.5 ns to 25 ns range, and is relatively small compared to main memory. The capacity of cache memory varies from just a few kilobytes to the low megabyte range depending on the cache level and model of processor. *Main memory* or *primary memory* has a slower access time than cache, but it has a significantly lower cost-per-bit. This lower cost means that main memory has far greater capacity than cache, with modern commodity computer systems ranging from hundreds of megabytes to multiple gigabytes of main memory. Registers, caches, and main memory are all classified as internal memory and are volatile [54, 115, 131].

The remaining types, such as secondary and removable memory, are classified as external memory and are non-volatile, meaning that they can be used for long-term storage. These memories

offer substantially greater capacity (up to terabytes of storage capacity) and lower cost-per-bit at the expense of slower access times and transfer rates. With CPUs improving in performance at a greater rate than main memory, there is the temptation to make use of larger quantities of the faster memory technologies (as seen in caches) in place of the current main memory technologies. However, this is prohibitively expensive and mostly unnecessary as the memory hierarchy is able to hide much of the access latency by exploiting *locality* through techniques such as *caching* and prefetching data [54, 115, 131].

There are a number of concepts relating to memory accesses that have an effect on the performance of the memory hierarchy. Some of the concepts are described below.

- **Hit** – *Hit* refers to a memory request that is satisfied by a given level of memory [115].
- **Miss** – A *miss* occurs when the requested data is not located in the desired level of memory and needs to be retrieved from a lower, slower level of memory [115].
- **Hit rate and miss rate** – The *hit rate* for a particular level of memory refers to the percentage of memory accesses satisfied by that level of memory. The *miss rate* refers to the percentage of memory accesses that cannot be satisfied by a given level of memory [115].
- **Hit time** – The *hit time* is the time it takes to access the requested data for a particular level of memory in the event of a request hit [115].
- **Miss penalty** – The *miss penalty* is the time it takes to retrieve the requested data from a lower level of memory, store it in the current level of memory, and deliver the data to the processor that requested it when a miss occurs [115].
- **Memory stall cycles** – This refers to the number of clock cycles during which the processor is stalled while waiting for data to be returned from memory. It is dependent on the clock speed of the processor and the miss penalty of memory in question [54].
- **Locality of reference** – *Locality of reference* refers to the tendency of memory accesses to cluster during a program's execution. This is particularly evident for loops and subroutines where a limited set of data and instructions are referenced repeatedly as the program iterates through the loop or makes frequent calls to a particular subroutine. *Spatial locality* refers to accesses that are clustered in a particular address space, whereas *temporal locality* is a form of locality whereby recently accessed elements tend to be accessed again within a short period of time. The last form, whereby instructions are usually accessed sequentially, is known as *sequential locality* [54, 115, 131].

As stated previously, it is too expensive to make exclusive use of the fastest memory technologies. However, by taking advantage of locality, the slower access times of memory lower in the memory hierarchy can be mitigated. This is achieved by ensuring that the data and instructions that are most likely to be used in the near future are available in the faster levels of memory. This results in a better hit rate for fast memories like the cache, thereby avoiding the performance impact of miss penalties that can stall the processor for hundreds to thousands of cycles [54].

Main memory and cache memory are described in greater detail below. We also introduce the concept of *virtual memory*.

Main Memory

The main memory of a computer system typically uses random access memory (RAM). Logically, main memory is represented as a linear array of addressable locations. It is connected to the system bus or front-side bus of the CPU via the memory controller in the Northbridge chip. More recent CPUs now come with integrated memory controllers, allowing main memory to be connected to the CPU directly [27, 115].

Main memory is generally considered to be a medium-speed memory in terms of the memory hierarchy. There are two main types of RAM: *static RAM (SRAM)* and *dynamic RAM (DRAM)*. SRAM is faster, but more expensive than DRAM. As such, it is used for cache memory instead of main memory. Unlike SRAM, DRAM uses capacitors that require constant refreshing to maintain cell state and introduce a delay when reading or writing values to a cell. DRAM cells are organised in an array of rows and columns that need to be individually selected to be accessed [27, 54, 115].

Main memory in modern commodity computer systems uses *Double Data Rate DRAM (DDR)* as opposed to the older *Synchronous DRAM (SDRAM)*. The memory controller dictates the clock frequency of the DRAM modules. The key difference between DDR and SDRAM is that DDR is able to transfer data on both the rising and falling edges of the clock cycle, thus doubling the effective frequency. There are currently three standards for DDR memory, namely DDR1, DDR2, and DDR3, each of which improves on the previous standard with increased frequencies and lower power requirements. The unit of transfer for modern memory controllers is typically around 64 bits, which can then be multiplied by the effective bus frequency to produce the maximum transfer rate [27].

Cache Memory

Cache memory is a very fast, but small memory that sits between the processor and main memory. It is usually implemented on the CPU itself using fast, expensive SRAM. The primary purpose of cache memory is to improve the *average memory access time* and approach an access time as close as possible to that of the fastest memory available, using only small amounts of this expensive memory type. This is achieved by attempting to cache frequently accessed data so that future requests for that data result in a cache hit and can be served directly from the much faster cache, as opposed to having the processor stall while it waits for the data from main memory [54, 115, 124, 131].

Cache memory is divided into a number of fixed-length blocks known as *cache lines* or *cache blocks*. Each cache line contains a fixed number of words, as defined by the *cache line size*, and a *tag* that identifies the contents of the line. The cache line size is defined by the processor's architecture and cannot be reconfigured for that particular model of CPU. The overall size of the cache determines the number of available cache lines. While increasing the cache size can improve performance by allowing more blocks to be cached, larger caches tend to have longer access times. For the purposes of caching, main memory is also considered to be divided into equal-size blocks as per the cache line size [54, 115, 124, 131]. This concept is illustrated in Figure 2.9.

When a word within a particular block of main memory is accessed, the whole block is copied into the appropriate line in cache memory. This ensures that nearby memory locations are available in the cache before they are requested by the processor, thereby taking advantage of spatial locality to reduce cache misses. Cache misses can be further reduced by keeping active cache lines in cache memory for as long as they are needed, which takes advantage of temporal locality. However, space in the cache is limited, so a block cannot permanently occupy the cache and eventually it will be necessary to evict the cache line to make space for new blocks. This is managed by the *cache mapping algorithm* and the *replacement algorithm* of the cache controller [124, 131].

A cache mapping algorithm maps blocks from main memory into their corresponding cache lines. It is also responsible for ascertaining which main memory block occupies a particular cache line. When the mapping algorithm assigns a memory block to a cache line, it records the information necessary to distinguish this block from other blocks with the same cache line mapping in the tag field of the cache line [124, 131]. The three cache mapping schemes are described below.

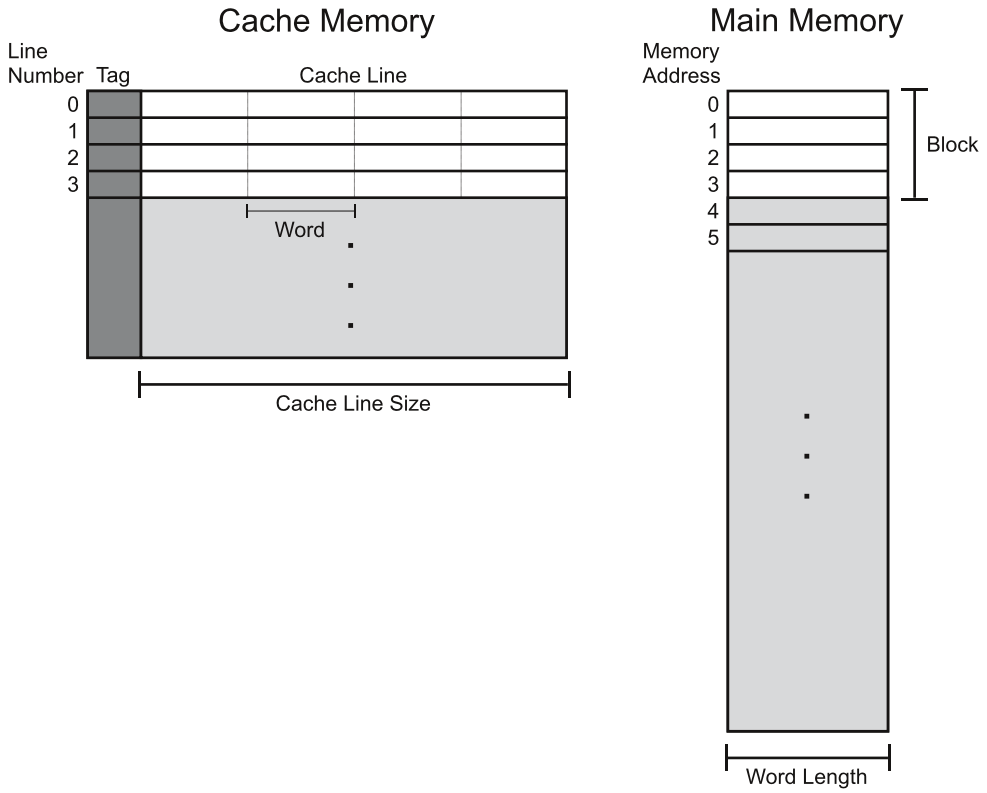


Figure 2.9: Cache memory structure showing the division of cache memory into cache lines and main memory into blocks [131].

- **Direct mapped cache** – This scheme maps each memory block to one possible cache line. This is the simplest of the mapping algorithms, but it has a significant downside. If two or more blocks with the same cache line mapping are currently being accessed, they will be forced to continually evict each other, even if there are other free cache lines (this is known as *thrashing*) [27, 54, 131].

- **Fully associative cache** – With a fully associative cache, blocks can be assigned a position anywhere in the cache. All cache line tags are checked simultaneously when searching for a particular memory block in the cache. This becomes complex and expensive as the cache size increases [27, 54, 131].

- **Set associative cache** – A set associative cache combines both of the above schemes by dividing the cache into sets of associative cache lines. With an n-way set associative cache, a block is first mapped to a specific associative set, and then, it can be mapped to any one of the n cache lines in that set [27, 54, 131].

When a cache becomes full, any subsequent cache misses require existing lines to be evicted

to make space for the new cache block. For associative caches, this is handled by the replacement algorithm (direct mapped caches just replace the mapped line regardless of free capacity). *Least-recently used* (LRU) is a popular replacement method that monitors cache line activity and replaces lines that have been unused for the longest time. Other methods include *random* replacement and the *first in, first out* (FIFO) method [54, 124].

Modern shared-memory multiprocessors, such as Intel's Nehalem based multicore CPUs, implement multiple levels of a cache. The first level of cache (L1) is usually very fast, but quite small (around 64KB). Each CPU core has its own private L1 data and L1 instruction caches. The second level of cache (L2) tends to be significantly larger and slightly slower than the L1 cache. Depending on the organisation of the processor, the L2 cache can either be shared between groups of CPU cores (larger L2 cache), as with early multicore designs, or each CPU core can have its own private L2 cache (smaller L2 cache). L2 is typically a unified data and instruction cache. CPUs implementing four or more cores will typically also come with a large shared L3 cache [27, 54, 131].

The use of shared L2 and L3 caches allows for faster shared-memory synchronisation between threads as different CPU cores have direct access to the fast, shared cache as opposed to having to communicate over the slower system bus. However, as with many shared systems, contention for cache resources becomes an issue, and the effects of maintaining *cache coherency* can affect performance [16, 84, 153]. The goal of cache coherence is to allow processors to make effective use of their private and shared caches while maintaining the consistency of shared data. The key concepts regarding cache coherence are described below.

- **Write policy** – If a word in a cache line has been modified and that cache line is removed from cache, it is necessary to write the modifed value back to the appropriate memory location, otherwise the data in that location will be invalid. The *write through* policy ensures that memory is always valid by modifying both the cache and main memory when a cache line is updated. However, this creates substantial memory traffic. An alternative policy, known as *write back*, makes changes to the cache only. It also marks the cache line as having been modified, so that when the line is removed, the data is written back to main memory only if the modified flag has been set. However, this can result in inconsistency [54, 131].
- **Directory protocol** – A centralised cache controller maintains a directory that records where cache line copies currently reside. Accesses to these cache lines go through this controller, which then grants exclusive access to the line by invalidating the copy held by other processors, forcing them to write back any modifications to these cache lines [54, 131].

- **Snooping protocols** – When a shared cache line in one processor is updated, the cache controller broadcasts this change on the processor interconnect network. All cache controllers then watch or *snoop* on the broadcast network, looking for cache line changes that affect them and allowing them to react accordingly [131].

- **MESI protocol** – The MESI protocol is the most commonly used cache coherence protocol. Cache line tags record one of four possible states.

  Modified (M) – The line has been modified and is only available in this cache.

  Exclusive (E) – The line is not in any other cache and does not differ from memory.

  Shared (S) – The line may be in another cache and does not differ from memory.

  Invalid (I) – The cache line contains invalid data.

  The cache controllers then snoop on a common bus and react to cache events by modifying the cache line state according to the prescribed MESI state transitions, invalidating cache lines as required [131].

Cache misses can occur for a number of reasons: a *compulsory* miss occurs the first time a block is read, a *true sharing* miss occurs as a result of one processor overwriting and invalidating a shared variable used by another processor, *false sharing* occurs when one processor modifies a non-shared variable in the same cache line as a variable being used by another processor causing the whole line to be invalidated, *conflict* misses occur when a processor makes changes to too many variables that each share the same associativity set, and *capacity* misses result from working data sets that are greater than the cache capacity [54, 68, 120].

Identifying the causes of cache misses, through profiling, and rectifying these issues is critical to improving multiprocessor performance [120]. Some cache use improvements include data *prefetching*, which involves explicitly preloading required data into memory or cache before the processor requests it, thus decreasing the chance of cache misses, and cache alignment and data structure padding, which help reduce false sharing [68, 92]. Maintaining *cache affinity*, whereby a thread is scheduled to the processor with the most relevant cache lines for that thread, is not particularly important for single multicore processors, however, there are performance benefits for systems with multiple multicore CPUs [84, 144].

Virtual Memory

Virtual memory acts as an extension to main memory by utilising a *page file* that is stored on the computer system's hard disk. This increases the address space available to processes, allowing

for more processes than main memory can support alone. Main memory is divided into equal-size chunks known as *page frames*. The virtual memory space is also divided into *pages* that are the same size as a page frame. Processes are then allocated pages in virtual memory, which need not be stored contiguously in main memory or even at all. Pages that have not been active for a while are moved to the page file on the hard disk and *paging* occurs when virtual pages are copied from the page file to RAM before they are likely to be used. A *page fault* occurs when a requested page is not available in main memory, thereby incurring a performance penalty as the page must be copied to RAM from the hard disk before it can be used [54, 115].

Virtual addresses need to be mapped to physical addresses in main memory. This is aided by the process's *page table*, which keeps records of the allocated pages and tracks their current location and usage. However, these page tables are stored in memory, so any reference to virtual memory requires an extra memory access for the page table. This can have a significant impact on performance, so an additional cache, known as the *translation lookaside buffer (TLB)*, is used to speed up access to frequently used page table entries. The virtual memory system is quite similar to cache memory as it takes advantage of locality and the increased capacity of memory lower in the hierarchy [54, 115].

2.4.3 Memory Organisation

The memory organisation of a computer system is closely tied to the processor organisation as described in Section 2.4.1. The two main memory organisations are described below with a focus on the performance implications of each organisation and how they affect the programmer.

UMA

Uniform Memory Access (UMA) systems are those in which all processors have read and write access to all regions of main memory through a shared interconnect such as a system bus. Since all processors share the same bus, the memory access times for any region of main memory is the same for all processors in the system. The cache coherence protocols are implemented at the hardware level and typically utilise the system bus to transmit cache-coherence signals [54, 131]. This concept is illustrated in Figure 2.10, which depicts N processors connected to main memory via a common system bus [131]. Modern SMP systems are a common example of UMA.

One of the advantages of a UMA system is that the operating system and application programmers do not have to consider the effects of varying memory access times for different processors.

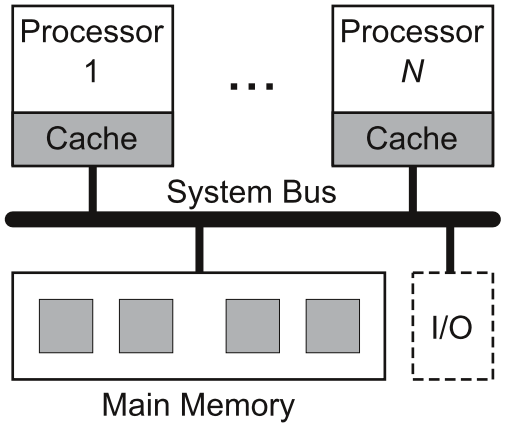


Figure 2.10: UMA/SMP processor organisation [131].

This makes memory allocation, thread allocation, and load balancing somewhat simpler. However, the downside is that there are practical limits to how many processors can be connected in this way. This arises from the fact that the bus traffic for memory access and cache coherence mechanisms increases as the number of processors increases. Eventually, as additional processors are added, the system bus becomes saturated and creates a bottleneck, thus degrading the performance of the system [27, 54, 131].

NUMA and CC-NUMA

In *Non-Uniform Memory Access (NUMA)* systems, all processors have read and write access to all regions of main memory. However, the processors are organised in such a way that groups of processors (typically SMP nodes) are connected internally via a local system bus, as well as being connected to other groups of processors via an interconnect network of some form (crossbar switches or multidimensional meshes). Each processor group has access to its own local main memory via its system bus [54, 131]. The key difference between UMA and NUMA arises when a processor accesses the main memory of another group of processors. Since the processor does not have direct access to the remote memory region via its own system bus, it incurs the cost of having to access the memory over the interconnect network. This introduces additional latency that varies with the *NUMA distance* or *NUMA factor* (dependant on node topology), as well as the speed or remaining bandwidth of the interconnect network [27, 93]. Therefore, the memory access time for a particular processor varies with the region of memory being accessed [54, 131].

A variation of NUMA is cache-coherent NUMA (CC-NUMA), which maintains cache coher-

ence between the separate nodes in the system. While implementations vary, the general idea is that when a processor requests data from memory, it attempts to satisfy this request from its different levels of cache memory. If the data line is not in the processor's cache, it is fetched from the appropriate location in main memory. If the data is located in the local main memory, the cache retrieves it over the local bus. However, if the data is not in the local main memory, the cache initiates a request for the data line over the interconnect network. The data is then transferred from the remote memory to the interconnect network via the remote bus, then to the local bus from the interconnect, and finally to the requesting cache via the local bus [131]. Cache coherence is maintained by some form of memory directory on each node, which records memory locations and cache line information. When a cache is modified, the relevant node is responsible for broadcasting this fact to all the nodes. Each node's directory is then checked and affected cache lines are either evicted or invalidated until the updated value is written back to memory. An alternative to the directory-based coherence mechanism is snooping [54, 131].

While not strictly NUMA, one can consider certain CPU cache topologies in a similar light. The best examples of this are Intel's early dual-core and quad-core CPUs, which, in the case of the dual-core offerings do not have a shared L2 cache, and in the case of the quad-core CPUs such as the Core 2 Q6600 and the later Q9400, two dual-core CPU dies, each with a separate internally shared L2 cache, are combined on a single CPU package. In the described cases, communication between L2 caches takes place over the system bus, incurring NUMA-like memory access penalties. This has repercussions for thread allocation in the OS as we would ideally want to place threads on the processor core closest to the L2 cache with the most relevant shared data, much like NUMA [27, 63, 64].

The CC-NUMA processor organisation is illustrated in Figure 2.11, which depicts two SMP nodes of N processors, connected via an interconnect network between the two system buses [131].

The main advantage of CC-NUMA over UMA systems is that the processor count can be increased without the bandwidth limitations associated with the single system bus of SMP/UMA systems, while still maintaining the single (shared) addressable memory space. This allows for increased parallelism, and therefore, increased performance provided that memory accesses are predominantly to the local memory and the various levels of cache memory are utilised effectively [54, 131].

However, the downside of NUMA can be quite severe as the performance drops considerably when applications make many remote memory accesses. Additionally, since NUMA based architectures do not transparently look or behave like SMP, the onus is on the programmer to account for the effects of remote memory accesses and to design systems that maintain good

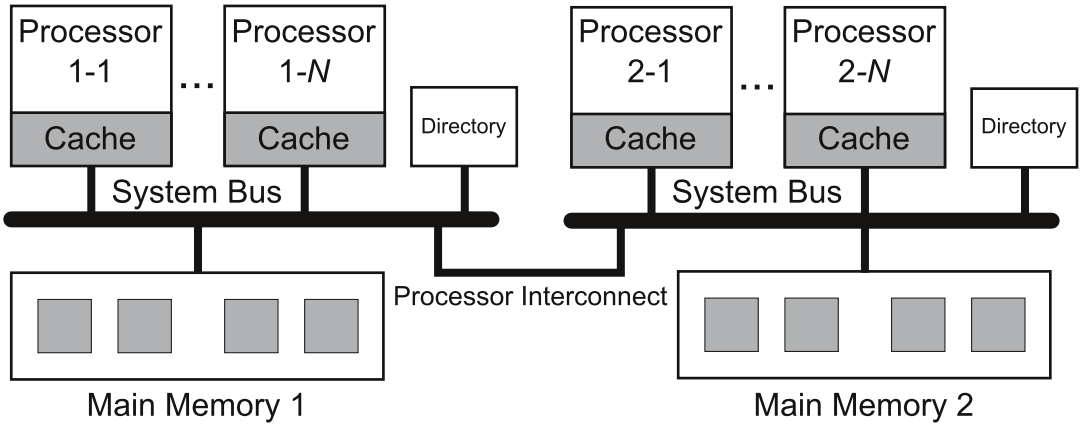


Figure 2.11: CC-NUMA processor organisation [131].

spatial and temporal locality [131]. Operating system developers have to accomodate the distributed memory organisation when implementing the logic for memory page allocation, process and thread allocation, and load balancing in NUMA systems. This logic should attempt to maximise locality of memory pages and minimise thread and process migration [27, 93].

Application programmers also need to consider situations where their program is run on NUMA systems. This is typically achieved using OS supplied functionality and libraries to modify or influence the memory and process allocation logic to suit the nature of the program better. More explicit optimisations may also be possible, such as implementing code in the program, which queries the NUMA node and policy information (using the available NUMA libraries), and adjusts program execution accordingly [27, 93].

2.5 Mutexes, Semaphores and Barriers

While the simplistic definition given for locks in Section 2.2.5 is sufficient to understand discussions involving locks and critical sections, there is more to locks than simply being either locked or unlocked. Locks, in their various forms, are described below and the concept of atomic operations is introduced.

2.5.1 Properties of Concurrent Operations and Locks

Since much lock specific terminology is used in the discussion that follows, brief definitions of the main concepts are provided below.

Contention

Lock *contention* occurs when multiple threads attempt to acquire the same lock at the same time. *High contention* refers to the situation where there are many threads contending for the lock, and *low contention* has few threads in contention [56]. High contention reduces scalability and increases the latency of operations associated with the contended lock [45].

Contention is typically linked to the *granularity* of the locks used to protect critical sections. *Coarse-grained* locking refers to locks that encompass large critical sections, resulting in the lock being held for longer, thereby causing high centention for the lock as multiple threads reach the lock and are forced to wait. *Fine-grained* locking is the opposite in that multiple small critical sections are used instead of locking large sections of code. Since the critical sections are short, the locks are not held for very long and contention is likely to be lower for each of the locks. However, increasing the number of locks increases the locking overheads, so care must be taken to strike an appropriate balance [45].

Lock contention can also be reduced using locks that implement exponential backoff, whereby the repeated attempts to acquire the lock after failure occur after successively longer periods of time if contention is detected [8, 105, 123].

Re-entrant Locks

A lock is said to be *re-entrant* if it can be locked or acquired more than once by the same thread. Re-entrant locks are important for methods that make recursive calls to themselves or make nested calls to other methods that acquire the same lock. If a recursive method acquires locks that are not re-entrant, the recursive call is likely to deadlock. Re-entrant locks typically record which thread acquired the lock and keep a counter of how many times the lock has been acquired by that thread. Whenever the lock is unlocked by the thread, the counter is decreased and once the counter indicates that it is no longer held by the owner thread, the lock is released and made available to other threads [56].

Non-blocking Operations

A *blocking* method or operation is one where an unexpected delay, such as a cache miss or thread pre-emption by the OS, in one thread can cause delays in other concurrently executing threads. Therefore, a *non-blocking* operation is one that does not cause threads to wait for

delays experienced by other threads, thereby allowing multiple threads to make progress without blocking one another [57, 56]. This is usually achieved using the primitive atomic operations described later.

An operation or method is *wait-free* if every call to it completes its execution in a finite number of steps, regardless of the execution speed of the method in other threads. *Bounded wait-free* methods have a limit to the number of steps that a method can take. The wait-free property ensures that all threads executing the method are able to make progress, but such methods can be inefficient and a weaker non-blocking property may be sufficient [55, 56].

A *lock-free* method or operation is one that only guarantees that some method invocation completes its execution in a finite number of steps. This makes it a weaker non-blocking property compared to wait-freedom and naturally, wait-freedom implies lock-freedom. Lock-free methods allow for some threads to starve, but in cases where this is unlikely, a fast lock-free algorithm is preferable to a slower wait-free implementation [56].

Linearisability

Sequential consistency is a correctness property that requires that concurrently executing method calls act as if they occurred sequentially, consistent with program order (there may be more than one ordering that satisfies this condition). Unfortunately, sequential consistency is not *compositional* in that it is not always possible to combine multiple sequentially consistent objects such that they are sequentially consistent as a whole [56]. There is, however, a stronger correctness condition for concurrent objects, referred to as *linearisability*. Linearisability imposes the restriction that method calls should appear to take effect instantaneously between their invocation and return, making it a non-blocking property. This implies sequential consistency, however, unlike sequential consistency, linearisability is compositional, making it appropriate for objects in large systems where modularity is required [57, 56].

2.5.2 Atomic Operations

The need for mutual exclusion arises from the nondeterministic nature of threads executing in parallel that read and modify shared memory. Statements written by the programmer may appear to execute correctly and without interruption, however, there is no guarantee that this is the case for concurrently executing threads with statements that modify shared variables. Even simply incrementing a variable by one using the `var++` post-increment statement can result in

multiple instructions being executed: the current value of the variable is read from memory, the value is incremented by one, and the updated value is stored back in memory. As previously described in Section 2.2.7, variable thread timing may result in interleaving execution of instructions from multiple threads, resulting in a race condition that can corrupt the value of the incremented variable [125]. This can even present itself at the instruction level where certain instructions take multiple instruction cycles to complete, particularly when the operands are large registers that have to be read in smaller segments. The lack of sequential consistency for memory reads and writes on modern multiprocessors allows for these reads and writes to be re-ordered, thereby introducing the potential for other threads to modify the shared data before it has been fully read and updated by the original thread [54, 56].

Therefore, when considering the order of operations being executed concurrently, the concept of *atomic*, *uninterruptible*, or *linearisable* operations becomes very important. An atomic operation is one that appears to complete instantaneously and without interruption or interference from concurrently executing operations. As such, atomic operations require exclusive access (mutual exclusion) to *concurrent objects*, which are shared data-structures (registers, variables, and objects) that are accessed by concurrent processes or threads [57, 55]. Additionally, atomic operations will typically fail outright without making any changes if the right conditions are not met, leaving memory in a consistent state and allowing the program to attempt the operation again.

Primitive Atomic Operations

Application programmers are able to specify atomic operations or sets of operations in their program using mutexes that protect critical sections of code. However, the mutex variable or data-structure is itself a shared object that is updated by concurrently executing threads or processes. Even a simple mutex that uses a boolean variable to store the state of the lock is susceptible to a race condition if multiple threads executing in parallel attempt to acquire the lock at exactly the same instant in time. These threads all read the mutex as being unlocked, set the mutex variable to its locked state and proceed into the critical section, thereby voiding the mutual exclusion property of the critical section [56].

Therefore, the high-level locks used by programmers need to be supported by primitive atomic operations implemented at the hardware level on the CPU. These are sometimes referred to as *read-modify-write* primitives and some of the more common examples are listed below [7].

- **Atomic Registers:** *Atomic registers* are read-write registers for which the read and write operations are linearisable. Read operations on the register return the last value written

and write operations are performed atomically, thereby preventing any in-between values from being read. Unfortunately, atomic registers are not suitable for implementing lock-free concurrent data structures and the atomic synchronisation primitives that follow are preferred [55, 56].

- **Fetch-and-Increment and Fetch-and-Decrement:** *Fetch-and-increment* and *fetch-and-decrement* are special atomic instructions that either increment or decrement the value stored in a register and return the old value. These instructions are particularly useful for implementing counting semaphores. A slight variation on these instructions is *fetch-and-add*, which atomically adds the specified increment amount to the value in a register and returns the old value [7, 56].

- **Test-and-Set (TAS):** The *test-and-set* instruction operates on a single byte in memory holding a boolean value. The instruction atomically sets the value to `true` and returns the old value. This instruction can be used to implement a simple test-and-set spinlock [56].

- **Compare-and-Swap (CAS):** The *compare-and-swap* instruction is implemented by several modern CPU architectures. The CAS instruction takes an *expected* value and an *updated* value as arguments. It then compares the current value of the register to be updated with the expected value and if they are equal, it replaces the value in the register with the updated value, otherwise it leaves the current register value as is. The operation then returns whether or not it successfully updated the register [7, 56]. This ensures that the swap only occurs if the register has not been updated by another thread or process.

  However, the CAS instruction can fall victim to the so-called *ABA problem*, whereby the instruction returns successfully when it should fail. This occurs when a thread reads a value A from a shared memory location and in between the read and the CAS instruction, another thread updates the location, setting it to the value B and then back to A. The CAS instruction from the first thread checks the value at the target location and sees that it matches the old value (A) and continues to update it, returning that the operation was successful. But it has failed to detect the change to the value B, resulting in an invalid modification to the memory location that could result in a corrupted object. Other solutions should be investigated if this is likely to occur for a given concurrent object [56, 125].

- **Load-Linked/Store-Conditional (LL/SC):** *Load-linked* and *store-conditional* are separate instructions that are combined to perform an atomic read-modify-write operation on a memory location. The LL instruction reads and returns the value from the target memory location. The SC instruction then checks if the memory location has been modified

since the thread issued the LL instruction. If it has been modified, the SC instruction fails, otherwise it succeeds and stores the new value in the target location. The SC instruction may also fail if there has been a context switch or if another LL instruction has been executed [54, 56]. Unlike CAS, LL/SC is non susceptible to the ABA problem as the SC instruction checks whether or not the memory location has been modified, and not if the value in the memory location matches the old value. The LL/SC instructions are typically found on the Alpha, PowerPC, MIPS, and ARM architectures [56].

A *consensus number* represents the maximum number of concurrent threads or processes for which a particular object or data structure can solve a simple consensus problem [55]. In [55], Herlihy shows that it is impossible to implement wait-free objects with a particular consensus number from objects with lower consensus numbers. He also provides consensus numbers for some of the above atomic primitives: atomic registers have a consensus number of one (1), fetch-and-increment, fetch-and-decrement and test-and-set have a consensus number of two (2), and compare-and-swap has an infinite consensus number [55].

2.5.3 Locking Strategies

There are two basic locking strategies that can be enforced by mutexes and semaphores to prevent threads from proceeding past the lock. The first strategy causes a thread to spin, where it essentially remains active (wasting CPU cycles) and repeatedly polls the lock while doing no useful work. When the lock eventually becomes available, the thread will detect this and attempt to acquire the lock for itself so that it can proceed past the lock. This is referred to as *spinning* or *busy-waiting* and locks using this strategy are commonly referred to as *spinlocks* [20, 54, 56, 105]. Spinlocks are best used for short critical sections where threads only wait for a short time to acquire the lock as costly scheduling operations are avoided. If threads have to wait for a long time, however, resources are wasted [8, 56].

A variation on spinning is the technique known as *local spinning*, whereby the lock attempts to spin in cache or local memory. This provides significant performance benefits over normal test-and-set spinlocks, especially for distributed shared-memory systems such as CC-NUMA. When threads spin using TAS, they broadcast on the shared bus and invalidate the cached copies of the lock for all processors with threads spinning on that lock. This results in cache misses for each spinning thread, which then has to fetch a new, yet unchanged value from memory. Local-spinning, using a *test and test-and-set* lock, avoids this by using a read-only loop to check the state of the lock and only using the TAS instruction when the lock appears to be free. Since

spinning threads have a locally cached copy of the lock, they do not generate bus traffic until the lock is released (which invalidates the cached locks) and they attempt to acquire the lock for themselves [8, 7, 56].

The second strategy forces a thread to block or sleep when it fails to acquire the lock, which involves the operating system's scheduler descheduling the thread, thereby putting it to sleep. When the lock becomes available, it signals the scheduler to reschedule the appropriate blocked thread, thus waking it up and allowing it to continue execution. Locks employing this strategy are called *blocking locks* and are most applicable when threads are likely to wait for long periods of time. Blocking locks with short wait times cause undesirable scheduling overhead as the scheduler is kept busy descheduling and rescheduling threads [8, 56]. Adaptive or hybrid locks, which spin for a short while and then block, try to balance the advantages and disadvantages of the two locking strategies and are found in certain operating systems [56].

2.5.4 Types of Locks

There are several different types of locks that can be used to provide mutually exclusive access to resources or critical sections. Each type of lock is suited to specific scenarios and as such, it is necessary to understand how these locks operate to make better informed decisions regarding which lock to use in a particular situation. The list of locks presented below is not exhaustive, however, it does provide a reasonable overview of some commonly used locks.

Spinlock Mutex

The simple TAS-based spinlock mutex, as described in Section 2.5.3, is one type of spinlock implementation. The *Filter lock* is another spinlock implementation, which is a generalisation of the two-thread *Peterson lock* for *n*-threads, which has multiple levels or "waiting rooms" where threads compete to gain access to the lower levels until the critical section is reached. Lamport's *Bakery lock* and the *Ticket lock* provide first-come-first-served access to threads attempting to acquire the lock, making them fair alternatives to the TAS spinlock [20, 56, 105].

Queue Lock

Queue locks overcome some of the issues associated with implementing scalable spinlocks. Instead of having threads spin on a single shared memory location, which can cause excessive

cache-coherence traffic, threads form a queue and are notified directly by their predecessor when it is safe for them to continue. This provides lower latency access to the critical section since threads are notified as soon as they are permitted to access the critical section. It also reduces cache-coherence traffic as each thread spins in a different memory location, improving the performance of the lock. First-come-first-served fairness is provided, much like the Bakery lock mentioned above [56, 105, 123].

Readers-Writers Lock

Readers-writers locks allow for increased concurrency if the majority of threads accessing an object protected by a critical section are only performing read operations, with fewer threads attempting to modify the object. There is no reason for multiple reader threads to be denied concurrent access to the critical section as they do not make any changes and pose no risk of causing a race condition. However, should a thread wish to modify an object in the critical section, the writer should be given exclusive access until it has completed its changes [56].

The readers-writers lock provides two lock objects: a *read lock* and a *write lock*. Threads requiring only read access acquire the read lock, whereas threads wishing to update the shared object acquire the write lock. While the read lock is held, other threads may also acquire the read lock, but no thread may acquire the write lock. The write lock may not be acquired if either the read lock or the write lock are currently held by another thread. This can result in the writer threads having to wait a long time to acquire the lock. Therefore, priority can be given to writers by preventing new readers from acquiring the read lock until existing readers have released their locks and the waiting writer thread has acquired and released the write lock [56].

Hierarchical Lock

Hierarchical locks attempt to take into account the hierarchical nature of distributed memory systems and the associated differences in access times. Hierarchical locks try to maintain node locality for lock acquisitions, only transferring the lock to remote nodes to avoid starvation and to keep some level of fairness. This results in much lower inter-node communication due to lock handover, thereby improving performance scalability. Backoff techniques can be used to achieve this by giving local nodes shorter backoff times compared to remote nodes. Queue-based hierarchical locks, implemented using local queues and a single global queue, can be used to ensure a degree of fairness [56, 123].

Semaphore

As described in Section 2.2.5, a semaphore is a synchronisation data structure that restricts simultaneous access to a shared resource to a certain number of threads or processes at any one time. A semaphore has a value or *capacity*, stored as an internal counter, that represents how many units of a particular resource are available. Aditionally, a semaphore provides methods for atomically increasing or decreasing the value of the semaphore. Resources are released using the `V` or `acquire()` method of the semaphore, which atomically increments the counter. Access to a resource is requested using the `P` or `release()` method, which atomically decrements the counter if the value is greater than zero, otherwise it causes the requestor to wait until resources are released by other threads [20, 56, 124].

A *counting semaphore* is a semaphore with a capacity greater than one, whereas a *binary semaphore* only has a capacity of one (restricting the value to either zero or one) [124]. Binary semaphores are similar to simple mutexes as they provide mutually exclusive access to a particular resource instead of a section of code. Counting semaphores are used when mutual exclusion is not necessary, but controlled access to a finite resource is required.

Monitors and Conditions

A *monitor* is a data structure that provides inbuilt synchronisation for its public methods by maintaining its own internal lock, which is used to enforce mutually exclusive access to the monitor and its methods. This ensures that there is only ever one thread accessing the monitor at any one time and thus alleviates the need for threads to handle the mutual exclusion themselves [20, 56].

Condition variables provide a means for threads to wait or sleep until a specific condition is met. Once the condition has been met, the waiting threads are notified or signalled to wake up and continue executing. Condition variables can be combined with monitors to provide conditional access to the monitor, allowing for threads to surrender their access to the monitor object until a required condition has been met [20, 102, 56].

Barrier

Although barriers have been adequately covered in Section 2.2.6, there are a number of different mechanisms for determining when threads can leave the barrier and how to notify these threads

that they need to wake up and continue executing. Barriers such as the *sense-reversing barrier* and other simple busy-waiting implementations can suffer from memory contention, but they tend to provide uniform notification time to waiting threads [56]. Tree-based mechanisms such as the *combining tree barrier* and *static tree barrier* reduce memory contention by spreading memory accesses across multiple barriers, but increase notification latency [56, 105].

Scoped Locks

While not strictly a separate type of lock, a scoped lock is a lock with an interface that enforces the *scoped locking pattern*. The scoped locking pattern utilises the contructor and destructor methods, provided by objects in object-oriented languages such as C++, to acquire and release the lock. This is achieved by placing the critical section code into a scoped block and declaring and initialising a scoped lock object at the start of the block, passing a mutex as an argument to the lock object's constructor. The lock object's constructor attempts to lock the mutex, spinning or blocking until successful, resulting in an instantiated lock object. Once the scoped block has been exited, the lock object's destructor is automatically called, releasing the lock on the mutex. Scoped locks are beneficial in that they do not require the programmer to remember to release the lock and locks are released when exceptions are thrown outside of the scoped block [125].

2.6 Software Metrics

Measurement is an important part of both scientific investigation and everyday life. Through the measurement of specific properties of objects, we can describe those objects in terms of the measured properties and perform meaningful comparisons between objects that share the same properties. These measurements can then be used to answer questions about how objects relate or compare to other objects, whether it be measuring the length of one's foot to determine which shoe size is likely to provide the best fit or measuring the execution time of a program on different processors to determine which processor upgrade will provide the greatest speedup for that program [31, 83].

An *entity* is an object or event that we wish to describe and an *attribute* is a characteristic feature or property of an entity that can be used to distinguish entities from one another. This leads us to the definition of measurement as the process of assigning numbers or symbols (a *measure* or *metric*) to attributes of entities according to a clearly defined set of rules. The quantification of attributes can be categorised as being either a *direct* quantification, where the attribute is

measured and quantified directly, or an *indirect* quantification, where other measurements are combined in some way to reflect a particular attribute using *calculation* [31, 83].

Software metrics refer to the activities involving the measurement of the internal and external attributes of software products (artifacts or deliverables such as code and documentation), processes (software-related activities), and resources (entities required by software processes). This includes aspects such as data collection, cost and effort estimation, performance evaluation, and structural and complexity metrics. *Internal attributes* refer to the characteristics of an entity that can be measured by examining the entity itself. These internal attributes can typically be measured statically, such as the number of lines of code in a program. *External attributes* refer to the characteristics or behaviour of an entity in relation to its environment. External attributes are usually measured dynamically as the entity interacts with its environment, such as measuring the execution time of a program for a given computer system [1, 31, 83]. Through the careful selection and use of software metrics, we can quantify attributes of interest and objectively compare the code of several different implementations of a program without resorting to subjective evaluations, thereby ensuring a more rigorous empirical investigation [31, 85].

In the remainder of this section, we go on to describe the internal product attributes and metrics relevant to our investigation, specifically those relating to length and complexity. Some external product attributes and metrics can be found in Section 2.3, which discusses performance metrics and models.

2.6.1 Code Size and Complexity

Software code is written by programmers in a particular programming language to perform a set of functions specified by the software requirements. This code is made up of a mixture of declarations, statements, expressions, comments, and whitespace. As such, code is an entity with measurable attributes such as size and complexity that can be measured statically. Code length, typically measured in *lines of code*, is the most apparent measure of size, although alternative measures such as those described in Halstead's software science are also available [31, 51, 83]. However, the application of these measures is not always as straightforward as their conceptual simplicity makes them out to be.

Lines of Code

Lines of code (LOC) is the most common measure of code length and represents the number of lines of source code in a program's source files. The fundamental problem with the LOC

measure is the definition of what actually constitutes a line of code. Program code can consist of non-executable elements, such as comments, compiler directives, and blank lines. Additionally, in some programming languages, it is possible to have multiple statements or instructions on a single line. Treating multi-statement lines as single lines belies the difference in effort involved in writing the multi-statement lines over the simpler single statement lines [31, 108, 146].

Blank lines and other formatting whitespace are typically added by the programmer to improve readability, but they do not require significant effort from the programmer to write, unlike statements and, to a certain extent, comments. Therefore, the removal of blank lines from the LOC counting scheme is usually justifiable. The same cannot be said for comments as they do require some effort from the programmer, although not as much as program statements, and it would be remiss to exclude them without further thought. Other elements such as non-executable declarations, headers, and compiler directives also serve to confuse matters as there may or may not be grounds to exclude them from the LOC counting scheme. The manner in which these cases are treated can have a significant effect on the resulting LOC value [31].

If the goal of the investigation is to determine the extent to which a particular program is commented, it is necessary to count both the total lines of code, including comments, and the number of comment lines on their own to be able to calculate the comment density. Therefore, the intended use of the LOC measure determines what should and should not be counted, resulting in many different counting schemes for the LOC measure. As such, any measurement of program length using lines of code must be explicit in stating what program elements contribute to the LOC measure and how they are counted [31, 85].

The most widely accepted definition for a line of code is any program statement, excluding comments and blank lines. However, it is recommended that this measure be referred to as *NCLOC* to highlight this definition and that a separate measure for comment lines be recorded as defined below. The program length is then an indirect measure of the sum of the non-comment and comment lines [31].

$$\begin{aligned} NCLOC &= \text{non-comment lines of code (excluding blank lines)} \\ CLOC &= \text{comment lines of code} \\ \text{program length } (LOC) &= NCLOC + CLOC \end{aligned}$$

The Software Engineering Institute further defines the LOC count to include all executable code and non-executable declarations and compiler directives. Additionally, the manner in which the code was produced is also considered, whether it be programmed, generated, reused, or modified [31]. The measurement of reuse and modification is important in the context of our investigation and is discussed further later on in this section.

Halstead's Software Science

Halstead's software science, developed by Maurice Halstead [51], provides alternative measures for program length and complexity. Instead of interpreting a program as lines of code in a source file, Halstead's metrics view program code as a combination of tokens that can be classified as either operators or operands. These tokens are then measured to provide the basic metrics defined below [31, 51, 52, 108, 145, 148].

$$\begin{aligned} \mu_1 &= \text{number of distinct operators} \\ \mu_2 &= \text{number of distinct operands} \\ N_1 &= \text{total number of operators} \\ N_2 &= \text{total number of operands} \end{aligned}$$

As with the LOC measure, the question of whether or not certain language constructs should be counted arises. The definition of the Halstead metrics excludes comments and blank lines as they do not contain program statements, and therefore, they count as neither operators nor operands. However, for certain tokens or constructs, it is not immediately clear whether they should be classified as operators or operands, or whether they should be included at all [52, 148]. Some examples of this are class and method declarations, as well as type qualifiers. The classification of such tokens then comes down to the interpretation of the user or the developers of automatic measurement tools [148]. It is, therefore, important to define how each construct is classified for the programming language being measured. When using an automatic measurement tool, the definitions used by the tool should be evaluated to ensure that they are consistent with the goals of the investigation.

From these metrics, Halstead defines a number of indirect measures relating to program length and complexity, some of which are defined below [31, 83, 108, 145].

- **Length** N is defined to be the sum of the total number of operators and the total number of operands as shown below.

  $$N = N_1 + N_2$$

  Since length is based on the number of operators and operands, it is less sensitive to code layout, such as multi-statement lines, than the LOC measure [108].

- **Vocabulary** μ is defined to be the sum of the number of distinct operators and the number of distinct operands.

  $$\mu = \mu_1 + \mu_2$$

- **Volume** V corresponds to the amount of computer storage required to represent the program and is similar to the number of mental comparisons required to write the program. The definition for program volume is given below.

$$V = N \log_2 \mu$$

- **Program difficulty** D is a measure of how easy it is to comprehend the program, which is proportional to the number of distinct operators.

$$D = \frac{\mu_1 N_2}{2\mu_2}$$

- **Program level** L is the inverse of the program difficulty.

$$L = \frac{1}{D}$$

- **Effort** E is the estimated mental effort required to write the program, which is proportional to the volume and difficulty of the program.

$$E = DV$$

  Since effort is concerned with the process of writing the program and not the attributes of the program code itself, it is classified as a process measure [108].

Halstead's metrics have been criticised for not providing clear enough definitions of the basic program attributes and their associated measures. In addition, many of the calculated metrics, namely programming time, intelligent content, and delivered bugs (not described above), have been criticised due to insufficient statistical backing for their relationship with the real world [31, 52, 83]. However, some of Halstead's metrics are still of practical use, particularly length and volume, which show a closer relationship to programming time than Halstead's original definition using the effort metric [52].

Code Complexity

The *cyclomatic complexity* metric was first proposed by Thomas J. McCabe in 1976 [104] as a means for calculating the complexity of a program. It has since seen widespread application in the software industry, particular by those involved with software testing. The *cyclomatic*

number $v(G)$ measures the number of *linearly independent* paths through the control flow graph G of a program. It is calculated using the number of edges e and number of nodes n of the directed control flow graph and the number of connected components p, as defined in (2.10) [1, 104, 83, 85, 147].

$$v(G) = e - n + 2p \tag{2.10}$$

Nodes in the graph correspond to groups of sequential statements in the program or method. Directed edges connect a node to other nodes that could possibly execute after it, thus corresponding to the decision points or conditional statements in the program, such as the `if-then` and `while` statements. For a single program or method, there can only be one connected component; otherwise, the number of connected components is equal to the number of programs, classes, or methods being measured [104].

McCabe also proposed some applications for the cyclomatic number to be used during development and testing, such as his testing methodology that involves testing each linearly independent path through the program. The number of required test cases to achieve full test coverage then corresponds to the cyclomatic number of the target program or module [104]. Limiting the cyclomatic complexity of program modules during development, which has achieved widespread support from the software industry, is another of his proposed applications. McCabe's recommendation was that the cyclomatic complexity of a module be limited to 10, failing which, the module should be split into smaller, less complex modules [31, 104, 151].

However, further research indicates that there is little evidence to support the notion of McCabe's cyclomatic complexity as a definitive measure of program complexity. Statistical analysis shows that cyclomatic complexity exhibits a greater correlation to code size and metrics such as LOC and Halstead's length measure, than it does to code complexity [1, 83, 85]. The benefit of using cyclomatic complexity over other metrics such as LOC, however, is that it is less sensitive to the programming language and code layout.

While there are many other measures for complexity, such as *Fan-In Fan-Out Complexity* and McCabe's data complexity measures, they have been excluded from discussion as they have little applicability to our investigation.

Code Reuse and Modification

The repetitive nature of many programming tasks leads to varying levels of code reuse. This reuse of existing code ranges from copying segments of code from a previous project to making

use of external libraries and utilities. Productivity is increased with code reuse as programmers are able to focus on new features and problems instead of duplicating existing work. Code quality is also improved because defects are less likely to be introduced when previously developed and tested methods or functions are reused. Therefore, it is necessary to account for code reuse when analysing program length and programmer effort, especially for this investigation where existing program implementations are modified to improve parallelism [31].

As stated previously, there are varying levels of code reuse and the subsequent modification of the copied code. To account for this, we measure the *extent of reuse* as the ratio of the number of modified lines to the total number of reused lines. The reused code can then be classified according to the extent of reuse: *reused verbatim* refers to code that was reused without modification, *slightly modified* code has less than 25 percent lines modified, *extensively modified* refers to code with more than 25 percent of the lines modified, and *new* code refers to code that has not been reused or adapted from elsewhere [31].

2.6.2 Concluding Remarks

The field of software metrics attempts to provide valid and useful measures for program size, complexity, and programmer effort. Unfortunately, many of the proposed measures are either ill-defined or lack a strong correlation to the desired attribute. However, the measures for lines of code, length, and volume from Halstead's software science, and McCabe's cyclomatic complexity do appear to have strong correlations with program size. They are also the easiest metrics to calculate automatically. For our purposes, these metrics, along with the extent of code reuse, provide a sufficient measure of programmer effort.

2.7 Summary

In this chapter, we have covered a range of concepts, all of which are necessary to gain a firm understanding of the many factors affecting parallel programming. We defined a number of parallel computing terms and concepts to aid in further discussions on the topic. We then presented some performance measures and models for assessing the performance and speedup of parallel programs. Such measures are necessary to gauge the effects of our software optimisation efforts.

Our discussions regarding computer organisation and the related hardware characteristics serves to highlight the factors that affect software performance. It is of critical importance that a full

understanding of the underlying computer system is gained, as it pertains to the performance of parallel programs. In particular, we noted the importance of cache memory for improving overall memory acess time, which is vital to achieving good application performance.

We also presented descriptions of the various mechanisms for ensuring mutual exclusion. This is necessary as program correctness is equally as important as program performance. Finally, we introduced some measures of program complexity and programmer effort to aid in our investigation of parallel programming APIs and libararies.

Chapter 3

Parallel Programming Tools

3.1 Introduction

A vast range of software development tools exist for a variety of programming languages and software development tasks. These tools are designed to improve the programmer's productivity and assist the programmer with common development tasks such as identifying and locating bugs, assessing program performance, and identifying code regions that are suitable for parallelisation.

For the purposes of our investigation, we provide a survey of popular software development tools for the C and C++ programming languages on modern multicore Linux computer systems. We begin by introducing the C and C++ languages. We introduce and describe several established parallel programming APIs and libraries for shared-memory multiprocessors. Then, we introduce some popular C/C++ compilers and integrated development environments. Finally, we introduce several debugging, program analysis, and profiling tools. It is important to note that this survey is by no means extensive and other relevant tools may have been excluded for a variety of reasons, such as system compatibility or licensing costs.

3.2 C and C++

The C programming language was developed in 1972 by Dennis Ritchie and Ken Thompson at AT&T Bell Laboratories for Unix and the DEC PDP-11 computer [90, 130]. It is a high-level, statically typed, procedural language originally designed for systems programming, but

also widely used for general-purpose application programming. C has a very minimal and concise syntax and it supports structured programming through scoped code blocks and functions, which contain all the executable code. The C standard libraries include the more complex functionality such as string handling and I/O functions. C also includes a pre-processor that supports file inclusion, conditional inclusion of code segments, and the definition of substitution macros [40, 90, 130].

C provides low-level access to memory and its constructs map closely to the underlying machine instructions, giving it a very thin layer of abstraction compared to other high-level languages such as Java. For this reason, it is sometimes referred to as a "middle-level" programming language. The low-level capabilities of C, coupled with its high-level constructs, make it a very powerful language, but also one prone to subtle programming errors. Despite the low-level characteristics of the language, standard C code is portable due to the widespread availability of C compilers for most computer platforms and architectures [40, 90, 130].

Initially, there was no formal standard for the C language and implementation details were left to the compiler writers. However, in 1983, the American National Standards Institute (ANSI) and International Organization for Standardization (ISO) began the standardisation process, which was completed in 1989. This resulted in the C89 standard, which is the most commonly implemented version of the language. The standard was revised in 1999 to include new features, some of which were borrowed from C++, resulting in the C99 standard [130].

C++ was developed in 1979 by Bjarne Stroustrup at Bell Laboratories based on the C language. It is an enhanced version of C, supporting object-oriented programming. Stroustrup designed C++ as a superset of the C language, which means that most C programs can be compiled using a suitable C++ compiler. Essentially, C++ is a statically typed, multi-paradigm (supporting both procedural and object-oriented programming), general-purpose programming language. Much like C, C++ supports both high-level and low-level language features, making it suitable for a wide range of applications such as systems software, device drivers, application software, computer games, and networking programs [40, 60, 130, 133].

Some of the primary features added to the language include classes, templates (support for generic classes), multiple inheritance, operator overloading, exception handling, and improvements to the type system. As with C, C++ also went through a number of revisions and standardisation by a joint ANSI and ISO committee. The first draft of the C++ standard was proposed in 1994, but the standard was later revised to include the Standard Template Library (STL) as well as many other new features and small changes. The STL is a powerful library that includes a variety of generic data structures and algorithms. The C++ standard, referred to as *Standard*

C++, was finalised by the ANSI/ISO C++ committee in late 1997 and released in 1998, with a subsequent update in 2003 [81, 130, 133].

Neither C nor C++ natively support parallel programming through inbuilt parallel constructs; relying instead on external threading libraries to provide the necessary parallel features [59]. However, this is set to change with the upcoming *C++0x* standard.

3.2.1 C++0x

An updated standard for the C++ programming language, known as C++0x, is currently in development by the International Electrotechnical Commission (IEC) and ISO. The new standard extends the core language and the C++ standard library, incorporating a number performance and usability enhancements, as well as new library features. Despite all the additions and changes, the C++0x standard is meant to be mostly compatible with existing Standard C++ code. Some of the more notable additions to the language and standard library include lambda functions, regular expressions, atomic operations, tuple types, and most importantly, threading facilities [12].

3.3 Concurrent Programming Environments

Once the programmer has identified the potential parallelism within a program, a concurrent programming environment is required to exploit this parallelism for improved speedup. Concurrent programming environments can be implemented as parallel libraries that are accessed from sequential languages or as language extensions and compiler APIs. They can also vary in the level of abstraction, from low-level multithreading techniques, to higher level abstractions such as Threading Building Blocks [103].

Pthreads [114] and Boost Threads [152] provide low-level interfaces to threads, conditions, and synchronisation constructs. However, this means that the programmer is responsible for thread management issues, such as synchronisation and load balancing. Unfortunately, this also introduces more opportunities for common parallel programming errors, including data race conditions, deadlock, and starvation. In contrast, OpenMP [116], Threading Building Blocks [125], and Cilk [135, 74] provide easy-to-use parallel abstractions that take care of many of the low-level aspects, such as thread management and load balancing.

3.3.1 POSIX Threads

The POSIX Threads (Pthreads) library is based on an international standard (IEEE Standard 1003.1, 2004 Edition) and supports explicit parallelism by providing methods and data-structures for the creation, joining, and termination of threads. It also provides data structures and methods for synchronisation (mutexes, reader-writer locks, semaphores, and condition variables) and signaling [20, 59, 60, 114, 124]. However, Pthreads was originally designed as a library for C, so it is more suited for use in pure C programs. While it is still possible to use pure Pthreads in C++ programs, it does not conform with the C++ object-oriented programming style and care must be taken when implementing such a solution [20, 59, 114, 124]. Pthreads is primarily supported on Unix-like platforms (Linux, Solaris, and BSD), but there are compatible versions for other operating systems such as Pthreads-w32 for Windows.

Pthreads functionality is made available through the inclusion of the "pthread.h" header file in the target program. The Pthreads library provides a set of data types, functions, and constants for implementing multithreaded C programs, all of which are prefixed with "pthread_". The Pthreads interfaces are categorised in Table 3.1 [20, 59, 60, 114, 124]:

Table 3.1: Pthreads interface categories.

pthread_	Basic threads functionality and related functions
pthread_attr_	Thread attributes objects
pthread_mutex_	Mutexes
pthread_mutexattr_	Mutex attributes objects
pthread_cond_	Condition variables
pthread_condattr_	Condition attributes objects
pthread_key_	Thread-specific data storage
pthread_rwlock_	Reader-writer locks
pthread_barrier_	Synchronisation barriers

Threads are declared using the `pthread_t` opaque data type and created with the `pthread_create (thread, attr, thread_function, arg)` function. Thread attributes, such as whether a thread is joinable, and thread priority, are declared using the `pthread_attr_t` opaque data type and initialised with the `pthread_attr_init (attr)` function. Threads are terminated by calling `pthread_exit (status)` within the thread function or cancelled by another thread using the `pthread_cancel` function. Thread attributes are destroyed with `pthread_attr_destroy (attr)`. Threads can be synchronised using `pthread_join (...)` and detached with `pthread_detach` [20, 59, 60, 114, 124].

Other synchronisation mechanisms include Pthreads mutexes and conditions. A mutex is declared with the `pthread_mutex_t` type and initialised either statically with

`PTHREAD_MUTEX_INITIALIZER`, or dynamically with the `pthread_mutex_init ()` function. It is also possible to set mutex attributes, much like thread attributes. A mutex can be locked using either `pthread_mutex_lock (mutex)`, which blocks while the mutex is locked, or `pthread_mutex_trylock (mutex)`, which attempts to lock the mutex and returns an error code if the lock is already held by another thread. A mutex is unlocked with the `pthread_mutex_unlock (mutex)` routine. However, the programmer must be careful to avoid *priority inversion*, whereby a thread with a lower priority executes before a thread with higher priority due to an unfavourable ordering of synchronisation operations [20, 59, 60, 114, 124].

Condition Variables allow threads to synchronise based on specific values or conditions, avoiding having to poll for the condition. Condition variables are used in conjunction with mutexes. A condition variable is declared with the `pthread_cond_t` type and initialised either statically with `PTHREAD_COND_INITIALIZER`, or dynamically with the `pthread_cond_init ()` function. They are destroyed using the `pthread_cond_destroy (condition)` routine. Threads then call the `pthread_cond_wait (condition, mutex)` routine while the mutex is locked to block until the specified condition is met or signaled. The `pthread_cond_signal (condition)` routine signals to a waiting thread that it is able to wake up. The `pthread_cond_broadcast (condition)` function is used to signal multiple threads waiting on the condition [20, 59, 60, 114, 124].

An example Pthreads program demonstrating the creation and joining of threads, as well as the use of mutexes, is presented in Listing 3.1.

3.3.2 Boost Threads Library

The Boost C++ libraries provide a wealth of functionality, including the Boost Threads library, which is a C++ threading library based on platform specific threading libraries, such as Pthreads and Win32 Threads. As with Pthreads, Boost Threads enables low-level threading, with explicit creation and termination of threads and mutexes. However, Boost Threads expands upon the capabilities of Pthreads with additional mutex types, lock functions, barriers, and more advanced conditional variables. Unlike Pthreads, Boosts Threads is designed for object-oriented programming, making use of C++ templates and function objects (functors), and therefore promotes a better object-oriented programming style [88, 152].

Boost Threads includes interfaces for thread management, synchronisation, mutexes, lock types and functions, condition variables, barriers, futures, and thread-local storage. The library is

```
#include <pthread.h>
#include <stdio.h>
#define NTHREADS 4

pthread_mutex_t m;
int id_sum = 0;

void* thread_worker(void* thread_id) {
    pthread_mutex_lock (&m);
    id_sum += (int)thread_id;
    printf ("Thread id: %d and current sum: %d\n", (int)thread_id, id_sum);
    pthread_mutex_unlock (&m);

    pthread_exit (NULL);
}

int main(void) {
    pthread_t* thread_id = new pthread_t[NTHREADS];
    pthread_mutex_init (&m, NULL);

    for (int i = 0; i < NTHREADS; i++)
        pthread_create ( &thread_id[i], NULL, thread_worker, (void*)i );
    for (int i = 0; i < NTHREADS; i++)
        pthread_join ( thread_id[i], NULL );

    pthread_mutex_destroy (&m);
    pthread_exit (NULL);
}
```

Listing 3.1: Simple Pthreads example.

accessible through the inclusion of the `boost/thread.hpp` header. The `thread` class is responsible for launching and managing threads and a `thread` object represents a single thread. Boost Threads also makes it easier to manage pools of threads through the `thread_group` class. Threads are launched by passing a function or a functor to the `thread` constructor. The `join()` member function causes the parent to thread wait until the execution of the child thread has completed [88, 152].

Boost Threads supports a variety of mutex types, with both recursive and non-recursive versions, as well as shared and exclusive ownership semantics. The primary Boost lock types, listed in order of efficiency, are `mutex`, `try_mutex`, `timed_mutex`, `recursive_mutex`, `recursive_try_mutex`, and `recursive_timed_mutex`. Boost Threads also supports scoped locks, which implement the resource acquisition is initialization (RAII) idiom for locking and unlocking a mutex. The Boost Threads library supports condition variables through the `condition_variable` class and its associated members. Thread-local storage is also provided through the `thread_specific_ptr` feature, which gives each thread its own private data storage that is protected from concurrent access [88, 152].

An example Boost Threads program demonstrating the creation and joining of threads, as well as the use of the scoped lock, is presented in Listing 3.2.

```
#include <iostream>
#include <boost/thread.hpp>
#define NTHREADS 4

int id_sum = 0;
boost::mutex m;

class Worker {
private:
    int thread_id;
public:
    Worker (tid)
    : thread_id(tid)
    {}
    ~Worker () {}

    void operator ()() {
        boost::mutex::scoped_lock lock(m);
        id_sum += (int)thread_id;
        std:cout << "Thread id: " << thread_id << " and current sum: " << id_sum << "\n";
    }
};

int main (void) {
    boost::thread_group threads;
    for (int t = 0; t < NTHREADS; t++) {
        threads.add_thread(new boost::thread(Worker, t));
    threads.join_all();
    return 0;
}
```

Listing 3.2: Simple Boost Threads example.

3.3.3 Threading Building Blocks

Intel Threading Building Blocks (TBB)[1] is a C++ template library that provides high-level, task and data parallel multithreading features targeted at improving multicore performance [71, 125]. It makes use of common C++ templates and coding style to facilitate multithreaded programming. The library also provides a set of highly-efficient parallel algorithm templates, such as *parallel_for* and *pipeline*, which further assist the programmer. Since TBB is a runtime library, it can be used with any suitable C++ compiler. It is also supported on a wide variety of operating systems and platforms making it portable [17, 71, 125].

TBB has a sophisticated thread management system that provides for a higher-level parallel programming abstraction. The creation, termination, and scheduling of threads is hidden from the programmer, allowing him to focus on the algorithm instead of thread management. The TBB scheduler assigns tasks to threads at runtime and can re-assign tasks through its "task stealing" mechanism to load-balance threads, while preserving cache locality and maximising

[1] http://www.threadingbuildingblocks.org/

speedup. The TBB library provides a variety of parallel algorithms, concurrent containers, memory allocators, task scheduling features, and synchronisation primitives. Some examples of the provided parallel algorithms include the `parallel_for`, `parallel_sort`, `parallel_scan`, `parallel_reduce`, `parallel_do`, and `pipeline` templates. TBB is thread-safe, which means that, with a bit of care, it can be combined with other parallel programming libraries [17, 71, 86, 125].

There are a number of aspects to the implementation of a parallel program using TBB. First, the task scheduler is initialised by declaring and instantiating a `task_scheduler_init` object. The task scheduler also allows the programmer to specify the number of threads to be used in the thread pool. For simple loop parallelisation, the `parallel_for`, `parallel_scan`, and `parallel_reduce` templates can be used. This involves the creation of a function object with an overloaded `()` operator, the required data members, and a copy constructor, much like Boost Threads [125]. However, the introduction of lambda function support in some modern compilers means that it is no longer necessary to define a new class as the function object can be implemented inline [86, 73].

To parallelise a for loop, the instantiated function object is passed to the `parallel_for` template, along with an iteration space argument. The `blocked_range` template is an example of an iteration space that splits the defined range into segments according to the specified *grain size*. This results in the creation of tasks for the segments of the splittable range, which are then scheduled for execution by the TBB runtime system. The selection of grain size tends to be a manual process that is very specific to the nature of the program. Selecting too large a grain size can limit parallelism, whereas small grain sizes increase overheads and reduce performance. However, TBB includes an `auto_partitioner` object that uses heuristics to select a grain size automatically and the use of this automatic partitioner is the default approach [17, 125].

More advanced, stream-based parallel algorithms, such as the `pipeline` and `parallel_do` templates, are also available. The `pipeline` template allows the programmer to define a number of processing stages or filters that work on independent tasks in parallel, much like an assembly line. Another important feature of TBB is the inclusion of concurrent, thread-safe containers for use in high-performance parallel programs. Example containers include the `concurrent_queue` and `concurrent_vector` template classes. TBB also provides mutexes and other synchronisation primitives for managing shared resources [86, 125].

An example Threading Building Blocks program is presented in Listing 3.3.

```
#include "tbb/task_scheduler_init.h"
#include "tbb/blocked_range.h"
#include "tbb/parallel_for.h"
#define NTHREADS 4

double sarray[100];

class Worker {
public:
    Worker () {}
    ~Worker () {}

    void operator()( const blocked_range<int>& range ) const {
        for (int i = range.begin(); i != range.end(); ++i)
            sarray[i] = sarray[i] * sarray[i];
    }
};

int main(void) {
    task_scheduler_init init(NTHREADS);
    Worker worker;

    for (int i = 0; i < 100; ++i)
        sarray[i] = i;
    parallel_for( blocked_range<int>(0, 100), worker, auto_partitioner() );

    return 0;
}
```

Listing 3.3: Short TBB parallel program example.

3.3.4 OpenMP

OpenMP[2] is a portable API for parallel programming on shared-memory computer systems. The OpenMP API is defined and updated by the *OpenMP Architecture Review Board (ARB)*, which consists of major hardware and software vendors. OpenMP augments C, C++, and Fortran by extending the languages with parallel directives. These directives can be added to specific loops or sections of code by the programmer to instruct the compiler to parallelise these so-called *parallel regions*. It also defines a number of environment variables for controlling the execution of OpenMP programs and includes a runtime library consisting of additional OpenMP functionality such as low-level access to locks. Since OpenMP is an API specification that must be implemented by the compiler, the performance of the resulting OpenMP programs is dependent on the compiler [21, 116, 127, 124, 129]. OpenMP is targeted at shared-memory multiprocessors such as the multicore CPUs found in modern computer systems. It does not, however, provide the means for managing the affinity of tasks on NUMA systems, although projects such as ForestGOMP aim to add NUMA support [18].

[2] http://openmp.org/wp/

Earlier versions of OpenMP focused primarily on data parallelism and coarse, function-level parallelism, however, with the release of OpenMP 3.0 came improved support for nested parallelism and the introduction of task parallelism through the *omp task* directive [21, 106, 116]. OpenMP includes a number of work-sharing, mutual exclusion, and synchronisation directives, each of which accepts a number of clauses. An OpenMP directive applies to a succeeding statement, compound statement, or OpenMP directive. The OpenMP work-sharing constructs have an implied barrier at the end of the construct, but these barriers can be suppressed to improve performance if it is safe to do so. The compiler then uses these directives to generate a concurrent version of the program following the *fork-join* programming model. OpenMP makes it easy to add parallelism incrementally and it is possible to recompile an OpenMP program to execute sequentially for debugging purposes [21, 116, 124].

The OpenMP compiler directives are listed and described below [21, 116].

`#pragma omp parallel` *[clause[[,] clause]...]*

Creates a team of threads and marks a region for parallel execution. The programmer must be careful where this construct is placed as the creation and termination of thread teams creates performance overheads. Accepts the following clauses: **if, num_threads, default, private, firstprivate, shared, copyin, and reduction**.

`#pragma omp for` *[clause[[,] clause]...]*

Specifies that loop iterations will be distributed and executed by an available team of threads. Accepts the following clauses: **private, firstprivate, lastprivate, reduction, schedule, collapse, ordered, and nowait**. It can also be combined with the **parallel** construct for a single work-sharing loop.

`#pragma omp single` *[clause[[,] clause]...]*

Specifies that only one thread in the team will execute the associated block, but not necessarily the master thread. Accepts the following clauses: **private, firstprivate, copyprivate, and nowait**.

`#pragma omp master`

Specifies that only the master thread will execute the associated block. No implied entry or exit barrier.

`#pragma omp sections` *[clause[[,] clause]...]*

{

[**`#pragma omp section`** *]*

block

[**#pragma omp section**
block]
...
}

Allows for the specification of *section* blocks that are distributed and executed by the thread team. Accepts the following clauses: **private, firstprivate, lastprivate, reduction, and nowait**. It can also be combined with the **parallel** construct for a single work-sharing region.

#pragma omp task *[clause[[,] clause]...]*

Defines an explicit independent task for execution by the thread team. Accepts the following clauses: **if**, **untied**, **default**, **private**, **firstprivate**, and **shared**.

#pragma omp taskwait

Creates a barrier that waits on the completion of child tasks.

#pragma omp flush *[(list)]*

Enforces memory consistency for the *list* of variables.

#pragma omp critical *[(name)]*

Creates a named critical section, enforcing mutual exclusion for the associated block.

#pragma omp barrier

Creates an explicit barrier for threads in the team.

#pragma omp atomic

Ensures that a variable or location is modified atomically. The assignment operator or expression can be one of the following: *x binop* `=` *expr*, *x*`++`, `++`*x*, *x*`--`, `or --`*x*.

#pragma omp ordered

Enforces a strict ordering for the associated block within a loop, according to the order of the loop iterations. This limits potential parallelism and should be used with care.

The clauses supplied to an OpenMP directive allow for the modification of data sharing attributes, thread management, and barriers, as described in Table 3.2 [21, 116].

Static scheduling splits the iteration space equally between threads. *Dynamic* scheduling splits the iteration space according the the supplied *chunk size*, much like TBB's *grain size*. *Guided* starts with larger chunks, and gradually reduces the *chunk size* as it progresses through the iteration space. The *auto* scheduling argument lets the OpenMP runtime system select the

Table 3.2: Clauses for OpenMP directives.

if(*condition***)**	Specifies conditional parallel execution of the associated parallel region.
num_threads(*int expr***)**	Sets the number of threads to be created in the thread pool.
default(shared | none)	Sets the default data sharing attributes for variables in the parallel region.
private(*list***)**	Specifies that the listed variables are private to a task.
firstprivate(*list***)**	Specifies that the listed variables are private to a task and that they must be initialised with the value of the original.
lastprivate(*list***)**	Specifies that the list variables are private to a task and that the original variable must be updated with the final value at the end of the region.
copyprivate(*list***)**	Broadcasts the values of the listed variables from one implicit task to other tasks in the same parallel region.
shared(*list***)**	Specifies that the listed variables are shared by threads in the parallel region.
copyin(*list***)**	Copies the listed `threadprivate` variables from the master thread to the other worker threads.
reduction(*operator:list***)**	Specifies that the listed variables be accumulated using the defined operator.
schedule(*kind[,chunk_size]***)**	Specifies the scheduling algorithm to be used.
collapse(*n***)**	Collapses the iteration spaces of *n* nested loops into one large iteration space.
ordered	Indicates that a for loop contains an *ordered* directive.
nowait	Suppresses the implied barrier at the end of a work-sharing constuct.

appropriate scheduling algorithm, while *runtime* scheduling allows the scheduling algorithm to be specified at runtime through the `OMP_SCHEDULE` environment variable [21, 116].

There are also a number of runtime library functions that allow the programmer to query and modify various aspects of the OpenMP runtime system. The runtime library also includes locks and timing routines. In addition to the runtime library, OpenMP makes use of a number of environment variables that allow for the specification of various runtime aspects, such as scheduling and the number of threads to be used [21, 116].

An example OpenMP program is presented in Listing 3.4.

```
1  #include <omp.h>
2
3  int id_sum;
4
5  int main (void) {
6      int i, n = 100;
7      id_sum = 0;
8
9      #pragma omp parallel default(none) shared(n, id_sum) private(i)
10     {
11         #pragma omp for
12         for (i = 0; i < n; ++i)
13         {
14             int tid = omp_get_thread_num();
15
16             #pragma omp critical
17             {
18                 id_sum += tid;
19                 std:cout << "Thread␣id:␣" << tid << "␣and␣current␣sum:␣" << id_sum << "\n";
20             }
21         }
22     }
23     return 0;
24 }
```

Listing 3.4: Short OpenMP parallel program example.

3.3.5 Cilk, Cilk++, and Cilk Plus

Cilk is a multithreaded programming language, developed at Massachusetts Institute of Technology, that extends the ANSI C language with high-level parallel constructs [38, 135]. Cilk adds five keywords, **cilk**, **spawn**, **sync**, **inlet**, and **abort**, which are translated, along with the rest of the program, to a valid C program called the *serial elision*. The serial elision implements the semantics of the parallel Cilk program and calls an efficient runtime system to handle issues such as parallel execution, resource management, scheduling, and communication. The Cilk scheduler makes use of an efficient "work-stealing" policy to ensure effective load balancing between multiple processors [38, 135].

An improved, commercial version of Cilk, called Cilk++, was under development by Cilk Arts. Cilk++ improved support for loops and shared data objects, as well as adding C++ support [39]. However, Cilk Arts was acquired by Intel Corporation in 2009. Intel refined Cilk++ and released it as Intel Cilk Plus[3], along with a specification and runtime application binary interface (ABI) for implementation in other compilers. Cilk Plus is currently only supported by the Intel C/C++ compiler on both 32- and 64-bit Windows and Linux platforms [73, 74].

Intel Cilk Plus introduces a set of C and C++ keywords for fine-grained task parallelism, as well as an improved array notation and elemental functions for data parallelism. These simple extensions make the implementation of parallel algorithms easy for both new and existing applications. It also includes a library for managing shared variables, called *reducers*, which create separate views for each task and reduces them back to the shared variable. The task parallel constructs are well suited to divide and conquer algorithms [73, 74, 44].

Cilk Plus only has three keywords, which are listed and described below. Two of these are also demonstrated in Listing 3.5 [73, 74].

- **`cilk_spawn`** specifies that a function call can be executed in parallel with the calling function.
- **`cilk_sync`** acts as a barrier, waiting for all spawned children tasks to be completed before progressing.
- **`cilk_for`** specifies that the iterations of a for loop can be executed in parallel.

```
1  #include <cilk/cilk.h>
2
3  void sample_qsort(int* begin, int* end) {
4      if (begin != end) {
5          --end;                          // Exclude last element (pivot)
6          int* middle = std::partition(begin, end, std::bind2nd(std::less<int>(),*end));
7          std::swap(*end, *middle);       // pivot to middle
8          cilk_spawn sample_qsort(begin, middle);
9          sample_qsort(++middle, ++end); // Exclude pivot
10         cilk_sync;
11     }
12 }
```

Listing 3.5: Cilk Plus parallel quicksort example [73].

The array notation introduced by Cilk Plus makes it easy to express array operations that can be executed in parallel as demonstrated by the element-wise multiplication example below.

[3] http://software.intel.com/en-us/articles/intel-cilk-plus/

Elemental functions allow the programmer to define their own functions that operate on independent array elements [73, 44].

```
1  // Parallel multiplication
2  mul_res[:] = mul_a[:] * mul_b[:];
3
4  // Elemental function
5  __declspec (vector) double elemental_func (double x, double y) {
6    return 2 * x * y;
7  }
8
9  // Syntax 1
10 cilk_for (int i = 0; i < N; ++i) {
11     elem_res[i] = elemental_func(elem_a[i], elem_b[i]);
12 }
13 // Syntax 2
14 elem_res[:] = elemental_func(elem_a[:], elem_b[:]);
```

Listing 3.6: Cilk Plus array notation and elemental functions [73, 44].

3.3.6 Parallel Performance Libraries

While the above parallel programming models allow the programmer to implement his/her own parallel algorithms, there are a variety of high-performance, multithreaded libraries that implement common scientific, data processing, signal processing, and multimedia functionality.

Intel Performance Libraries

- **Intel Integrated Performance Primitives (IPP)**[4] is a library of multithreaded, highly-optimised, low-level functions for multimedia, data processing, matrix operations, communications, and cryptography. IPP is supported on Windows, Linux, and Mac OS X running on x86 and x86-64 compatible platforms. The functions provided by the IPP library provide a stateless interface, allowing them to be integrated into C and C++ programs with ease. IPP functions are optimised to take advantage of advanced SSE instructions and parallelism. Function categories include video and audio encoding and decoding, computer vision, cryptography, data compression, image conversion and processing, signal processing, speech recognition and coding, and vector and matrix mathematics operations.

[4] http://software.intel.com/en-us/articles/intel-ipp/

- **Intel Math Kernel Library (MKL)**[5] is a library of extensively threaded and optimised mathematics functions for science, engineering, and financial applications. MKL is supported on x86 and x86-64 compatible platforms, and Windows, Linux, and Mac OS X based operating systems. MKL is supported natively in C, C++, and Fortran programs and can be used in C# and Java via code wrappers. Core library functionality includes linear algebra functions (BLAS, LAPACK, and ScaLAPACK), direct and iterative solvers for large sparse linear systems of equations, vector mathematics and random number generators, and fast Fourier transform functions. All functions are highly-optimised with extensive use of SSE instructions and parallelised using OpenMP.

- **Intel Array Building Blocks (ArBB)**[6] is a new C++ template library for taking advantage of data parallel problems on multiprocessor computer systems. The library uses standard C++ extensions, so it is widely compatible with standard C++ compilers and IDEs, and its associated runtime library supports Windows and Linux. ArBB provides generalised vector parallel algorithms that are highly scalable and optimised for parallel and SIMD (SSE) capable CPUs, while preventing deadlock and data race conditions. Intel ArBB is currently in beta testing and will be available as a part of Intel's Parallel Building Blocks suite, which includes Cilk Plus and Thread Building Blocks.

AMD Performance Libraries

- **AMD FrameWave**[7] is an open source library developed by AMD consisting of optimised, low-level functions for image processing, signal processing, and video applications, as well as basic arithmetic functions. FrameWave is similar to Intel IPP, but it is optimised for AMD processors. However, it supports all x86 and x86-64 compatible CPUs running Windows, Linux, and Solaris based operating systems. The library is optimised to make use of SSE instructions and uses multithreading for improved parallel performance.

- **AMD LibM**[8] is a C99 library of optimised basic mathematical functions for x86-64 based platforms. The library features 104 optimised scalar and vector maths functions that can be used in place of the standard maths routines for improved performance and accuracy.

[5] http://software.intel.com/en-us/articles/intel-mkl/
[6] http://software.intel.com/en-us/articles/intel-array-building-blocks/
[7] http://developer.amd.com/cpu/Libraries/framewave/Pages/default.aspx
[8] http://developer.amd.com/cpu/Libraries/LibM/Pages/default.aspx

- **SSEPlus**[9] is an open source library that provides optimised emulation of SSE instructions and SIMD algorithms with the intent of simplifying the development of SIMD-optimised applications. The emulation of SIMD instructions allows for a single program implementation to take full advantage of multiple target architectures and updates to the library automatically take advantage of new instruction sets without the need to update the program itself.

- **AMD String Library**[10] provides a subset of the standard C library string functions that are optimised for AMD CPUs. These functions, ffsll, strchr, strrchr, memchr, strlen, strnlen, index and rindex, replace those provided by the standard libraries and are designed to take advantage of AMD processors and the SSE4a instruction set. AMD String Library is support on x86-64 versions of Linux running on compatible AMD CPUs.

3.4 Compilers

Programs written in high-level languages such as C++ and Java need to be translated into the appropriate machine code for the target platform before they can be executed. This is the responsibility of the *compiler* or a set of compiler programs. A compiler translates source code from the *source language* to its equivalent representation in the *target language*. Programming languages are typically characterised as being either compiled or interpreted. Compiled languages are translated into machine code for a specific platform, producing executable binaries or object code. Interpreted languages are compiled into platform-independent *bytecode*, which is then interpreted by a platform-specific *virtual machine* (VM) for the language. The VM translates the bytecode to the appropriate machine code while the program is executing [5]. Both C and C++ are compiled languages with a variety of different compilers for a vast number of platforms.

The parsing and compilation process is typically divided into a number of phases. The front-end includes the pre-processing of source code files, lexical analysis (the identification of tokens), syntax analysis (the identification of syntactic structures), and semantic analysis (type checking and other semantic checks). The middle-end contains the majority of the code optimisation stages, and the back-end is responsible for code generation. The code optimisation phase performs a number of transformations on the code to produce a faster, yet functionally equivalent version of the program. The types and degrees of optimisation are dependent on the compiler

[9] `http://developer.amd.com/cpu/Libraries/sseplus/Pages/default.aspx`
[10] `http://developer.amd.com/cpu/Libraries/AMDStringLibrary/Pages/default.aspx`

being used and include transformations such as removing unreachable code, function inlining, loop vectorisation and other loop transformations, and in some cases, *automatic parallelisation* [5, 34, 89, 139].

Automatic parallelisation is becoming an increasingly important feature for modern compilers given the rise of the multicore CPU. A compiler with automatic parallelisation attempts to generate multithreaded code from the original sequential code, thereby relieving the programmer of this error-prone and often tedious process [15, 61, 98, 140]. The primary targets for the parallelisation effort are the looping structures since the majority of a program's execution time is typically found within these structures. This requires extensive data dependence analysis and alias analysis to ensure that it is safe to execute the loop iterations in parallel and determine which shared variables need to be protected. However, this is a particularly difficult task for the compiler and the automatic parallelisation may fail to parallelise loops with complicated dependency issues [15, 34, 41, 61, 89, 98, 122].

Other significant optimisations include interprocedural optimisation (IPO) and profile-guided optimisation (PGO). IPO analyses the entire program, instead of focusing on single functions or blocks of code, and attempts to optimise frequently used procedures. This is usually achieved by inlining small functions that are called frequently, removing dead code branches, and reordering code to improve memory access [23, 34, 50, 89]. PGO is performed in stages: first, the program is instrumented and compiled; the program is then executed and profiling data is gathered; and finally, the program is recompiled using the profiling data to guide the optimisation process [24, 107].

3.4.1 GNU Compiler Collection

The GNU Compiler Collection (GCC)[11] is an open-source compiler distribution with support for a wide variety of platforms and architectures. At the time of writing, the latest stable release of GCC is version 4.5.2. GCC includes compilers for a number of programming languages, including Ada, C, C++, Fortran, Java, and Objective-C, among others. The C compiler is invoked with the `gcc` command and the C++ compiler is invoked with `g++` [43]. Several important compiler options for the GCC C/C++ compilers are described in Appendix A.1.

Each language has its own front-end that parses the input source files and translates into an intermediate tree representation that is then passed to a common, language-independent "middle-end" and back-end. The majority of the code optimisation occurs in the middle-end, where a

[11] `http://gcc.gnu.org/`

variety of compiler optimisations transform the program to produce a faster version, which is then passed to the back-end. The back-end then finalises the register allocations and generates the architecture-specific machine code [43].

GCC supports a number of more advanced optimisations such as interprocedural optimisation, profile-guided optimisation, automatic vectorisation, and limited automatic parallelisation through the Graphite framework. GCC also supports the generation of parallel code using OpenMP 3.0 compiler directives [41, 42, 43].

3.4.2 Intel C and C++ Compilers

Intel C++ Compiler XE 2011[12] (previously known as the Intel C++ Compiler Professional Edition) is a commercial C and C++ compiler for Linux, Mac OS, and Microsoft Windows based operating systems, running on IA-32 (32-bit Intel Architecture / i386), x86-64 (AMD64 and Intel 64), and Itanium 2 compatible processors. It is included in the Intel C++ Composer XE 2011, Intel C++ Studio XE 2011, Intel Parallel Studio XE 2011, and Intel Cluster Studio 2011 suites of optimising compilers, high-performance libraries, and software development tools. A Fortran compiler (Intel Fortran Composer XE 2011) is also available [70, 72, 73]. However, unlike GCC which is freely available, the Intel development tools have expensive licensing costs depending on the type of license and number of users[13]. Discounted academic and non-commerical licenses are, however, available, as well as limited-time evaluation versions for testing.

The Intel C and C++ compilers support an extensive array of optimisations and the included libraries are themselves highly optimised. Some of the advanced optimisations include highly effective automatic vectorisation using SSE instructions, automatic parallelisation, IPO, PGO, and other high-level optimisations such as loop interchange, loop unrolling, loop fusion, and data prefetch to name but a few. In addition to the automatic vectorisation and parallelisation features, the Intel compilers support parallelisation using OpenMP 3.0 and Intel Cilk Plus, which is an updated version of the Cilk++ parallel programming language (described in Section 3.3.5) [15, 34, 72, 73, 69, 139]. The Intel compilers are also capable of producing debugging and profiling information in standard formats that are compatible with common debuggers and profilers, as well as Intel's own debugging and profiling tools [72, 73].

The Intel compilers and some of the Intel performance libraries are criticised for explicitly optimising programs for better performance on Intel CPUs compared to compatible non-Intel

[12] http://software.intel.com/en-us/articles/intel-sdp-products/
[13] http://software.intel.com/en-us/articles/buy-or-renew/

variants such as those produced by AMD and VIA. The primary difference between programs running on an Intel CPU and a non-Intel CPU when compiled with the Intel compiler relates to the use of the SSE instruction sets for vectorisation and other SSE-based optimisations. When compiling a program with SSE support (using the `-axSSE`*n* compiler option), such as SSE3, Supplemental SSE3 (SSSE3), or SSE4, the Intel compiler generates multiple code paths for functions containing SSE instructions. These code paths differ in the level of SSE instructions used, from a generic version with minimal or no SSE support, to the fully optimised version using the highest SSE instruction set as specified during compile-time. Additionally, the compiler integrates an automatic runtime CPU dispatcher into the program executable that detects which SSE instruction sets the current CPU supports and executes the appropriate code path [33, 34, 79, 100].

However, the CPU dispatcher also checks whether the program is running on an Intel CPU. If not, the dispatcher selects the generic code path regardless of the capabilities of the CPU, thereby limiting the performance of the program on non-Intel CPUs. There are several methods for circumventing the CPU dispatcher and allowing for optimal execution on non-Intel processors. However, the method for permanently removing the dispatcher requires modifications to the Intel compilers and libraries, which is against the end-user licensing agreement (EULA) [33, 34, 79, 100]. Despite this, the Intel compilers and libraries produce highly optimised code and are recommended for performance critical applications [34].

The C compiler is invoked with the `icc` command, while the C++ compiler is invoked with the `icpc` command or using `icc` with the `-x c++` compiler option specified before each C++ file [72, 73]. Several important compiler options for the Intel C/C++ compilers are described in Appendix A.2.

3.4.3 Low Level Virtual Machine (LLVM)

The Low Level Virtual Machine (LLVM)[14] is a modular, language-agnostic compiler infrastructure designed for code optimisation using its own intermediate code form. LLVM supports front-ends for a number of languages, including C and C++. The primary front-ends for C and C++ are llvm-gcc (and dragonegg for GCC 4.5), llvm-g++ and Clang[15]. The GCC based front-ends replace the GCC code generator with an LLVM implementation, while Clang is a new, native LLVM compiler that supports C, Objective-C and C++. Clang provides faster compile times, more expressive diagnostics and tighter IDE integration along with a static analyser

[14] `http://llvm.org/`
[15] `http://clang.llvm.org/`

that is capable of automatically identifying certain bugs in the user's source code. While these compilers are very promising, they are in constant development are not yet mature enough for reliable and accurate performance analysis compared to GCC and the Intel compiler suite.

3.5 Integrated Development Environments

Integrated development environments (IDEs) are applications that combine various software development tools and features for one or more programming languages into a single programming environment or user interface with the aim of improving programmer productivity compared to that using separate editing, building, and debugging tools. Common elements found in IDEs include a source code editor, automated build tools, compilers for the specified language or interfaces to system compilers, and various debugging and analysis tools. Modern IDEs typically provide a source code editor with syntax highlighting, code completion, inline error highlighting, and various code refactoring features. The debugging and program analysis tools tend to be tightly integrated into the editor, thereby making it easier to identify issues in the code. Some IDEs even include visual GUI editors, which greatly simplify the task of creating and modifying an application's user interface [58, 95, 117, 149].

For larger projects, particularly applications written in object-oriented languages, advanced IDEs provide multiple views or browsers for navigating and inspecting the application's modules, classes, and files. Integrated version control support is another feature that is typically found in modern IDEs [117, 149]. All of the above features combine to make programmer's job easier by automating many software development tasks and improving access to important tools [58].

Two popular IDEs for C and C++ development on Linux are introduced below.

3.5.1 Eclipse

Eclipse[16] is a highly extensible, open source software development platform primarily written in Java. At its core, Eclipse is a basic IDE with a generic text editor, which is coupled to a powerful plug-in system. The IDE and editor are then extended and augmented to support particular programming languages and additional software development tools using plug-ins, such as the Java Development Toolkit (JDT) for Java programming and the C/C++ Development

[16] `http://www.eclipse.org`

Toolkit (CDT) for C and C++ programming. However, the Eclipse platform is not limited to programming languages and there are a variety of plug-ins for other tasks such as typesetting using LaTeX [149].

The CDT[17] plug-in for Eclipse provides a fully featured IDE for C and C++ software development. Eclipse CDT provides support for project creation and navigation, as well as automated project build tools using a variety of build toolchains. The source code editor supports features such as syntax highlighting, code completion and code templates, refactoring, and code formatting. In addition to the build tools, support for various visual debugging and profiling tools is provided [138]. Many of the Intel software development tools for Linux, such as the C/C++ compiler, Intel debugger, and VTune performance analyser, provide plug-ins for integration with the Eclipse platform.

An example screenshot of the Eclipse CDT IDE is provided in Figure 3.1.

3.5.2 NetBeans IDE

NetBeans[18] is an opensource, modular IDE written in Java. It provides support for software development in various languages, including Java, C/C++, and PHP among others. The NetBeans IDE Bundle for C/C++[19] features support for C and C++ projects with tools to create and manage build configurations, as well as version control. As with most advanced IDEs, the NetBeans C/C++ editor features syntax highlighting, code formatting, code templates, and code assistance through code completion and refactoring. It also features advanced class and file navigation capabilities for efficient navigation through complex projects [117]. Figure 3.2 shows a screenshot of the NetBeans C/C++ IDE.

NetBeans also integrates software development tools such as a runtime profiler, GNU Debugger support with close editor integration, and support for a variety of popular compilers and the configuration thereof. Furthermore, it integrates tools for creating and modifying GUIs based on the Qt application framework[20] [117]. The Oracle Solaris Studio[21] software development environment is based on the NetBeans IDE.

[17] http://www.eclipse.org/cdt/
[18] http://netbeans.org/features/index.html
[19] http://netbeans.org/features/cpp/index.html
[20] http://qt.nokia.com/
[21] http://www.oracle.com/technetwork/server-storage/solarisstudio/

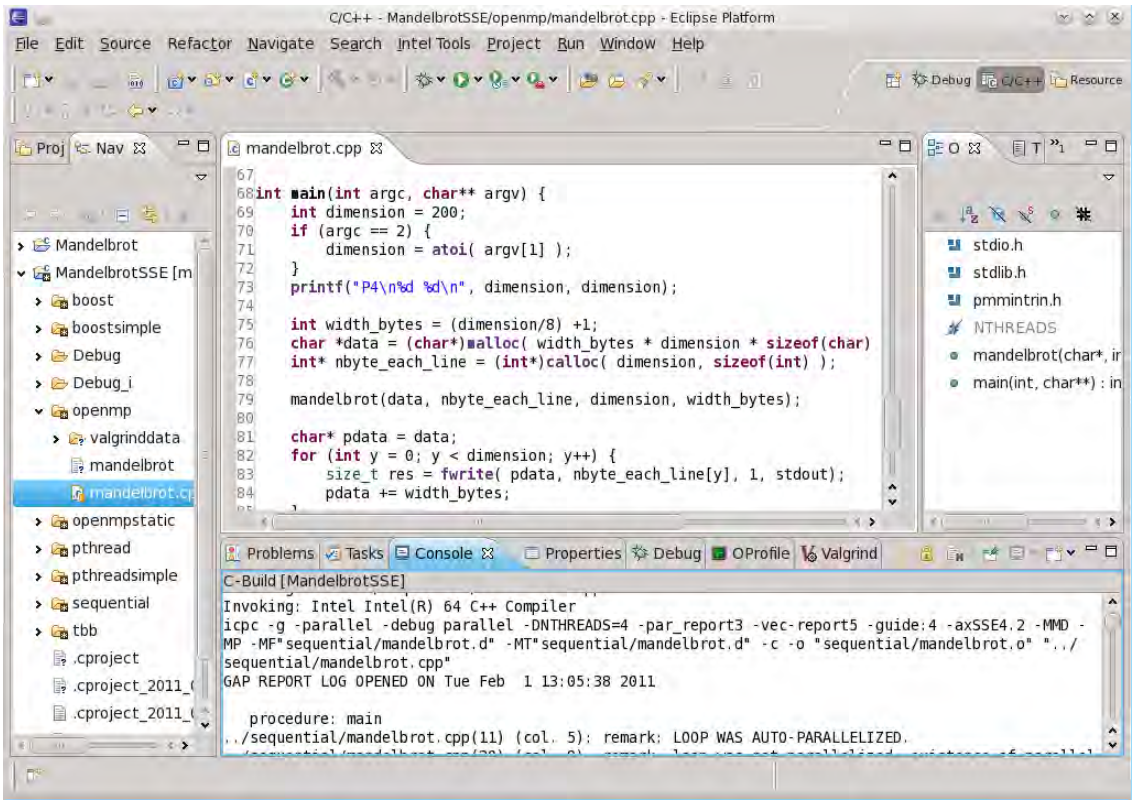


Figure 3.1: The Eclipse C/C++ Development Toolkit IDE.

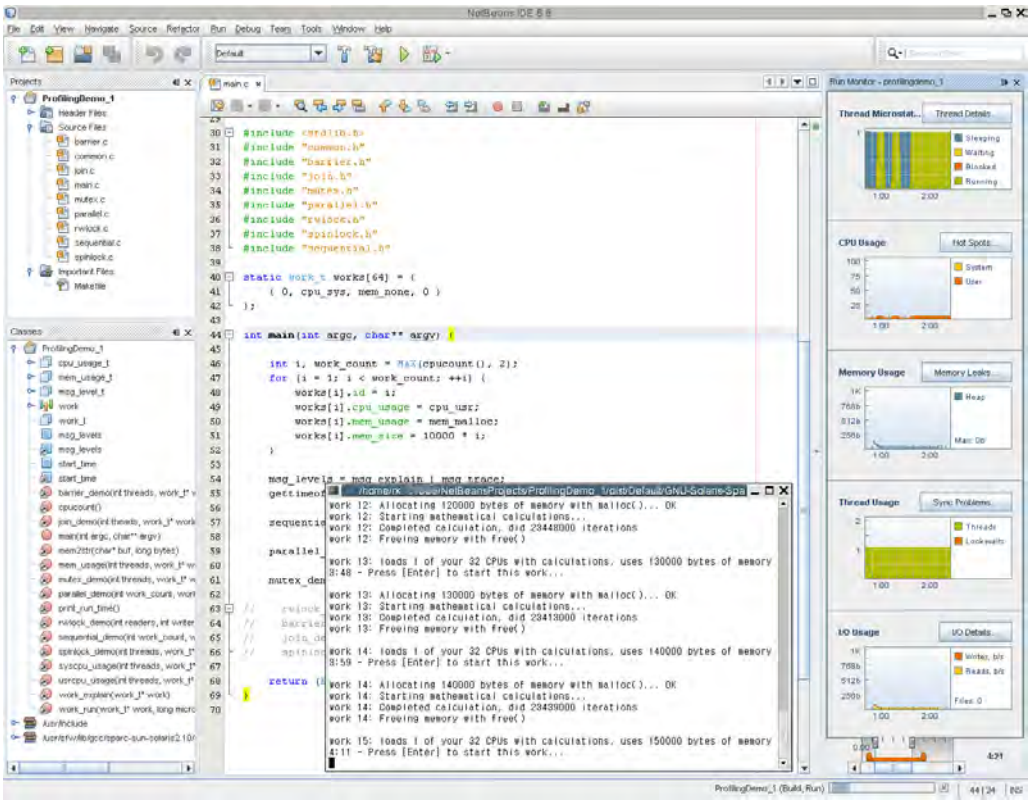


Figure 3.2: The NetBeans C/C++ IDE. Taken from [117].

Oracle Solaris Studio

Oracle Solaris Studio (formerly Sun Studio) is a suite of software development tools for Solaris and Linux operating systems. It features an extended version of the NetBeans IDE, which integrates additional tools for debugging and optimising programs on multicore CPUs, as well as support for the Oracle Sun Studio C/C++ compilers. These compilers are highly-optimised for Oracle Sun hardware and feature automatic parallelisation capabilities and OpenMP 3.0 support. Solaris Studio includes a more advanced, fully integrated debugger and a thread analyzer, which are capable of detecting memory leaks and threading errors such as race conditions and deadlock. Profiling support has also been improved with the Solaris Studio Performance Analyzer, which is capable of identifying program hotspots and providing performance tuning advice [118]. Oracle Solaris Studio is heavily focused towards development for the Oracle Solaris operating system and Oracle Sun hardware (SPARC processors).

3.6 Debugging and Profiling Tools

Debugging is the process of detecting, reproducing, locating, analysing, and fixing programming defects present in a particular program. There are various techniques for debugging software depending on the needs of the developer. Print or tracing debugging is a manual debugging process that involves the insertion of print statements into the program source code to indicate state and program flow. This can be tedious, so the use of an interactive, source-level or symbolic (as opposed to machine code-level) debugger tool is recommended. Post-mortem debugging involves the analysis of the program's trace information or memory dump after the program has crashed. Static code analysis tools are also available and feature the ability to analyse the program source code and locate semantic errors that could lead to program defects [17, 126, 128, 154].

Interactive symbolic debuggers are tools that allow the developer to debug the program while it is still running. They typically feature the ability to monitor and inspect program state (such as variables), set breakpoints that pause execution when a particular condition or program region is encountered, single-step through the program one statement at a time (at the source code level), and alter certain aspects of the program while it is executing. This makes debuggers incredibly powerful tools for locating and diagnosing defects. Many debuggers are integrated into the programmer's development environment, making them easier to use [126, 128, 154].

Software profiling is concerned with the measurement and analysis of the execution characteristics of a program as opposed to static code analysis, which analyses the program structure

without actually running the program. Profiling allows the programmer to identify performance issues and optimisation opportunities within a program based on call counts, execution times, and performance impacting events, such as cache misses and branch mispredictions. Profiling data can be captured using either instrumentation or sampling, or a combination of the two [17, 25, 34, 36, 113].

Instrumentation involves adding instructions to the program that record the characteristics being examined, such as call counts and execution time. Instrumentation can be carried out manually by inserting source-level profiling instructions or directives, or automatically using instrumentation tools or special compiler options. Binary instrumentation adds profiling instructions to the compiled binary, whereas runtime instrumentation instruments code just prior to execution using an intermediary execution supervisor. However, the added instrumentation instructions often introduce overheads that affect the performance of the target program [17, 34, 36, 113].

Sampling involves a statistical analysis of profiling events, such as hardware performance counter events. The profiler uses interrupts to periodically check and record the state of the performance counters and timing information and associates this data with the instructions being executed at that time. Therefore, performance degrading instructions are more likely to be recorded by the profiler as they will trigger performance events more frequently. The profiler is then able to build up a statistical summary of those sections of code associated with performance events. Sampling is a low-overhead profiling technique that has little effect on the program execution time, but it is less accurate [17, 36, 113]. However, there are techniques for compensating for the effect of overhead as a result of instrumentation and profiling [101].

Below, we present a short survey of commonly used free and commercial debugging and profiling tools.

3.6.1 GNU Debugger (GDB)

GNU Project debugger[22], commonly referred to as "GDB", is an open source, command-line interface (CLI), symbolic debugger. Its primary features include the ability to start, monitor, and control the execution of a program running on either the local computer or a remote host. It is also possible to attach GDB to a program that is already running. GDB allows the programmer to monitor and change the values of a program's internal variables, pause program execution by setting breakpoints on specific lines of code or functions, and step through program execution line-by-line. More importantly, GDB has support for debugging multithreaded programs,

[22] http://www.gnu.org/software/gdb/

including: automatic notification of new threads, the ability to inquire about existing threads, switching the current focus to another thread, and setting thread-specific breakpoints [17, 36].

One of the primary drawbacks of GDB is that it does not include a user-friendly GUI by default. However, there are several front-ends, such as the Data Display Debugger (DDB)[23], and it is also integrated into certain IDEs such as Eclipse. GDB can debug programs written in a variety of languages, including Ada, C, C++, Objective-C, and Pascal, with partial support for many others. It is available for most Unix variants, such as Linux and Solaris, as well as Microsoft Windows and is currently at version 7.2 [36, 150].

3.6.2 Intel Debugger (IDB)

Intel Debugger (IDB) for Linux is a commercial symbolic application debugger that forms part of Intel Composer XE 2011 for Linux. IDB supports debugging for programs compiled with the Intel and GCC compilers (C, C++, and Fortran) on IA-32 and x86-64 compatible platforms. However, the Intel compilers provide enhanced debugging information for use with IDB. IDB supports typical debugging features such as attaching to one or more running processes (local or remote), breakpoints, execution stepping, variable inspection and modification, variable watch lists, and support for language-specific features such as C++ templates and the STL [65, 9].

More advanced features include the ability to debug optimised code and support for parallel programs, which include multithreaded applications that use Pthreads and OpenMP. IDB provides detailed information about thread states and interactions, as well as details of OpenMP locks, threads, and teams. It also allows for switching focus between threads and setting thread-specific breakpoints. Additionally, IDB can detect thread data sharing events in programs that use Pthreads, OpenMP, Intel Cilk Plus, and Intel Threading Building Blocks [65, 9].

IDB comes with both a command line interface and a GUI based on the Eclipse platform as shown in Figure 3.3.

3.6.3 Intel Inspector XE 2011

Intel Inspector XE 2011 (formerly Intel Thread Checker)[24] is a memory and thread error checking tool for Windows and Linux found in the Intel Parallel Studio XE 2011 tool suite. It features

[23] http://www.gnu.org/software/ddd/
[24] http://software.intel.com/en-us/articles/intel-inspector-xe/

powerful error checking capabilities for serial and parallel programs written in C, C++, and Fortran. It is applicable to parallel programs that make use of Pthreads, Thread Building Blocks, OpenMP, and Cilk Plus. Its memory checking feature is able to detect memory leaks, memory corruption, memory allocation and de-allocation issues, and inconsistent memory API usage. The thread checking feature is able to identify data race conditions, deadlocks, memory accesses between threads, and incorrect threading API usage. It is also capable of locating the causes of these problems, right down to the actual lines of source code responsible for the errors. Intel Inspector XE is even able to detect non-deterministic errors regardless of whether or not the specific error condition arose during execution [17, 75].

Intel Inspector XE makes use of dynamic binary instrumention, which means that it is not necessary to adjust the application's build configuration before using the tool. This, coupled with the intuitive standalone GUI, makes it very easy to use throughout development [75]. An example screenshot is provided in Figure 3.4.

3.6.4 GNU Profiler (gprof)

The GNU Profiler (gprof)[25] is an open source, command line, statistical profiler that is distributed with many Linux systems. Gprof requires that the target program be compiled with profiling support enabled, using, for example, the `-pg` option of GCC. The program must then be executed with the usual arguments, run to completion, and exit successfully. This produces a file in the current working directory, named 'gmon.out', which contains the profiling data. Profiling adds some overhead, in the region of five to thirty percent, to the program's execution time [17, 37, 47].

The profiling data is then analysed using gprof, which produces a flat profile, dynamic call graph, or annotated source code. The flat profile shows how many times each function was called and how much time was spent executing the function. The call graph summarises which functions called other functions and gives an estimate of the amount of time spent in a function, including calls to other functions. Finally, gprof is also able to produce annotated source code listings with execution count details. However, gprof only produces textual output using a command line interface, making it inconvenient to use for large projects. It also does not handle multithreaded programs, which makes it unsuitable for parallel programs using thread-based libraries [37, 47, 49].

[25] `http://sourceware.org/binutils/docs/gprof/index.html`

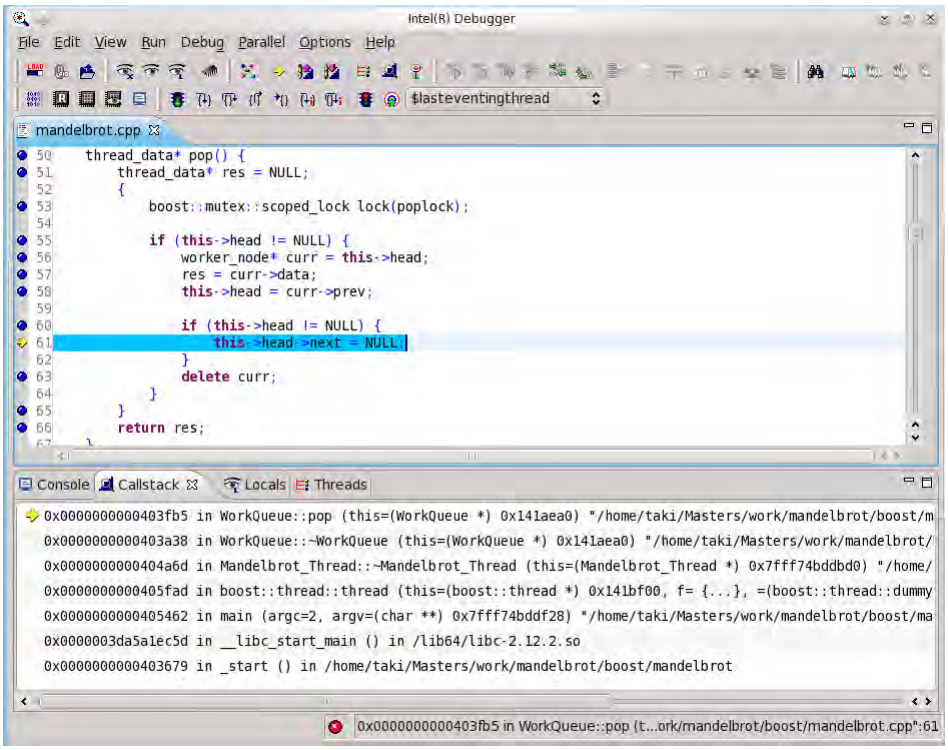


Figure 3.3: The Eclipse-based IDB graphical user interface.

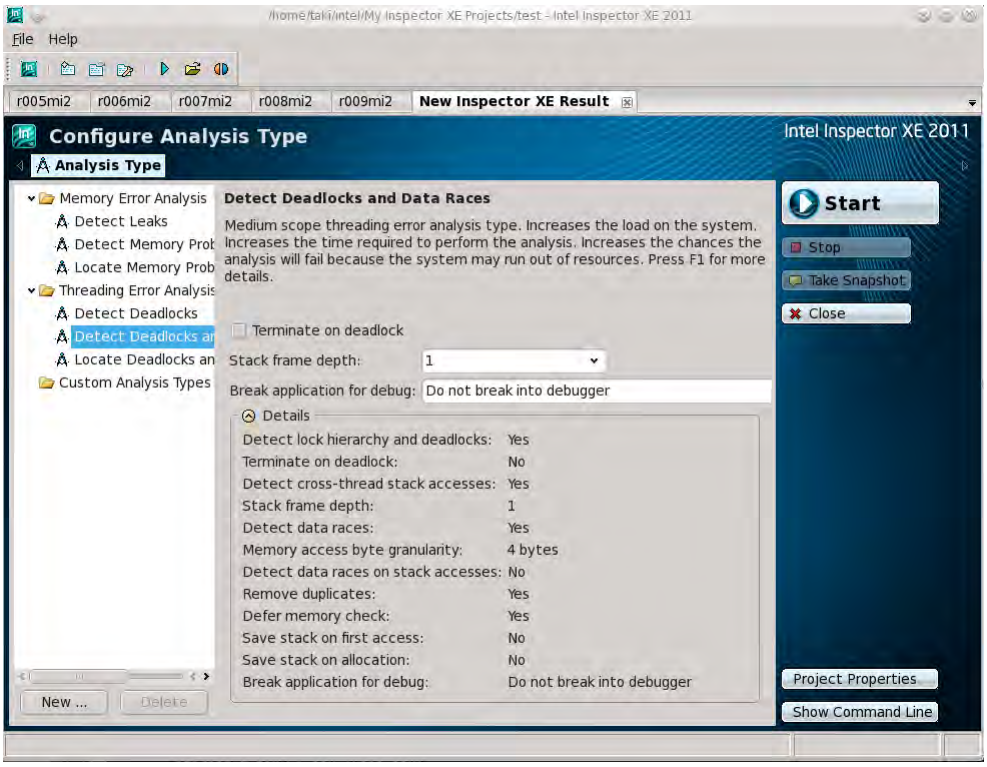


Figure 3.4: Intel Inspector XE 2011.

3.6.5 OProfile

OProfile[26] is a low overhead, system-wide profiler for Linux released under the GNU General Public License (GPL). It makes use of a kernel driver and a daemon to collect sample data from a variety of hardware performance counters on the CPU. It also supplies several tools for analysing and reporting on the resulting sampling data [27, 97]. OProfile is supported by most modern Linux kernels (through a loadable kernel module) running on a variety of CPUs, including those produced by Intel and AMD. OProfile does not require any special compiler support besides the inclusion of debug symbols if source code annotation is desired [97].

OProfile makes use of sampling to gather its profiling data. The profiling data can then be analysed and summarised using the provided **opreport** and **opannotate** tools, which produce flat profiles, call-graphs, and annotated source code listings. This information can then be used to identify performance hotspots, poor cache utilisation, and other performance issues. OProfile does not include a GUI for controlling and analysing profiling runs, however, Eclipse provides integrated profiling support using OProfile [97, 119].

3.6.6 Valgrind

Valgrind[27] is an open source software development tool suite for detecting and locating memory related errors, as well as profiling and thread-related error detection. It is supported by most major Linux distributions, running on x86, x86-64, and PowerPC based platforms. Valgrind began as a memory error-checking tool for Linux, but it has since advanced to become an extensible framework for dynamic program analysis tools. The Valgrind tool suite features command line tools for the automatic detection of memory and threading related bugs, many of which are difficult to discover manually, as well as performance profiling [49, 111, 113, 112, 143, 142]. The tools report their findings using a command line interface and only the Callgrind and Cachegrind tools have a GUI interface for displaying results [49].

Valgrind uses heavyweight dynamic binary instrumentation, which means that the program's build configuration does not need to be modified prior to analysis. It also means that Valgrind can be used for programs written in any language, however, it is primarily aimed at C and C++ due to the nature of memory management in these languages. Essentially, Valgrind translates the machine code of the target program into an intermediate representation (IR) that is then instrumented by the Valgrind tool plug-in. This instrumented IR is then recompiled in a just-in-time

[26] `http://oprofile.sourceforge.net`
[27] `http://valgrind.org/`

fashion and collects the relevant profiling and debugging data during execution. Unfortunately, this comes with a significant overhead and execution times are substantially higher than those of the original program [111, 113, 112, 143, 142].

Memcheck

Memcheck (`--tool=memcheck`) is a memory error detection tool for detecting out-of-bounds memory accesses, undefined values, incorrect allocation and de-allocation of memory, and memory leaks. Memcheck inserts extensive instrumentation into the target program and replaces the standard memory allocators with its own heavily instrumented version, thereby keeping track of all memory reads and writes, allocations, and de-allocations. This allows it to detect discrepancies in the target program's memory management code [49, 143].

Callgrind and Cachegrind

Cachegrind (`--tool=cachegrind`) is a cache usage and branch prediction profiler that simulates the L1, I1, and last-level (LL) caches. Since the caches are simulated, Cachegrind is able to produce very detailed profile data for cache reads, writes, and misses for the varying levels of cache memory, as well as branch mispredictions. Summary statistics and annotated source code listings can be generated with the `cg_annotate` command [27, 49, 113, 142].

Callgrind (`--tool=callgrind`) is a call-graph profiling tool that collects instruction execution counts, function call counts, and function call relationships. It also features cache and branch prediction simulation similar to that of Cachegrind. The resulting profile data is summarised and displayed on the command line. The `callgrind_annotate` command takes the profile data, prints a sorted list of profiled functions, and annotates the supplied source code for the target program [113, 142].

Unlike many of the other Valgrind tools, the Callgrind and Cachegrind tools produce profile data that can be analysed using the GUI-based KCachegrind program. KCachegrind provides graphical visualisations of the profile data, which are much easier to interpret than the text-based output of the command line tools [49, 113].

DRD and Helgrind

Helgrind (`--tool=helgrind`) and DRD (`--tool=drd`) are Valgrind tools for detecting errors in multithreaded programs that are written in C and C++ using Pthreads, or Pthreads-based

thread abstractions. DRD features improved support for the Boost Threads library and programs parallelised using OpenMP. These tools are capable of detecting data race conditions, potential deadlock situations, and misuses of the Pthreads API. The detection algorithms use a combination of static and dynamic analysis to improve speed and accuracy, but they are prone to false positives, which can be suppressed using special configuration files [111, 143].

3.6.7 AMD CodeAnalyst for Linux

AMD CodeAnalyst[28] is an open source, system-wide, GUI-based statistical profiler for x86 and x86-64 based systems running Windows and Linux. CodeAnalyst for Linux is based on OProfile and shares the basic features of OProfile. CodeAnalyst features time-based profiling (TBP), event-based profiling (EBP), and instruction-based profiling (IBS). IBS and some of the EBS features rely on specific hardware support provided by AMD CPUs, and are therefore not available on non-AMD platforms. TBP identifies the performance hotspots in the program, which are the parts of the program with the highest execution time. EBP is used to diagnose specific performance issues within a hotspot, such as cache misses, using hardware event counters. IBS is a more advanced and more accurate form of sampling, supported by AMD Family 10h processors and up, that precisely associates performance events with the responsible instructions, thereby aiding the diagnosis of performance issues [3, 28, 29].

The CodeAnalyst GUI provides a number of features to assist in profiling, such as offering a number of predefined profiling configurations using either TBP, EBP, or IBS. It also provides data aggregation and performance summaries using tables and charts, along with the ability to drill down from a system-wide overview to the source code or instruction level. Multithreaded program analysis is supported with the ability to generate thread profiles, thereby assisting in the diagnosis of processor core affinity, memory locality, and memory access issues. Profiling data can be imported from the OProfile command line tools, allowing for scripted profiling. It also allows the user to compare different profiling runs in a session to assess the impact of any optimisation efforts [3, 28, 29].

An example screenshot of the CodeAnalyst for Linux GUI is provided in Figure 3.5.

[28] `http://developer.amd.com/cpu/CodeAnalyst/Pages/default.aspx`

3.6.8 Intel VTune Amplifier XE 2011 for Linux

Intel VTune Amplifier XE 2011 (formerly Intel VTune Performance Analyzer with Intel Thread Profiler)[29] is a kernel-level (loadable kernel module), system-wide, statistical profiler for 32-bit and 64-bit Windows and Linux systems. It incorporates the functionality of Intel VTune Performance Analyzer, Intel Parallel Amplifier, and Intel Thread Profiler, as well as a number of additional improvements, such as better support for Intel Core i7-based CPUs. VTune supports both Intel and non-Intel CPUs, however, certain hardware-specific features are only supported on genuine Intel processors. VTune Amplifier XE for Linux features both a command line interface and a very powerful and user-friendly GUI interface based on the Eclipse platform [17, 66, 67, 78].

VTune requires little or no changes to the target program's build configuration and supports C/C++, Fortran, and assembly based programs, built with the normal compiler and optimisation flags. It can even be attached to running programs. The profiler uses accurate, low-overhead, event-based sampling techniques to gather the relevant profile data, with optional overhead compensation [66, 78, 101].

VTune provides a number of easy-to-use performance analysis wizards and pre-defined EBS profiling experiments, such as the call-graph wizard and the basic sampling wizard, which identifies hotspots in terms of the number of unhalted clock cycles. Other pre-defined profiles include cache miss, TLB miss, and branch misprediction analysis. VTune also features thread profiling capabilities, such as thread timeline visualisation and lock analysis. The resulting profile information is summarised and presented in a variety of formats using tables, graphs, timelines, and annotated source code listings. These results can be filtered to highlight particular areas of interest and the interface allows the user to drill down from a high-level, system-wide overview to the source code level. VTune even highlights optimisation and parallelisation opportunities and provides appropriate suggestions [17, 66, 78, 101].

An example screenshot of the VTune for Linux GUI is provided in Figure 3.6.

[29] `http://software.intel.com/en-us/articles/intel-vtune-amplifier-xe/`

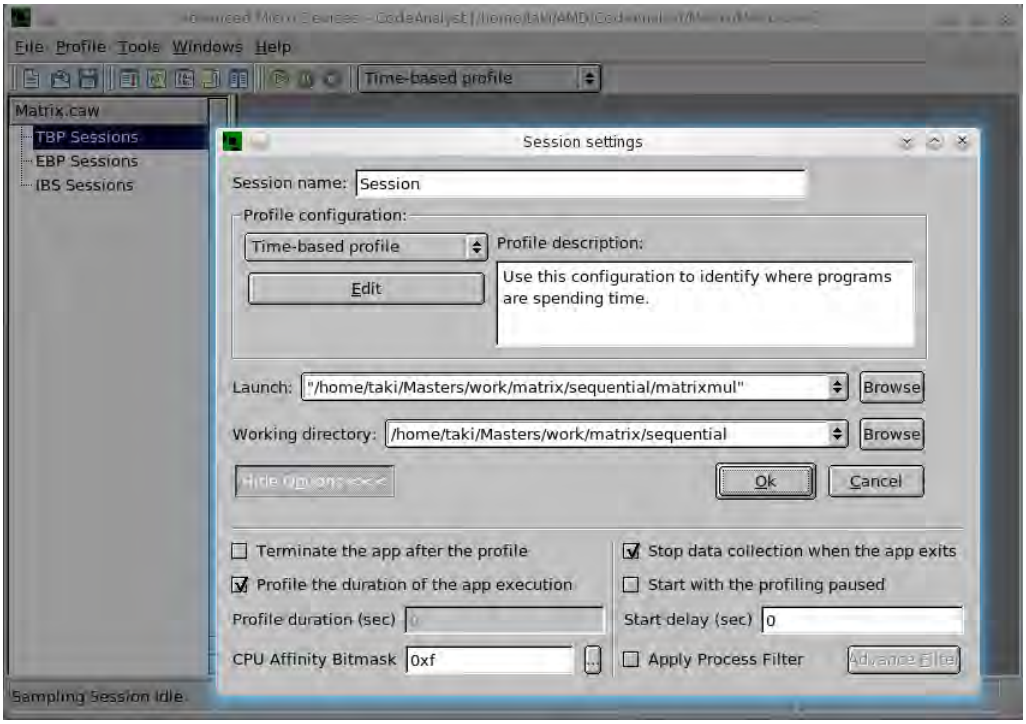


Figure 3.5: AMD CodeAnalyst for Linux.

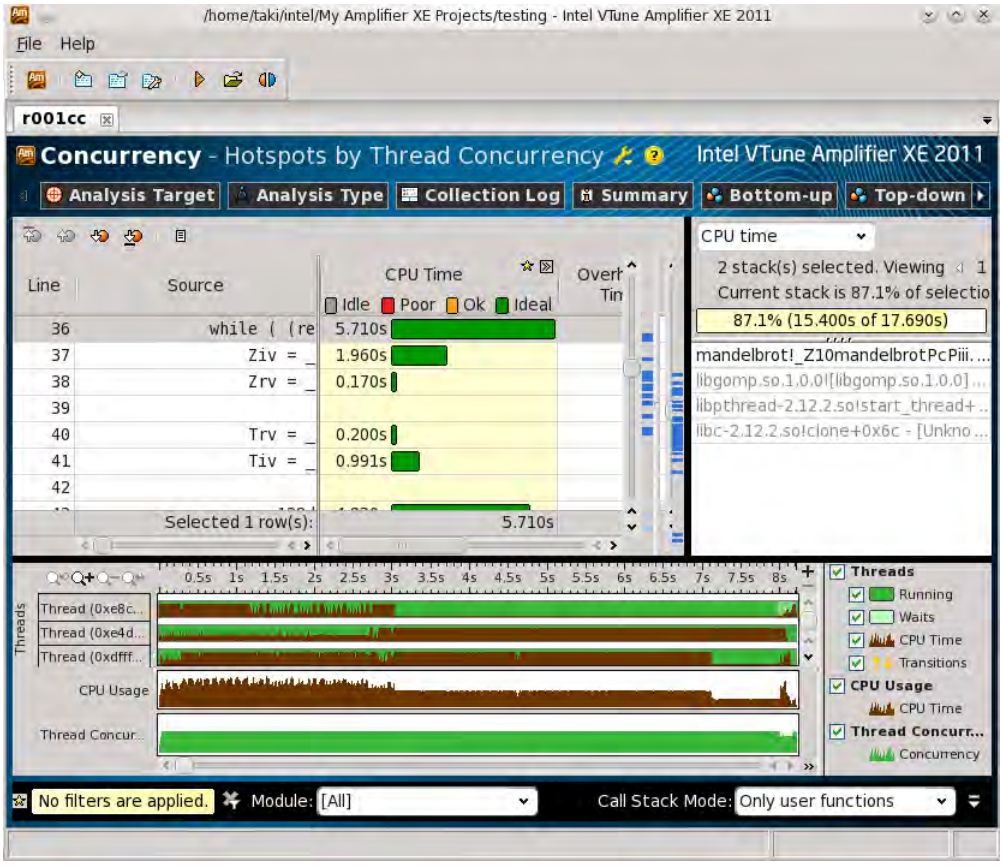


Figure 3.6: Intel VTune Amplifier XE 2011 for Linux.

Chapter 4

Parallel Programming Techniques

4.1 Introduction

The process of optimising a program or application to improve its performance requires diligence and a methodical approach. Simply attempting to optimise every minor piece of code only leads to wasted time and energy for insignificant performance gains, or even performance degradation [25]. In this chapter, we present a common approach to performance analysis and tuning, which describes a straightforward, iterative approach to both sequential and parallel optimisation. We then describe a number of high-level sequential optimisations that assist in the tuning of a sequential program before attempting parallelisation. Parallel programming is not a new phenomenon. As such, there is a wealth of information and experience available relating to the creation of high-performance parallel programs. A number of parallel programming patterns and models have emerged that provide a structured approach to creating parallel programs [103]. We describe some of these patterns and, finally, we briefly discuss several optimisations for improving parallel performance.

4.2 Performance Analysis and Tuning

For effective results, performance analysis and tuning requires a methodical, iterative approach that relies on empirical performance analysis to guide optimisation efforts and implementation decisions. Premature optimisation, or optimisations based on the programmer's guesses as to the source of performance bottlenecks, often result in disappointment and can affect the

correctness and maintainability of the program [13, 17]. Optimisation and tuning is possible at a number of stages in the software development process: at the design stage where overall program structure is defined, at the algorithm stage where the appropriate algorithms and data structures are developed or selected, and at the source code level where code tranformations are performed provided a suitable sequential program is available. The most important consideration or goal when performing optimisations at any level is that the correctness of the program must be maintained [13].

4.3 The Tuning Process

The performance tuning process is divided into a number of phases that are followed in order. The process is repeated until the performance of the program reaches a satisfactory level or the point of diminishing returns. The tuning process is described below and illustrated in Figure 4.1 [17, 25, 67, 73].

1. **Measure Performance Data:** The first step to effective optimisation is performance measurement and profiling using realistic workloads. The choice of performance metrics should be appropriate to the nature of the program and provide a reliable and consistent means for comparing the performance between different versions of the program. It is important to capture baseline performance and profiling information so that quantitative performance comparisons can be made. Overall performance indicators include metrics such as total elapsed time, latency, and throughput. Performance events and code execution times gathered through profiling provide invaluable information for identifying performance issues and determining whether an optimisation has been effective [13, 73]. There is a wide variety of profiling tools available for various platforms. Some of the more popular tools are listed and described in Section 3.6.

2. **Analyse Data and Identify Performance Issues and Hotspots:** If the measured performance is unsatisfactory, the profiling information must be analysed to identify and locate hotspots and performance issues. Hotspots are usually the best place to begin optimising as they often provide the greatest overall performance improvement when optimised appropriately. This follows the so-called “80-20” optimisation rule (also referred to as the “90-10” rule), which states that 80 percent of a program’s execution time occurs in 20 percent of the code. The optimisation of hotspots may also resolve or expose other performance issues. Thereafter, other performance issues, such as excessive cache misses or branch mispredictions, can be analysed. It is advisable to focus on specific issues for

optimisation instead of trying to resolve all the issues at once [13, 73]. How one analyses the profiling data depends on the format of the profiler output and the functionality of the profile analysis tools supplied with the profiler. GUI-based profilers, such as Intel VTune and AMD CodeAnalyst, assist by summarising the performance data, highlighting performance issues, and locating the source of the issues down to the source code level. In some cases, they even provide optimisation advice [73, 78]. More in-depth analysis techniques can be employed to highlight specific issues, such as loop-centric profiling to identify parallelism and data profiling to identify cache bottlenecks [110, 120].

3. **Devise Optimisations for a Specific Issue:** Once a hotspot or performance issue has been identified and selected for optimisation, devise a number of possible optimisations to resolve the issue. These optimisations can include using more aggressive compiler optimisation options and compiler directives, making use of high-performance libraries, changes to the algorithm or data structures to improve data access patterns and synchronisation, manual source-level optimisations such as loop transformations and manual vectorisation, and parallelisation using implicit or explicit parallelisation. Compiler options and directives can provide cheap, unobtrusive performance improvements. Intrusive source-level optimisations require more effort, increase the likelihood of introducing errors, and reduce portability. If parallelisation is the goal, hotspots provide a good starting point for identifying potential parallelism [17, 73].

4. **Implement the Optimisation:** This involves making the changes to the program to implement the desired optimisations, which might only require recompiling the program with new compiler options, or it could require making changes to the code. Attempting to implement multiple optimisations simultaneously is not advised since the effects of the different optimisations can be obscured, limiting one's ability to gauge the effectiveness of specific optimisations. Therefore, small, incremental changes are suggested. OpenMP supports incremental parallelisation due to the simple, directive-based approach, making it a good choice for introducing parallelism into an existing application. It is also advisable to make use of a version control system so that changes can be reversed easily if they are found to be ineffective [17, 21, 73].

5. **Test and Debug the Program:** Testing is a vital step in the process. Each optimisation must be tested, both for effectiveness and correctness. Optimisations must not affect the accuracy of results or correct functioning of the program. If errors are discovered, they must be debugged and corrected before moving on to further optimisations. Error checking tools such as Valgrind and Intel Inspector can assist in detecting memory and threading errors. The effectiveness of the optimisation must also be evaluated and changes

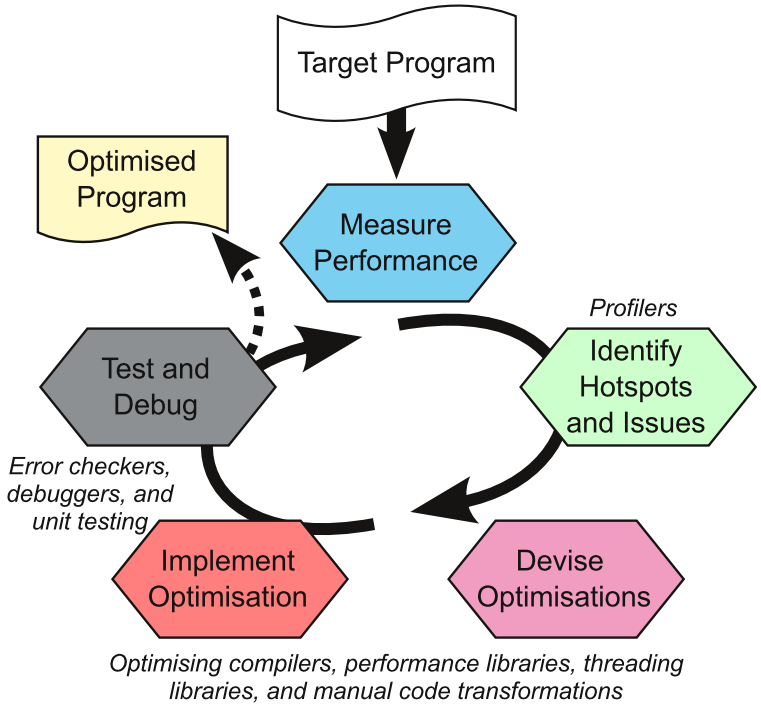


Figure 4.1: Approach to performance analysis and tuning.

that do not produce acceptable improvements should be reverted, particularly if they affect the maintainability of the program [17, 73]. The tuning process is then repeated until an acceptable improvement in the overall performance level is achieved.

4.4 Sequential Optimisations

While it may seem odd to discuss sequential optimisation techniques in a chapter dedicated to parallel programming techniques, many of these optimisations are beneficial to shared-memory multiprocessor systems. For instance, performance issues resulting from inefficient cache usage in a sequential program are likely to be amplified in the parallel implementation. Therefore, it is necessary to start with an optimised version of the serial program when attempting to implement parallelism [17, 21]. The optimisations discussed below are primarily high-level optimisations targeted at the structure of the code and data, as opposed to low-level C and C++ optimisations, which are beyond the scope of this thesis.

4.4.1 Loop Optimisations

As described in Chapter 2, modern shared-memory multiprocessor systems have a hierarchical memory architecture. The smaller, faster levels of cache memory are located closer to the processor cores on the CPU die, while the much larger and slower main memory is accessible via the system bus. This has obvious performance implications when attempting to write optimal code as loading data from cache memory is magnitudes faster than acquiring the data from main memory. Since data is loaded into the cache in fixed length *cache lines*, code should be structured to exploit this by accessing data stored in contiguous sections of memory. Appropriate structuring of data access can help ensure that data is available in the cache, thereby reducing expensive cache misses [21, 27, 54, 89]. Many of the loop optimisations described below are performed by good optimising compilers, such as the Intel C++ compiler [73, 80]. However, various structural issues, such as unnecessary loop-carried dependencies and loop invariant statements, may prevent the compiler from performing these optimisations. Therefore, it is sometimes necessary to optimise and restructure the code manually to assist the compiler in performing further optimisations [21].

Loop Interchange

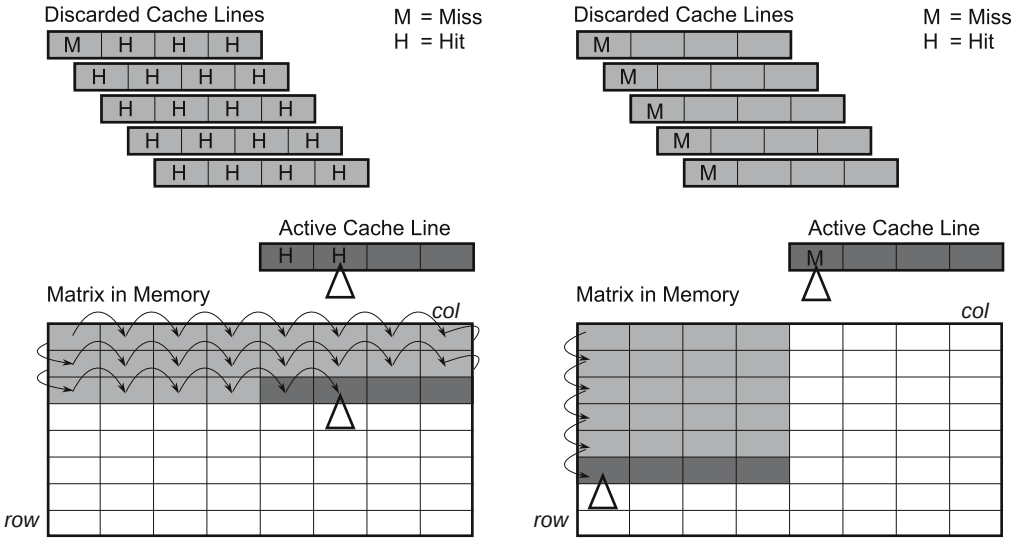
Two-dimensional arrays provide a very clear example of caching effects. In C and C++, two-dimensional arrays are stored in memory such that rows are arranged contiguously, where elements in the same row are located adjacent to each other. This, coupled with the cache line mechanism, makes it possible to optimise data access by ensuring that elements of the array are accessed row by row (row-wise access), thereby improving spatial locality. Listing 4.1 demonstrates optimal cache usage through row-wise access as the use of each cache line is maximised before it is evicted to make way for new blocks [21, 27, 54, 89].

```
1 double matrix[MSIZE][MSIZE];
2 double sum_sq = 0.0;
3 init_matrix();
4
5 for (int row = 0; row < MSIZE; row++)
6   for (int col = 0; col < MSIZE; col++)
7     sum_sq +=   matrix[row][col]
8               * matrix[row][col];
```

Listing 4.1: Row-wise array access.

```
1 double matrix[MSIZE][MSIZE];
2 double sum_sq = 0.0;
3 init_matrix();
4
5 for (int col = 0; col < MSIZE; col++)
6   for (int row = 0; row < MSIZE; row++)
7     sum_sq +=   matrix[row][col]
8               * matrix[row][col];
```

Listing 4.2: Column-wise array access.



(a) Row-wise **(b)** Column-wise

Figure 4.2: Cache usage for row-wise (a) and column-wise (b) matrix access.

On the contrary, the code in Listing 4.2 shows inefficient column-wise access. For sufficiently large matrices, this results in cache lines being discarded after reading a single element from the line, only to fetch the same block at a later stage to read another element from the same row. These long strides in memory make it hard for the cache controller to prefetch the next required elements, resulting in excessive cache misses and cache line evictions [27]. This concept is represented diagrammatically in Figure 4.2 and the performance of the two implementations is summarised in Table 4.1 for `MSIZE` equal to 4096.

The *loop interchange* optimisation involves restructuring loops with column-wise access, as shown in Listing 4.2, to make use of the more efficient row-wise access pattern demonstrated in Listing 4.1. However, this optimisation cannot be applied blindly to all loops exhibiting column-wise access. The programmer must be careful to ensure that the correctness of the program is not affected [21, 89]. This involves analysing the array access patterns and applying the following rule:

Table 4.1: Loop interchange performance summary.

	Row-wise	Column-wise
Wall clock time	0.205s	0.418s
Memory references	302,100,031	302,100,031
L1 data misses	4,195,850	18,875,914
L1d miss rate	1.39%	6.25%

> If any memory location is referenced more than once in the loop nest and if at least one of those references modifies its value, then their relative ordering must not be changed by the transformation [21].

This loop optimisation can vastly improve the performance of inefficient loops, particularly for larger array dimensions, where cache evictions become more frequent.

Blocking or Loop Tiling

Blocking or *loop tiling* is an advanced loop transformation for dealing with large data sets and bad memory access patterns. Blocking breaks the loop into smaller subsets, known as *blocks* or *tiles*, which are able to fit into the cache and require fewer page tables, improving both cache and TLB miss rates [21, 27, 34, 54, 89]. This technique can be used for loops where loop interchange is ineffective due to a combination of both strided and sequential loop accesses as demonstrated in Listing 4.3 [21, 89].

```
1 double res[MSIZE][MSIZE];
2 double orig[MSIZE][MSIZE];
3 init_matrix();
4
5 for (int col = 0; col < MSIZE; col++)
6   for (int row = 0; row < MSIZE; row++)
7     res[i][j] = orig[j][i]
```

Listing 4.3: Combined column-wise and row-wise array accesses.

Blocking is performed by replacing a single loop with two loops, the outermost of which is incremented in appropriately sized chunks, as shown in Listing 4.4. The value of `CHUNK` is dependent on cache size and determines the size of each tile. The result is that the two innermost loops are constrained to a smaller section of the overall iteration space, thereby improving both spatial and temporal locality [21, 27, 54, 89].

```
1 double res[MSIZE][MSIZE];
2 double orig[MSIZE][MSIZE];
3 init_matrix();
4
5 for (int tile = 0; tile < MSIZE; tile += CHUNK)
6   for (int row = 0; row < MSIZE; row++)
7     for (int col = 0; col < min(MSIZE - tile, CHUNK); col++)
8       res[i][tile + col] = orig[tile + col][i]
```

Listing 4.4: Loop tiling for matrix operations [21].

Loop Unrolling

Loop unrolling is used for loops with a fixed number of iterations, where the loop body is small and the loop execution overheads (loop control variable incrementation, completion test, and branch to loop body) contribute a significant portion to the loop's execution time. Loop unrolling involves unpacking several loop iterations and duplicating the loop body such that a single pass of the loop effectively executes multiple loop iterations. This increases the workload of the loop body, thereby reducing the effect of the loop overheads, as well as improving cache usage, branch prediction, and instruction-level parallelism. The number of iterations that are unrolled is known as the *unroll factor*. *Complete loop unrolling* removes the loop overheads entirely by removing the loop and duplicating the loop body by the total number of iterations. For unroll factors that do not fully divide into the iteration count, an additional cleanup loop is required to handle the remaining loop iterations. Manually unrolling loops is seldom required as modern compilers are particularly good at performing this loop transformation [4, 21, 80, 89, 115]. Listings 4.5 and 4.6 show, respectively, the before and after code for loop unrolling. However, loop unrolling must be balanced with code size reduction, which benefits instruction cache usage [27].

```
1 for (int i = 0; i < ASIZE; i++) {
2   arr_a[i] = arr_a[i] * 2;
3   arr_b[i] = arr_a[i] + 2;
4 }
5
6
```

Listing 4.5: Loop before unrolling.

```
1 for (int i = 0; i < ASIZE; i+=2) {
2   arr_a[i] = arr_a[i] * 2;
3   arr_b[i] = arr_a[i] + 2;
4   arr_a[i+1] = arr_a[i+1] * 2;
5   arr_b[i+1] = arr_a[i+1] + 2;
6 }
```

Listing 4.6: Unroll factor of 2.

Unroll and Jam

Unroll and jam is an advanced form of loop unrolling for nested loops where the inner loop does not benefit from loop unrolling, but the outermost loop does. In essence, the outermost loop is unrolled by an appropriate unroll factor, and then the duplicated inner loops are combined or *jammed* together to form one inner loop with a loop body consisting of the duplicated loop bodies [21, 89].

Loop Fusion

Loop fusion is a transformation that involves the merging of multiple loops that share the same iteration space and, ideally, data. By combining the loop bodies of multiple loops, cache and register re-use is increased, loop overheads are reduced, and instruction-level parallelism is improved due to the increased workload of the loop body. However, care must be taken to ensure that a valid ordering of data is preserved. An example of the before and after code for loop fusion can be seen in Listings 4.7 and 4.8, respectively [21, 69, 89, 115].

```
1 for (int i = 0; i < ASIZE; i++) {
2   arr_a[i] = i * 2;
3   arr_b[i] = arr_a[i] + 2;
4 }
5 for (int i = 0; i < ASIZE; i++)
6   arr_z[i] = arr_a[i] * arr_b[i];
```

Listing 4.7: Loop before loop fusion.

```
1 for (int i = 0; i < ASIZE; i++) {
2   arr_a[i] = i * 2;
3   arr_b[i] = arr_a[i] + 2;
4   arr_z[i] = arr_a[i] * arr_b[i];
5 }
6
```

Listing 4.8: Fused loop.

Loop Fission

Loop fission is a technique used to split a loop into a number of separate loops, thereby improving cache usage or enabling further optimisations that are hindered by the current loop structure. It can be used to remove loop-carried dependencies, making it easier to parallelise or vectorise the loops. It is also useful for large loop nests where the loop data does not fit into the cache. Listings 4.9 and 4.10 demonstrate loop fission and the subsequent use of loop interchange [21, 69, 89, 115].

```
1 for (int c = 0; c < MSIZE; c++) {
2   oscil[c] = (c % 9) - 4;
3   for (int r = 0; r < MSIZE; r++)
4     res[r][c] = src[r][c] * src[r][c];
5 }
```

Listing 4.9: Suboptimal loop structure that cannot be interchanged.

```
1 for (int c = 0; c < MSIZE; c++)
2   oscil[c] = (c % 9) - 4;
3 for (int r = 0; r < MSIZE; r++)
4   for (int c = 0; c < MSIZE; c++)
5     res[r][c] = src[r][c] * src[r][c];
```

Listing 4.10: Loop fission and subsequent loop interchange.

4.4.2 Aligning, Padding, and Sorting Data Structures

Cache usage can be improved through optimisation techniques such as ensuring natural alignment of data structures by padding and sorting data structure members. *Data structure alignment* refers to the way in which data structures are arranged in memory. A data structure is said to be aligned if it is placed at a memory address that is a multiple of the largest power of two data member size when allocated space in memory. Misaligned data access can have a significant effect on performance, particularly if it results in data from the same structure being split over cache line boundaries as two memory accesses are required instead of just one [4, 27, 34, 80].

Compilers typically align data members according to the word length, or their natural alignment, which is based on the size of the data type. This is an important consideration when defining the layout of a **struct** or **class**, as the ordering of data members may result in unused space where a larger member is preceded by a smaller member that does not extend to the natural alignment boundary of the larger data member. This is resolved by sorting the data members such that the largest members are allocated first. It is possible to go a step further and pad the data structure with unused data members such that the size of the data structure is a multiple of the largest data member's type size. When working with a collection of a particular data structure, it may be necessary to pad the structure so that it fits the cache line better and does not result in sections of the same structure being split over cache line boundaries. If only a few of the data members of a particular structure are accessed frequently, cache usage can be optimised by splitting the structure so that the frequently accessed members are stored sequentially, while the remaining members are stored in a separate data structure [4, 27, 34, 80]. Data structure sorting and padding is demonstrated in Listings 4.11 and 4.12.

```
1 struct unsorted {
2   int age;        // 4 bytes
3   char gender;    // 1 byte
4                   // 3 unused bytes
5   double balance; // 8 bytes
6   char fname[11]; // 10 bytes
7                   // 2 unused bytes
8 }; // 28 bytes total (5 unused bytes)
```

Listing 4.11: Unsorted struct resulting in unused space.

```
1 struct sorted {
2   double balance; // 8 bytes
3   int age;        // 4 bytes
4   char gender;    // 1 byte
5   char fname[10]; // 10 bytes
6   char pad;  // pad to 8 byte boundary
7 }; // 24 bytes total (1 unused byte)
8
```

Listing 4.12: Sorted struct allowing for better data alignment.

The language specifications for C and C++ do not permit compilers to sort data structures, so it is necessary to re-organise data structures by hand. However, many compilers allow the programmer to specify the alignment of a variable or data structure explicitly. The GNU C/C++

Compiler supports this through the `__attribute__ ((aligned(n)))` directive, and other compilers such as the Microsoft and Intel C/C++ compilers provide the `__declspec(align(n))` directive for this purpose [4, 34, 80].

4.4.3 Avoid Branching

Modern CPUs are very good at exploiting instruction-level parallelism using deep instruction pipelines, out-of-order execution, and speculative execution. However, speculative execution relies on accurate *branch prediction* for the outcome of branch instructions, such as those generated by **if** and **switch** statements. The processor's branch prediction mechanisms build up a history of branch outcomes and use this history to improve the chance of correctly predicting the outcome of a branch. A branch misprediction occurs when an incorrect branch outcome is taken, resulting in stalls in the instruction pipeline as speculative results are discarded and the correct branch is executed. This can have a significant effect on performance, especially if it occurs in a critical loop [4, 21, 34, 80].

Branch mispredictions are typically the result of complex branch conditions, where the result is fairly random, or many branches located very close together, particularly if the branches are in the body of a loop. These factors hinder the processor's ability to predict the correct outcome. If profiling reveals that there is a high branch misprediction rate (10 percent or less is considered normal) for a section of code, then it may be necessary to restructure the code to simplify the branches, space them out by a few instructions, or remove them completely. It is sometimes possible to remove a branch and replace it with bit-masking, which can provide additional performance improvements if combined with loop unrolling and SSE vectorisation [4, 21, 34, 80].

4.4.4 Vectorisation

Vectorisation using SIMD instructions, such as those provided by the SSE instruction sets, can provide significant speedup, particularly for computationally intensive loops and mathematical calculations. SSE is particularly well-suited to multimedia and streaming applications. Vectorisation of suitable code can be performed either explicitly or automatically by a compiler that supports automatic vectorisation, such as GCC and the Intel C/C++ compiler. SSE provides a variety of 64-bit and 128-bit packed data types (vectors) that store multiple float, double, and integer values in the XMM registers on the processor. The SSE instructions can then perform

operations on all the vector elements simultaneously, thereby introducing additional instruction-level parallelism. However, the use of SSE instructions requires that the vector data structures be aligned by 16 bytes [4, 34, 80, 89].

Automatic vectorisation simply requires specifying the relevant compiler options when compiling the program (see A.1 and A.2). The compiler identifies sections of code that may benefit from vectorisation and converts the code to make use of the appropriate SSE instructions. The Intel C/C++ Compiler is particularly good at performing automatic vectorisation, however, certain conditions can prevent the compiler from vectorising a section of code that would benefit from such an optimisation. For the compiler to perform vectorisation, the arrays and data structures involved must be aligned to 16 bytes and it must be clear to the compiler that any pointers or references do not alias. Data alignment is covered in Section 4.4.2. It is also possible to tell the compiler that specific data structures are aligned by adding a compiler directive, `#pragma vector aligned`, above the relevant structures. The `#pragma ivdep` directive and the `restrict` variable qualifier inform the compiler that pointers do not alias. Vectorisation of loops works best when the loop iteration count is a multiple of the number of vector elements [4, 34, 80]. It is also possible to access compiler diagnostics for the vectorisation process. Depending on the compiler, the diagnostics specify which loops were vectorised and the reasons for not vectorising a loop (e.g., data dependencies) [34, 73, 89].

Explicit vectorisation is required for loops that the compiler is unable to vectorise. This involves manually packing, shuffling, and unpacking the vectors and performing the appropriate SSE operations using the SSE intrinsics functions provided by the compiler libraries[1], as well as unrolling the loops to the required level. This is also necessary for compilers that support SSE, but not automatic vectorisation. Explicit vectorisation requires extensive code modifications and adds a significant number of instructions to the program, which reduces the readability of the code and increases the likelihood of introducing errors. Some compilers provide libraries with classes that abstract away the low-level intrinsic functions. These classes make it easier to implement SSE in the program, but it is usually advisable to seek alternatives such as existing SSE-optimised libraries that implement the desired functionality [34, 80].

4.5 Parallel Programming Patterns

Parallel programming is often characterised as a difficult process, requiring extensive knowledge and experience to achieve success. Therefore, a methodical and informed approach to

[1] These functions are typically prefixed with `__mm_` and are named according to the type of operation and the data types of the operands. Various packed data types are also provided for use with the intrinsic functions [34, 80].

finding and expressing concurrency is necessary to enable and foster parallel programming abilities in a wider range of programmers. *Patterns* and *pattern languages* have been used to great effect in the object-oriented programming realm to assist software developers in the creation of complex, modular software systems. Patterns describe good solutions to common software development problems and encapsulate the thought processes and experience of domain experts [17, 102, 103, 125]. Mattson, Sanders, and Massingill [102] present a pattern language for parallel programming that, when combined with the programmer's understanding of the problem domain, allows for the creation of a detailed parallel design and implementation. The pattern language follows four key stages or design spaces, progressing from the design phase to the implementation of parallelism in the program. Others have taken this concept a step further and produced parallel algorithmic skeletons that make use of language features such as templates [30]. The design spaces of the parallel programming pattern language are summarised below, however, for a more in-depth description, it is best to consult the original text [102].

4.5.1 Identifying Concurrency

In the *Finding Concurrency* design space, the primary problem is that of identifying and exposing exploitable concurrency, as well as identifying and managing dependencies associated with concurrency. This involves decomposing the problem into a collection of computationally intensive tasks, working on mostly independent units of data. Accordingly, the key aspects of the problem and its data need to be fully understood by the developer and there needs to be a sufficient workload to justify parallelisation. If an existing program is being parallelised, the hotspots identified through profiling are typically a good place to begin looking for exploitable concurrency [102, 103].

Decomposition Patterns

The *Task Decomposition* and *Data Decomposition* patterns are concerned with dividing up the problem so that it can execute concurrently.

- The **Task Decomposition** pattern addresses the problem of how one decomposes the program into sequences of instructions or *tasks* that can be executed concurrently with minimal interference from instructions in other tasks. This is achieved by analysing the computationally intensive segments of the problem, the primary data structures, and the interactions between the two. The problem is then broken up by defining a number of

tasks and the manner in which these tasks access the data. It is important to define these tasks to be largely independent to reduce the overheads of managing data and ordering dependencies, while, at the same time, making sure that the tasks allow for good load balancing over the available processors. It is best to start with as many tasks as can be identified and merge tasks where appropriate. Examples of possible tasks include the iterations of a loop, independent function calls (often referred to as *functional decomposition*), and groups of sequential operations [17, 102, 124].

- The **Data Decomposition** pattern addresses the problem of how the data can be partitioned into units or segments that can be operated on independently. The key concern in this approach is the manner in which the data is produced and used to solve the problem. This approach is appropriate as a starting point if computation centers around large data structures or if similar, independent operations are performed on different parts of the data. For a task-based design, the data decomposition will follow the data requirements of the tasks. For a primarily data-based approach, the design follows the nature of the data structures and how the data is to be distributed and processed as concurrent tasks. Common data structures include large arrays of varying dimensions and recursive data structures, such as trees. Data decomposition is a common approach for many scientific applications [17, 35, 102, 124].

However, task and data decomposition are just different takes on the same basic problem decomposition and one decomposition can often be expressed in terms of the other. The importance of highlighting the distinction between the two and emphasising one approach over the other lies in its ability to make the design easier to comprehend. The flexibility, efficiency, and simplicity of the design must also be considered when addressing the problem decomposition as these relate to the scalability and maintainability of the program [35, 102].

Dependency Analysis Patterns

The *Group Tasks*, *Order Tasks*, and *Data Sharing* patterns are usually applied in order and provide approaches for grouping tasks and dependency analysis.

- The **Group Tasks** pattern addresses the problem of how the tasks can be grouped to simplify the management of dependencies. This step takes the results of the task and data decompositions and analyses the dependencies among tasks. While tasks are defined to be as independent as possible, there is still a structure to the set of tasks that imposes

certain constraints on concurrent execution, such as temporal dependencies and other ordering constraints. By grouping tasks with similar constraints, it becomes easier to perform dependency analysis and manage the constraints of whole groups instead of each separate task. This is typically achieved by looking back at the decomposition process and identifying tasks that can be grouped, followed by merging groups that share the same constraint, and looking at the constraints between groups that could imply further grouping. The larger the task groups, the greater the flexibility and scaling [102].

- The **Order Tasks** pattern solves the next problem of how the groups of tasks, as defined above, are ordered to satisfy constraints between the tasks. This involves identifying the temporal dependencies, requirements for simultaneous execution, and lack of constraints between the tasks. These can usually be identified by analysing the data flows between tasks and assessing external ordering constraints such as sequential I/O. The ordering of tasks should be sufficiently restrictive as to ensure correctness, while avoiding unnecessary restrictions that negatively influence efficiency and flexibility [17, 60, 102].

- The **Data Sharing** pattern addresses the problem of how data is shared among tasks. Although the data decomposition process identified chunks of data that are local to tasks, there are usually situations where data must be shared, such as updates to global data structures or when a task requires access to another task's local data. The goal, therefore, is to identify such data sharing situations and define mechanisms for efficiently managing the shared accesses to avoid race conditions, using synchronisation constructs such as locks and semaphores. Shared access can be *read-only* (no synchronisation required), *effectively-local* due to partitioned access (no synchronisation required), or read-write (mutual exclusion required) [17, 102].

Design Evaluation Pattern

The *Design Evaluation* pattern involves analysing the earlier design decisions and identifying any weaknesses in the design so that they can be addressed before moving onto the algorithm design. This step is important because it is often difficult to correct early design mistakes later on in the development process. If multiple decompositions are possible, it may be necessary to compare them to the current design and make adjustments where necessary. The design should be evaluated on quality and suitability for the platform. Quality is defined by the design's efficiency, flexibility, and simplicity. Flexibility refers to how easy it is to adjust aspects of the design, such as the number of tasks, task scheduling, and data distribution granularity. Efficiency is concerned with the effective utilisation of available resources, which involves minimising

communication and synchronisation overheads, as well as achieving good load balancing. Simplicity refers to how easy it is to maintain and debug the resulting design. Platform suitability looks at how well the design maps to the number of available processors or execution units, how data is shared between processors (shared memory or distributed memory), and the types and speeds of the communication and synchronisation characteristics of the platform [35, 102].

4.5.2 Algorithm Structure

The *Algorithm Structure* design space takes the design from the Finding Concurrency space, which includes the tasks, data, groups, and dependencies, and refines it to create a framework for developing the parallel algorithm. The algorithm design process follows a simple decision tree, which is based on the primary features of the problem, resulting in one of six algorithm design patterns. In addition, the efficiency, simplicity, portability, and scalability of the design must be considered and an appropriate balance must be struck between these conflicting design considerations. Other considerations that must be taken into account include the target hardware and software platforms, as well as the programming environment [102, 103].

Concurrency usually falls into one of three primary organisations: organisation by tasks, organisation by data, and organisation by data flow. However, some algorithm designs may combine aspects of more than one organisation, resulting in a design that combines the different algorithm structures in some or other manner. The decision tree starts with the selection of an appropriate organisation branch. *Organise by tasks* is appropriate when the tasks themselves are the dominant feature of the problem decomposition. The *Task Parallelism* pattern is used if the tasks can be gathered into a linear set and the *Divide and Conquer* pattern is used if the problem is solved through recursive division of the problem into subproblems. The *organise by data decomposition* branch is selected if the data or data structures are the dominant factor in the problem. Linear decomposition of data in one or more dimensions results in the *Geometric Decomposition* pattern, whereas a recursive data decomposition, involving recursive data structures, favours the *Recursive Data* pattern. Finally, the *organise by data flow* branch should be chosen if the data flow between tasks imposes an ordering constraint on the groups of tasks. A regular, static ordering follows the *Pipeline* pattern, whereas, the *Event-Based Coordination* pattern is appropriate for an irregular, dynamic, and unpredictable flow of data [102].

- The **Task Parallelism** pattern is concerned with the development of *task parallel* algorithms that are based on the tasks themselves. The tasks are typically associated with the

iterations of a loop and may be known from the outset or they may arise dynamically during execution. It is important to consider whether the tasks are completely independent of each other and if the problem can be solved without completing all the tasks. Tasks should be defined such that there are at least as many of them as there are processors and each task should be sufficiently compute-intensive. Tasks need to be carefully structured to minimise overheads associated with managing dependencies such as ordering constraints and shared data. Problems without dependencies between tasks are usually referred to as *embarrassingly parallel* problems. However, it is sometimes possible to remove or separate dependencies through code modifications, such as using thread-local temporary variables and creating local copies of data that can be operated on independently and then combined with the results of the other tasks. An example of this is the *reduction* operation [102].

The assignment of tasks to execution units, known as scheduling, is another key factor in algorithm design as it is responsible for effective load balancing of tasks. *Static scheduling* uses a fixed allocation scheme where tasks are grouped into blocks according to their execution time and assigned to execution units. It is a fairly low overhead approach, but it can lead to load imbalance if the execution times of tasks vary significantly or if the capabilities and current workloads of the processors vary. In such cases, *dynamic scheduling* is preferred. Dynamic scheduling assigns tasks to a task queue, which is then emptied by the execution units as they complete tasks and acquire new ones from the queue. *Work stealing* is a variation on this that involves idle execution units stealing uncompleted tasks from busy execution units [35, 102, 125].

- The **Divide and Conquer** pattern solves recursive problems by recursively splitting the problem into smaller subproblems and merging the separate results back into the final solution. This approach generates good potential concurrency since the subproblems are solved independently. However, at some point, the overheads of splitting and merging the subproblems may overcome the benefits, so it is advisable to switch over to a sequential base case at an appropriate subproblem size. Divide and conquer is usually implemented using a simple fork/join approach that forks off new tasks and joins them on completion. Since the tasks are created dynamically and may not have equal workloads, a dynamic load balancing approach using the master/worker strategy may be advisable. Dependencies are uncommon in divide and conquer algorithms, but in certain cases, access to shared data must be managed appropriately [35, 102].

- The **Geometric Decomposition** pattern is concerned with how the algorithm can be designed to work with a data structure that has been decomposed into independent blocks

or chunks. For linear data structures, this usually involves decomposing the structure into contiguous subregions. Tasks are structured to perform operations on these subregions or chunks, taking into account any data that is required from neighbouring chunks through data sharing. It is also important to ensure that the data needed to update a particular chunk is available when required. The granularity of the data decomposition has an effect on the scalability and efficiency of the program. Fine-grained decomposition results in a large number of smaller chunks, improving scalability while increasing overheads. Inversely, coarse-grained decomposition creates fewer, larger chunks, thereby reducing overheads at the expense of scalability. The shape of the chunks can also have an effect on performance due to memory access factors. Another key factor in this pattern is the structuring of tasks and data accesses to ensure that non-local data is available before it is required, reducing expensive execution stalls [35, 102].

- The **Recursive Data** pattern structures the tasks to expose parallelism in the recursive structure of the data. This may involve looking at the structure in different ways to identify possible transformations on inherently sequential traversals that will allow for the elements to be operated on concurrently without affecting correctness, which can be a difficult task. Possible decompositions include decomposing the structure into individual elements that are assigned to different threads, structural decompositions that involve a loop where iterations perform operations on all elements or specific elements simultaneously, and synchronisation-based decompositions that involve simultaneously updating all elements (SIMD is an example of this) [35, 102].

- The **Pipeline** pattern describes an approach to performing concurrent computations on a stream of data by forwarding segments of the stream through a number of independent stages. Each stage performs a specific computation and these stages are capable of parallel execution using a number of threads, thereby allowing the different stages to perform operations on different segments of the stream simultaneously. This kind of processing is analogous to a manufacturing assembly line and common computing examples include the processor's instruction pipeline, vector processing, signal processing, and computer graphics. This approach is suited to tasks that have a strong, regular ordering that is amenable to pipelined execution. It is implemented by assigning each task or stage to a thread and routing the output from earlier stages to the inputs of later stages. So, while one stage is busy processing a segment of data, the previous stage is processing the next segment of data in the stream. The stages in the pipeline can be structured to form both linear and non-linear data flows through the pipeline. If the computational intensity of the stages differs, it may be necessary to group stages together on a processor, or further decompose the computationally intensive stages. Factors to consider when evaluating the

performance of the implementation include the throughput and latency of the pipeline [35, 102, 124, 125].

- The **Event-Based Coordination** pattern is used for problems where groups of semi-independent tasks interact in an irregular or unpredictable way. As with the pipeline pattern, the data flow dictates the ordering of tasks. This problem can be solved by defining the data flow in terms of *events* that are generated by a particular task and processed by another task. Therefore, the events define the ordering constraints and the data that is to be processed by the tasks. Firstly, the tasks are defined according to the processing requirements of the events. Then, the event flow between tasks is defined using an appropriate communication mechanism such as message-passing or shared queues between linked tasks. Factors such as deadlocks, effective task scheduling, and efficient communication all need to be dealt with to ensure that the implementation is both correct and efficient [102].

4.5.3 Supporting Structures

The next stage in the process is the *Supporting Structures* design space, which addresses the high-level constructs used to map the algorithms to the source code level. These constructs include *loop parallelism*, *boss/worker*, *single program multiple data (SPMD)*, *fork/join*, *shared data*, *shared queue*, and *distributed array*. These constructs are used to convert the high-level design into structures that can be implemented in source code. However, these patterns do not represent exclusive program structures as they can be combined or implemented in terms of each other. Factors such as the clarity of abstraction, scalability, efficiency, maintainability and alignment with the target environment are all important considerations for the design of the supporting structures. Certain structures are better suited to the constructs provided by certain parallel environments, so this must be taken into account [17, 35, 102, 124].

- The **Loop Parallelism** pattern addresses the parallelisation of computationally intensive loops where the iterations can be executed in parallel. This pattern structures the loop execution such that it generates tasks for parallel computation. It is often used for incremental parallelism to improve sequential programs by parallelising intensive loops. This is achieved by identifying program hotspots and bottlenecks centered around loops, eliminating or managing loop-carried dependencies, parallelising the loops, and optimising the scheduling of iterations. However, cache effects must be taken into account when performing loop parallelisation as the memory access patterns can have a significant effect

on performance. Loop parallelism is suited to task parallelism and geometric decomposition, but it is unable to represent the other approaches effectively [17, 35, 102, 124].

- The **Master/Worker** pattern involves a master thread that maintains a pool of worker threads and a collection of tasks. The worker threads repeatedly request tasks from the master thread and execute these tasks until all the tasks have been completed. This approach is particularly good for dynamic load balancing of unpredictable tasks. It is usually implemented by initialising the problem and creating a collection of tasks that are typically stored in a shared queue structure according to the Shared Queue pattern. A set of worker threads is spawned, each of which enters a loop that checks for the availability of tasks in the shared task queue and terminates if no more tasks are available. The results from the separate computations are then combined. Master/worker is a good approach to task parallelism, but it is generally not suitable for the other approaches [17, 35, 102, 124].

- The **Fork/Join** pattern is used when one thread must split off or *fork* computations to other threads and then *join* the child threads when they have completed their section of the computation. This pattern is usually applicable to problems with dynamically created tasks that cannot be structured using simple parallel loops, such as recursive tasks structures and irregular sets of connected tasks. This approach can be implemented as a fixed pool of forked threads that perform computations on tasks taken from a task queue, and terminate after all parallel computations have completed. A simpler approach is also possible. In this approach, threads are forked to perform specific tasks and are joined at the end of the parallel region. However, there are overheads associated with frequently forking and joining threads, so the more complex approach may be required if the overheads are unsatisfactory. Fork/join is particularly appropriate for divide and conquer, pipeline, and event-based coordination algorithms [17, 35, 102, 124].

- The **SPMD** pattern addresses the situation where multiple threads all execute the same instructions, function, or program, but on different sets of data. It is usually implemented by initialising multiple threads, each with their own identifier; running the same program or function on each thread while differentiating their execution based on their identifier; distributing the data either through local replication or segmentation of global data; and finalising the execution by combining results and cleaning up shared state. SPMD is well suited to task parallelism and data decomposition, and to a lesser extent, divide and conquer and pipeline algorithms [17, 35, 102, 124].

- The **Shared Data** pattern addresses the problem of sharing data between multiple tasks, while maintaining correctness and performance. Its usage involves defining the set of operations that are performed on the shared data structure. Then, these operations are

encapsulated in a synchronisation construct that ensures correct, consistent updates and accesses to the shared data. If possible, it attempts to segregate access so that different tasks do not interfere with one another when using the shared data structure. It is also important to keep the size of critical sections to a minimum for performance reasons [102].

- The **Shared Queue** pattern represents a queue data type that allows for correct and safe usage by multiple concurrent threads. A shared queue can be implemented as a normal queue abstract data type, which is then modified according to the Shared Data pattern. Dequeue calls to the queue can be either blocking or non-blocking, depending on the requirements of the parallel algorithm. The simplest implementation locks the whole structure for exclusive access by one thread. However, this can limit parallel performance, so more efficient concurrency-control protocols may be required. An example of this involves making use of the non-interfering nature of the enqueue and dequeue operations to allow simultaneous access to the head and tail of the queue when the queue contains a sufficient number of tasks. Distributed shared queues can also help improve performance by taking the pressure off a single shared queue [17, 35, 102].

- The **Distributed Array** pattern is used for the decomposition of arrays with one or more dimensions into sub-arrays that can be distributed among threads. The primary challenge is to structure array access so that elements required by each thread are located nearby at the appropriate time during execution. The memory hierarchy must also be taken into account when distributing segments of the array. There are a number of common array distributions that can assist in this process. *One-dimensional block* distribution decomposes the array into contiguous segments that are assigned to tasks or threads. *Two-dimensional block* distribution assigns each task or a thread a rectangular sub-block from the array. *Cyclic* or *block-cyclic* distributions generate a number of blocks exceeding the thread count, which are then assigned in a round-robin fashion [17, 35, 102].

4.5.4 Implementation Mechanisms

The last stage of the design process is the *Implementation Mechanisms* design space, which deals with the language constructs and library routines for implementing parallelism in the program. It includes constructs and routines for thread and process management, scheduling adjustment, synchronisation, and communication. In this stage, the high-level concurrent design from the preceding stages is implemented in source code using the low-level operations provided by the parallel programming environment to produce a parallel program. As such, an appropriate parallel programming environment must be selected, including the language,

supporting libraries, and parallel API or library. One important consideration when selecting and using external libraries is the concept of *thread-safety*. A thread-safe library protects its shared components using synchronisation mechanisms, allowing the library to be used by concurrent tasks without the possibility of race conditions. If a library is not thread-safe, access to the library may need to be serialised to ensure correctness when executing multiple concurrent tasks [17, 102, 103, 125].

There are two primary approaches to implementing parallelism at the source level using threading libraries: *implicit threading* and *explicit threading*. With implicit threading, the threading library or API takes care of most of the details of thread management and synchronisation by providing higher-level abstractions that make it easier for the programmer to implement parallelism, while reducing the flexibility and expressiveness somewhat. Some examples of implicit threading are automatic parallelising compilers, OpenMP, Cilk, and Threading Building Blocks. Explicit threading, on the other hand, forces the programmer to implement and manage all aspects of thread creation and termination, scheduling and load balancing, synchronisation, and communication. This improves expressiveness and flexibility at the expense of increased difficulty, resulting in a greater possibility for errors. Examples of explicit threading libraries include Pthreads, Windows Threads, and Boost Threads. A number of parallel programming libraries are described in Chapter 3. Some parallel programming models or environments are better suited to implementing certain supporting structures than others, so an appropriate model must be selected based on the algorithm [17, 60, 124].

4.6 Parallel Optimisations

There are several performance issues that arise from the parallel execution of multithreaded programs on shared-memory multiprocessor computer systems, particularly when multiple physical CPUs are involved. As described in Section 2.4, processor cores are able to communicate through either shared on-chip caches or via the system bus or processor interconnect network. Data that is shared between processors or cores needs to be kept consistent and overall cache coherence must be maintained to ensure that multiple processors modifying the same data all see the correct values. The synchronisation and cache coherence mechanisms required to maintain a valid memory state create the potential for contention, negatively affecting performance and reducing potential parallelism if the contention is not managed carefully [153]. Additionally, good parallel performance and scaling can only be achieved if all of the available processors are kept sufficiently busy. Therefore, concurrent tasks must be distributed across the available processors according to their workload using load balancing techniques [17, 80, 124].

4.6.1 Data Sharing

While modern multicore CPUs with large shared data caches do allow for cheap sharing of data between threads running on different cores, sharing of frequently modified data should be kept to a minimum. As per the MESI cache coherence protocol implemented by most commodity multicore CPUs, when a thread attempts to access data that is in a modified state in the L1 cache of another processor core, the modified cache line must first be evicted and written back to main memory before reading it into the L1 cache of the requesting core. This introduces a performance penalty over reading the data straight from the L1 or L2 data cache and creates cache and bus contention if evictions occur frequently. Therefore, data sharing between processors should be kept to a minimum, particularly for data that is modified frequently. For large shared data structures, techniques such as blocking can improve locality and reduce contention for data in the same region of memory. Data sharing bottlenecks can also be dealt with at the design stage by devising implementations that partition shared data between threads so that minimal sharing occurs [69, 80, 153].

The above situation is typically referred to as *true sharing*, where the sharing of data is intentional. On the other hand, *false sharing* occurs when two or more separate pieces of data, modified by threads running on different cores, happen to share the same cache line. This causes frequent cache line evictions for each core as they issue request for ownership (RFO) messages for the cache line, even though the specific data being accessed has not been modified by the other threads. This can incur a significant performance penalty if the same cache line is updated frequently, particularly when multiple physical processors are involved [4, 21, 69, 80, 124, 125]. Data sharing and false sharing bottlenecks can also be identified through profiling by focusing on data-related performance events. Intel CPUs provide a number of useful events for identifying occurrences of false sharing. Code inspection can then be used to identify the specific data structures causing the false sharing [69, 120]. False sharing can be reduced by padding or aligning data structures and arrays to cache line boundaries. The use of *thread-local* copies of data can also help to reduce the chance of false sharing. It is also important to ensure that synchronisation variables are alone on a cache line to prevent false sharing as a result of frequently accessed locks and data sharing cache lines [4, 80].

4.6.2 Reduce Lock Contention

Contention over shared data has been discussed in Section 4.6.1, but partitioning of shared data is not always possible. In such cases, it necessary to implement mutual exclusion or other

synchronisation mechanisms to ensure that no race conditions arise as a result of simultaneous access to shared data. However, mutual exclusion serialises access to the critical section. If the critical section is short or if threads do not attempt to access the critical section frequently, the impact on performance may be minimal. However, if the critical section takes a significant amount of time to traverse and other threads are kept waiting, the impact on performance and scaling can be significant, especially if the critical section is in a performance hotspot [45, 69, 137].

Lock contention can be reduced by ensuring that critical sections are as short as possible, are specific to the data being protected, and do not encompass computational intensive sections of code that are safe to execute in parallel (fine-grained locking). A simple demonstration of this concept can be seen in Listings 4.13 and 4.14. However, mutexes and other synchronisation constructs are not without their own overheads. Excessive use of small critical sections can also reduce performance due to the overhead of checking and updating the state of the lock. If the overhead of using multiple critical sections exceeds the performance benefits of fine-grained locking, it may necessary to revert to coarse-grained locking [45, 69, 137]. Barriers are another type of synchronisation that should be avoided if possible. Barriers act as a synchronisation point for groups of threads, forcing threads to wait until all the threads in the group reach that point, limiting scalability. Unnecessary barriers, such as the implied barriers at the end of certain OpenMP constructs, should be removed or suppressed (using the **nowait** clause in OpenMP) to allow threads to continue with useful work [21].

```
1  function perform_work() {
2      lock(mutex1);
3
4      update_data(shared_data1);
5
6      other_work();
7
8      process(shared_data2);
9
10     compute();
11
12     unlock(mutex1);
13 }
```

Listing 4.13: Coarse-grained locking.

```
1  function perform_work() {
2      lock(mutex1);
3      update_data(shared_data1);
4      unlock(mutex1);
5
6      other_work();
7
8      lock(mutex2);
9      process(shared_data2);
10     unlock(mutex2);
11
12     compute();
13 }
```

Listing 4.14: Fine-grained locking.

Lock contention can also arise from suboptimal algorithm or lock type choices. A program with shared data that is frequently accessed by multiple readers and modified infrequently by a single writer will perform poorly if a spin mutex is used, as each reader must acquire an exclusive lock on the data, which is not always necessary because the data is only being read. In this case, a

reader-writer lock is more appropriate as multiple readers can access the data simultaneously until the writer thread attempts to modify the data. Wait-free or non-blocking algorithms may also be possible, reducing contention (cache contention may still exist) and improving scalability [56, 105, 137]. However, the synchronisation routines provided by the available threading libraries should be used instead of developing and using custom synchronisation routines, particularly if the target program is intended to be run on a range of different platforms. It is often difficult to implement correct, portable, and efficient lock routines for multiple platforms unless one is an expert in the field, so this task is best left to the parallel library developers [69].

4.6.3 Load Balancing

Efficient parallel performance is dependent on keeping all available processors busy performing useful work. Uneven workload distribution and resource contention leads to processors sitting idle. *Load balancing* attempts to minimise idle processor time by ensuring that work is distributed equally amongst the available processors and that there is no over- or under-subscription of worker threads. However, load balancing is dependent on a variety of program and system factors, including the number of processors, workload size, work distribution policy, and the nature of the problem [17, 21, 35, 69, 125].

For existing implementations, many of the listed factors have already been specified or determined, meaning that the algorithm or work distribution policy may need to be modified to achieve better load balancing. For new programs, it is important to design the algorithms and work distribution policies to ensure good load balancing and maximise processor utilisation. Parallel APIs and libraries such as OpenMP and Threading Building Blocks provide methods for controlling the distribution of work between tasks or threads. Static workloads, where each unit of work takes roughly the same amount of time, are fairly easy to load balance as each processor can be given an equal sized chunk of the workload. Static scheduling is the default for OpenMP, whereas Threading Building Blocks divides the workload into smaller chunks that are allocated to tasks. Uneven workloads are more difficult to schedule efficiently as the execution time varies between units of work. In such situations, dynamic or guided scheduling is more appropriate as smaller chunks of work are easier to load balance during execution. However, dynamic scheduling typically incurs slight performance overheads compared to static scheduling [17, 21, 69, 125].

For explicit threading using Pthreads or Boost Threads, all the workload scheduling and load balancing must be implemented by the programmer. Static scheduling is easy to implement as it merely requires splitting the workload between the available threads or processors. Dynamic

scheduling requires more thought. Task queues or master/worker style workload distribution can be reasonably effective if implemented efficiently, but scalability may be limited for larger processor counts owing to increased communication and synchronisation. Overlapping of I/O and computation must also be used wherever possible to hide the latency of I/O operations. OpenMP and Threading Building Blocks manage the creation and termination of threads, including how many threads are used, unless explicit thread counts are specified. For explicit threading, the number of threads used is controlled by programmer. Therefore, the programmer must be careful to avoid over-subscription, where the number of threads exceeds the number of logical processors, resulting in context switch overheads, and under-subscription, where the number of threads is lower than the processor count resulting in the under-utilisation of available resources [17, 21, 69, 125].

Chapter 5

Methodology

The first three research goals have already been addressed in Chapters 2, 3, and 4, leaving the fourth and final research goal for the remaining chapters of this thesis. Unlike the previous chapters, which focused on the relevant parallel programming literature, the fourth goal requires an empirical evaluation of selected parallel programming APIs and libraries. This evaluation takes a two-fold approach: an analysis of parallel programming models and compilers based on performance and an analysis of parallel programming models based on programmer effort for each of the sample problems. The research design for achieving this goal is presented below, together with a description of the system specifications and the use of an automated benchmarking tool for collecting performance data.

5.1 Research Design

The aims of this research require that various parallel programming models, tools, and optimisations be evaluated to gauge their effectiveness at improving software performance on multiprocessor computer systems. Accordingly, the investigative technique is that of a formal experiment, in which a controlled investigation of the impact of the different parallel models and compilers is undertaken.

For the performance analysis, the independent variables are the parallel programming model and the compiler, whereas, the second experiment only has one independent variable, the parallel programming model. In both cases, the experimental objects are the optimised sequential

programs and their related source code, which include the classic matrix multiplication algorithm, the Mandelbrot set algorithm, and a deduplication kernel developed by Princeton University. The treatments, represented by the independent variables, that are applied to each experimental object include the Intel Composer XE 2011 and GNU Compiler Collection compilers, and the OpenMP, Threading Building Blocks, automatic parallelisation, Cilk Plus, Pthreads, and Boost Threads parallel programming models. In both cases, the experimental subject is the author, who represents a novice-level parallel programmer. The response variables or performance metrics for the investigation are the wall clock timing (includes initialisation and finalisation routines) and parallel speedup for the execution of the parallel program or experimental object. The response variables or code metrics for the second investigation are the number of modified lines of code, Halstead's volume and length metrics, and the cyclomatic code complexity. The experiments follow a *crossed* design where each treatment factor is tested in conjunction with every other treatment factor. Thereafter, the ratio of speedup to additional programming effort for each of the parallel programming models and compilers is calculated and used to provide recommendations regarding the selection of parallel programming model.

The approach taken to optimising and parallelising each program follows the methodologies and techniques described in Chapter 4. First, the sequential programs are optimised using the iterative performance tuning approach. Exploitable parallelism is identified and decomposed to produce a suitable parallel design that is then implemented using each of the parallel programming models. All code is placed under version control using Subversion to allow for changes to be reverted and different versions of the code to be compared. Since the cognitive process of exposing parallelism is common to all the implementations of the same program, the programmer effort is constrained to the task of actually implementing the design using the specific parallel programming models. As such, the basic attribute of code size, measured using lines of code and Halstead's volume metric, can provide a sufficient estimation of programmer effort.

5.2 System Specifications and Software Versions

All profiling, debugging, and performance measurements are performed on a computer system with the hardware specifications given in Table 5.1, running the versions of software listed in Table 5.2. To ensure that the testing platform remains as consistent as possible, all power saving features are disabled and CPU frequency scaling is set to maximum performance for each processor core. The Intel Turbo Burst feature is also disabled for the CPU to prevent certain cores from attaining higher clock speeds as a result of minimal load on the other cores. Final benchmarking runs are performed with the bare minimum set of programs running on the

system to ensure that the performance results are not influenced by the execution of external programs. This means that the X Windowing system and any other unnecessary services are terminated.

Table 5.1: System hardware specifications.

Component	Description
CPU model	Intel Core i5-750
Number of cores	4
Core frequency	2.67GHz
L1 data cache size (per core)	32KB
L1 instruction cache size (per core)	32KB
L2 unified cache size (per core)	256KB
L3 unified cache size (shared)	8192KB
Set-associativity	L1d: 8-way, L1i: 4-way, L2: 8-way, L3: 16-way
RAM	4GB TEAM Xtreem DDR3 1333MHz
RAM modules	2 x 2GB
RAM timings (tCL-tRCD-tRP-tRAS)	7-7-7-18
Cache line size	64B
Motherboard	MSI P55-GD65
Hard disk	Western Digital Black 500GB SATA2

Table 5.2: System software specifications.

Software	Version
Operating System	Fedora Core 13
Kernel	Linux 2.6.34.7-66.fc13.x86_64
GNU Compiler Collection	4.5.1
Intel C/C++ Compiler	Intel Composer XE 2011 (12.0.0.084)
Glibc	2.12.2-1
OpenMP	3.0 for both GCC and Intel C/C++
Threading Building Blocks	3.0
Boost Threads	1.41.0-11
Intel MKL	10.3
Intel IPP	7.0.1

5.3 Automated Benchmarking Tool

Owing to the repetitive nature of testing and benchmarking changes, as well as the number of different test cases, an automated benchmarking tool is used. This ensures that the benchmarking process remains consistent and is less tedious.

Instead of developing an automated benchmarking tool from scratch, an existing tool was found and subsequently modified. The Computer Language Benchmarks Game [46] is a website that compares language implementations using a variety of benchmark programs. To automate the process of benchmarking dozens of language implementations for each test program, the author of the site, Isaac Gouy, developed a benchmark tool written in Python [46]. The source code for the benchmark tool is available under the 3-clause BSD license, thus allowing it to be downloaded and freely modified to meet our additional requirements related to result logging.

The tool repeatedly (given a user-specified repetition count) executes and measures the benchmark program's CPU time, elapsed time (wall clock time), resident memory usage, and CPU load while the program is running and generates raw measurement data files in the comma-separated values (CSV) file format. This data can then be analysed by the user to compare the performance of different program implementations and compiler optimisation options. It also generates basic result summaries for each program implementation by averaging the results over each run for a particular program input value. These summaries can be used to quickly evaluate the effect of changes without having to perform a full data analysis. To ensure that each program implementation executes correctly, the resulting output of each program run is compared with other implementations. Any inconsistencies are reported to the user and the offending execution runs are flagged as invalid in the measurement data files [46].

5.3.1 Extensions to the Benchmarking Tool

Better result and run information logging has been implemented, such that instead of overwriting previous results and run logs, the measurements and logs for each run are stored according to the date and time of the run. Additional information is also recorded for each run, such as the environment and build options used for the run. All the files for the benchmark tool and the test programs are added to a Subversion (SVN) repository to allow for version control of the benchmark suite and its associated configuration files. The SVN revision information is then recorded along with the logs for each run. This provides a number of useful features, such as the ability to revert to a specific set of changes to replicate a test run and compare source code files to identify code that has changed between runs. The tool has also been configured to allow for the testing of different compiler options using the same compiler.

Chapter 6

Implementation

6.1 Introduction

This chapter describes the actual steps taken during the process of profiling, analysing, optimising, parallelising, and testing the three target programs. Three progressively more complex programs are used as the basis of our investigation: a classic matrix multiplication example (Section 6.2) for demonstration purposes, a Mandelbrot Set algorithm (Section 6.3) to highlight load balancing issues, and a deduplication kernel (Section 6.4) as an example of pipeline parallelism. For each program, we perform an initial analysis of the baseline performance data, apply any relevant sequential optimisations, and measure the performance and profiling data again to ensure that the optimisations have had the desired effect.

Next, we decompose the problem for parallel execution and formulate an appropriate parallel design. The design is then implemented using each of the selected parallel programming models, which include automatic parallelisation, Boost Threads, Cilk Plus, OpenMP, Pthreads, and Threading Building Blocks, where applicable. The parallel implementations are checked for threading and memory errors and the output is compared to the original sequential version. Profiling is also performed to identify any threading-related performance issues. Once the programs have been optimised sufficiently, a final benchmarking run is performed and the code metrics are measured.

For each of the parallel implementations, we have defined the number of threads to be adjustable at compile-time by specifying `-DNTHREADS=`<*n*> as a compiler preprocessor option, where *n* is the desired number of threads (four threads is the default if the option is not specified).

6.2 Matrix Multiplication

Matrix multiplication is used extensively in scientific computing, particularly for applications that must solve linear algebra problems. As the matrix multiplication algorithm is very amenable to parallel execution, as we shall see, it provides a simple test case for demonstrating the various parallel programming models. The matrix multiplication operation is defined in (6.1), where two square matrices, A and B, are multiplied together and stored in matrix Res. The subscripts represent the row and column indices of the element being addressed in that particular matrix, while dim is the dimension of the matrices [102]. The relevant sections of the baseline sequential program code, called **matrixmul** are given in Listing 6.1, while the full program source code can be found in Appendix B.1.

$$Res_{r,c} = \sum_{k=0}^{dim-1} A_{r,k} \cdot B_{k,c} \tag{6.1}$$

```
1  double** matrix_a;      // matrix a
2  double** matrix_b;      // matrix b
3  double** matrix_res;    // result matrix
4
5  void mul_matrices(void) {
6      // multiply matrix a and matrix b and store in result matrix
7      for (int r = 0; r < dimension; r++) {
8          for (int c = 0; c < dimension; c++) {
9              double sum = 0.0;
10             for (int k = 0; k < dimension; k++) {
11                 sum = sum + matrix_a[r][k] * matrix_b[k][c];
12             }
13             matrix_res[r][c] = sum;
14         }
15     }
16 }
17
18 int main(int argc, char** argv) {
19     ...
20     init_matrices();
21     mul_matrices();
22     ...
23 }
```

Listing 6.1: Classic matrix multiplication algorithm (**matrixmul**).

6.2.1 Initial Profiling and Analysis

The first step in the performance tuning process is to measure the baseline performance of the matrix multiplication program. The **matrixmul** program was compiled using GCC with `-O0`

`-g` as compiler options and the timing measurements were repeated five times with *dimension* equal to 2048. The initial timings are summarised in Table 6.1. The program was profiled using Valgrind with the Cachegrind tool and Intel VTune, as well as being checked for memory errors using Intel Inspector and Valgrind's Memcheck tool. However, no memory access errors or leaks were detected by either tool. VTune reported an L3 cache miss rate of 0.531%, execution stall rate of 0.39%, and a high cycles per instruction (CPI) measurement of 1.221. Cachegrind confirmed these results and indicated an L1 data cache miss rate of 8.2%. These results indicate the need for optimisations to improve the instruction and data access characteristics of the program. As expected, the vast majority of the execution time and memory accesses are linked to the innermost loop statement.

Table 6.1: Runtime performance of the original, unoptimised sequential **matrixmul** program.

Mean Wall clock time	Geometric Mean	Standard Deviation
161.0396s	161.0388s	0.5706

6.2.2 Sequential Optimisations

The first attempt to improve the performance of the program centers around the use of compiler optimisation options (Table 6.2), which do not require any programming effort and can often provide good speedup. In all cases, speedup is measured against the original sequential program results listed in Table 6.1.

Table 6.2: Performance of **matrixmul** with sequential optimisations.

	Wall clock time (seconds)					
Compiler	g++ -O1	g++ -O2	g++ -O3	icpc -O0	icpc -O2	icpc -O3
Mean	131.8142	130.6012	130.8114	159.3842	130.2306	130.0406
Geo. Mean	131.8095	130.5957	130.803	159.3821	130.2284	130.0390
Std. Dev.	1.240314	1.336184	1.650024	0.9266608	0.85369	1.773753
Speedup	1.222x	1.233x	1.231x	1.01x	1.236x	1.238x

Using the Intel C/C++ compiler with the `-opt-report` option reveals that loop interchange was not performed due to an imperfect loop nest and that the innermost loop was not vectorised as the compiler deemed it to be inefficient. However, loop unrolling was performed on the innermost loop, so this optimisation should not be applied manually. Since the compiler was unable to perform the loop interchange, the loops have been restructured manually to remove the temporary sum and improve the access pattern as shown in Listing 6.2. The resulting timings are presented in Table 6.3.

```
1  void mul_matrices(void) {
2      // multiply matrix a and matrix b and store in result matrix
3      for (int r = 0; r < dimension; r++) {
4          for (int c = 0; c < dimension; c++) {
5              for (int k = 0; k < dimension; k++) {
6                  matrix_res[r][k] = matrix_res[r][k] + matrix_a[r][c] * matrix_b[c][k];
7              }
8          }
9      }
10 }
```

Listing 6.2: Matrixmul with improved array access pattern.

Table 6.3: Performance of **matrixmul** with improved array access.

	Wall clock time (seconds)			
Compiler	g++ -O0	g++ -O3	icpc -O0	icpc -O3
Mean	79.4798	14.5112	105.938	41.138
Geo. Mean	79.47977	14.5119	105.938	41.138
Std. Dev.	0.07340095	0.1531992	0.05404165	0.01781853
Speedup	2.026x	11.1x	1.52x	3.914x

The correctness of the resulting implementation is tested by outputting and comparing the full result matrix against the output of the original program. This introduces significant I/O overhead, which obscures the results of the optimisations. Therefore, the full check is only performed once for each optimisation or change. Thereafter, the check is substituted with a statement that only outputs a single element of the result matrix to prevent aggressive compiler optimisations from eliminating the matrix calculations entirely (because the result is unused).

Profiling shows that the L1 data cache miss rate has been reduced to 0.5% from 8.2%, but the number of data references increased by 50%. Fortunately, this increase in references does not impact peformance as the vast majority of these references can be served from the cache. The Intel compiler appears to have poor performance compared to GCC, however, the situation changes when interprocedural optimizations (IPO) (icpc: -ipo, g++: -fwhole-program) are applied (see Table 6.4). This seems to indicate that GCC performs more aggressive IPO than the Intel compiler at optimisation level `-O3`.

Table 6.4: Performance of **matrixmul** with interprocedural optimisations.

	Wall clock time (seconds)	
Compiler	g++ -O3 w/ IPO	icpc -O3 w/ IPO
Mean	14.5128	8.6408
Geo. Mean	14.51279	8.640792
Std. Dev.	0.01400714	0.01329286
Speedup	11.096x	18.637x

These compiler optimisations provide a vast improvement in performance, with the Intel compiler giving an 18.637x speedup over the original sequential version. Profiling of the IPO optimised programs shows that the L1 data miss rate has increased to 8.1%. But, this is coupled with a decrease in the instruction reference count to 4.82% and a decrease in the data reference count to 6.75% compared with the previous version, which explains the improvement in performance. The data cache miss count is the same as in the previous version, indicating that these are most likely compulsory misses. Loop tiling was attempted, but it severly degraded the performance when compiling with the Intel compiler and only provided a minor performance increase with GCC. Therefore, the change was reverted and the sequential code from Listing 6.2 was used for parallelisation.

Profile-guided optimisation (PGO) is another compiler-based optimisation that can improve program performance by tuning the optimisation process according to the runtime profile generated by executing the instrumented version of the program binary. Results show that PGO was able to improve the performance of the GCC-compiled program by two seconds, but there was no improvement with the Intel compiler, and the use of PGO on the parallel implementations either had no effect or degraded performance for both compilers. Therefore, it was decided that PGO would not be used in further testing of the **matrixmul** program.

6.2.3 Parallel Implementations

Matrix multiplication is an example of an embarrassingly parallel problem where the parallel decomposition is fairly straightforward and efficient. As such, we will not discuss the parallel design in too much detail. The problem can be organised as tasks, where a task is represented by the loop body of the outermost loop. This leads to the Task Parallelism pattern supported by the loop parallelism structure. The data decomposition uses a simple Geometric Decomposition that divides the data according to the rows of `matrix_a` and `matrix_res`. There are no ordering constraints and no data sharing constraints since the two input matrices are only accessed with read operations and write access to the result matrix is partitioned by row, which means that shared access to the result matrix is considered effectively-local. Since the workload is the same for each row, load balancing can be achieved using a simple static scheduling approach that distributes the data in equal size groups of contiguous rows. The parallel implementations for each of the parallel programming models are presented below. The speedups listed are relative to the best sequential performance for that specific compiler. All further implementations are compiled with IPO and -O3 compiler optimisation flags enabled.

Automatic Parallelisation

The successful automatic parallelisation of a program by the compiler is a very desirable outcome as it does not require the programmer to formulate and implement a parallel design and, therefore, requires very little effort. However, the automatic parallelisation capabilities in GCC and the Intel compiler are limited to simple loop structures [69]. GCC was unable to generate a program that produced any speedup over the original version, whereas the Intel compiler provided only a marginal benefit. Following the advice provided by the Intel compiler's diagnostic reports yielded no improvement. In some cases, the suggested compiler directives used to assist auto-parallelisation interfered with the process, resulting in worse speedup than without the directives. Although not mentioned previously, automatic vectorisation was attempted for the baseline sequential program, however, neither compiler was able to perform effective vectorisation. Compiler diagnostics reveal that the inner loop was identified as a target for vectorisation, but that the compiler (icpc in this case) deemed the vectorised version to be too inefficient to warrant performing the optimisation. Unlike the sequential version, the automatically parallelised program (as well as the other parallel implementations) benefits greatly from automatic vectorisation. The performance summary is presented in Table 6.5, where "auto-parallel" refers to the automatically parallelised program and " + SSE4.2" refers to the inclusion of automatic vectorisation using the SSE4.2 extended instruction set.

Table 6.5: Performance of **matrixmul** with automatic parallelisation and vectorisation.

	Wall clock time (seconds)	
Compiler	icpc auto-parallel	icpc auto-parallel + SSE4.2
Mean	6.7788	2.4858
Geo. Mean	6.7788	2.485796
Std. Dev.	0.03453549	0.004816638
Speedup	1.275x	3.476x

OpenMP

The OpenMP implementation (OMP) uses a straightforward **parallel for** construct with static scheduling as shown in Listing 6.3.

Performance testing reveals a significant speedup for both compilers (Table 6.6). However, the Intel compiler appears to produce a more efficient parallel program, achieving a linear speedup with the number of processor cores.

```
1 void mul_matrices(void) {
2     // multiply matrix a and matrix b and store in result matrix
3     #pragma omp parallel for num_threads(NTHREADS) default(none) \
4                 shared(matrix_a, matrix_b, matrix_res, dimension)
5     for (int r = 0; r < dimension; r++)
6         for (int c = 0; c < dimension; c++)
7             for (int k = 0; k < dimension; k++)
8                 matrix_res[r][k] = matrix_res[r][k] + matrix_a[r][c] * matrix_b[c][k];
9 }
```

Listing 6.3: Parallel matrix multiplication using OpenMP.

Table 6.6: Performance of OpenMP **matrixmul** with and without automatic vectorisation.

	Wall clock time (seconds)			
Compiler	g++ OMP	g++ OMP + SSE4.2	icpc OMP	icpc OMP + SSE4.2
Mean	4.1328	3.9666	2.1316	0.9346
Geo. Mean	4.132594	3.966441	2.1316	0.934591
Std. Dev.	0.04616492	0.03963963	0.0005477226	0.004615192
Speedup	3.511x	3.66x	4.054x	9.25x

The program output was compared against the sequential version and found to be correct. The program was then analysed for memory and threading errors using Valgrind and Intel Inspector XE. Intel Inspector found no problems with memory access, which was confirmed by Valgrind's Memcheck tool. No threading errors were identified by Intel Inspector XE, whereas Valgrind's Helgrind tool identified 545 data race errors. However, these can be ignored as an inspection of the code reveals that they are false positives resulting from the unprotected, yet safe accesses to the shared matrices. Profiling of the OpenMP implementation revealed a high level of concurrency with minimal waiting times, which is evident from the linear speedup achieved by the icpc-compiled program.

Thread Building Blocks

For the Threading Building Blocks implementation (TBB), the `mul_matrices` function has been replaced with an inline lambda function within the program's main function. Lambda function support is only available in compilers that implement parts of the C++0x standard. Both GCC and the Intel C++ compiler support lambda functions and this functionality is enabled when the `-std=c++0x` compiler option is specified. The TBB `parallel_for` template is used to parallelise the outer loop and the grain size is managed automatically by the runtime system using `auto_partitioner()`, as shown in Listing 6.4.

```
#include "tbb/task_scheduler_init.h"
#include "tbb/blocked_range.h"
#include "tbb/parallel_for.h"

int main(int argc, char** argv) {
    ...
    task_scheduler_init init(NTHREADS);

    init_matrices();
    parallel_for( blocked_range<int>(0, dimension),
       [](const blocked_range<int>& range){
            // multiply matrix a and matrix b and store in result matrix
            for (int r = range.begin(); r != range.end(); r++)
                for (int c = 0; c < dimension; c++)
                    for (int k = 0; k < dimension; k++)
                        matrix_res[r][k] = matrix_res[r][k] + matrix_a[r][c] * matrix_b[c][k];
        }, auto_partitioner() );

    ...
}
```

Listing 6.4: Parallel matrix multiplication using Thread Building Blocks.

Table 6.7: Performance of TBB **matrixmul** with and without automatic vectorisation.

	Wall clock time (seconds)			
Compiler	g++ TBB	g++ TBB + SSE4.2	icpc TBB	icpc TBB + SSE4.2
Mean	4.1468	3.995	3.4854	3.422
Geo. Mean	4.146472	3.994981	3.485086	3.421326
Std. Dev.	0.05797586	0.01360147	0.05248619	0.07638063
Speedup	3.5x	3.632x	2.479x	2.53x

The TBB library appears to be the limiting factor for performance when compiling with the Intel compiler. The speedup for the GCC-compiled executable differs only marginally compared to the OpenMP implementation. Both Memcheck and Intel Inspector XE detected memory leaks in the TBB library, but these are beyond the scope of the application itself. As with OpenMP, Intel Inspector does not find any threading errors, while Helgrind produces false postives for data race issues. Profiling reveals that the TBB parallel implementation has good thread concurrency and near ideal CPU usage despite the lackluster speedup for the icpc-compiled program, while memory and instruction access characteristics are mostly unchanged from the sequential version.

Pthreads

The Pthreads implementation requires more extensive code modifications than the previous parallel programming models. Within the program's main function, a pool of threads is defined and spawned, passing the thread's identification number as an argument to the thread function.

The main thread waits until the spawned threads are joined before proceeding. The thread function, `mul_matrices`, calculates the chunk size and loop bounds based on the number of threads and the thread identifier, as seen in Listing 6.5.

```
#include <pthread.h>

void* mul_matrices(void* thread_id) {
    // calculate loop bounds from the thread id
    const long r_block = (long)ceil((double)dimension / NTHREADS);
    const long r_lower = (long)thread_id * r_block;
    const long r_upper = std::min(((long)thread_id + 1) * r_block, (long)dimension);

    // multiply matrix a and matrix b and store in result matrix
    for (int r = r_lower; r < r_upper; r++)
        for (int c = 0; c < dimension; c++)
            for (int k = 0; k < dimension; k++)
                matrix_res[r][k] = matrix_res[r][k] + matrix_a[r][c] * matrix_b[c][k];

    return (NULL);
}
int main(int argc, char** argv) {
    ...
    pthread_t thread_id[NTHREADS];
    init_matrices();

    for (int i = 0; i < NTHREADS; i++)
        pthread_create( &thread_id[i], NULL, mul_matrices, (void*)i );
    for (int i = 0; i < NTHREADS; i++)
        pthread_join( thread_id[i], NULL );
    ...
}
```

Listing 6.5: Parallel matrix multiplication using Pthreads.

Table 6.8: Performance of Pthreads **matrixmul** with and without automatic vectorisation.

	Wall clock time (seconds)			
Compiler	g++ Pthread	g++ Pthread + SSE4.2	icpc Pthread	icpc Pthread + SSE4.2
Mean	4.187	3.968	1.9412	0.7666
Geo. Mean	4.186804	3.967959	1.940252	0.7665936
Std. Dev.	0.04539273	0.02014944	0.06882732	0.003507136
Speedup	3.465x	3.657x	4.453x	11.272x

This simple Pthreads implementation provides exceptional performance as evidenced by the superlinear speedup of the icpc-compiled executable (Table 6.8). This superlinear speedup can be explained by the memory hierarchy of modern CMPs, which afford multicore CPUs greater cache space than singlecore CPUs [21, 35]. Error checking reveals that there are no memory or threading related errors besides the false positives reported by Helgrind. Profiling indicates that the program has good concurrency and CPU usage and further attempts to improve performace merely add overheads that outweigh the benefits of the more sophisticated load balancing.

Boost Threads

The Boost Threads implementation follows much the same structure as the Pthreads version, except that the blocks and loop bounds are calculated in the program's main function as opposed to in the function object. The function object that performs the matrix multiplication is defined as a `struct` with an overloaded `operator()` function, as shown below in Listing 6.6.

```
#include <boost/thread.hpp>

void mul_matrices(const int r_lower, const int r_upper) {
    // multiply matrix a and matrix b and store in result matrix
    for (int r = r_lower; r < r_upper; r++)
        for (int c = 0; c < dimension; c++)
            for (int k = 0; k < dimension; k++)
                matrix_res[r][k] = matrix_res[r][k] + matrix_a[r][c] * matrix_b[c][k];
}
int main(int argc, char** argv) {
    ...
    boost::thread_group threads;
    int r_lower = 0;
    const int r_block = (int)ceil((double)dimension / NTHREADS);
    init_matrices();

    for (int t = 0; t < NTHREADS-1; t++) {
        int r_upper = r_lower + r_block;
        threads.add_thread(new boost::thread(mul_matrices, r_lower, r_upper));
        r_lower = r_upper;
    }
    mul_matrices(r_lower, dimension);
    threads.join_all();
    ...
}
```

Listing 6.6: Parallel matrix multiplication using Boost Threads.

Table 6.9: Performance of Boost Threads **matrixmul** with and without automatic vectorisation.

	Wall clock time (seconds)			
Compiler	g++ Boost	g++ Boost + SSE4.2	icpc Boost	icpc Boost + SSE4.2
Mean	4.1588	4.0204	3.4946	3.3778
Geo. Mean	4.158518	4.020274	3.494116	3.377628
Std. Dev.	0.05440772	0.03564828	0.06480972	0.03819293
Speedup	3.49x	3.61x	2.472x	2.558x

The Boost Threads implementation suffers from the same performance limiting factors that affected the TBB implementation using the Intel compiler. A dynamic load balancing version using the Master/Work approach yields little additional speedup, so it appears that the static load balancing is not at fault. Again, error checking reveals that there are no memory or threading related errors besides the false positives reported by Helgrind. Profiling information shows that the Boost Threads implementation exhibits similar characteristics to the TBB implemen-

tation, with near ideal concurrency and good CPU usage. This conflicts with the measured speedup attained using the Intel compiler (Table 6.9), indicating that the performance issue is not within the program itself, but most likely stems from the compiler and optimisation flags used to compile the Boost library.

Cilk Plus

The Cilk Plus implementation is very straightforward, only requiring the outermost **for** statement to be replaced with **cilk_for**. Finer task granularity can be achieved by changing the inner loops to use **cilk_for**. However, the overheads of the finer granularity result in descreased performance, so the implementation found in Listing 6.7 is more appropriate. A Divide and Conquer approach is also possible using the **cilk_spawn** and **cilk_sync** keywords. This would, however, require more extensive code modifications and the performance of the current implementation is already very good. As such this alternative approach is excluded from consideration. Unfortunately, Cilk Plus is currently only supported by the Intel compiler.

```
1 #include <cilk/cilk.h>
2
3 void mul_matrices(void) {
4     cilk_for (int r = 0; r < dimension; r++)
5         for (int c = 0; c < dimension; c++)
6             for (int k = 0; k < dimension; k++)
7                 matrix_res[r][k] = matrix_res[r][k] + matrix_a[r][c] * matrix_b[c][k];
8 }
```

Listing 6.7: Parallel matrix multiplication using Cilk Plus.

Table 6.10: Performance of Cilk Plus **matrixmul** with and without automatic vectorisation.

	Wall clock time (seconds)	
Compiler	icpc Cilk	icpc Cilk + SSE4.2
Mean	1.9924	0.7736
Geo. Mean	1.991722	0.773534
Std. Dev.	0.05846623	0.01128273
Speedup	4.338x	11.171x

Again, we see superlinear speedup from one of the parallel implementations, which is due, in part, to the efficient Cilk scheduler and to the cache memory effects (Table 6.10). Intel Inspector XE found no memory related errors, while Memcheck located a number of issues in the Cilk libraries, which are beyond the control of the programmer. Runtime profiling with VTune and Valgrind shows that the Cilk Plus implementation exhibits good concurrency with no significant performance issues.

Intel MKL

We have demonstrated that it is possible to achieve good speedup with our own parallel implementations. However, as stated in Section 3.3.6, highly optimised solutions for many problems already exist and matrix multiplication is one such problem. The Intel Math Kernel Library provides routines for basic linear algebra, including a variety of matrix multiplication routines [77]. The MKL BLAS routines are accessible via a Fortran interface, however, CBLAS provides a C/C++ wrapper for the BLAS routines. MKL also provides routines for the aligned allocation of memory [77]. Our implementation of matrix multiplication simply initialises the matrices and calls the appropriate MKL routine to perform the multiplication as shown in Listing 6.8.

```
1  #include <mkl.h>
2
3  void init_matrices(void) {
4      // initialise the matrices with random values
5      matrix_a = (double*)mkl_malloc(sizeof(double)*dimension*dimension, 128);
6      ...
7  }
8  int main(int argc, char** argv) {
9      ...
10     init_matrices();
11     // call MKL routine to multiply two matrices of doubles
12     cblas_dgemm(CblasRowMajor, CblasNoTrans, CblasNoTrans, dimension, \
13                     dimension, dimension, 1.0, matrix_a, dimension,  \
14                     matrix_b, dimension, 1.0, matrix_res, dimension);
15     ...
16     mkl_free(matrix_a);
17     return 0;
18 }
```

Listing 6.8: Parallel matrix multiplication using MKL.

Table 6.11: Performance of MKL **matrixmul** with and without automatic vectorisation.

	Wall clock time (seconds)	
Compiler	g++ MKL + SSE4.2	icpc MKL + SSE4.2
Mean	0.6022	0.7362
Geo. Mean	0.6021775	0.7361903
Std. Dev.	0.00580517	0.004207137
Speedup	24.1x	11.737x

As shown in the performance summary in Table 6.11, it is clear that the use of existing libraries has the potential to provide very good speedup with minimal effort.

6.3 Mandelbrot Set Algorithm

The Mandelbrot set is a mathematical set of points in the complex plane, named after the mathematician, Benoît Mandelbrot, who studied the set and published significant research on it. The boundary of this set of points forms a fractal pattern, which is often referred to as the "Mandelbrot fractal". A complex number, c, is in the Mandelbrot set if the absolute value of z_n, starting with $z_0 = 0$, never exceeds a certain value, regardless of the size of n. The value of z_n is calculated by iterating the equation defined in (6.2) repeatedly [10, 26].

$$z_{n+1} \mapsto z_n^2 + c \tag{6.2}$$

The "escape time" algorithm is the simplest algorithm for generating points in the Mandelbrot set. In this algorithm, the calculation is iterated for each x,y point in the desired plot area until an escape or bailout condition is reached. The Mandelbrot set is only interesting between -1.0 and 1.0 on the imaginary axis (y-axis) and between -2.0 and 1.0 on the real axis (x-axis), therefore, the points on the plot must be scaled to fit within this region [10, 26]. The escape condition can vary in complexity depending on the method chosen for calculating the bailout point. The method chosen for our algorithm uses a more computationally complex algorithm that detects the escape condition sooner than other simple escape conditions. This is achieved by using the Pythagorean theorem to calculate the distance from the origin and bailout if the distance is greater than two (no complex number with either part greater than two can form part of the Mandelbrot set). In addition to this escape condition, an iteration limit is used to control the depth or detail of the fractal [10, 26]. This variation in the number of iterations performed for each point creates a simple and easily reproducible example of an uneven workload that requires careful load balancing in concurrent implementations.

The relevant sections of the baseline sequential program code, named **mandelbrot**, are shown in Listing 6.9, while the full program source code can be found in Appendix B.2. The source code for this program is based on various versions of the program code for the Mandelbrot benchmark from The Computer Language Benchmarks Game [46].

6.3.1 Initial Profiling and Analysis

The baseline performance of the original sequential Mandelbrot set program is presented in Table 6.12. The **mandelbrot** program was compiled using GCC with `-O0 -g` as the compiler

```
1  void mandelbrot(char* data, int* nbyte_each_line, int dimension, int width_bytes) {
2      const int iterations = 100;
3      const double limit = 2.0 * 2.0;
4      double Cimag, Creal, Zreal, Zimag;
5
6      for (int y = 0; y < dimension; ++y) {
7          char* pdata = data + (y * width_bytes);
8
9          // scale y to -1.0, 1.0 on the imaginary axis
10         Cimag = ((double)y * 2.0 / dimension) - 1.0;
11         bool le_limit;
12
13         for (int x = 0; x < dimension; ++x) {
14             // scale x to -2.0, 1.0 on the real axis
15             Creal = ((double)x * 3.0 / dimension) - 2.0;
16             Zreal = Creal;
17             Zimag = Cimag;
18
19             le_limit = true;
20             // iterate z = z^2 + c until iteration cutoff or limit is reached
21             for (int i = 0; i < iterations && le_limit; ++i) {
22                 double Zrealtemp = Zreal;
23                 Zreal = (Zreal * Zreal) - (Zimag * Zimag) + Creal;
24                 Zimag = 2.0 * Zrealtemp * Zimag + Cimag;
25
26                 le_limit = ((Zreal * Zreal) + (Zimag * Zimag) <= limit);
27             }
28
29             // mark whether or not the pixel is within the mandelbrot
30             // set in the pdata array.
31         }
32     }
33 }
```

Listing 6.9: Escape time Mandelbrot set algorithm (**mandelbrot**).

options and the timing measurements were repeated five times with *dimension* equal to 16000. For the purposes of profiling, a *dimension* of 8000 is used for convenience when profiling with tools such as Valgrind, which add considerable performance overheads. As in the previous example, profiling is performed using Callgrind (a Valgrind tool) and VTune, and the implementations are checked for memory and threading errors using Memcheck, Helgrind, and Intel Inspector XE. Profiling reveals that 95.37% of the CPU time is spent in the innermost loop of the algorithm (lines 21 to 27 of Listing 6.9), indicating that this region is the primary hotspot. Both Callgrind and VTune report a 0% miss rate for all levels of the cache. This is to be expected as the Mandelbrot set algorithm does not consume input data and only accesses data that is calculated during execution, which will be present in the local cache due to temporal locality. VTune also reports an operation retire stall rate of 0.4%, resulting from execution stalls and instruction starvation, which are likely caused by the system running at full capacity (as suggested by VTune). This indicates that the base sequential algorithm does not suffer from any significant performance issues. No memory leaks or access errors were detected by either Memcheck or Intel Inspector XE.

Table 6.12: Performance of the original, unoptimised **mandelbrot** program.

Mean Wall clock time	Geometric Mean	Standard Deviation
81.937 s	81.937 s	0.01923538

6.3.2 Sequential Optimisations

A variety of simple compiler optimisation options have been tested and measured, as shown below. However, it appears that optimisation levels beyond `-O2` do not result in any noticeable speedup. The `-ffast-math` optimisation for GCC was also tested, along with the equivalent options for the Intel compiler. These floating point mathematical optimisations improved the performance by roughly four seconds, but since they affect the accuracy of certain floating point calculations, the resulting program output from these optimised versions was incorrect. Therefore, these optimisations were excluded from further testing. The speedups presented in Table 6.13 are measured against the original sequential program performance as given in Table 6.12.

Table 6.13: Performance of **mandelbrot** with sequential optimisations.

	Wall clock time (seconds)				
Compiler	g++ -O2	g++ -O3	icpc -O0	icpc -O2	icpc -O3
Mean	45.5754	45.5786	85.8382	41.5824	41.581
Geo. Mean	45.5754	45.5786	85.83072	41.5824	41.581
Std. Dev.	0.01013903	0.01101363	1.275116	0.008988882	0.01534601
Speedup	1.78x	1.8x	0.955x	1.97x	1.971x

In addition to the above compiler optimisations, IPO and PGO were tested with the results presented in Table 6.14. The only significant performance increase from this set of optimisation options is the profile-guided optimisation performed by GCC, which brings its performance in line with the Intel compiler. PGO is not used in further optimisation testing for the sake of convenience.

Table 6.14: Performance of **mandelbrot** with IPO and PGO compiler optimisations.

	Wall clock time (seconds)			
Compiler	g++ w/ IPO	g++ w/ PGO	icpc w/ IPO	icpc w/ PGO
Mean	45.5926	41.8018	41.6232	41.5578
Geo. Mean	45.59259	41.8018	41.62318	41.5578
Std. Dev.	0.02808558	0.01377316	0.04600217	0.01265701
Speedup	1.8x	1.96x	1.97x	1.97x

Profiling reveals an increase in the cache miss rates. However, the actual miss counts are same, while the number of memory references have been decreased dramatically, causing the perceived increase in the miss rates. Therefore, it was not deemed necessary to attempt to optimise the cache access. The Callgrind summary data for the execution of the unoptimised (`icpc -O0 -g`) and optimised (`icpc -O3 -ipo -prof-use -g`) binaries are shown in Figures 6.1 and 6.2, respectively, for *dimension* equal to 4000. Again, no memory related errors were detected by VTune and Memcheck.

```
D   refs:      10,295,210,137  (7,790,125,991 rd + 2,505,084,146 wr)
D1  misses:            73,152  (       40,015 rd +        33,137 wr)
LLd misses:            37,607  (        5,076 rd +        32,531 wr)
D1  miss rate:            0.0% (          0.0%   +           0.0%  )
LLd miss rate:            0.0% (          0.0%   +           0.0%  )
```

Figure 6.1: Callgrind summary for unoptimised **mandelbrot** program.

```
D   refs:           3,357,904  (    853,223 rd + 2,504,681 wr)
D1  misses:            73,595  (     40,413 rd +    33,182 wr)
LLd misses:            37,739  (      5,177 rd +    32,562 wr)
D1  miss rate:            2.1% (        4.7%   +       1.3%  )
LLd miss rate:            1.1% (        0.6%   +       1.3%  )
```

Figure 6.2: Callgrind summary for optimised **mandelbrot** program.

The Intel C/C++ compiler optimisation diagnostics, accessed via the `-opt-report` and `-vec-report` options, reveal that a variety of loop optimisations, such as loop interchange, loop unrolling, and loop vectorisation, are not possible due to loop carried dependencies and unsupported loop structures. An inspection of the code confirms this. None of the loops contain loop invariant statements that can be moved out of the loops and the inner loops contain loop carried dependencies that cannot be removed because they are required by the algorithm. As with the matrix multiplication example, all further Mandelbrot implementations are compiled with the respective IPO and -O3 compiler optimisation flags enabled, `icpc -O3 -ipo` and `g++ -O3 -fwhole-program`, respectively.

Vectorisation

While the compilers are unable to perform automatic vectorisation of the loop structures, it is possible to manually implement SSE vectorisation using the intrinsics functions provided by the

compiler. The double precision floating point variables and calculations within the nested loops have been converted to SSE packed data types and vector operations. The loop over x has also been unrolled to allow for two loop iterations to be executed simultaneously since the packed data types store two double values, which are then processed as a vector [34]. The resulting code is presented in Listing 6.10 and performance measurements in Table 6.15. The program output has also been checked for correctness against the previous non-SSE implementation. Unfortunately, this optimisation affects the readability and portability of the code, but since the target platform is primarily commodity x86-based multicore CPUs, most of which support SSE3 and up, portability is not a primary concern.

```
#include <pmmintrin.h>

void mandelbrot(char* data, int* nbyte_each_line, int dimension, int width_bytes) {
    const int iterations = 100;
    __m128d val1      = _mm_set1_pd(1.0);
    __m128d val2      = _mm_set1_pd(2.0);
    __m128d limit     = _mm_set1_pd(4.0);
    __m128d scale_y = _mm_set1_pd(2.0 / dimension);
    __m128d scale_x = _mm_set1_pd(3.0 / dimension);

    for (int y = 0; y < dimension; ++y) {
        char* pdata = data + (y * width_bytes);

        // scale y to -1.0, 1.0 on the imaginary axis
        __m128d Cimag = _mm_set1_pd(y);
        Cimag = _mm_sub_pd(_mm_mul_pd(Cimag, scale_y), val1);

        for (int x = 0; x < dimension; x += 2) {
            // scale x to -2.0, 1.0 on the real axis
            __m128d Creal = _mm_set_pd(x, x + 1);
            Creal = _mm_sub_pd(_mm_mul_pd(Creal, scale_x), val2);

            __m128d Zreal = Creal;
            __m128d Zimag = Cimag;
            __m128d Treal = _mm_mul_pd(Creal, Creal);
            __m128d Timag = _mm_mul_pd(Cimag, Cimag);

            int result = 3;
            int i = 0;
            while ( (result != 0) && (i++ < iterations) ) {
                Zimag = _mm_add_pd(_mm_mul_pd(_mm_mul_pd(Zreal, Zimag), val2), Cimag);
                Zreal = _mm_add_pd(_mm_sub_pd(Treal, Timag), Creal);

                Treal = _mm_mul_pd(Zreal, Zreal);
                Timag = _mm_mul_pd(Zimag, Zimag);
                __m128d delta = _mm_cmple_pd(_mm_add_pd(Treal, Timag), limit);

                int mask = _mm_movemask_pd(delta);
                result &= mask;
            }

            // mark whether or not the pixel is within the mandelbrot
            // set in the pdata array.
        }
    }
}
```

Listing 6.10: Vectorised Mandelbrot set algorithm.

Table 6.15: Performance of **mandelbrot** with SSE vectorisation.

	Wall clock time (seconds)	
Compiler	g++ SSE4.2	icpc SSE4.2
Mean	22.212	22.8114
Geo. Mean	22.21199	22.81138
Std. Dev.	0.01872165	0.02961925
Speedup	3.704x	3.592x

As can be seen from the results, the increased instruction-level parallelism has almost doubled the speedup. Profiling shows that there has been no significant change in the execution and memory access characteristics of the program, indicating that no performance issues have been introduced and that the optimised version is still making full use of the capacity of the CPU cores. This vectorised implementation is the new baseline for the **mandelbrot** program and all parallelisation efforts will be based on it.

6.3.3 Parallel Implementation

The Mandelbrot set algorithm is another example of a problem that is very amenable to parallel execution. As with the matrix multiplication example, the problem is primarily organised by tasks, leading to the Task Decomposition pattern of the Identifying Concurrency design space. The primary task is to determine whether, for each point in the desired plot area, a particular point falls within the Mandelbrot set. This can be further decomposed into the task of performing the iterative calculation to update z and the task of determining whether the value of z falls within the appropriate limits. The inner loops of the algorithm (Listing 6.9) are responsible for carrying out these tasks and are identified as the execution hotspots through profiling and performance analysis in Section 6.3.1. Other tasks include scaling the points to the appropriate range on the imaginary and real axes and storing the result in the output data structure.

With the relevant tasks identified, the next step is to apply the dependency analysis patterns. The Group Tasks and Order Tasks patterns specify that tasks must be grouped and ordered according to their temporal and ordering constraints. There is an ordering constraint on the tasks that perform the update calculations for z and determine whether z is within limits, as the limit check relies on the results of the calculation. Additionally, each successive iteration of the z update calculation and subsequent limit check is dependent on the results of the previous iteration, which introduces an ordering constraint. Therefore, the calculation and limit check tasks are grouped and an ordering constraint is placed on the iterations of the task group. Another ordering constraint is the requirement that the co-ordinates of the point be scaled to the appropriate range before starting with the calculations for that point. Since these tasks must be

performed in order and the only computationally intensive tasks are the repeated calculations and limit checking, the tasks for a particular point can be grouped into one large independent task. The only data sharing aspect of the problem is the storing of the results in a shared result array. However, each task writes to an exclusive section of the array, making the shared accesses effectively-local, which means that no explicit synchronisation is required.

Since the problem is organised as a linear set of tasks (task for each point on the plot), the Task Parallelism pattern from the Algorithm Structure design space is used. The tasks are associated with the iterations of the loops and are known from the outset. However, unlike the matrix multiplication problem, each task does not necessary have the same computation time as another task due to the nature of the iterative calculations and escape conditions. This results in uneven workloads, which means that static scheduling is unlikely to produce an efficient workload distribution. Therefore, a dynamic scheduling approach is the best choice for this problem. Static scheduling versions of the implementations will be developed in addition to the dynamic versions to demonstrate the effect of bad load balancing. The Loop Parallelism, Master/Worker, and Shared Queue patterns from the Supporting Structures design space are, therefore, required to support the parallel design. According the Loop Parallelism pattern, the iteration space of the outer loop is partitioned into chunks that are executed concurrently by multiple processors. The size of these chunks is dependent on the scheduling algorithm. For static scheduling, the iteration space is divided equally, whereas, with the dynamic scheduling approach using the Master/Worker pattern, a larger number of smaller chunks are added to a shared task queue (which must be implemented by the programmer according to the Shared Queue pattern for explicit threading approaches).

Finally, the above parallel design must be implemented in code using the appropriate language constructs and parallel libraries, as per the Implementation Mechanisms design space. The implementations for automatic parallelisation, OpenMP, Threading Building Blocks, Pthreads, Boost Threads, Cilk Plus, and the shared task queue are presented and described below. Performance libraries such as Intel Math Kernel Library and Intel Integrated Performance Primitives do not provide any relevant functions for implementing the Mandelbrot program. The speedups listed are relative to the best sequential performance for that specific compiler.

Automatic Parallelisation

As with the matrix multiplication problem in Section 6.2.3, automatic parallelisation of the sequential program has been attempted. Yet again, the automatic parallelisation feature within

GCC (`g++ -floop-parallelize-all -ftree-parallelize-loops=4`) was unable to improve the performance of the Mandelbrot program. On the other hand, the Intel C/C++ compiler (using `icpc -parallel -par-schedule-auto`) was able to generate an efficient parallel version of the program after placing the `#pragma parallel` above the outermost loop. The performance summary is shown in Table 6.16.

Table 6.16: Performance of auto-parallelised **mandelbrot**.

	Wall clock time (seconds)	
Compiler	g++ auto-parallel	icpc auto-parallel
Mean	22.2954	5.8472
Geo. Mean	22.29539	5.846668
Std. Dev.	0.02235621	0.0887733
Speedup	0.996x	3.902x

OpenMP

The OpenMP (OMP) **parallel for** construct has been used to parallelise the outermost **for** statement of the Mandelbrot set algorithm as shown in Listing 6.11. The `schedule(static)` clause indicates that this implementation makes use of static scheduling.

```
1  void mandelbrot(...) {
2      ...
3      #pragma omp parallel for default(shared) schedule(static) num_threads(NTHREADS)
4      for (int y = 0; y < dimension; ++y) {
5          ...
6          for (int x = 0; x < dimension; x+=2) {
7              ...
8          }
9          ...
10     } /* end of omp parallel for */
11 } /* end of void mandelbrot(...) */
```

Listing 6.11: Parallel **mandelbrot** using OpenMP with static scheduling.

However, as discussed in the parallel design stage, the workload of the Mandelbrot algorithm is unbalanced. Therefore, a more effective OpenMP implementation using dynamic scheduling (with a *chunk* size based on the input dimension divided by 64, which was chosen through manual experimentation) was also developed and is presented in Listing 6.12. Table 6.17 shows the performance summary for the static and dynamic implementations.

The dynamic scheduling implementations provide a clear performance improvement over their static scheduling counterparts. Additionally, it appears that GCC produces a slightly faster pro-

```
1  void mandelbrot(...) {
2      ...
3      #pragma omp parallel for default(shared) schedule(dynamic,dimension >> 6) \
4                                                        num_threads(NTHREADS)
5      for (int y = 0; y < dimension; ++y) {
6          ...
7      } /* end of omp parallel for */
8  } /* end of void mandelbrot(...) */
```

Listing 6.12: Parallel **mandelbrot** using OpenMP with dynamic scheduling.

Table 6.17: Performance of OMP **mandelbrot** with static and dynamic scheduling.

	Wall clock time (seconds)			
Compiler	g++ OMP static	g++ OMP dynamic	icpc OMP static	icpc OMP dynamic
Mean	8.646	5.6788	9.014	5.8074
Geo. Mean	8.645974	5.678797	9.013996	5.8074
Std. Dev.	0.02382226	0.006760178	0.009539392	0.002701851
Speedup	2.569x	3.911x	2.531x	3.928x

gram executable, but the performance difference is minor (approximately 2%). Profiling reveals that the dynamically scheduled implementation has good concurrency and CPU usage, with no significant performance issues. The statically scheduled version shows poor CPU utilisation, resulting from the unbalanced workload (Figure 6.3). Neither implementation has any threading and memory access issues (potential data race issues reported by Valgrind are false positives related to the effectively-local writes to the result array).

Thread Building Blocks

Unlike OpenMP, Thread Building Blocks (TBB) requires more extensive modifications to the original code to implement parallelism. The Mandelbrot algorithm code has been moved into

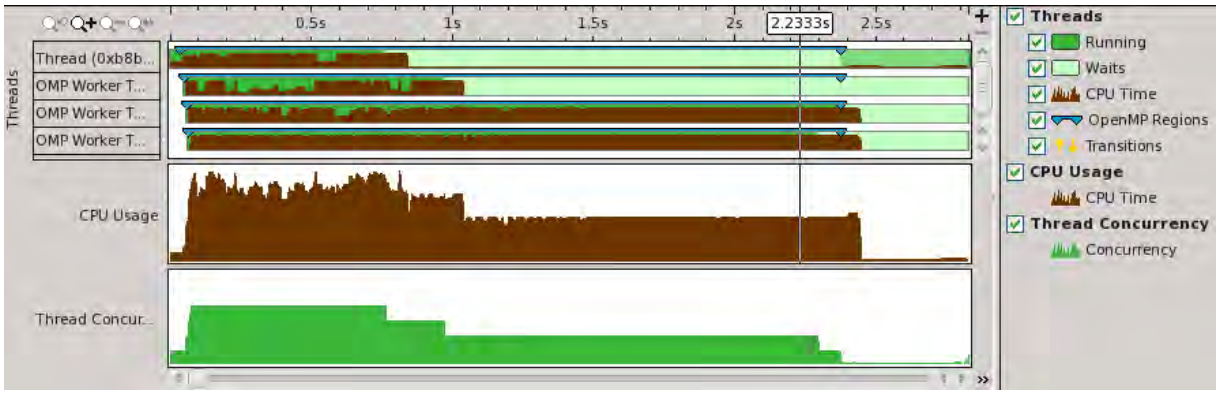


Figure 6.3: VTune concurrency profiling for the statically scheduled OpenMP **mandelbrot**.

its own class, `MandelPar`, and placed within an overloaded `operator()` function, along with the required data elements, which are declared and initialised as private data members within the class.

The only change that needs to be made to the algorithm itself involves modifying the loop bounds for the outer **for** loop such that it uses `r.begin()` for the initial value and `r.end()` for the loop termination boundary, where `r` is the TBB `blocked_range` object supplied as an argument to the operator function [125]. The call to the **mandelbrot** function has been replaced with a call to the TBB `parallel_for` template.

In the program's `main` function, the TBB task scheduler is initialised along with the `MandelPar` instance to be used with the `parallel_for`. The TBB `parallel_for` template is then called with the `MandelPar` instance and an instance of the `blocked_range` template initialised with the iteration space and *block size* (similar to OpenMP's chunk size) set to the input dimension divided by 64. In addition to the explicit block size implementation, an implementation using TBB's `auto_partitioner` feature was also tested [125]. Listing 6.13 shows the TBB **mandelbrot** implementation using an explicit block size. The performance summary for these two implementations is presented in Table 6.18.

```
1  #include "tbb/task_scheduler_init.h"
2  #include "tbb/blocked_range.h"
3  #include "tbb/parallel_for.h"
4  using namespace tbb;
5
6  class MandelPar {
7  private:
8      char* data;
9      int* nbyte_each_line;
10     int width_bytes, dimension;
11 public:
12     MandelPar(char* d, int* nb, int dim, int wb)
13     : data(d), nbyte_each_line(nb), dimension(dim), width_bytes(wb) {}
14
15     void operator()( const blocked_range<int>& r ) const {
16         ...
17         for (int y = r.begin(); y != r.end(); ++y) {
18             ...
19             for (int x = 0; x < this->dimension; x += 2) {
20                 ...
21             }
22             ...
23 } } }; /* end of MandelPar class */
24
25 int main(int argc, char** argv) {
26     ...
27     task_scheduler_init init(NTHREADS);
28     MandelPar mpar (data, nbyte_each_line, dimension, width_bytes);
29     parallel_for (blocked_range<int>(0, dimension, dimension >> 6), mpar);
30     ...
31 } /* end of int main(...) */
```

Listing 6.13: TBB **parallel_for** implementation with **MandelPar** class.

Table 6.18: Performance of TBB **mandelbrot** with and without the auto-partitioner.

	Wall clock time (seconds)			
Compiler	g++ TBB	g++ TBB auto-part.	icpc TBB	icpc TBB auto-part.
Mean	5.8478	5.8766	5.9832	5.8948
Geo. Mean	5.847364	5.875111	5.98273	5.894466
Std. Dev.	0.0799356	0.1482845	0.08400714	0.07051383
Speedup	3.799x	3.781x	3.813x	3.87x

From the results, we can see that there is very little difference in performance between the implementations and the two compilers. Profiling reveals that both the explicit and auto-partitioner implementations exhibit good CPU usage with no performance issues. Error checking with Valgrind and Intel Inspector XE confirm that there are no threading errors or memory access issues, although, Intel Inspector identifies a memory leak in the TBB library.

Cilk Plus

The Cilk Plus implementation is very straightforward. The outermost **for** statement is replaced with **cilk_for** and the number of worker threads is set in the program's main function, as shown in Listing 6.14. Since Cilk Plus is only supported by the Intel compiler, the performance summary in Table 6.19 only lists the performance details using icpc.

```
1  #include <cilk/cilk.h>
2  #include <cilk/cilk_api.h>
3
4  void mandelbrot(char* data, int* nbyte_each_line, int dimension, int width_bytes) {
5      ...
6      cilk_for (int y = 0; y < dimension; ++y) {
7          ...
8          for (int x = 0; x < dimension; x += 2) {
9              ...
10         }
11         ...
12 }  }
13
14 int main(int argc, char** argv) {
15     ...
16     char nworkers[5];
17     sprintf(nworkers, "%d", NTHREADS);
18     __cilkrts_set_param("nworkers", nworkers);
19
20     mandelbrot(data, nbyte_each_line, dimension, width_bytes);
21     ...
22 } /* end of int main(...) */
```

Listing 6.14: Cilk Plus **mandelbrot** implementation.

Although the performance of the Cilk Plus implementation is slightly inferior to the performance of the other parallel implementations, the measured speedup is still very good. Profiling

Table 6.19: Performance of Cilk Plus **mandelbrot**.

	Wall clock time (seconds)
Compiler	icpc Cilk
Mean	6.0174
Geo. Mean	6.016267
Std. Dev.	0.1304964
Speedup	3.792x

reveals that CPU utilisation and concurrency are good, but are very close to being classified as poor by VTune. There are no other performance issues or memory access errors, but Valgrind and Intel Inspector report a possible data race relating to accesses to the `nbyte_each_line` array. This issue was corrected by inserting a **cilk_sync** statement after the call to the **mandelbrot** function to ensure that all Cilk threads are synchronised before outputting the result.

Shared Task Queue

While the implicit threading models, such as OpenMP and Threading Building Blocks, have built-in thread scheduling mechanisms, the explicit threading models require the programmer to implement such scheduling manually. For Pthreads and Boost Threads, static scheduling does not require any complex supporting structures. However, for dynamic scheduling, a shared task queue is required to support the Master/Worker threading pattern.

As such, we have developed a simple, specialised queue data structure to store tasks and allow threads to retrieve new tasks when their current task has been completed. The abridged source code for the `thread_data` and `worker_node` structs, as well as the `WorkQueue` class, is given in Listing 6.15. The `thread_data` struct stores the loop bounds for each task. The `worker_node` struct (queue node data structure) contains pointers to the preceding and following nodes and a pointer to that node's task data. The `WorkQueue` class keeps track of the head and tail nodes and provides public functions for pushing nodes to the tail of the queue (`WorkQueue::push`) and popping nodes off the head and returning the popped task data (`WorkQueue::pop`).

Unfortunately, this initial implementation is not thread-safe and it does not perform any memory management, as evidenced by the data race and memory leak issues detected by Valgrind and Intel Inspector XE. Sample threading and memory issue reports from Intel Inspector are given in Figures 6.4 and 6.5 respectively.

The *WorkQueue* class was modified to make it thread-safe and resolve the memory leak. The memory leak was resolved by implementing an appropriate de-constructor and deleting the

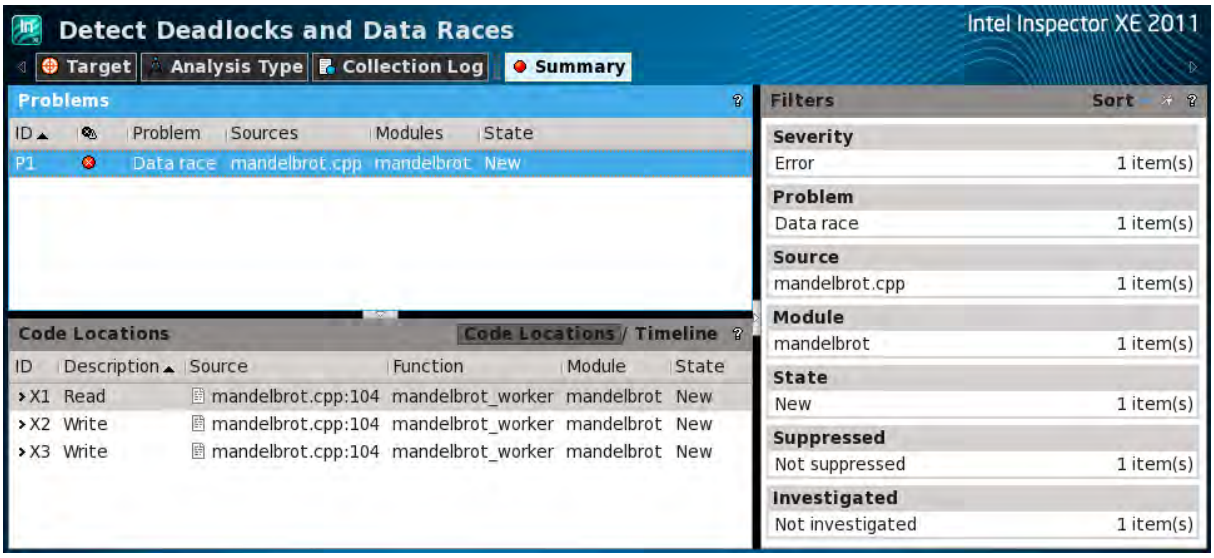


Figure 6.4: Data race error detected by Intel Inspector in the shared task queue.

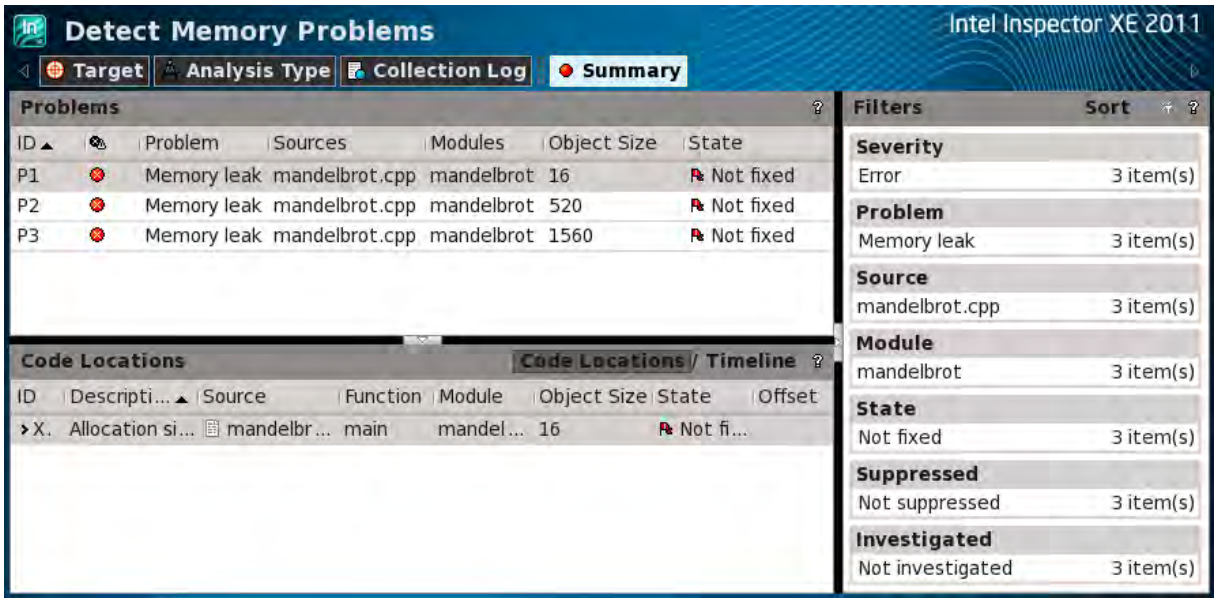


Figure 6.5: Memory leak detected by Intel Inspector in the shared task queue.

```
typedef struct {
    int start;
    int end;
} thread_data;

struct worker_node {
    worker_node* next;
    worker_node* prev;
    thread_data* data;
};

class WorkQueue {
private:
    worker_node* head;
    worker_node* tail;

public:
    WorkQueue() {
        this->head = NULL;
        this->tail = NULL;
    }

    thread_data* pop() {...}

    void push(thread_data* data) {...}
};
```

Listing 6.15: Shared task queue data structure for dynamic scheduling.

worker nodes once they have been popped off the queue. The data race issue was resolved by implementing mutual exclusion within the methods of the class itself, making it a kind of monitor object, although it would also be possible to implement the mutual exclusion in the "client" program. Making the data structure thread-safe instead of relying on client programs to implement mutual exclusion means that application programmers do not have to remember to protect accesses to the structure, however, it does limit portability. For the Pthreads version, a private class member of type `pthread_mutex_t` was added (named `qlock`) and initialised in the constructor. The Pthreads lock and unlock functions were then used to protect changes to the data structure as shown in Listing 6.16.

```
thread_data* pop() {
    pthread_mutex_lock (&(this->qlock));
    // return NULL if queue is empty, else pop and
    // return head node of the queue
    pthread_mutex_unlock (&(this->qlock));
    return res;
}

void push(thread_data* data) {
    ...
    pthread_mutex_lock (&(this->qlock));
    // add new node to the tail of the queue
    pthread_mutex_unlock (&(this->qlock));
}
```

Listing 6.16: Protecting concurrent access to the task queue using Pthreads mutex.

For the Boost Threads version, a private class member, named `qlock`, of type `boost::mutex` was added to the task queue class. The Boost Threads scoped lock object was used to protect changes to the data structure as shown in Listing 6.17.

```
1  thread_data* pop() {
2      {
3          boost::mutex::scoped_lock lock(this->qlock);
4          // return NULL if queue is empty, else pop and
5          // return head node of the queue
6      }
7      return res;
8  }
9
10 void push(thread_data* data) {
11     ...
12     {
13         boost::mutex::scoped_lock lock(this->qlock);
14         // add new node to the tail of the queue
15     }
16 }
```

Listing 6.17: Protecting concurrent access to the task queue using Boost Threads mutex and scoped lock.

While our task queue structure has been implemented from scratch for the explicitly threaded program implementations, it is entirely possible to use generic queue data structures from external libraries or reuse existing queue implementations, provided that the data structures are thread-safe. The final implementation for the Pthreads version of the shared task queue can be found in Appendix C.1 (Listing C.1).

Pthreads

Pthreads (PT) is an explicit threading model that requires the programmer to manage threading details such as thread creation, scheduling, and termination. There are two different implementations using Pthreads: one based on a simple static scheduling approach, and the other making use of the shared task queue and dynamic scheduling.

The statically scheduled Pthreads **mandelbrot** was implemented as follows. A thread pool consisting of *NTHREADS* threads is declared and initialised and an array of size *NTHREADS* is declared for the storage and communication of thread specific data to the respective threads. For each thread, the thread specific data, such as the start and end bounds for the parallel loop (the chunk size is calculated by dividing the input dimension by the number of threads), and common data, such as the plot dimension, are stored in the respective array element data structure (of type `struct mandelbrot_data`). Then the threads are spawned or created, passing in

a void pointer to the appropriate array element as the thread's argument. Finally, the threads are joined once they finish executing their share of the workload. This creates a thread for each processor, assigns each thread an equal chunk of the plot to process (but not an equal share of the computational workload) and joins the threads before outputting the result. The source code for the program's `main` function is provided in Listing 6.18.

```
#include <pthread.h>

typedef struct {
    int start;
    int end;
    int dimension;
    int width_bytes;
    int* nbyte_each_line;
    char* data;
} mandelbrot_data;

int main(int argc, char** argv) {
    ...
    pthread_t thread_id[NTHREADS];
    static mandelbrot_data thread_data_array[NTHREADS];
    int cur_start = 0;
    int chunk = (int)ceil((double)dimension / NTHREADS);

    for (int i = 0; i < NTHREADS; i++) {
        thread_data_array[i].data = data;
        thread_data_array[i].dimension = dimension;
        thread_data_array[i].nbyte_each_line = nbyte_each_line;
        thread_data_array[i].width_bytes = width_bytes;
        thread_data_array[i].start = cur_start;

        int cur_end = std::min(cur_start + chunk, dimension);
        thread_data_array[i].end = cur_end;
        cur_start = cur_end;

        pthread_create(&thread_id[i], NULL, mandelbrot_worker, (void*) &thread_data_array[i]);
    }
    for (int j = 0; j < NTHREADS; j++)
        pthread_join(thread_id[j], NULL);
    ...
} /* end of int main(...) */
```

Listing 6.18: Pthreads **mandelbrot** `main` function using static scheduling.

The Pthreads worker function implementation is shown in Listing 6.19. The worker function takes the thread argument and casts it back to a pointer to a `mandelbrot_data` struct and uses the `start` and `end` data structure members as the bounds for the outermost loop. Other common data, such as the input dimension, is also retrieved from the data structure.

As with OpenMP, a dynamic scheduling solution is also possible with Pthreads. The implementation is somewhat more complex than the static scheduling version above. Our implementation makes use of the Master/Worker threading approach and a shared task queue (6.3.3) that stores uncompleted tasks, which are created by dividing the iteration space into chunks of work and

```
1  void* mandelbrot_worker(void* threadarg) {
2      mandelbrot_data* my_data = (mandelbrot_data*)threadarg;
3      ...
4      for (int y = my_data->start; y < my_data->end; ++y) {
5          ...
6          for (int x = 0; x < my_data->dimension; x += 2) {
7              ...
8          }
9          ...
10     }
11     return NULL;
12 } /* end of void mandelbrot_worker(...) */
```

Listing 6.19: Pthreads **mandelbrot_worker** function using static scheduling.

pushed onto the queue. The program's `main` function is presented in Listing 6.20, which shows the declaration of the thread pool and common thread data, the instantiation of the the task queue (line 17), the filling of the task queue by dividing the input dimension into a larger number of small chunks (lines 19 to 25), and the creation and joining of the threads in the thread pool.

```
1  typedef struct {
2      int dimension;
3      int width_bytes;
4      int* nbyte_each_line;
5      WorkQueue* workload;
6      char* data;
7  } mandelbrot_data;
8
9  int main(int argc, char** argv) {
10     ...
11     pthread_t thread_id[NTHREADS];
12     static mandelbrot_data thread_data_static;
13     thread_data_static.data = data;
14     thread_data_static.dimension = dimension;
15     thread_data_static.nbyte_each_line = nbyte_each_line;
16     thread_data_static.width_bytes = width_bytes;
17     thread_data_static.workload = new WorkQueue();
18
19     int block = dimension >> 6;
20     for (int k = 0; k < dimension; k += block) {
21         thread_data* new_load = new thread_data();
22         new_load->start = k;
23         new_load->end = (k + block > dimension) ? dimension : k + block;
24         thread_data_static.workload->push(new_load);
25     }
26
27     for (int i = 0; i < NTHREADS; i++)
28         pthread_create(&thread_id[i], NULL, mandelbrot_worker, (void*)&thread_data_static);
29     for (int j = 0; j < NTHREADS; j++)
30         pthread_join(thread_id[j], NULL);
31     ...
32 } /* end of int main(...) */
```

Listing 6.20: Pthreads **mandelbrot** `main` function using dynamic scheduling.

The dynamic scheduling version of the Pthreads worker function, `mandelbrot_worker`, differs from its static counterpart in that the loop bounds are taken from the currently assigned task. The worker function has been modified to include an infinite **while** loop that repeatedly requests and processes tasks from the task queue and exits when there are no longer any tasks to process. The workload is fixed and tasks are added to the queue before the worker threads begin executing, so it is not necessary to implement thread signalling to inform threads of new tasks or the end of the tasks. Threads can simply terminate once the task queue is empty. The `mandelbrot_worker` is shown in Listing 6.21 and the performance summary for both the static and dynamic implementations is presented in Table 6.20.

```
1  void* mandelbrot_worker(void* threadarg) {
2      mandelbrot_data* my_data = (mandelbrot_data*)threadarg;
3      thread_data* my_work;
4      ...
5      while (true) {
6          my_work = my_data->workload->pop();
7          if (my_work == NULL) {
8              pthread_exit(NULL);
9              break;
10         }
11         else {
12             for (int y = my_work->start; y < my_work->end; ++y) {
13                 ...
14                 for (int x = 0; x < my_data->dimension; x += 2) {
15                     ...
16                 }
17                 ...
18             }
19             delete my_work;
20     }   }
21     return NULL;
22 }
```

Listing 6.21: Pthreads `mandelbrot_worker` function using dynamic scheduling.

Table 6.20: Performance of Pthreads (PT) **mandelbrot** with static and dynamic scheduling.

	Wall clock time (seconds)			
Compiler	g++ PT static	g++ PT dynamic	icpc PT static	icpc PT dynamic
Mean	8.6718	5.654	8.8572	5.8258
Geo. Mean	8.671697	5.653992	8.85717	5.825791
Std. Dev.	0.04705529	0.01067708	0.02549902	0.01118928
Speedup	2.561x	3.929x	2.575x	3.916x

The Pthreads performance results are similar to the OpenMP results, showing excellent speedup for the dynamically scheduled implementation and significantly inferior performance for the statically scheduled implementation. Profiling information for the statically scheduled implementation reveals that the CPU is not fully utilised throughout the program's execution due to the unbalanced workloads of the loop iterations assigned to each thread. This results in poor

speedup. There is no significant lock contention resulting from the locks required by the shared task queue. Neither implementation has any performance issues and there are no memory access or threading errors according to VTune and Valgrind.

Boost Threads

The threading approach behind the Boost Threads implementations is essentially the same as for Pthreads, so the only significant difference is the library functions and constructs used to implement parallelism. The statically scheduled program is implemented by initialising a thread pool (using Boost's `thread_group` class), instantiating an object of the `Mandelbrot_Thread` class, dividing the iteration space into equal sized chunks, creating a Boost thread (passing it a reference to the `Mandelbrot_Thread` object) and adding it to the thread pool, and finally, joining all the threads upon completion. The source code for the statically scheduled Boost implementation's `main` function is provided in Listing 6.22.

```
1  #include <boost/thread.hpp>
2
3  int main(int argc, char** argv) {
4      ...
5      int cur_start = 0;
6      int block = (int)std::ceil((double)dimension / NTHREADS);
7      boost::thread_group threads;
8      static Mandelbrot_Thread m(dimension, width_bytes, data, nbyte_each_line);
9
10     for (int t = 0; t < NTHREADS; t++) {
11         int cur_end = std::min(cur_start + block, dimension);
12         threads.add_thread(new boost::thread(boost::ref(m), cur_start, cur_end));
13         cur_start = cur_end;
14     }
15     threads.join_all();
16     ...
17 }
```

Listing 6.22: Boost Threads **mandelbrot** `main` function with static scheduling.

As with the TBB implementation in Section 6.3.3, the Mandelbrot worker function is moved to an overloaded `()` operator of a function object, `Mandelbrot_Thread`, along with copies of any necessary data [152]. The loop bounds for the outermost loop are then taken from the arguments passed to the function object, as shown in Listing 6.23.

In the dynamically scheduled implementation's `main` function (Listing 6.24), thread pool and `Mandelbrot_Thread` objects are instantiated, but instead of dividing the iteration space equally, the iteration space is divided into smaller chunks and pushed onto a shared task queue (lines 6

```
1  class Mandelbrot_Thread {
2  private:
3      int dimension;
4      int width_bytes;
5      char* data;
6      int* nbyte_each_line;
7
8  public:
9      Mandelbrot_Thread(int dim, int wb, char* d, int* nb)
10     : dimension(dim), width_bytes(wb), data(d), nbyte_each_line(nb) {}
11
12     void operator()(const int start, const int end) {
13         ...
14         for (int y = start; y < end; ++y) {
15             ...
16             for (int x = 0; x < this->dimension; x += 2) {
17                 ...
18             }
19             ...
20 } } };
```

Listing 6.23: Boost Threads `Mandelbrot_Thread` class with static scheduling.

to 12). Then, the threads are spawned and added to the task pool and joined upon completion (lines 14 to 17).

```
1  #include <boost/thread.hpp>
2
3  int main(int argc, char** argv) {
4      ...
5      boost::thread_group threads;
6      static Mandelbrot_Thread m(dimension, width_bytes, data, nbyte_each_line);
7
8      int block = dimension >> 6;
9      for (int k = 0; k < dimension; k += block) {
10         thread_data* new_load = new thread_data();
11         new_load->start = k;
12         new_load->end = (k + block > dimension) ? dimension : k + block;
13         m.add_job(new_load);
14     }
15
16     for (int t = 0; t < NTHREADS; t++) {
17         threads.add_thread(new boost::thread(boost::ref(m)));
18     }
19     threads.join_all();
20     ...
21 } /* end of int main(...) */
```

Listing 6.24: Boost Threads **mandelbrot** `main` function with dynamic scheduling.

For the dynamically scheduled function object (Listing 6.25), the Mandelbrot algorithm has been moved to within an infinite **while** loop. For each iteration of the loop, a new task is popped off the shared task queue (accessed via a private class member that is a pointer to the `WorkQueue` object). If there are no more tasks to process, the loop is terminated with a **break** statement, otherwise the Mandelbrot algorithm is executed using the loop boundaries provided

by the newly acquired task (lines 24 to 29).

```
1  class Mandelbrot_Thread {
2  private:
3      int dimension;
4      int width_bytes;
5      int* nbyte_each_line;
6      WorkQueue* workload;
7      char* data;
8
9  public:
10     Mandelbrot_Thread(int dim, int wb, char* d, int* nb)
11     : dimension(dim), width_bytes(wb), data(d), nbyte_each_line(nb) {
12         this->workload = new WorkQueue();
13     }
14     ~Mandelbrot_Thread() {
15         delete this->workload;
16     }
17     void add_job(thread_data* data) {
18         this->workload->push(data);
19     }
20
21     void operator()() {
22         thread_data* my_work = NULL;
23         ...
24         while (true) {
25             my_work = this->workload->pop();
26             if (my_work == NULL)
27                 break;
28             else {
29                 for (int y = my_work->start; y < my_work->end; ++y) {
30                     ...
31                     for (int x = 0; x < this->dimension; x += 2) {
32                         ...
33                     }
34                     ...
35                 }
36                 delete my_work;
37 } } } }; /* end of Mandelbrot class */
```

Listing 6.25: Boost Threads `Mandelbrot_Thread` class with dynamic scheduling.

The runtime performance summary for both the statically and dynamically scheduled implementations is given in Table 6.21. As with the OpenMP and Pthreads implementations, there is a clear difference in performance between the static and dynamic versions of the program, with static scheduling exhibiting mediocre speedup compared to dynamic scheduling, which has near linear speedup.

Profiling information for both versions of the program indicate that there are no instruction or memory access issues, but the statically scheduled version exhibits poor CPU usage as depicted in Figure 6.6. We can see that for the majority of the program's execution, only two threads are performing any work, which is caused by the uneven workload. Error checking reveals that there are no memory access or threading errors within the program itself, but Intel Inspector detects a memory leak in the Boost Threads library (Valgrind's Memcheck tool does not detect this error).

Table 6.21: Performance of Boost Threads **mandelbrot** with static and dynamic scheduling.

	Wall clock time (seconds)			
Compiler	g++ Boost static	g++ Boost dynamic	icpc Boost static	icpc Boost dynamic
Mean	8.6414	5.6578	8.8854	5.8676
Geo. Mean	8.641356	5.657799	8.88538	5.86754
Std. Dev.	0.03084315	0.003962323	0.02131431	0.02970354
Speedup	2.57x	3.926x	2.567x	3.888x

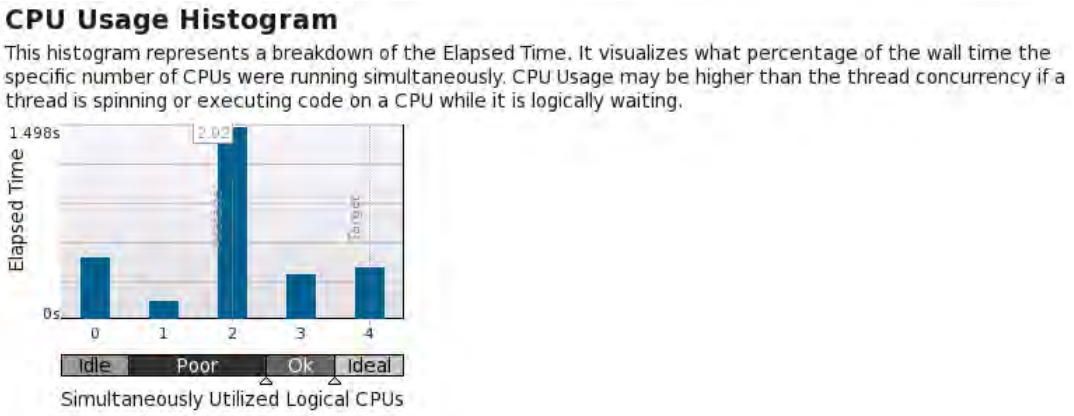


Figure 6.6: VTune CPU usage summary for the statically scheduled Boost Threads **mandelbrot**.

6.4 Deduplication Kernel

The deduplication kernel, **dedup**, was developed by Princeton University and forms part of the Princeton Application Repository for Shared-Memory Computers (PARSEC) benchmark suite [14]. *Deduplication* is a compression method for compressing a data stream using a combination of global and local compression, thereby removing redundant data segments to achieve high compression ratios [155]. The deduplication kernel provided by the PARSEC benchmark suite[1] makes use of a pipelined programming model that mimics real-world implementations, such as the new-generation disk-based deduplication storage systems used for enterprise data protection [14, 155].

The pipeline consists of five stages or pipeline filters that are responsible for performing the deduplication process on an input stream and producing the deduplicated output. The first stage, *DataProcess*, is responsible for breaking the input stream into coarse-grained chunks for further processing. The second stage, *FindAllAnchors*, further segments the data into fine-grained blocks based on the content of the data using rolling fingerprints. The third stage, *ChunkProcess*, computes the SHA1 hash value for each segment. If the hash already exists in the global hash table, the segment is marked as a duplicate, the data is replaced with the hash value, and the segment is sent to the output stage (bypassing the compression stage). If the hash is not found in the hash table, an entry is created for it and the data segment is sent to the compression stage. The compression stage, *Compress*, compresses the data segment using the Ziv-Lempel algorithm (GZIP) and updates the segment's entry in the hash table before passing the segment on to the output stage. Finally, the output stage, *SendBlocks*, assembles the deduplicated output in the correct order from the segments it receives using the hash values and compressed data associated with the corresponding entries in the global hash table [14].

The **dedup** source code is significantly longer than the previous examples and consists of multiple files, so only small segments of code are provided to highlight those aspects of the program under discussion. An abridged version of the encoding function (`Encode`) for the sequential **dedup** program is provided in Listing 6.26.

6.4.1 Initial Profiling and Analysis

The original **dedup** kernel provided by the PARSEC benchmarking suite is written in C. Since some of the parallel programming libraries only support C++, all of the C code files were

[1] `http://parsec.cs.princeton.edu/index.htm`

```
void Encode(config * conf) {
  // open input and output files

  //queue allocation & initialization:
  // only one queue between stages for sequential implementation
  const int nqueues = 1;
  anchor_que = (struct queue *)malloc(sizeof(struct queue) * nqueues);
  send_que = (struct queue *)malloc(sizeof(struct queue) * nqueues);
  int threads_per_queue = 1;
  ...
  //call queue_init with threads_per_queue
  queue_init(&anchor_que[0], QUEUE_SIZE, 1);
  queue_init(&send_que[0], QUEUE_SIZE, threads_per_queue);
  ...
  DataProcess(nqueues, fd); // enqueues chunks to the anchor queue

  //do the processing: the serial implementation combines the FindAllAnchors,
  //  ChunkProcess, and Compress stages for cache efficiency. Dequeues chunks
  //  from anchor queue, processes them, and enqueues segments to send queue
  SerialIntegratedPipeline(0);

  SendBlock(nqueues, conf); // dequeues from send queue and produces output

  queue_destroy(&anchor_que[0]);
  queue_destroy(&send_que[0]);
  ...
}
```

Listing 6.26: Deduplication encoding function for **dedup** kernel.

renamed to C++ files and any compilation issues resulting from the conversion were resolved. The code was also modified to comply with our own benchmarking tool and PARSEC specific code was removed.

The **dedup** kernel program was compiled using G++ with `-O0 -g` as compiler options and the timing measurements were repeated five times with a 184 MB data file (*dataset184.dat*) as the input. The initial timings and profiling results are summarised in Table 6.22. The program was profiled with a 31 MB data file (*dataset31.dat*) using Callgrind and Intel VTune Amplifier XE, as well as being checked for memory errors using Intel Inspector and Valgrind's Memcheck tool. Both VTune and Callgrind reported an L3 cache miss rate of less than 0.5%, while Callgrind reported a 0% L1 instruction miss rate and a 3.0% L1 data miss rate. Both Callgrind and VTune indicated that there was a high branch misprediction rate, with significant execution stalls.

However, drilling down into the profiling information using VTune revealed that most of these performance events originated from other modules, such as *libz* (compression), which accounts for 49.8% of the events, *libc* (memory allocation and copying), and *libcrypto* (SHA1 hash generation), with only 4.9% of the hardware performance events originating from the **dedup** code. These profiling results indicate that there is very little room for improving the original code through manual optimisation, since most of the performance issues are within libraries

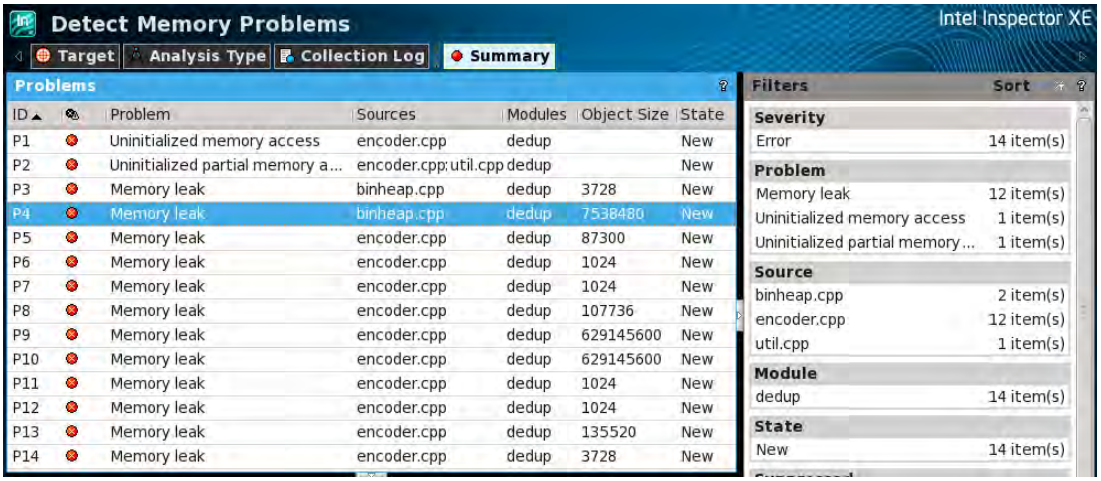


Figure 6.7: Memory errors detected by Intel Inspector in **dedup**.

provided by the system. This conclusion is supported by the high level optimisation report (`-opt-report` compiler option) produced by the Intel C++ compiler. Hotspot analysis shows that the `sub_Compress` function of the Compress stage accounts for the majority of CPU time. However, this CPU time is being spent inside libz's GZIP compression function.

Memory access error checking using Valgrind's Memcheck tool and Intel Inspector XE revealed several memory leaks in the original program (Figure 6.7). Some of these leaks were quite severe and resulted in over 600 MB of memory leakage by the **dedup** program. Intel Inspector's easy-to-use graphical interface proved indispensable in locating and resolving the memory issues. The Intel debugger was also found to be very useful in tracking down the memory issues through its single-stepping and variable watching features. The full version of the corrected serial pipeline implementation can be found in Appendix B.3.

Table 6.22: Runtime performance of the original, unoptimised sequential **dedup** program.

Mean Wall clock time	Geometric Mean	Standard Deviation
12.2234 s	12.22339 s	0.01645600

6.4.2 Sequential Optimisations

Since manual optimisations are unlikely to provide worthwhile benefits relative to the amount of effort required to implement them, only automatic compiler optimisations have been evaluated. The **dedup** program code was compiled using both the GNU C++ compiler and the Intel C++ compiler with increasing levels of optimisation enabled (`-O1`, `-O2`, and `-O3`). The performance summary for these compiler optimisations, using the GNU compiler (g++) and the Intel

compiler (icpc), is given in Table 6.23. In all cases, speedup is measured against the original sequential program results (Table 6.22).

Table 6.23: Performance of **dedup** with increasing compiler optimisation levels.

	Wall clock time (seconds)					
Compiler	g++ -O1	g++ -O2	g++ -O3	icpc -O1	icpc -O2	icpc -O3
Mean	10.814	10.8406	10.8206	10.6156	10.6252	10.615
Geo. Mean	10.81398	10.84060	10.82059	10.61559	10.62519	10.615
Std. Dev.	0.021012	0.011216	0.0175585	0.0183657	0.01467	0.001581
Speedup	1.13x	1.128x	1.13x	1.151x	1.15x	1.152x

In addition to the standard optimisations tested above, other advanced compiler optimisations such as interprocedural optimisation and profile-guided optimisation were tested. IPO was enabled for G++ using the `-O3`, `-fwhole-program`, and `-flto` compiler flags. The `-flto` option enables link-time optimisations, which expands the scope of interprocedural optimisations to multiple object files. IPO was enabled for the Intel compiler using the `-O3` and `-ipo` compiler flags. PGO was enabled using `-fprofile-generate` and `-fprofile-use` for G++, and `-prof-gen` and `-prof-use` for the Intel C++ compiler, along with the IPO and `-O3` compiler flags. The timing results with IPO and PGO enabled are given in Table 6.24.

Table 6.24: Performance of **dedup** with interprocedural and profile-guided optimisations.

	Wall clock time (seconds)			
Compiler	g++ w/ IPO	g++ w/ IPO + PGO	icpc w/ IPO	icpc w/ IPO + PGO
Mean	10.8066	10.7588	10.618	10.6042
Geo. Mean	10.80660	10.7588	10.618	10.6042
Std. Dev.	0.005899	0.0031145	0.004848	0.002588
Speedup	1.131x	1.136x	1.151x	1.153x

The performance results for the additional compiler optimisations show minor improvements over the original unoptimised version. Performance does not improve noticeably beyond the first optimisation level for each compiler (`-O1`). However, the highest performance is achieved using `-O3`, IPO, and PGO, even if the performance increase from `-O1` is minor. The Intel compiler produces a slightly faster executable (15.3% improvement) compared to GCC (13.6% improvement). Profiling of the optimised executables shows only minor improvements over the initial analysis for performance events such as instruction retire stalls. The `-O3`, IPO, and PGO compiler options are used for all further testing.

Automatic vectorisation using the SSE4.2 compiler options (`-msse4.2 -mfpmath=sse` for G++ and `-xSSE4.2` for the Intel compiler) was also tested. Unfortunately, there was no performance improvement from automatic SSE vectorisation, with the G++ compiled executable

giving a wall clock time of 10.8206 s and the icpc compiled executable taking 10.6148 s. The vectorisation report from the Intel compiler (`-vec-report`) revealed that the loops in the program did not provide worthwhile vectorisation or had loop carried dependencies that prevented vectorisation.

To ensure that the disk subsystem is not affecting the performance of the program, the data files were copied to and accessed from a ram-disk (storage space located in RAM), thereby eliminating the potential bottleneck of the system's slower hard disks. Then, the program was timed using each of the available data file sizes (10 MB, 31 MB, 184 MB, and 672 MB data files) and compared to the timing runs with the data files accessed from the hard disk. There was no difference in performance when using the ram-disk, which indicates that the disk subsystem does not impose a bottleneck on the program's performance.

Intel Integrated Performance Primitives

As noted in Section 6.4.1, calls to the libz compression library are a significant performance hotspot in the **dedup** kernel program. The Intel Integrated Performance Primitives (IPP) library provides an optimised drop-in replacement (no need to modify function calls) for the libz library [76]. To make use of the IPP library compression functions, the libz library was recompiled with IPP support (patches provided by Intel) and the IPP libz library was linked to the target program at compile time. The **dedup** kernel was benchmarked using the IPP libz library and compared to the performance of the program with the original libz library. To make use of the IPP libz library, the compiler linking options were changed to the following: `-lipp_z -lippcore -lippdc -lipps -liomp5 -lpthread`. Performance analysis reveals that the IPP version did not improve performance, which is likely due to the small segment sizes not providing enough computational work for each call to the compression function, thereby limiting any potential speedup. Additionally, the program output differs between the two versions. Therefore, the IPP optimised version of **dedup** has been excluded from further analysis.

6.4.3 Parallel Implementation

Unlike the previous two example programs, which were loop based task parallel problems, the deduplication kernel is implemented as a pipeline where data passes through several processing filters before the output emerges at the end of the pipeline. Deduplication favours the Data Decomposition pattern of the Identifying Concurrency design space, since the focus is on the data and how it is segmented and processed. The pipeline stages have already been described

in Section 6.4, which details the segmentation of the data by each pipeline stage and the use of a hash table to store compressed segments. Each pipeline stage can be viewed as a task that performs processing on a segment of data as it passes through the pipeline. Dependency analysis is straightforward as data segments can be processed independently by parallel instances of the same pipeline stage. However, there is a clear ordering dependency between adjacent stages in the pipeline. There is also an ordering constraint on the I/O operations as the data must be read and written in the correct order. Therefore, the pipeline stages can be grouped and ordered using the Group Tasks and Order Tasks patterns, with the input and output pipeline stages requiring sequential execution. The Data Sharing pattern is particularly important as adjacent stages communicate via shared producer-consumer style queues. The global hash table is also shared between all the tasks. These shared data structures must be protected using the appropriate synchronisation constructs.

As noted previously, deduplication organises tasks by data flow and, therefore, follows the Pipeline pattern of the Algorithm Structure design space. The deduplication pipeline has a primarily linear data flow (the compression stage can be skipped if the data associated with a particular hash value has been compressed previously) and since the computational intensity of the stages differ, the pipeline filter tasks can be grouped together on the same processor. The deduplication pipeline makes use of a wide variety of structures from the Supporting Structures design space. A combination of the Fork/Join and SPMD patterns is used to fork a pool of threads for each of the parallel pipeline stages, which follow the SPMD pattern and execute the same instructions on different data segments. The threads are joined once all of the data segments have been processed and the results have been combined and reordered. Since multiple threads are forked for each parallel stage, load balancing is handled by the operating system [14]. The Shared Data and Shared Queue patterns are also applicable as the global hash table must be protected from concurrent access through mutual exclusion and the appropriate care must be taken to protect the shared queues between pipeline stages. Conditional variables can be used to prevent busy-waiting by blocking threads while they wait for more data, thereby allowing other threads to continue until the sleeping threads are signalled that there is more data to process.

Finally, the parallel design described above must be implemented using the appropriate language constructs and parallel libraries, bringing us to the Implementation Mechanisms design space. The attempts at implementing the design using automatic parallelisation, OpenMP, Threading Building Blocks, Pthreads, Boost Threads, and Cilk Plus are presented below. However, certain parallel libraries are better suited to the implementation of the Pipeline pattern, while others do not provide all the necessary features for an effective implementation. The speedups listed for each implementation are relative to the best sequential performance for that

specific compiler.

Automatic Parallelisation

Automatic parallelisation proved unsuccessful for both the GNU C/C++ and Intel C/C++ compilers. In the case of the Intel compiler, the performance was worse than the sequential version. Consulting the compiler diagnostics and guided auto-parallelisation (`-par-report` and `-guide`) features of the Intel compiler revealed that there are very few loops that are suitable for automatic parallelisation and the loops that can be parallelised create performance overheads that outweigh the potential benefit of parallelisation.

Pthreads

The PARSEC benchmark suite provided an existing parallel implementation of the **dedup** kernel program using Pthreads that closely matches the parallel design described above [14]. Their implementation uses a separate thread pool for each parallel pipeline stage, where each thread pool contains at least as many threads as the number of available processor cores (originally specified at runtime, but modified so that it is specified at compile time using a preprocessor symbol). This allows any particular stage to make full use of the CPU if required [14]. Lock contention is avoided by scaling the number of queues between stages according to the number of threads and by giving each hash table entry its own lock. The `Encode` function is responsible for creating the queues and forking and joining the appropriate number of threads for each pipeline stage function as shown in Listing 6.27.

```
1  void Encode(config * conf) {
2    const int nqueues = (conf->nthreads / MAX_THREADS_PER_QUEUE) +
3                        ((conf->nthreads % MAX_THREADS_PER_QUEUE != 0) ? 1 : 0);
4    ... // Queue allocation
5    int threads_per_queue;
6    for(int i = 0; i < nqueues; i++) {
7      if (i < nqueues -1 || conf->nthreads %MAX_THREADS_PER_QUEUE == 0) {
8        //all but last queue
9        threads_per_queue = MAX_THREADS_PER_QUEUE;
10     } else //remaining threads work on last queue
11       threads_per_queue = conf->nthreads %MAX_THREADS_PER_QUEUE;
12     queue_init(&chunk_que[i], QUEUE_SIZE, threads_per_queue);
13     queue_init(&anchor_que[i], QUEUE_SIZE, 1);
14     queue_init(&send_que[i], QUEUE_SIZE, threads_per_queue);
15     queue_init(&compress_que[i], QUEUE_SIZE, threads_per_queue);
16   }
17
18   // Variables for 3 thread pools and 2 pipeline stage threads.
19   // The first and the last stage are serial (mostly I/O).
20   pthread_t threads_anchor[MAX_THREADS], threads_chunk[MAX_THREADS],
```

```
21      threads_compress[MAX_THREADS], threads_send, threads_process;
22
23    struct thread_args data_process_args;
24    .. // Set file handle and nqueues for data_process_args
25    pthread_create(&threads_process, NULL, DataProcess, &data_process_args);
26
27    int i;
28    struct thread_args chunk_thread_args[conf->nthreads];
29    for (i = 0; i < conf->nthreads; i ++) {
30      chunk_thread_args[i].tid = i;
31      pthread_create(&threads_chunk[i], NULL, ChunkProcess, &chunk_thread_args[i]);
32    }
33    struct thread_args anchor_thread_args[conf->nthreads];
34    for (i = 0; i < conf->nthreads; i ++) {
35       anchor_thread_args[i].tid = i;
36       pthread_create(&threads_anchor[i], NULL, FindAllAnchors, &anchor_thread_args[i]);
37    }
38    struct thread_args compress_thread_args[conf->nthreads];
39    for (i = 0; i < conf->nthreads; i ++) {
40      compress_thread_args[i].tid = i;
41      pthread_create(&threads_compress[i], NULL, Compress, &compress_thread_args[i]);
42    }
43
44    struct thread_args send_block_args;
45    // Set conf and nqueues for send_block_args
46    pthread_create(&threads_send, NULL, SendBlock, &send_block_args);
47
48    /*** parallel phase -- join all threads ***/
49    pthread_join(threads_process, NULL);
50    for (i = 0; i < conf->nthreads; i ++)
51      pthread_join(threads_anchor[i], NULL);
52    for (i = 0; i < conf->nthreads; i ++)
53      pthread_join(threads_chunk[i], NULL);
54    for (i = 0; i < conf->nthreads; i ++)
55      pthread_join(threads_compress[i], NULL);
56    pthread_join(threads_send, NULL);
57    ... // Destroy queues
58  }
```

Listing 6.27: Encoding function for parallel **dedup** using Pthreads.

The shared queue structure is manipulated using the `enqueue` and `dequeue` functions, which must ensure that only one thread is able to access a particular queue at a time. An example of the mutual exclusion constructs provided by Pthreads (mutex and condition variables) can be seen in Listing 6.28, which shows the queue initialisation and the `dequeue` function. A simple mutex in combination with a condition variable is sufficient for ensuring mutual exclusion without high lock contention. Reader-writer locks are not appropriate for the queue data structure as any accesses to the queue require both reads and writes.

The runtime performance of the Pthreads implementation using the GNU and Intel C/C++ compilers is shown in Table 6.25. While there is a definite performance improvement over the sequential version, the speedups of between 2.1x and 2.2x fall very short of linear speedup with the number of processors. This can be attributed to the sequential input and output stages. This is confirmed by profiling as illustrated in Figure 6.8, which shows how the wait times associated

```
1  void queue_init(struct queue * que, int size, int threads) {
2    pthread_mutex_init(&que->mutex, NULL);
3    pthread_cond_init(&que->empty, NULL);
4    pthread_cond_init(&que->full, NULL);
5    ... // Other queue data members
6  }
7
8  int dequeue(struct queue * que, int * fetch_count, void ** to_buf) {
9    pthread_mutex_lock(&que->mutex);
10   while (que->tail == que->head && (que->end_count) < que->threads)
11     pthread_cond_wait(&que->empty, &que->mutex);
12   if (que->tail == que->head && (que->end_count) == que->threads) {
13     pthread_cond_broadcast(&que->empty);
14     pthread_mutex_unlock(&que->mutex);
15     return -1;
16   }
17   for ((*fetch_count) = 0; (*fetch_count) < ITEM_PER_FETCH; (*fetch_count)++) {
18     to_buf[(*fetch_count)] = que->data[que->tail];
19     que->tail++;
20     if (que->tail == que->size) que->tail = 0;
21     if (que->tail == que->head) {
22       (*fetch_count)++;
23       break;
24     }
25   }
26   pthread_cond_signal(&que->full);
27   pthread_mutex_unlock(&que->mutex);
28   return 0;
29 }
```

Listing 6.28: Thread-safe queue using Pthreads mutexes and condition variables.

Table 6.25: Performance of Pthreads **dedup** implementation.

	Wall clock time (seconds)	
Compiler	g++ Pthreads	icpc Pthreads
Mean	4.8526	4.9878
Geo. Mean	4.85174	4.980707
Std. Dev.	0.1023562	0.3016947
Speedup	2.218x	2.126x

with the sequential stages affect concurrency. It is unlikely that this problem can be resolved without greatly increasing the program's complexity with more sophisticated thread scheduling. Profiling also reveals that there are no performance issues within **dedup** itself and that data sharing performance is good (data segments are copied between stages to ensure local access). The output produced by the Pthreads program is identical to the output for the sequential version. Error checking with Valgrind and Intel Inspector XE confirm that there are no threading or memory related problems.

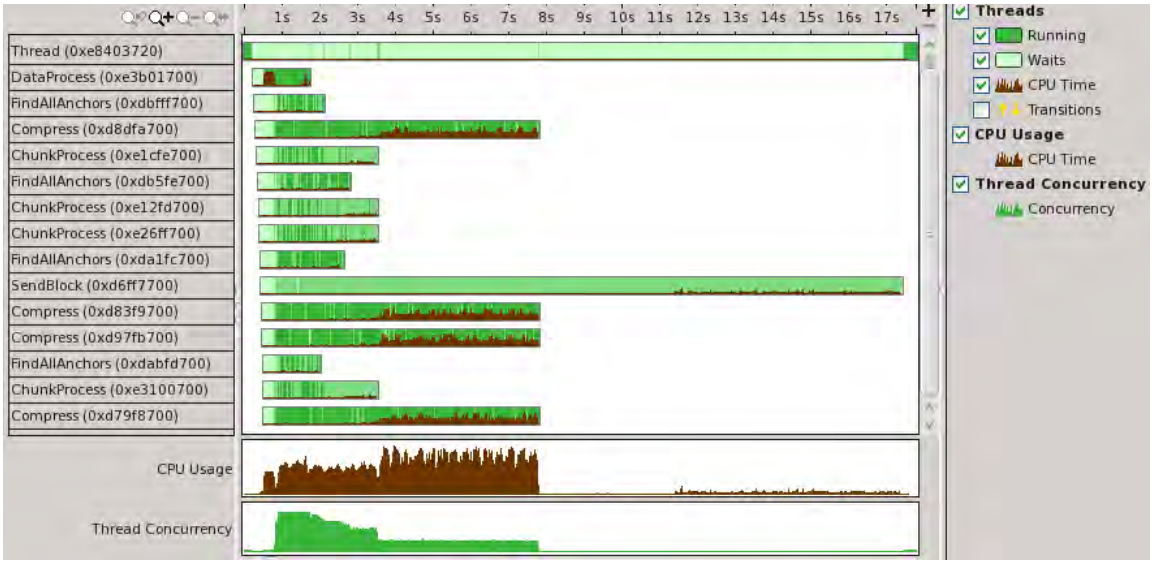


Figure 6.8: VTune thread concurrency analysis of Pthreads **dedup**.

Boost Threads

The Boost Threads implementation follows the same structure as the Pthreads version, replacing the Pthreads data types and function calls with Boost Threads objects and member functions. Boost simplifies certain aspects of the implementation by providing classes for thread groups and allowing function arguments to be passed directly to the function when creating threads as opposed to using void pointers and additional data structures. The `Encode` function is responsible for creating the queues and forking and joining the appropriate number of threads for each pipeline stage function as shown in Listing 6.29.

Examples of the mutual exclusion constructs provided by Boost (mutex, scoped locks, and condition variables) can be seen in Listing 6.30, which shows the queue initialisation and the `dequeue` function. The use of scoped lock objects prevents deadlock if an exception is thrown while a lock is held.

Since the Boost Threads version follows the same design as the Pthreads version, the performance is essentially the same, as evidenced by the performance results in Table 6.26. The profiling results also indicate that there is very little difference in terms of the performance counters data. The output produced by the Boost Threads program is identical to the output for the sequential version and Valgrind and Intel Inspector XE did not detect any threading or memory related errors.

```
1  void Encode(config * conf) {
2    ... // Open data file
3    ... // Queue allocation & initialization
4
5    // Variables for 3 thread pools and 2 pipeline stage threads.
6    // The first and the last stage are serial (mostly I/O).
7    boost::thread_group threads_anchor, threads_chunk, threads_compress;
8    boost::thread * threads_send = NULL;
9    boost::thread * threads_process = NULL;
10   int i;
11
12   // Thread for first pipeline stage (input)
13   threads_process = new boost::thread(DataProcess, nqueues, fd);
14   // Create 3 thread pools for the intermediate pipeline stages
15   for (i = 0; i < conf->nthreads; i++)
16     threads_chunk.add_thread(new boost::thread(ChunkProcess, i));
17   for (i = 0; i < conf->nthreads; i++)
18     threads_anchor.add_thread(new boost::thread(FindAllAnchors, i));
19   for (i = 0; i < conf->nthreads; i++)
20     threads_compress.add_thread(new boost::thread(Compress, i));
21   // Thread for last pipeline stage (output)
22   threads_send = new boost::thread(SendBlock, nqueues, conf);
23
24   /*** parallel phase -- join all threads ***/
25   (*threads_process).join();
26   threads_anchor.join_all();
27   threads_chunk.join_all();
28   threads_compress.join_all();
29   (*threads_send).join();
30
31   ... // Destroy queues
32
33   delete threads_send;
34   delete threads_process;
35 }
```

Listing 6.29: Encoding function for parallel **dedup** using Boost Threads.

OpenMP

In the previous examples, OpenMP has been shown to be an easy to implement alternative to explicit threading. However, OpenMP does not provide native support for pipeline based processing, which makes implementing such a design more difficult. Research is underway to add pipeline or stream based processing support to OpenMP, but these features are not part of the current standard and are only supported in experimental builds of the GNU C/C++ compiler [121]. As such, the approach to implementing an OpenMP version of **dedup** is similar to that of Pthreads and Boost Threads. The OpenMP version uses the same queue structures between the pipeline stages and spawns OpenMP tasks for each pipeline stage as shown in Listing 6.31.

As with the Boost Threads and Pthreads implementations, access to the queue structures is protected via mutual exclusion using locks as demonstrated in Listing 6.32. The OpenMP runtime

```
1  void queue_init(struct queue * que, int size, int threads) {
2    que->mutex = new boost::mutex;
3    que->empty = new boost::condition_variable;
4    que->full = new boost::condition_variable;
5    ... // Other queue data members
6  }
7
8  int dequeue(struct queue * que, int * fetch_count, void ** to_buf) {
9    boost::unique_lock<boost::mutex> lock(*que->mutex);
10   while (que->tail == que->head && (que->end_count) < que->threads) {
11     que->empty->wait(lock);
12   }
13   if (que->tail == que->head && (que->end_count) == que->threads) {
14     que->empty->notify_all();
15     return -1;
16   }
17   for ((*fetch_count) = 0; (*fetch_count) < ITEM_PER_FETCH; (*fetch_count)++) {
18     to_buf[(*fetch_count)] = que->data[que->tail];
19     que->tail++;
20     if (que->tail == que->size) que->tail = 0;
21     if (que->tail == que->head) {
22       (*fetch_count)++;
23       break;
24     }
25   }
26   que->full->notify_one();
27   return 0;
28 }
```

Listing 6.30: Thread-safe queue using Boost Threads mutexes and condition variables.

Table 6.26: Performance of Boost Threads **dedup** implementation.

	Wall clock time (seconds)	
Compiler	g++ Boost Threads	icpc Boost Threads
Mean	4.9458	4.9156
Geo. Mean	4.94401	4.910308
Std. Dev.	0.1491147	0.2597697
Speedup	2.176x	2.16x

provides lock types (`omp_lock_t`) and functions for locking (`omp_set_lock`) and unlocking (`omp_unset_lock`) these locks. However, OpenMP does not provide condition variables. This functionality was replicated by spinning on a locally cached copy of a shared variable and flushing memory to receive updated values for the variable. The lock is released while the thread spins and is acquired again when the thread is signalled to continue.

Performance testing reveals improved speedup over the Pthreads and Boost Threads implementations for both compilers (Table 6.27). This is likely due to the improved task scheduling implemented by the OpenMP runtime system. However, the scaling is still far from perfect. The GNU OpenMP implementation shows significantly better performance than the Intel implementation in this case.

```
1  void Encode(config * conf) {
2    ... // Open data file
3    ... // Queue allocation & initialization
4
5    #pragma omp parallel default(none) shared (fd, conf)
6    {
7        #pragma omp single nowait
8        {
9            int i;
10           #pragma omp task firstprivate(fd)
11           DataProcess(nqueues, fd);
12
13           for (i = 0; i < conf->nthreads; i++) {
14               #pragma omp task firstprivate(i)
15               FindAllAnchors(i);
16           }
17           for (i = 0; i < conf->nthreads; i++) {
18               #pragma omp task firstprivate(i)
19               ChunkProcess(i);
20           }
21           for (i = 0; i < conf->nthreads; i++) {
22               #pragma omp task firstprivate(i)
23               Compress(i);
24           }
25
26           #pragma omp task firstprivate(conf)
27           SendBlock(nqueues, conf);
28   }  }
29   ... // Destroy queues
30 }
```

Listing 6.31: Encoding function for parallel **dedup** using OpenMP.

The output from the OpenMP **dedup** kernel is identical to the output for the sequential version, so there are no apparent defects in the parallel implementation of the program. Since condition variables have been approximated, it is very important to ensure that there are no deadlock or data races problems. Fortunately, this is the case as Intel Inspector XE did not detect any threading errors.

Cilk Plus

As with OpenMP, Cilk Plus is better suited to data and simple task based parallelism. Unfortunately, Cilk Plus does not provide any explicit mutual exclusion constructs, which makes it unsuitable for implementing a Cilk Plus version of the **dedup** kernel program. While the queue structure can be implemented using Cilk Plus reducers [73], the locks and condition variables required by the hash table are not supported. It is possible to use the mutex constructs provided by other threading libraries such as Threading Building Blocks and Pthreads [73], but it was decided such a path would skew the validity of the experiment.

```
1  void queue_init(struct queue * que, int size, int threads) {
2    omp_init_lock(&que->mutex);
3    que->empty = 0;
4    que->full = 0;
5    ... // Other queue data members
6  }
7
8  int dequeue(struct queue * que, int * fetch_count, void ** to_buf) {
9    omp_set_lock(&que->mutex);
10   while (que->tail == que->head && (que->end_count) < que->threads) {
11     omp_unset_lock(&que->mutex);
12     #pragma omp flush
13     while (que->empty != 1) {
14         #pragma omp flush
15     }
16     omp_set_lock(&que->mutex);
17     que->empty = 0;
18     #pragma omp flush
19   }
20   if (que->tail == que->head && (que->end_count) == que->threads) {
21     #pragma omp flush
22     que->empty = 1;
23     #pragma omp flush
24     omp_unset_lock(&que->mutex);
25     return -1;
26   }
27   for ((*fetch_count) = 0; (*fetch_count) < ITEM_PER_FETCH; (*fetch_count)++) {
28     to_buf[(*fetch_count)] = que->data[que->tail];
29     que->tail++;
30     if (que->tail == que->size) que->tail = 0;
31     if (que->tail == que->head) {
32       (*fetch_count)++;
33       break;
34     }
35   }
36   #pragma omp flush
37   que->full = 1;
38   #pragma omp flush
39   omp_unset_lock(&que->mutex);
40   return 0;
41 }
```

Listing 6.32: Thread-safe queue using OpenMP locks.

Thread Building Blocks

The Threading Building Blocks implementation of **dedup** differs substantially from the other implementations described above. TBB provides explicit support for pipeline based processing through its `pipeline` template. A TBB pipeline is implemented by defining filter classes to represent the stages of the pipeline. These filter classes must provide an overloaded `operator()` function that accepts an input token from a previous stage and returns a token to the following stage. This means that there is no longer a need for the shared queues used by the other implementations as this is handled by the Threading Building Blocks runtime system. As shown in Listing 6.33, an instance of each pipeline filter class is declared and added to the pipeline object in the appropriate order. The pipeline is then instructed to begin execution by calling its `run`

Table 6.27: Performance of OpenMP **dedup** implementation.

	Wall clock time (seconds)	
Compiler	g++ OpenMP	icpc OpenMP
Mean	3.3894	4.0978
Geo. Mean	3.382169	4.096963
Std. Dev.	0.2554981	0.09353983
Speedup	3.181x	2.588x

function with the desired number of simultaneous tokens. This allows the programmer to specify how many tokens the pipeline can have in flight at any one time, thereby limiting the number of tokens produced by the input stage until a token has reached the end of the pipeline [125]. This ensures that the pipeline is never oversubscribed. After the pipeline execution has completed, the filters are cleared and the program can exit.

```
1  void Encode(config * conf) {
2    ... // Open data file
3
4    tbb::task_scheduler_init init; // initialise the TBB scheduler
5    tbb::pipeline pipeline;        // create the TBB pipeline
6
7    // create and add the data processing / file reading stage to the pipeline
8    DataProcessFilter data_process_stage(fd);
9    pipeline.add_filter(data_process_stage);
10
11   // create and add the find all anchors stage to the pipeline
12   FindAllAnchorsFilter find_anchors_stage;
13   pipeline.add_filter(find_anchors_stage);
14
15   // create and add the chunk process stage to the pipeline
16   ChunkProcessFilter chunk_process_stage;
17   pipeline.add_filter(chunk_process_stage);
18
19   // create and add the compress stage to the pipeline
20   CompressFilter compress_stage;
21   pipeline.add_filter(compress_stage);
22
23   // create the send block stage to the pipeline
24   SendBlockFilter send_block_stage(conf);
25   pipeline.add_filter(send_block_stage);
26
27   // start the pipeline and clear the filters when done
28   pipeline.run(conf->nthreads);
29   pipeline.clear();
30 }
```

Listing 6.33: Encoding function for parallel **dedup** using TBB.

Since the TBB implementation no longer makes use of the queue structures between filter stages, the implementation of the filter functions had to modified to support the updated pipeline design. An example filter is provided in Listing 6.34, which shows the ChunkProcess stage implemented as a TBB filter class. The filter class inherits from `tbb::filter` and implements

the overloaded `operator()` function that accepts an input token from the previous stage. The initialisation statement, `tbb::filter(false)`, in the class constructor indicates that this stage can execute in parallel. This is set to true in the input and output stages to prevent parallel execution [125]. Also shown in the example is the use of the mutex and scoped lock constructs provided by TBB to protect access to the shared hash table. While the TBB implementation of **dedup** required extensive modifications to the original code, it is the subjective opinion of the author that the TBB version of the program would be easier to implement than the other versions if the various parallel versions of the program were being implemented from scratch. This is echoed by similar research carried out by Kegel *et al.* [86, 87].

```
class ChunkProcessFilter: public tbb::filter {
private:
    send_buf_item * sub_ChunkProcess(data_chunk chunk) {
      send_buf_item * item; send_body * body = NULL;
      u_char * key = (u_char *)malloc(SHA1_LEN);
      Calc_SHA1Sig(chunk.start, chunk.len, key);

      struct hash_entry * entry;
      /* search the cache */
      tbb::mutex *ht_lock = hashtable_getlock(cache, (void *)key);
      tbb::mutex::scoped_lock lock (*ht_lock);
      if ((entry = hashtable_search(cache, (void *)key)) == NULL) {
        // Cache miss: put it in the hashtable and flag for compression
      } else {
        // Cache hit: flag as a fingerprint and exclude from compression
      }
      if (chunk.start) MEM_FREE(chunk.start);
      return item;
    }
public:
    ChunkProcessFilter() : tbb::filter(false) {}

    void * operator()(void * itemm) {
      anchored_data_chunks * inchunks = (anchored_data_chunks *)itemm;
      send_buf_list * ret;
      ret = (send_buf_list *)malloc(sizeof(send_buf_list));
      ret->size = inchunks->size;
      ret->items = new send_buf_item *[inchunks->size];

      for (int i = 0; i < inchunks->size; i++) {
          data_chunk chunk;
          chunk.start = inchunks->chunks[i]->start;
          chunk.len = inchunks->chunks[i]->len;
          chunk.cid = inchunks->chunks[i]->cid;
          chunk.anchorid = inchunks->chunks[i]->anchorid;

          ret->items[i] = sub_ChunkProcess(chunk);
          MEM_FREE(inchunks->chunks[i]);
      }
      delete [] inchunks->chunks;
      MEM_FREE(inchunks);
      return ret;
    }
}; // end of ChunkProcessFilter class
```

Listing 6.34: ChunkProcess filter class for parallel **dedup** using TBB.

The performance of the TBB implementation (Table 6.28) is significantly better than that of the other parallel implementations with a speedup close to 3.4x, which is acceptable given the serial input and output stages. The Intel compiler produces a slightly faster program, but the difference is negligible. The improved performance can be attributed to Threading Building Blocks's explicit pipeline support and efficient runtime scheduler [125]. Error checking reveals that the output is correct and analysis with Intel Inspector XE confirms that there are no threading issues.

Table 6.28: Performance of TBB **dedup** implementation.

	Wall clock time (seconds)	
Compiler	g++ TBB	icpc TBB
Mean	3.1722	3.1594
Geo. Mean	3.171457	3.158701
Std. Dev.	0.07759317	0.07503533
Speedup	3.392x	3.357x

Chapter 7

Results

The runtime performance results and static code metrics for **matrixmul**, **mandelbrot**, and **dedup** are presented below, along with an analysis and discussion of the results. The Halstead Length (N) and Volume (V) and McCabe's cyclomatic complexity (v(G)) metrics were measured using Crystal FLOW for C++[1]. For **matrixmul** and **mandelbrot**, the source lines of code (SLOC) measurements were calculated by hand. The line counts and code differences for **dedup** were gathered using the Unified CodeCount (UCC)[2] tool due to the much larger code size of the program compared to the previous examples. Performance measurements were gathered using the benchmarking tool described in Section 5.3.

The compiler options used to compile each program for the respective compilers, optimisation levels, and parallel programming model are listed below. Modifications to the listed options for a particular program or implementation are described in the respective sections. For the sequential compiler optimisations, the speedup for each test is calculated against the G++ compiler with optimisations disabled (`-O1`). Whereas for the parallel implementations, the speedup for each test is calculated against the fastest sequential performance using the same compiler as was used for the compilation of the parallel program. This is to ensure that the speedup is not affected by the performance difference between compilers.

- **Common compiler options –** `-fomit-frame-pointer -fno-builtin -pipe`
- **O1 –** g++: `-O1`
  icpc: `-O1`

[1] `http://www.sgvsarc.com/Prods/CFLOW/Crystal_FLOW.htm`
[2] `http://sunset.usc.edu/research/CODECOUNT/`

- **O2 –** g++: `-O2`
  icpc: `-O2`
- **O3 –** g++: `-O3`
  icpc: `-O3`
- **IPO –** g++: `-fwhole-program`
  icpc: `-ipo`
- **SSE –** g++: `-msse4.2 -mfpmath=sse`
  icpc: `-xSSE4.2`
- **PGO –** g++: `-fprofile-generate / -fprofile-use`
  icpc: `-prof-gen / -prof-use`
- **Auto-parallelisation –** icpc: `-parallel -par-runtime-control -par-schedule-auto -par-num-threads=<n>`
- **Boost Threads –** g++: `-lboost_thread-mt -DNTHREADS=n`
  icpc: `-lboost_thread-mt -DNTHREADS=n`
- **Cilk Plus –** icpc: `-DNTHREADS=n`
- **OpenMP –** g++: `-fopenmp -DNTHREADS=<n>`
  icpc: `-openmp -DNTHREADS=<n>`
- **Pthreads –** g++: `-pthread -DNTHREADS=<n>`
  icpc: `-pthread -DNTHREADS=<n>`
- **Intel TBB –** g++: `-ltbb -std=c++0x -DNTHREADS=<n>`
  icpc: `-tbb -std=c++0x -DNTHREADS=<n>`
- **Intel MKL –** g++: `-lmkl_intel_ilp64 -lmkl_gnu_thread -lmkl_core -fopenmp -lpthread`
  icpc: `-mkl`

7.1 Matrix Multiplication

7.1.1 Runtime Performance

A runtime performance summary of the sequential compiler optimisations using the GNU C/C++ (g++) and Intel C/C++ (icpc) compilers for **matrixmul** is given in Table 7.1, while the performance of the various parallel implementations, with number of threads (n) equal to four, is given in Table 7.2. All measurements were taken with a matrix dimension of 2048x2048. Figure 7.1 shows a set of boxplots comparing the performance of the parallel implementations and C/C++ compilers. The parallel programs were compiled with the O3, IPO, and SSE options in addition to the options listed for the respective parallel model or library. An analysis of the results is performed in Section 7.1.3.

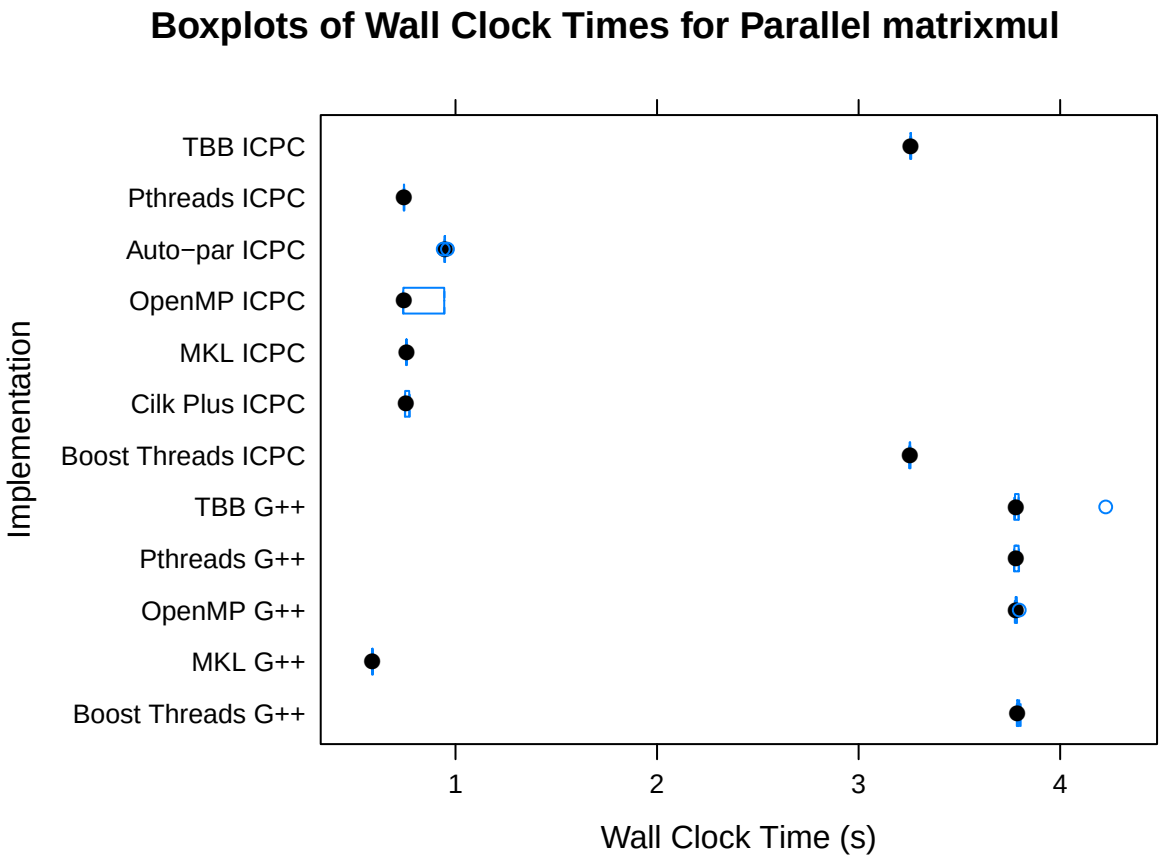


Figure 7.1: Runtime performance for parallel implementations of **matrixmul** using the GNU and Intel C/C++ compilers.

Table 7.1: Performance of compiler optimisations for sequential **matrixmul** program.

Optimisation	Compiler	Wall Clock Time (s)				Speed-up	CPU Time (s)	CPU Load	Memory Usage (MB)	Size (KB)
		Mean	Geo. Mean	Std. Dev.	Var.		Mean	Mean	Mean	Mean
-O0	g++	79.5802	79.580200	0.0080436	0.0000647	–	79.524	25%	97.060	0.554
	icpc	106.3374	106.337400	0.0058992	0.0000348	0.75	106.276	25%	97.073	0.554
-O1	g++	41.4922	41.492200	0.0019235	0.0000037	1.92	41.455	25%	97.061	0.554
	icpc	44.6620	44.662000	0.0035355	0.0000125	1.78	44.622	25%	97.073	0.554
-O2	g++	14.5476	14.547996	0.0039115	0.0000153	5.47	14.531	25%	97.060	0.554
	icpc	40.9238	40.923799	0.0099850	0.0000997	1.94	40.874	25%	97.072	0.554
-O3	g++	14.5446	14.544998	0.0029665	0.0000088	5.47	14.530	25%	97.060	0.554
	icpc	41.3378	41.337800	0.0010954	0.0000012	1.93	41.301	25%	97.072	0.554
-O3 IPO	g++	14.5420	14.519996	0.0038730	0.0000150	5.48	14.527	25%	97.063	0.554
	icpc	7.4098	7.409800	0.0014832	0.0000022	10.74	7.401	25%	97.074	0.554
-O3 IPO SSE	g++	14.5626	14.562600	0.0020736	0.0000043	5.46	14.550	25%	97.060	0.554
	icpc	7.4164	7.416396	0.0089051	0.0000793	10.73	7.405	25%	97.072	0.554
-O3 IPO SSE PGO	g++	14.5750	14.575000	0.0024495	0.0000060	5.46	14.562	25%	97.061	0.554
	icpc	61.4730	61.472989	0.0406510	0.0016525	1.29	61.423	25%	97.081	0.554

Table 7.2: Performance of parallel **matrixmul** implementations.

Parallel Model	Compiler	Wall Clock Time (s)				Speed-up	CPU Time (s)	CPU Load	Memory Usage (MB)	Binary Size (KB)
		Mean	Geo. Mean	Std. Dev.	Var.		Mean	Mean	Mean	Mean
Sequential -O3 IPO	g++	14.5420	14.519996	0.0038730	0.0000150	–	14.527	25%	97.063	0.554
	icpc	7.4098	7.409800	0.0014832	0.0000022	1.96	7.401	25%	97.074	0.554
Auto-parallelisation	icpc	0.9480	0.947975	0.0076485	0.0000585	7.82	2.847	75%	97.678	0.554
Boost Threads	g++	3.7920	3.791993	0.0081854	0.0000670	3.83	14.679	97%	97.169	0.781
	icpc	3.2544	3.254399	0.0024083	0.0000058	2.28	12.583	97%	97.183	0.781
Cilk Plus	icpc	0.7592	0.759131	0.0114324	0.0001307	9.76	2.596	86%	97.589	0.649
Intel MKL	g++	0.5878	0.587798	0.0016432	0.0000027	24.70	1.917	82%	106.126	0.531
	icpc	0.7566	0.756599	0.0011402	0.0000013	9.79	1.933	68%	110.469	0.531
OpenMP	g++	3.7832	3.783192	0.0084971	0.0000722	3.84	14.696	97%	97.138	0.634
	icpc	0.8228	0.816969	0.1111067	0.0123447	9.07	2.530	79%	97.667	0.634
Pthreads	g++	3.7830	3.782986	0.0115974	0.0001345	3.84	14.663	97%	97.099	0.785
	icpc	0.7444	0.744399	0.0011402	0.0000013	9.95	2.553	86%	97.111	0.785
Intel TBB	g++	3.8698	3.865872	0.1992943	0.0397182	3.76	14.755	96%	97.811	0.703
	icpc	3.2584	3.258400	0.0020736	0.0000043	2.27	12.565	97%	97.832	0.703

7.1.2 Code Metrics

Table 7.3: Static code metrics for the implementations of the **matrixmul** program.

Implementation	SLOC	Added	Removed	Modified	N	V	v(G)
Sequential	48	–	–	–	645.03	180	10
Auto-parallelisation	48	0	0	0	645.03	180	10
Boost	62	15	1	2	877.06	216	11
Cilk Plus	58	9	0	1	747.20	191	9
Intel MKL	33	3	18	14	533.57	141	5
OpenMP	52	4	0	0	727.44	190	10
Pthreads	63	16	1	2	913.73	226	12
Intel TBB	57	17	10	1	821.85	207	9

7.1.3 Discussion

It is clear from the sequential performance results (Table 7.1) that the increasing levels of compiler optimisations are very effective at improving the runtime performance of the **matrixmul** program. However, the GNU C++ compiler appears to perform more effective optimisations than the Intel compiler without interprocedural optimisations (IPO) enabled. With IPO enabled, the Intel compiler takes the lead, almost doubling the speedup attained by the respective G++ compiled executable. This indicates that the G++ compiler performs more aggressive single file optimisations at the standard optimisation levels. Other compiler-based optimisations such as automatic SSE vectorisation and profile-guided optimisation (PGO) had little effect on performance and in the case of the Intel compiler, PGO substantially reduced the performance relative to just using IPO. The remaining characteristics, such as memory usage and CPU load (25% usage represents one processor core with 100% load), are essentially the same between compilers.

The performance of the parallel implementations vary quite substantially between parallel models and, in some cases, between the compiler used for a particular parallel programming model or library. The overall best performance is achieved using the Intel Math Kernel Library (MKL), which is an optimised mathematics library as opposed to a parallel programming model. This shows that it is worthwhile looking for the required functionality in existing optimised libraries instead of developing custom solutions. Of the two compilers, the GNU C/C++ compiler has the best performance when used in conjunction with MKL. Beyond the MKL library, it is clear that

the Intel compiler is the best choice for parallel implementations of **matrixmul**. This is likely due to the improved SSE vectorisation performed by the Intel compiler, which, in combination with favourable caching effects, resulted in speedups far in excess of linear speedup. As stated in Section 6.2.2, SSE vectorisation was only effective for the parallel implementations and had no effect on the sequential code. The Boost Threads and Threading Building Blocks implementations do not appear to have benefited from automatic vectorisation, which is likely due to the use of C++ templates affecting the ability of the compiler to perform the vectorisation, since Boost and TBB are the only libraries that make use of templates.

Of the parallel programming models, Pthreads and Cilk Plus had the best runtime performance. If we compare Boost Threads and Pthreads using G++, we find that their performance is effectively the same. This is to be expected as Boost Threads essentially provides an object-oriented interface for Pthreads in C++. Another particularly interesting result is that of the automatic parallelisation by the Intel compiler (automatic parallelisation using G++ was ineffective and excluded from analysis), which produced excellent speedup for little to no additional effort. The CPU load characteristics indicate that many of the parallel implementations make good use of the available processor cores.

The code metrics for the various implementations of the **matrixmul** program show that the explicit threading approaches (Boost Threads and Pthreads) required more substantial changes and resulted in a more complex program when compared to the other threading approaches. This indicates that the lower level of abstraction provided by these explicit threading libraries results in more programming effort required to implement the respective parallel implementations. However, it must be noted that for the Cilk Plus implementation, the code required to limit the number of available threads artificially increases the extent of modification compared to the other implementations where such functionality fits seamlessly into other required modifications. Despite this, the high level of abstraction provided by Cilk Plus results in a lower program complexity and minimal increase in program volume compared to other parallel implementations. As mentioned previously, automatic parallelisation requires additional programming effort to implement as the compiler performs all the work.

7.2 Mandelbrot Set Algorithm

7.2.1 Runtime Performance

A runtime performance summary of the sequential compiler optimisations using the GNU C/C++ (g++) and Intel C/C++ (icpc) compilers for **mandelbrot** is given in Table 7.4, while the performance of the various parallel implementations, with number of threads (n) equal to four, is given in Table 7.5. All measurements were taken with an input dimension of 16000. Figure 7.2 shows a set of boxplots comparing the performance of the parallel implementations using each of the C/C++ compilers.

For the presented performance data, "Dynamic (Dyn.)" refers to dynamic thread scheduling and "Static" refers to static thread scheduling. All tests are compiled with SSE enabled (due to the explicit use of SSE intrinsic functions) and the parallel programs are compiled with O3 and IPO in addition to SSE and the listed options. The results for the **mandelbrot** program are analysed and discussed in Section 7.2.3.

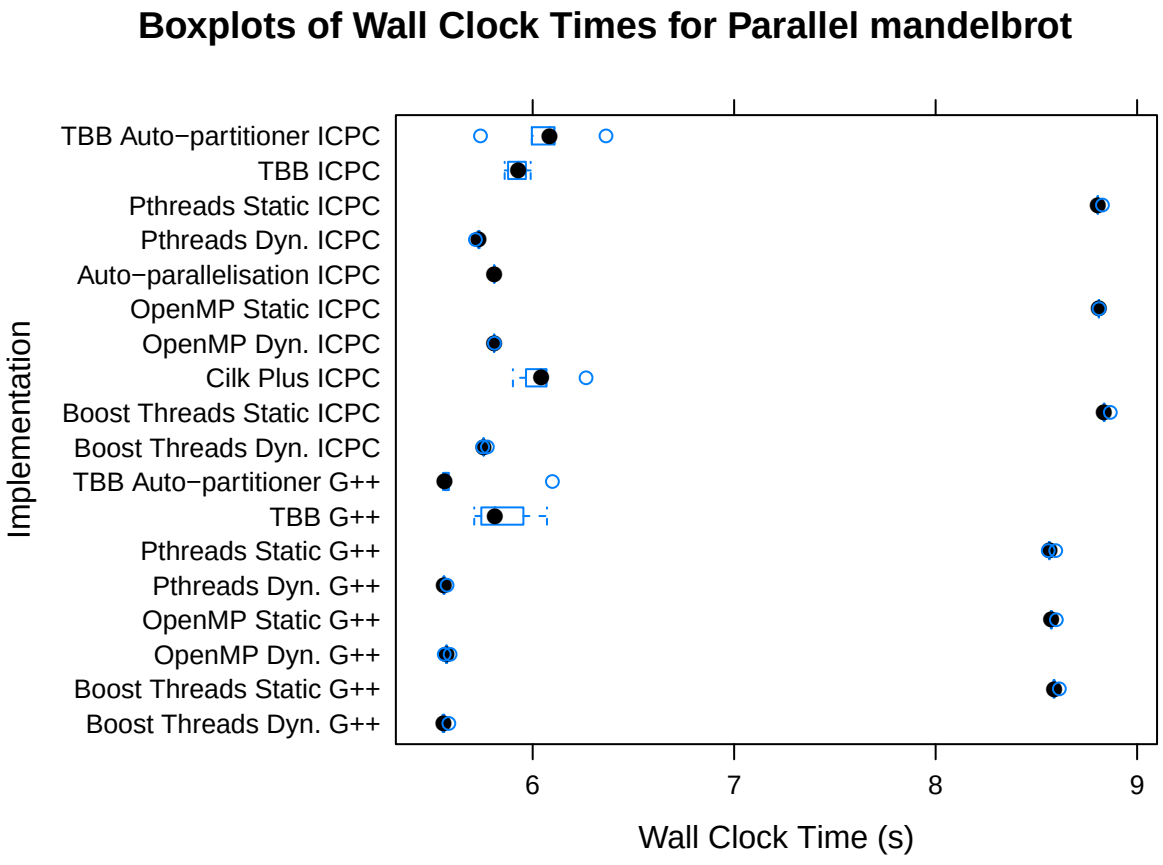


Figure 7.2: Runtime performance for parallel implementations of **mandelbrot** using the GNU and Intel C/C++ compilers.

Table 7.4: Performance of compiler optimisations for sequential **mandelbrot** program.

Optimisation	Compiler	Wall Clock Time (s)				Speed-up	CPU Time (s)	CPU Load	Memory Usage (MB)	Binary Size (KB)
		Mean	Geo. Mean	Std. Dev.	Var.		Mean	Mean	Mean	Mean
-O0	g++	92.8840	92.883999	0.0110000	0.0001210	–	92.820	25%	30.550	0.866
	icpc	80.2872	80.287199	0.0134052	0.0001797	1.16	80.244	25%	30.407	0.866
-O1	g++	22.5308	22.530799	0.0089833	0.0000807	4.12	22.516	25%	28.794	0.866
	icpc	22.7254	22.725397	0.0126610	0.0001603	4.09	22.712	25%	29.272	0.866
-O2	g++	22.2540	22.253997	0.0123895	0.0001535	4.17	22.243	25%	29.259	0.866
	icpc	22.8234	22.823397	0.0122801	0.0001508	4.07	22.810	25%	28.757	0.866
-O3	g++	22.2614	22.261398	0.0105024	0.0001103	4.17	22.249	25%	29.259	0.866
	icpc	22.8184	22.818397	0.0121367	0.0001473	4.07	22.807	25%	28.498	0.866
-O3 IPO	g++	22.2510	22.250999	0.0084261	0.0000710	4.17	22.238	25%	29.312	0.866
	icpc	22.8282	22.828198	0.0095760	0.0000917	4.07	22.811	25%	28.755	0.866
-O3 IPO PGO	g++	24.1028	24.102798	0.0108950	0.0001187	3.85	24.093	25%	29.106	0.866
	icpc	27.9436	27.943468	0.0961681	0.0092483	3.32	27.930	25%	30.204	0.866

Table 7.5: Performance of parallel **mandelbrot** implementations.

Parallel Model	Compiler	Wall Clock Time (s)				Speed-up	CPU Time (s)	CPU Load	Memory Usage (MB)	Binary Size (KB)
		Mean	Geo. Mean	Std. Dev.	Var.		Mean	Mean	Mean	Mean
Sequential -O3 IPO	g++	22.2510	22.250999	0.0084261	0.0000710	–	22.238	25%	29.312	0.866
	icpc	22.8282	22.828198	0.0095760	0.0000917	0.97	22.811	25%	28.755	0.866
Auto-parallelisation	icpc	5.8094	5.809400	0.0005477	0.0000003	3.93	22.661	98%	26.827	0.866
Boost Threads Static	g++	8.5928	8.592793	0.0120291	0.0001447	2.59	22.2388	65%	30.459	1.150
	icpc	8.8420	8.841991	0.0140535	0.0001975	2.58	22.949	65%	30.998	1.150
Boost Threads Dyn.	g++	5.5620	5.561988	0.0127083	0.0001615	4.00	22.1174	100%	25.225	1.558
	icpc	5.7592	5.759194	0.0090388	0.0000817	3.96	22.899	100%	25.500	1.558
Cilk Plus	icpc	6.0488	6.047565	0.1372942	0.0188497	3.77	24.112	100%	30.998	0.962
OpenMP Static	g++	8.5788	8.578794	0.0113886	0.0001297	2.59	22.4612	66%	30.351	0.937
	icpc	8.8102	8.810200	0.0004472	0.0000002	2.59	23.119	66%	30.960	0.937
OpenMP Dynamic	g++	5.5736	5.573592	0.0105499	0.0001113	3.99	22.2648	100%	25.130	0.945
	icpc	5.8092	5.809200	0.0004472	0.0000002	3.93	22.734	99%	26.719	0.945
Pthreads Static	g++	8.5698	8.569790	0.0148560	0.0002207	2.60	22.169	65%	30.377	1.192
	icpc	8.8092	8.809195	0.0105688	0.0001117	2.59	22.838	65%	30.495	1.192
Pthreads Dynamic	g++	5.5622	5.562196	0.0073959	0.0000547	4.00	22.116	100%	25.151	1.600
	icpc	5.7300	5.729995	0.0080932	0.0000655	3.98	22.773	100%	25.422	1.600
TBB	g++	5.8586	5.857052	0.1510639	0.0228203	3.80	22.380	96%	31.232	1.106
	icpc	5.9248	5.924594	0.0552874	0.0030567	3.85	23.146	98%	31.193	1.106
TBB Auto Partitioner	g++	5.6718	5.667931	0.2384905	0.0568777	3.93	22.157	98%	30.666	1.112
	icpc	6.0586	6.055270	0.2242450	0.0502858	3.77	23.111	96%	31.245	1.112

7.2.2 Code Metrics

Table 7.6: Static code metrics for the implementations of the **mandelbrot** program.

Implementation	SLOC	Added	Removed	Modified	N	V	v(G)
Sequential	63	–	–	–	208	914.05	9
Auto-parallelisation	64	1	0	0	209	921.00	9
Boost Static	87	25	1	8	263	1233.82	10
Boost Dynamic	152	90	1	4	412	2027.03	25
Cilk Plus	73	9	0	1	220	1004.16	8
OpenMP Static	67	4	0	0	214	955.70	9
OpenMP Dynamic	67	4	0	0	217	976.46	9
Pthreads Static	96	34	1	8	309	1438.07	11
Pthreads Dynamic	159	97	1	8	455	2221.67	23
TBB Explicit Partioning	82	19	0	9	241	1104.98	9
TBB Auto Partitioner	82	19	0	9	239	1093.36	9

7.2.3 Discussion

Unlike **matrixmul**, the performance difference between compilers and optimisation levels for **mandelbrot** from level `-O1` and higher are marginal, excluding PGO, which once again results in reduced performance. That being said, the GNU C++ compiler produces faster executables in terms of sequential runtime performance compared to the Intel compiler. It is immediately apparent from the performance comparison of the various parallel implementations (Figure 7.2) that the dynamically scheduled versions perform markedly better than their statically scheduled counterparts. Comparing the CPU load between implementations (Table 7.5) shows that the statically scheduled versions only use two thirds of the available processing power, whereas dynamic scheduling makes full use of the four processor cores. Additionally, the dynamically scheduled programs use less RAM during runtime as threads work on smaller chunks of the workload. The performance comparison also shows that the GNU C++ compiler produces faster, more consistent performance compared to the Intel compiler.

Excluding the statically scheduled implementations, the runtime performance of the different parallel implementations is essentially the same when using G++, while the Pthreads implementation is the fastest of the executables compiled with the Intel compiler. Once again, the automatic parallelisation feature of the Intel compiler was able to produce an efficient parallel

program with minimal programmer intervention. The Cilk Plus implementation was also able to produce respectable, if slightly slower, performance in comparison to the other implementations. In respect of the two Threading Building Blocks implementations, the choice of whether or not to `auto_partitioner()` for the `parallel_for` construct is dependent on the compiler as the Intel compiler favours manual block size selection for our example program, whereas G++ produces better performance using automatic range partitioning.

For embarrassingly parallel problems such as the Mandelbrot set algorithm and matrix multiplication problems examined here, it is apparent from the speedups achieved by our parallel program implementations that Amdahl's pessimistic outlook on scaling does not apply as these programs are able to approach or even exceed linear speedup. However, the results for the **dedup** kernel (Section 7.3.1) reveal that this is certainly not the case for all problems types.

Regarding other parallel software development tools, the Valgrind, Intel Inspector XE, and Intel debugger (IDB) error checking and debugging tools were found to be helpful in detecting and locating data race and memory leak issues in the shared queue data structure that was developed for the dynamically scheduled Boost Threads and Pthreads implementations.

Once again, code metrics reveal that automatic parallelisation requires the least programming effort to implement. It is followed by OpenMP, Cilk Plus, Threading Building Blocks, and the statically scheduled Boost Threads and Pthreads implementations, which are presented in order of increasing effort. While the dynamically scheduled Boost Threads and Pthreads implementations produce very good performance, they require significantly more effort to implement than the statically scheduled versions. This is as a result of the need for a task queue for delegating smaller chunks of work to the threads. While we implemented our own queue structure, it is possible to make use of existing data structures provided that they are thread-safe or can be made thread-safe by enclosing access to the data structure in critical sections using the appropriate lock constructs. Regardless of this, implementing dynamic scheduling using low-level threading libraries requires more thought and effort in comparison to simple static scheduling approaches. In contrast, switching between static and dynamic scheduling in OpenMP is trivial and can be done at compile time or even runtime by modifying the appropriate environment variables. These initial findings, along with similar research [86, 87], show that the use of implicit parallel programming models, such as Cilk Plus, OpenMP, and Threading Building Blocks, requires less programming effort and results in lower program complexity compared to explicit threading using low-level libraries such as Boost Threads and Pthreads.

7.3 Dedup Kernel

7.3.1 Runtime Performance

The compiler options used for **dedup** differ slightly from the previous two examples. The options that differ are listed below. The parallel programs are compiled with the IPO and SSE options in addition to the parallel library specific options.

- **Common compiler options –** `-fomit-frame-pointer -fno-builtin -pipe -fno-strict-aliasing -D_XOPEN_SOURCE=600 -lcrypto -lz`
- **IPO –** g++: `-flto -fwhole-program`
  icpc: `-ipo`

The runtime performance summary of the sequential compiler optimisations using the GNU C/C++ (g++) and Intel C/C++ (icpc) compilers for **dedup** is presented in Table 7.7. The performance of the four parallel implementations, with number of threads (n) equal to four, is given in Table 7.8. All test runs were performed using a 672 MB data file. Previous testing (Section 6.4.2) revealed that the performance of the disk subsystem did not have an effect on the performance of the program and that it was not necessary to make use of a ram-disk These results are analysed and discussed in Section 7.3.3.

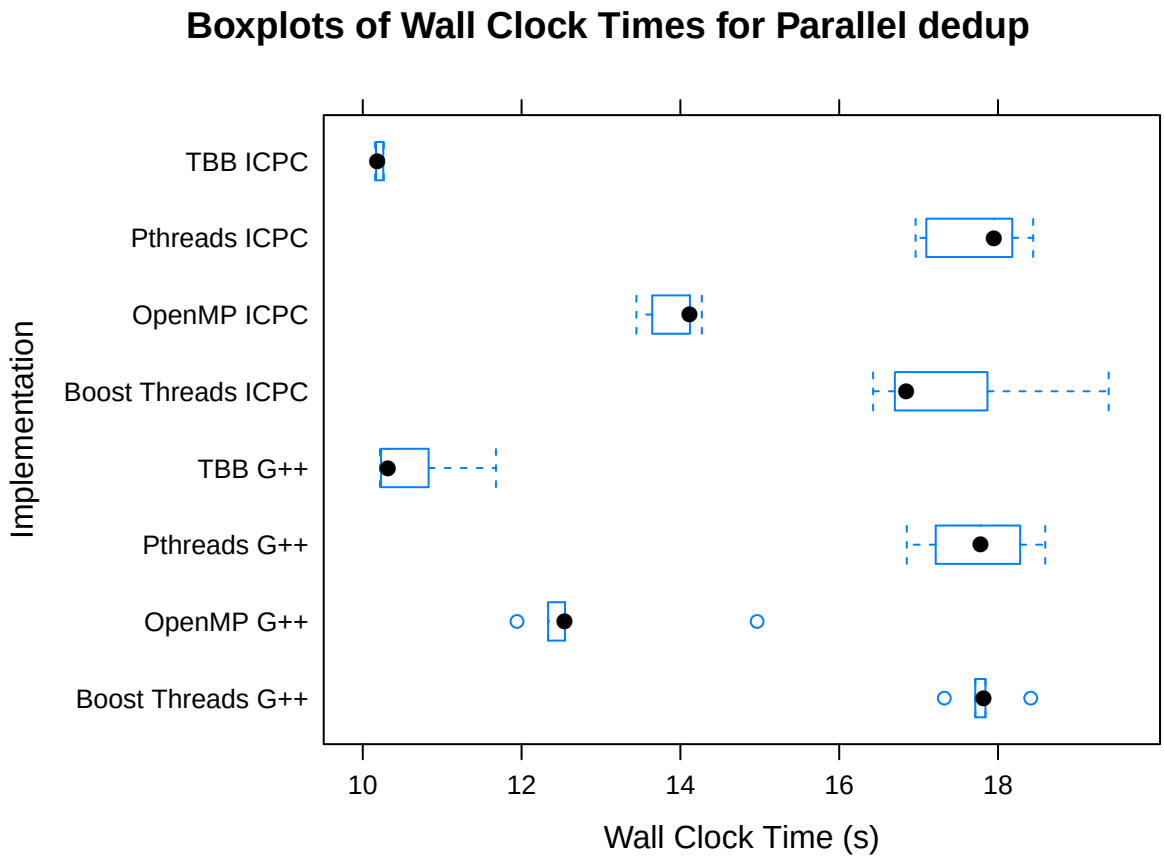


Figure 7.3: Runtime performance for parallel implementations of **dedup** using the GNU and Intel C/C++ compilers.

Table 7.7: Performance of compiler optimisations for sequential **dedup** program.

Optimisation	Compiler	Wall Clock Time (s)				Speed-up	CPU Time (s)	CPU Load	Memory Usage (MB)	Binary Size (KB)
		Mean	Geo. Mean	Std. Dev.	Var.		Mean	Mean	Mean	Mean
-O0	g++	43.6730	43.661460	1.1275520	1.2713735	–	43.199	25%	1816.731	1.110
	icpc	43.5432	43.530830	1.1706971	1.3705317	1.00	43.262	25%	1805.699	1.110
-O1	g++	37.8744	37.862060	1.0900047	1.1881103	1.15	37.553	25%	1806.474	1.110
	icpc	37.3650	37.353380	1.0532208	1.1092740	1.17	37.182	25%	1806.365	1.110
-O2	g++	37.8116	37.797930	1.1508839	1.3245338	1.16	37.650	25%	1806.472	1.110
	icpc	37.5238	37.510890	1.1122530	1.2371067	1.16	37.309	25%	1806.470	1.110
-O3	g++	37.9594	37.947870	1.0544903	1.1119498	1.15	37.666	25%	1820.911	1.110
	icpc	37.6794	37.662930	1.2600884	1.5878228	1.16	37.329	25%	1806.467	1.110
-O3 IPO	g++	37.9588	37.946620	1.0845913	1.1763382	1.15	37.695	25%	1806.472	1.110
	icpc	38.5878	38.484950	3.2591673	10.6221717	1.13	37.514	25%	1821.215	1.110
-O3 IPO SSE	g++	37.9202	37.901780	1.3412154	1.7988587	1.15	37.703	25%	1806.473	1.110
	icpc	37.4846	37.472140	1.0937179	1.1962188	1.17	37.347	25%	1820.906	1.110
-O3 IPO SSE PGO	g++	38.1936	38.180960	1.1090324	1.2299528	1.14	37.942	25%	1806.466	1.110
	icpc	38.2808	38.265900	1.2102453	1.4646937	1.14	38.170	25%	1813.423	1.110

Table 7.8: Performance of parallel **dedup** implementations.

Parallel Model	Compiler	Wall Clock Time (s)				Speed-up	CPU Time (s)	CPU Load	Memory Usage (MB)	Binary Size (KB)
		Mean	Geo. Mean	Std. Dev.	Var.		Mean	Mean	Mean	Mean
Seq. -O3 IPO SSE	g++	37.9202	37.901780	1.3412154	1.7988587	–	37.703	25%	1806.473	1.110
	icpc	37.4846	37.472140	1.0937179	1.1962188	1.01	37.347	25%	1820.906	1.110
Boost Threads	g++	17.8220	17.818600	0.3899308	0.1520460	2.13	51.257	71%	1879.532	1.177
	icpc	17.4470	17.414190	1.2170996	1.4813315	2.15	50.190	71%	1876.580	1.177
OpenMP	g++	12.8664	12.824650	1.1992591	1.4382223	2.96	42.982	85%	1884.319	1.162
	icpc	13.9190	13.915380	0.3537061	0.1251080	2.69	53.227	95%	1881.326	1.162
Pthreads	g++	17.7440	17.732200	0.7226707	0.5222530	2.14	51.121	71%	1870.334	1.172
	icpc	17.7260	17.716090	0.6611286	0.4370910	2.12	50.869	70%	1884.657	1.172
Intel TBB	g++	10.6542	10.640030	0.6246589	0.3901987	3.56	37.323	89%	1912.961	1.156
	icpc	10.2056	10.205490	0.0520461	0.0027088	3.67	36.422	89%	1913.261	1.156

7.3.2 Code Metrics

Table 7.9: Static code metrics for the implementations of the **dedup** program.

Implementation	SLOC	Added	Removed	Modified	N	V	v(G)
Sequential	1837	–	–	–	5443	26641.85	354
Boost	2013	217	41	0	6068	30735.85	385
OpenMP	2041	245	41	0	6158	31116.11	388
Pthreads	2044	257	50	0	6316	32003.83	390
Intel TBB	1867	203	173	9	5383	26828.43	338

7.3.3 Discussion

The sequential performance results for the **dedup** program (Table 7.7) do not reveal any interesting differences between the two compilers, with the Intel C++ compiler producing only marginally faster program executables compared to G++. However, the parallel performance results given in Table 7.8 and illustrated in Figure 7.3 show a distinct difference in performance between the various parallel programming models.

The first and most obvious observation that can be drawn from the results is that the Threading Building Blocks implementation is significantly faster than the other parallel models. This is due to the nature of the **dedup** kernel, which is implemented using a pipeline architecture. TBB provides a powerful high-level abstraction for implementing stream or pipeline processing parallel designs where the other parallel programming models do not. Surprisingly, OpenMP exhibits a distinct performance improvement over the explicit threading approaches despite its lack of a pipeline processing abstraction. The improved performance is likely due to OpenMP's more sophisticated thread scheduling capabilities. The Boost Threads and Pthreads implementations rely on the operating system scheduler to manage thread execution, which can lead to suboptimal performance if the threads associated with less computationally intensive pipeline stages interfere with the execution of threads associated with the compression stage. There is no significant performance difference between the explicit threading approaches as they both suffer from the same thread concurrency issue. The only case where performance differs significantly between compilers is the OpenMP based implementation where the GNU C++ OpenMP runtime appears to outperform Intel's OpenMP implementation.

Further analysis of the data shows that none of the implementations make full use of the CPU during program execution and the effect of this can be seen on the parallel speedup of the various

programs implementations. From the speedup values, one can calculate the value for the parallel fraction f of each implementation and predict program speedup for increasing processor counts. However, the lack of appropriate hardware to test such predictions prevents us from performing an in-depth investigation.

The effect of not providing appropriate parallel abstractions for pipeline architectures is evident in the code metrics for **dedup** (Table 7.9). For parallel programming models without a pipeline abstraction (Boost Threads, OpenMP, and Pthreads), the program length and complexity increased by roughly 10%. Whereas, for Threading Building Blocks, the length and complexity actually decreased due to the more natural mapping of the pipeline design to the capabilities of the library. The way this affected the code metrics was that the TBB implementation no longer needed the queue data structure and its associated methods, so that code was removed. The pipeline stages were also simplified as they no longer had to interact with the queue and could simply take the next segment of work from the function argument. During development, it was noted that the lack of appropriate synchronisation constructs in OpenMP resulted in a program that was more complex, despite the simplified thread pool and pipeline initialisation.

On a more subjective note, with the increased complexity of the **dedup** program compared to the previous examples, the value of easy-to-use parallel software development tools, such as Valgrind, Intel Inspector XE, Intel VTune Amplifier XE, and the Intel debugger (IDB), became apparent and these tools were found to be indispensable. This was particularly true for Valgrind's Memcheck and Helgrind tools, as well as Intel Inspector, as debugging the complex memory allocation patterns and extensive use of mutual exclusion would likely have been difficult without such tools to aid the process.

7.3.4 Overall Findings

By taking the ratio of the parallel program length (V) against the original sequential program length, we derive a measure for the additional effort required to implement the parallel program relative to the original sequential program. This allows us to compare different programs with vastly different program lengths. Furthermore, we can take the parallel speedup for each parallel programming model as a normalised measure of performance that can be compared to the speedups for other sample programs provided the performance is measured on the same computer system with the same number of processor cores. Then, we take the geometric means[3] of the speedups and of the additional effort ratios for the sample programs in respect to the parallel

[3]The geometric mean is used as gives us a more consistent number with which to perform comparisons [115].

programming model and compiler used. The geometric means of the program speedups are then divided by the geometrics mans of the effort ratios to produce a value that can be used to rank the parallel programming models accordingly.

The results of our analysis of the programs and parallel programming models described above are given in Table 7.10. In the case of the **mandelbrot** program, the dynamically scheduled implementations were selected as they had the best performance. The `auto_partitioner` TBB implementation was selected over the manual partitioning approach as it is more flexible with regards to input size. For the automatic parallelisation approach (Auto-parallel), a speedup of 1 was used to indicate that the parallelisation effort did not yield any improvement and that the program was effectively the same as the sequential version. The results for the Cilk Plus implementation are only valid for simple loop-based task parallelism as the only implementations of Cilk Plus are for the `matrixmul` and `mandelbrot` programs.

Our preliminary analysis of the performance to effort ratio indicate that OpenMP is the best choice when it comes to selecting a parallel programming model as it provides consistently good performance and typically requires far less programming effort compared to the other parallel programming models. The next best choice is Threading Building Blocks as it provides reasonable performance in conjunction with powerful parallel programming abstractions that are well-suited to solving a range of parallel problem types. These results confirm that it is usually better to select a parallel programming model with a higher level of abstraction. Another implication of these results is that automatic parallelisation should always be attempted as it has the potential to provide good performance improvements with little to no programming effort required should the target program make use of computationally intensive loop structures. However, the proposed parallel design must be taken into consideration along with these recommendations as certain parallel programming models are simply not suited to particular parallelisation approaches. This is evident with Cilk Plus, which is better suited to recursive task-based parallelism and task-parallel loops and does not provide an abstraction for pipeline or explicit mutual exclusion constructs. While Boost Threads and Pthreads have the lowest performance to effort ratio, they are also the most flexible parallel models due to their low-level threading functionality.

Table 7.10: Performance to Effort ratio calculations for **matrixmul**, **mandelbrot**, and **dedup**.

Parallel Model	Compiler	Program Performance (Speedup)				Effort (Increase in Program Length)				Performance
		Matrix.	Mandel.	Dedup	Geo. Mean	Matrix.	Mandel.	Dedup	Geo. Mean	Effort
Auto-parallel	g++	1.00	1.00	1.00	1.000	1.000	1.005	1.000	1.002	0.998
	icpc	7.82	3.93	1.00	3.132					3.127
Boost Threads	g++	3.83	4.00	2.13	3.196	1.360	1.981	1.115	1.443	2.215
	icpc	2.28	3.96	2.15	2.688					1.863
Cilk Plus	g++	NA	NA	NA	NA	1.158	1.058	NA	1.107	NA
	icpc	9.76	3.77	NA	6.066					5.480
OpenMP	g++	3.84	3.99	2.96	3.566	1.128	1.043	1.131	1.100	3.242
	icpc	9.07	3.93	2.69	4.577					4.161
Pthreads	g++	3.84	4.00	2.14	3.203	1.417	2.188	1.160	1.532	2.091
	icpc	9.95	3.98	2.12	4.379					2.858
Intel TBB	g++	3.76	3.93	3.56	3.747	1.274	1.149	0.989	1.131	3.312
	icpc	2.27	3.77	3.67	3.155					2.789

Chapter 8

Conclusion

While it is clear that parallel computers are now seeing use in almost all aspects of modern computing thanks to the widespread availability of multicore processors, it is certainly not the case that all software is capable of harnessing the ever increasing parallel computing power available in modern and upcoming CPUs. Harnessing this power is important as the CPU power wall has put an end to free software performance increases resulting from higher clock speeds. However, creating efficient parallel software is difficult for the novice parallel programmer as parallel programming models require a different mindset and approach compared to the traditional sequential programming methodology. Parallel programming also requires a greater understanding of both the hardware and software factors that affect the performance of parallel programs, such as the effect of cache memory and memory bus bandwidth on programs that share data between processors. In addition to this, a wide variety of parallel programming models and software development tools are available and it may be necessary for a programmer to master more than one of these to able to successfully parallelise different types of problem. Without the appropriate software tools and knowledge, it is exceedingly difficult to develop the high performance parallel software required in our age of parallel computing.

As such, we set out to address the needs of the novice parallel programmer by highlighting and condensing the required background information, presenting a selection of software development tools and parallel programming models, and recommending an approach to performance tuning and parallel software design. Finally, we aimed to assist in the selection of an appropriate parallel programming model by presenting an analysis of the performance of different parallel programming models relative to the amount of programming effort required. This was achieved by presenting an extensive array background of information, which included summaries of the key parallel programming concepts, as well as the impact of the hardware organisation and architecture on the performance of parallel programs. This provides a solid theoretical grounding

to inform later parallel software development efforts. Thereafter, we presented a short survey of commonly available commercial and free software development tools that support multithreading. The survey also included introductions to the various parallel programming models that were used later in our investigations.

One of the key realisations during our study was the importance of a structured and methodical approach to software performance tuning and parallel program design. To this end, we presented an iterative approach to software tuning that relies on the gathering of relevant performance data to inform optimisation efforts and verify the effectiveness and correctness of the implemented optimisations. We also introduced and discussed a number of manual code optimisations and the situations in which they can be applied. The development of an effective parallel program design has a large effect on the efficiency and scalability of the resulting program. As such, we presented a descriptive summary of a structured approach to parallel design that makes uses pattern language to capture and express the expertise of experienced parallel programmers. The resulting patterns are then selected according to the nature of the problem and applied in a specific order to build up an appropriate parallel design for the target problem. However, the variety of problem types and parallel designs, along with the limitations each parallel programming model, highlighted the problem with selecting a parallel programming model based entirely on the performance and programming effort of the models. The implication of this is that the choice of parallel programming model may be limited by the capabilities of the parallel model in relation to the requirements of the parallel design. Nevertheless, the evaluation of parallel programming models in terms of performance and effort is still a worthwhile pursuit for problems where the design does not limit the choice to a single library or API.

Our investigation into the performance and programming effort of the various parallel programming models was carried out as an experiment whereby three different sequential programs were parallelised using each of the selected parallel programming models where such a parallelisation was possible. The ratio of runtime performance (speedup) to the additional programming effort required to implement the parallel program based on program length was calculated for each implementation. This provided us with performance to effort index. The results showed that high-level parallel programming models provide the best combination of performance to programming effort and that explicit threading approaches can provide good performance in certain circumstances, but require markedly more programmer effort to implement efficiently. Once again, it is important to note the recommendations from our analysis should be used in conjunction with an appropriate parallel design as the problem type and parallel design affect the suitability of certain parallel programming models.

The work carried out in this thesis represents an initial investigation into the effectiveness of

parallel programming models relative to programming effort. As such, there are opportunities for future work to expand upon and improve the observations presented here. First and foremost, a more rigorous statistical analysis is required, which includes a larger sample of potential parallel programs with a range of different problem characteristics, the inclusion of several novice parallel programmers as experimental subjects, and a more in-depth analysis of the performance data and code metrics. Secondly, there is a range of other parallel programming models that have not been examined, such as FastFlow and QuickThreads for instance, as well as parallel libraries and APIs for other programming languages and environments. The study could be expanded to include these additional parallel programming models. Additionally, an investigation into the most effective approaches to load balancing and thread scheduling can be undertaken to provide recommendations for particular type of scheduling problems.

Bibliography

[1] A. Abran, M. Lopez, and N. Habra. An Analysis of the McCabe Cyclomatic Complexity Number. In *12th International Workshop on Software Measurement - IWSM2004*, pages 391–405, Königs Wusterhausen, Germany, 2004. Shaker Verlag.

[2] ACM. The ACM Computing Classification System [1998 Version] [online]. October 2009. URL: `http://www.acm.org/about/class/ccs98-html` [cited 29 November 2009].

[3] Advanced Micro Devices. AMD CodeAnalyst Performance Analyzer for Linux [online]. 2010. URL: `http://developer.amd.com/cpu/CodeAnalyst/codeanalystlinux/Pages/default.aspx` [cited Dec. 27, 2010].

[4] Advanced Micro Devices. Software Optimization Guide for AMD Family 10h and 12h Processors. Technical report, Advanced Micro Devices, December 2010. URL: `http://support.amd.com/us/Processor_TechDocs/40546.pdf`.

[5] A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, 2nd edition edition, 2006.

[6] M. Aldinucci, M. Meneghin, and M. Torquati. Efficient smith-waterman on multi-core with fastflow. In *Proceedings of the 2010 18th Euromicro Conference on Parallel, Distributed and Network-based Processing*, PDP '10, pages 195–199, Washington, DC, USA, 2010. IEEE Computer Society.

[7] J. H. Anderson, Y. Kim, and T. Herman. Shared-memory Mutual Exclusion: Major Research Trends Since 1986. *Distrib. Comput.*, 16(2-3):75–110, 2003.

[8] T. E. Anderson. The Performance of Spin Lock Alternatives for Shared-Memory Multiprocessors. *IEEE Trans. Parallel Distrib. Syst.*, 1:6–16, January 1990.

[9] B. Andersson, S. Blair-Chappell, and R. Mueller-Albrecht. Intel® Debugger for Linux*. Tutorial, Intel Corporation, Developer Products Division, 2010. URL: `http://software.intel.com/en-us/articles/idb-linux/`.

[10] Anonymous. The Mandelbrot Set [online]. 2010. URL: `http://warp.povusers.org/Mandelbrot/` [cited Jan. 13, 2011].

[11] B. Barney. Introduction to Parallel Computing [online]. Oct. 12, 2010. URL: `https://computing.llnl.gov/tutorials/parallel_comp/` [cited Nov. 10, 2010].

[12] P. Becker. Working Draft, Standard for Programming Language C++. Technical Report N2369=07-0229, ISO/IEC JTC 1, Information Technology, Subcommittee SC 22, Programming Language C++, November 2010. URL: `http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2010/n3225.pdf`.

[13] J. L. Bentley. *Writing Efficient Programs*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1982.

[14] C. Bienia. *Benchmarking Modern Multiprocessors*. PhD thesis, Princeton University, January 2011.

[15] A. J. C. Bik, D. L. Kreitzer, and X. Tian. A Case Study on Compiler Optimizations for the Intel® Core™ 2 Duo Processor. *Int. J. Parallel Program.*, 36(6):571–591, 2008.

[16] G. E. Blelloch, R. A. Chowdhury, P. B. Gibbons, V. Ramachandran, S. Chen, and M. Kozuch. Provably Good Multicore Cache Performance for Divide-and-Conquer Algorithms. In *SODA '08: Proceedings of the nineteenth annual ACM-SIAM symposium on Discrete algorithms*, pages 501–510, Philadelphia, PA, USA, 2008. Society for Industrial and Applied Mathematics.

[17] C. Breshears. *The Art of Concurrency: A Thread Monkey's Guide to Writing Parallel Applications*. O'Reilly Media, Inc., 2009.

[18] F. Broquedis, N. Furmento, B. Goglin, P.-A. Wacrenier, and R. Namyst. ForestGOMP: An Efficient OpenMP Environment for NUMA Architectures. *International Journal of Parallel Programming*, May 2010.

[19] K. Carlson. SD West: Parallel or Bust [online]. Mar. 7, 2008. URL: `http://www.ddj.com/hpc-high-performance-computing/206902441` [cited Jun. 2, 2009].

[20] R. H. Carver and K.-C. Tai. *Modern Multithreading: Implementing, Testing, and Debugging Multithreaded Java and C++/Pthreads/Win32 Programs*. Wiley-Interscience, 2005.

[21] B. Chapman, G. Jost, and R. v. d. Pas. *Using OpenMP: Portable Shared Memory Parallel Programming (Scientific and Engineering Computation)*. The MIT Press, 2007.

[22] E. G. Coffman, M. Elphick, and A. Shoshani. System Deadlocks. *ACM Comput. Surv.*, 3(2):67–78, 1971.

[23] K. D. Cooper, K. Kennedy, and L. Torczon. Interprocedural Optimization: Eliminating Unnecessary Recompilation. *SIGPLAN Not.*, 21:58–67, July 1986.

[24] A. Das, J. Lu, H. Chen, J. Kim, P.-C. Yew, W.-C. Hsu, and D.-Y. Chen. Performance of Runtime Optimization on BLAST. In *Proceedings of the International Symposium on Code Generation and Optimization*, CGO '05, pages 86–96, Washington, DC, USA, 2005. IEEE Computer Society.

[25] S. Debray. Writing Efficient Programs: Performance Issues in an Undergraduate CS Curriculum. *SIGCSE Bull.*, 36(1):275–279, 2004.

[26] Digital Equipment Corporation. *Introduction to the Mandelbrot Set*, January 1997. Order Number: AA-Q62LC-TE. URL: `http://www.mun.ca/hpc/hpf_pse/manual/hpf0019.htm` [cited Jan. 13, 2011].

[27] U. Drepper. What Every Programmer Should Know About Memory, 2007. URL: `http://www.akkadia.org/drepper/cpumemory.pdf`.

[28] P. J. Drongowski. Instruction-Based Sampling: A New Performance Analysis Technique for AMD Family 10h Processors. Technical report, Advanced Micro Devices, Inc., Boston Design Center, November 2007. URL: `http://developer.amd.com/assets/AMD_IBS_paper_EN.pdf`.

[29] P. J. Drongowski. An introduction to analysis and optimization with AMD CodeAnalyst^{TM} Performance Analyzer. Technical report, Advanced Micro Devices, Inc., Boston Design Center, September 2008. URL: `http://developer.amd.com/Assets/Introduction_to_CodeAnalyst.pdf`.

[30] J. Falcou, J. Sérot, T. Chateau, and J. T. Lapresté. QUAFF: efficient C++ design for parallel skeletons. *Parallel Comput.*, 32(7):604–615, 2006.

[31] N. Fenton and S. L. Pfleeger. *Software Metrics: A Rigorous and Practical Approach (2nd edition)*. PWS Publishing Co., Boston, MA, USA, 1997.

[32] M. Flynn. Some Computer Organizations and Their Effectiveness. *IEEE Transaction on Computers*, 21(C):948–960, 1972.

[33] A. Fog. Intel's "cripple AMD" function [online]. October 12, 2010. URL: `http://www.agner.org/optimize/blog/read.php?i=49` [cited Dec. 27, 2010].

[34] A. Fog. Optimizing software in C++: An optimization guide for Windows, Linux and Mac platforms [online]. September 25, 2010. URL: `http://www.agner.org/optimize/optimizing_cpp.pdf` [cited Dec. 27, 2010].

[35] I. Foster. *Designing and Building Parallel Programs: Concepts and Tools for Parallel Software Engineering*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.

[36] Free Software Foundation. Debugging with GDB: the GNU Source-Level Debugger [online]. Dec., 27 2010. URL: `http://sourceware.org/gdb/current/onlinedocs/gdb/` [cited Dec. 27, 2010].

[37] Free Software Foundation. *GNU gprof*, December 2010. URL: `http://sourceware.org/binutils/docs/gprof/index.html`.

[38] M. Frigo. Multithreaded Programming in Cilk. In *PASCO '07: Proceedings of the 2007 International Workshop on Parallel Symbolic Computation*, pages 13–14, New York, NY, USA, 2007. ACM.

[39] M. Frigo, P. Halpern, C. E. Leiserson, and S. Lewin-Berlin. Reducers and other Cilk++ hyperobjects. In *Proceedings of the twenty-first annual symposium on Parallelism in algorithms and architectures*, SPAA '09, pages 79–90, New York, NY, USA, 2009. ACM.

[40] M. Gabbrielli and S. Martini. *Programming Languages: Principles and Paradigms*. Springer Publishing Company, Incorporated, 1st edition, 2010.

[41] GCC Team. Automatic parallelization in Graphite [online]. August 17, 2009. URL: `http://gcc.gnu.org/wiki/Graphite/Parallelization` [cited Dec. 27, 2010].

[42] GCC Team. Autopar's(in trunk) Algorithms [online]. May 26, 2009. URL: `http://gcc.gnu.org/wiki/AutoparRelated` [cited Dec. 27, 2010].

[43] GCC Team. Using the GNU Compiler Collection: GCC 4.5.2 Manual [online]. December 18, 2010. URL: `http://gcc.gnu.org/onlinedocs/gcc-4.5.2/gcc/` [cited Dec. 27, 2010].

[44] R. Geva. Elemental functions: Writing data-parallel code in C/C++ using Intel® Cilk™ Plus. White paper, Intel Corporation, Month 2010.

[45] B. Goetz. Threading lightly, Part 2: Reducing contention [online]. Sept., 5 2001. URL: `http://www.ibm.com/developerworks/java/library/j-threads2.html?ca=drs-` [cited Apr. 3, 2009].

[46] I. Gouy. Computer Language Benchmarks Game [online]. May 2009. URL: `http://shootout.alioth.debian.org/u32q/faq.php` [cited 28 May 2009].

[47] S. L. Graham, P. B. Kessler, and M. K. McKusick. gprof: a Call Graph Execution Profiler. *SIGPLAN Not.*, 39:49–57, April 2004.

[48] J. L. Gustafson. Reevaluating Amdahl's Law. *Communications of the ACM*, 31:532–533, 1988.

[49] R. Hackney, L. Momii, C. Riggs, and A. Dingle. Profiler Tools Selection for Curricular Support. *J. Comput. Small Coll.*, 21(1):177–182, 2005.

[50] M. W. Hall, K. Kennedy, and K. S. McKinley. Interprocedural Transformations for Parallel Code Generation. In *Proceedings of the 1991 ACM/IEEE conference on Supercomputing*, Supercomputing '91, pages 424–434, New York, NY, USA, 1991. ACM.

[51] M. H. Halstead. *Elements of Software Science (Operating and programming systems series)*. Elsevier Science Inc., New York, NY, USA, 1977.

[52] P. G. Hamer and G. D. Frewin. M.H. Halstead's Software Science - A Critical Examination. In *ICSE '82: Proceedings of the 6th international conference on Software engineering*, pages 197–206, Los Alamitos, CA, USA, 1982. IEEE Computer Society Press.

[53] B. Hayes. Computing in a Parallel Universe. *American Scientist*, 95:476–480, 2007.

[54] J. L. Hennessy and D. A. Patterson. *Computer Architecture: A Quantitative Approach (4th edition)*. Morgan Kaufmann, September 2006.

[55] M. Herlihy. Wait-Free Synchronization. *ACM Trans. Program. Lang. Syst.*, 13:124–149, January 1991.

[56] M. Herlihy and N. Shavit. *The Art of Multiprocessor Programming*. Morgan Kaufmann, March 2008.

[57] M. P. Herlihy and J. M. Wing. Linearizability: A Correctness Condition for Concurrent Objects. *ACM Trans. Program. Lang. Syst.*, 12:463–492, July 1990.

[58] D. Hou and Y. Wang. An Empirical Analysis of the Evolution of User-Visible Features in an Integrated Development Environment. In *Proceedings of the 2009 Conference of the Center for Advanced Studies on Collaborative Research*, CASCON '09, pages 122–135, New York, NY, USA, 2009. ACM.

[59] C. Hughes and T. Hughes. *Parallel and Distributed Programming Using C++*. Prentice Hall Professional Technical Reference, 2003.

[60] C. Hughes and T. Hughes. *Professional Multicore Programming: Design and Implementation for C++ Developers*. Wrox Press Ltd., Birmingham, UK, UK, 2008.

[61] C. S. Ierotheou, H. Jin, G. Matthews, S. P. Johnson, and R. Hood. Generating OpenMP code using an interactive parallelization environment. *Parallel Comput.*, 31(10-12):999–1012, 2005.

[62] Intel Corporation. Intel Research Advances 'Era Of Tera' [online]. February 2007. URL: `http://www.intel.com/pressroom/archive/releases/2007/20070204comp.htm` [cited Dec. 27, 2010].

[63] Intel Corporation. *Intel® Core™2 Extreme Quad-Core Processor QX6000$^{\Delta}$ Sequence and Intel® Core™2 Quad Processor Q6000$^{\Delta}$ Sequence Datasheet*. Intel Corporation, August 2007. URL: `http://download.intel.com/design/processor/datashts/31559205.pdf` [cited Sept. 21, 2010].

[64] Intel Corporation. *Intel® Core™2 Extreme Processor QX9000$^{\Delta}$ Series, Intel® Core™2 Quad Processor Q9000$^{\Delta}$, Q9000S$^{\Delta}$, Q8000$^{\Delta}$, and Q8000S$^{\Delta}$ Series Datasheet*. Intel Corporation, August 2009. URL: `http://download.intel.com/design/processor/datashts/318726.pdf` [cited Sept. 21, 2010].

[65] Intel Corporation. *Intel® Debugger: Command Reference*. Intel Corporation, 2009. Document number: 319698-009US.

[66] Intel Corporation. Intel® VTune™ Performance Analyzer 9.1 for Linux. Online, 2009.

[67] Intel Corporation. Using Intel® VTune™ Performance Analyzer to Optimize Software on Intel® Core™ i7 Processors. Online, February 2009. URL: `http://software.intel.com/file/15529`.

[68] Intel Corporation. Avoiding and Identifying False Sharing Among Threads [online]. February 2010. URL: http://software.intel.com/en-us/articles/avoiding-and-identifying-false-sharing-among-threads/ [cited Dec. 17, 2010].

[69] Intel Corporation. Intel Guide for Developing Multithreaded Applications [online]. March 2010. URL: http://software.intel.com/en-us/articles/intel-guide-for-developing-multithreaded-applications/ [cited Dec. 27, 2010].

[70] Intel Corporation. Intel Software Development Product List [online]. 2010. URL: http://software.intel.com/en-us/articles/intel-sdp-products/ [cited Dec. 27, 2010].

[71] Intel Corporation. Intel Thread Building Blocks [online]. December 2010. URL: http://software.intel.com/en-us/articles/intel-tbb/ [cited Dec. 27, 2010].

[72] Intel Corporation. *Intel® C++ Compiler 11.1 User and Reference Guides*. Intel Corporation, 2010. Document number: 304968-023US. URL: http://software.intel.com/sites/products/documentation/hpc/compilerpro/en-us/cpp/lin/compiler_c/index.htm.

[73] Intel Corporation. *Intel® C++ Compiler XE 12.0 User and Reference Guides*. Intel Corporation, 2010. Document number: 323273-120US - Intel C++ Composer XE 2011 - Linux* OS. URL: http://software.intel.com/sites/products/documentation/hpc/composerxe/en-us/cpp/lin/index.htm.

[74] Intel Corporation. Intel® Cilk™ Plus: A Simple Path to Parallelism. Technical report, Intel Corporation, 2010. URL: http://software.intel.com/sites/products/evaluation-guides/docs/cilk-plus-evaluation-guide.pdf.

[75] Intel Corporation. Intel® Inspector XE Thread and Memory Checker [online]. 2010. URL: http://software.intel.com/en-us/articles/intel-inspector-xe/ [cited Dec. 27, 2010].

[76] Intel Corporation. *Intel® Integrated Performance Primitives for Intel® Architecture Reference Manual*. Intel Corporation, 2010. Intel IPP 7.0. URL: http://software.intel.com/sites/products/documentation/hpc/composerxe/en-us/ippxe/ipp_manual_lnx/index.htm.

[77] Intel Corporation. *Intel® Math Kernel Library Reference Manual*. Intel Corporation, 2010. Document number: 630813-031US. URL: http://software.intel.com/sites/products/documentation/hpc/compilerpro/en-us/cpp/lin/mkl/refman/index.htm.

[78] Intel Corporation. Intel® VTune™ Amplifier XE Performance Profiler. Online, November 2010. URL: http://software.intel.com/sites/products/collateral/XE/vtune_amplifier_xe_brief.pdf.

[79] Intel Corporation. Optimization Notice [online]. 2010. URL: `http://software.intel.com/en-us/articles/optimization-notice/` [cited Dec. 17, 2010].

[80] Intel Corporation. *Intel® 64 and IA-32 Architectures Optimization Reference Manual*. Intel Corporation, January 2011. Order Number: 248966-023a. URL: `http://www.intel.com/Assets/PDF/manual/248966.pdf`.

[81] N. M. Josuttis. *The C++ Standard Library: A Tutorial and Reference*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.

[82] N. P. Jouppi. Superscalar vs. Superpipelined Machines. *SIGARCH Comput. Archit. News*, 16:71–80, June 1988.

[83] S. H. Kan. *Metrics and Models in Software Quality Engineering*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2nd edition, 2002.

[84] V. Kazempour, A. Fedorova, and P. Alagheband. Performance Implications of Cache Affinity on Multicore Processors. In *Euro-Par '08: Proceedings of the 14th international Euro-Par conference on Parallel Processing*, pages 151–161, Berlin, Heidelberg, 2008. Springer-Verlag.

[85] J. P. Kearney, R. L. Sedlmeyer, W. B. Thompson, M. A. Gray, and M. A. Adler. Software Complexity Measurement. *Commun. ACM*, 29(11):1044–1050, 1986.

[86] P. Kegel, M. Schellmann, and S. Gorlatch. Using OpenMP vs. Threading Building Blocks for Medical Imaging on Multi-cores. In *Euro-Par '09: Proceedings of the 15th International Euro-Par Conference on Parallel Processing*, pages 654–665, Berlin, Heidelberg, 2009. Springer-Verlag.

[87] P. Kegel, M. Schellmann, and S. Gorlatch. Comparing programming models for medical imaging on multi-core systems. *Concurrency and Computation: Practice and Experience*, 2010.

[88] B. Kempf. The Boost.Threads Library. *j-CCCUJ*, 20(5):6–??, May 2002. URL: `http://www.drdobbs.com/cpp/184401518`.

[89] K. Kennedy and J. R. Allen. *Optimizing Compilers for Modern Architectures: A Dependence-Based Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2002.

[90] B. W. Kernighan and D. M. Ritchie. *The C Programming Language*. Prentice Hall Press, Upper Saddle River, NJ, USA, 1988.

[91] B. Kuhn, P. Petersen, and E. O'Toole. OpenMP versus threading in C/C++. *Concurrency - Practice and Experience*, pages 1165–1176, 2000.

[92] C. Kyriacou, P. Evripidou, and P. Trancoso. Cacheflow: Cache Optimisations for Data Driven Multithreading. *Parallel Process. Lett.*, 16(2):229–244, 2006.

[93] C. Lameter. Local and Remote Memory: Memory in a Linux/NUMA System. Technical report, Linux Kernel, 2006. URL: `http://www.kernel.org/pub/linux/kernel/people/christoph/pmig/numamemory.pdf`.

[94] J. Laudon and L. Spracklen. The Coming Wave of Multithreaded Chip Multiprocessors. *Int. J. Parallel Program.*, 35(3):299–330, 2007.

[95] I. Lee. Integrated environments. *SIGSOFT Softw. Eng. Notes*, 8:60–62, March 1983.

[96] J. Lee and V. J. Mooney, III. A Novel Deadlock Avoidance Algorithm and Its Hardware Implementation. In *CODES+ISSS '04: Proceedings of the 2nd IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis*, pages 200–205, New York, NY, USA, 2004. ACM.

[97] J. Levon. OProfile [online]. November 2009. URL: `http://oprofile.sourceforge.net/about/` [cited Apr. 28, 2010].

[98] S.-W. Liao, A. Diwan, R. P. Bosch, Jr., A. Ghuloum, and M. S. Lam. SUIF Explorer: An Interactive and Interprocedural Parallelizer. *SIGPLAN Not.*, 34:37–48, May 1999.

[99] M. L. Liu. *Distributed Computing Principles and Applications*. Addison-Wesley, 2004.

[100] M. Mackey. Intel's compiler: is crippling the competition acceptable? [online]. 2005. URL: `http://www.swallowtail.org/naughty-intel.shtml` [cited Dec. 27, 2010].

[101] A. D. Malony and S. S. Shende. Overhead Compensation in Performance Profiling. *Parallel Process. Lett.*, 15(1-2):19–35, 2005.

[102] T. Mattson, B. Sanders, and B. Massingill. *Patterns for Parallel Programming*. Addison-Wesley Professional, 2004.

[103] T. Mattson and M. Wrinn. Parallel programming: can we PLEASE get it right this time? In *DAC '08: Proceedings of the 45th annual conference on Design automation*, pages 7–11, New York, NY, USA, 2008. ACM.

[104] T. J. McCabe. A Complexity Measure. In *Proceedings of the 2nd international conference on Software engineering*, ICSE '76, pages 407–, Los Alamitos, CA, USA, 1976. IEEE Computer Society Press.

[105] J. M. Mellor-Crummey and M. L. Scott. Algorithms for Scalable Synchronization on Shared-Memory Multiprocessors. *ACM Trans. Comput. Syst.*, 9:21–65, February 1991.

[106] D. A. Mey, S. Sarholz, and C. Terboven. Nested Parallelization with OpenMP. *Int. J. Parallel Program.*, 35(5):459–476, 2007.

[107] Microsoft Developer Network. Profile-Guided Optimizations [online]. 2010. URL: `http://msdn.microsoft.com/en-us/library/e7k32f4k(v=VS.100).aspx` [cited Dec. 27, 2010].

[108] E. E. Mills. Software Metrics SEI Curriculum Module. Technical Report SEI-CM-12-1.1, Software Engineering Institute Carnegie Mellon University, Seattle, Washington, December 1988.

[109] J. C. Mogul and K. K. Ramakrishnan. Eliminating Receive Livelock in an Interrupt-driven Kernel. *ACM Trans. Comput. Syst.*, 15(3):217–252, 1997.

[110] T. Moseley, D. A. Connors, D. Grunwald, and R. Peri. Identifying Potential Parallelism via Loop-centric Profiling. In *CF '07: Proceedings of the 4th international conference on Computing frontiers*, pages 143–152, New York, NY, USA, 2007. ACM.

[111] A. Mühlenfeld and F. Wotawa. Fault Detection in Multi-Threaded C++ Server Applications. *Electron. Notes Theor. Comput. Sci.*, 174:5–22, June 2007.

[112] N. Nethercote and J. Seward. Valgrind: A Framework for Heavyweight Dynamic Binary Instrumentation. In *Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '07, pages 89–100, New York, NY, USA, 2007. ACM.

[113] N. Nethercote, R. Walsh, and J. Fitzhardinge. Building Workload Characterization Tools with Valgrind. *IEEE Workload Characterization Symposium*, 0:2, 2006.

[114] B. Nichols, D. Buttlar, and J. P. Farrell. *Pthreads programming*. O'Reilly & Associates, Inc., Sebastopol, CA, USA, 1996.

[115] L. Null and J. Lobur. *Essentials of Computer Organization and Architecture*. Jones and Bartlett Publishers, Inc., USA, 1st edition, 2003.

[116] OpenMP Architecture Review Board. *OpenMP Application Program Interface*, 3.0 edition, May 2008. URL: `http://www.openmp.org`.

[117] Oracle Corporation. NetBeans IDE 6.9 Features: C and C++ Development [online]. 2010. URL: `http://netbeans.org/features/cpp/index.html` [cited Dec. 27, 2010].

[118] Oracle Corporation. Oracle Solaris Studio Features [online]. 2010. URL: `http://www.oracle.com/technetwork/server-storage/solarisstudio/overview/index-jsp-138069.html` [cited Dec. 27, 2010].

[119] P. Panchamukhi. Smashing performance with OProfile: Identifying performance bottlenecks in real-world systems [online]. October 2003. URL: `http://www.ibm.com/developerworks/linux/library/l-oprof.html` [cited Dec. 27, 2010].

[120] A. Pesterev, N. Zeldovich, and R. T. Morris. Locating Cache Performance Bottlenecks Using Data Profiling. In *EuroSys '10: Proceedings of the 5th European conference on Computer systems*, pages 335–348, New York, NY, USA, 2010. ACM.

[121] A. Pop and A. Cohen. A Stream-Computing Extension to OpenMP. In *Proceedings of the 6th International Conference on High Performance and Embedded Architectures and Compilers*, HiPEAC '11, pages 5–14, New York, NY, USA, 2011. ACM.

[122] K. Psarris. Program analysis techniques for transforming programs for parallel execution. *Parallel Computing*, 28(3):455–469, 2002.

[123] Z. Radović and E. Hagersten. RH Lock: A Scalable Hierarchical Spin Lock, May 2002.

[124] T. Rauber and G. Rünger. *Parallel Programming: for Multicore and Cluster Systems*. Springer, 1st edition. edition, March 2010.

[125] J. Reinders. *Intel Threading Building Blocks: Outfitting C++ for Multi-core Processor Parallelism*. O'Reilly, 2007.

[126] J. B. Rosenberg. *How Debuggers Work: Algorithms, Data Structures, and Architecture*. John Wiley & Sons, Inc., New York, NY, USA, 1996.

[127] R. Rufai, M. Bozyigit, J. Alghamdi, and M. Ahmed. Multithreaded Parallelism with OpenMP. *Parallel Process. Lett.*, 15(4):367–378, 2005.

[128] P. J. Salzman and R. Somers. Introduction to Debugging and GDB [online]. December 2010. URL: `http://www3.sympatico.ca/rsquared/gdb/intro.html#whatisadebugger` [cited Jan. 03, 2011].

[129] M. Sato. OpenMP: parallel programming API for shared memory multiprocessors and on-chip multiprocessors. In *ISSS '02: Proceedings of the 15th International Symposium on System Synthesis*, pages 109–111, New York, NY, USA, 2002. ACM.

[130] H. Schildt. *C++ from the Ground Up, Third Edition*. McGraw-Hill, Inc., New York, NY, USA, 3 edition, 2003.

[131] W. Stallings. *Computer Organization and Architecture: Designing for Performance (7th Edition)*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 2005.

[132] J. Stokes. 16- and 48-core monster chips on tap at next week's ISSCC [online]. February 2010. URL: `http://arstechnica.com/business/news/2010/02/16--and-48-core-monster-chips-on-tap-at-next-weeks-isscc.ars` [cited Dec. 27, 2010].

[133] B. Stroustrup. *The C++ Programming Language*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 3rd edition, 2000.

[134] X.-H. Sun and Y. Chen. Reevaluating Amdahl's law in the multicore era. *J. Parallel Distrib. Comput.*, 70(2):183–188, 2010.

[135] Supercomputing Technologies Group. *Cilk 5.4.6 Reference Manual*. Massachusetts Institute of Technology, 1998.

[136] H. Sutter. The Concurrency Revolution. *Dr. Dobb's Journal*, February 2005. URL: `http://www.drdobbs.com/184401916`.

[137] H. Sutter. Sharing Is the Root of All Contention. *Dr. Dobb's Journal*, October 2009. URL: `http://www.ddj.com/go-parallel/article/showArticle.jhtml?articleID=214100002`.

[138] The Eclipse Foundation. Eclipse CDT (C/C++ Development Tooling) [online]. 2010. URL: `http://www.eclipse.org/cdt/` [cited Dec. 27, 2010].

[139] X. Tian, M. Girkar, A. Bik, and H. Saito. Practical Compiler Techniques on Efficient Multithreaded Code Generation for OpenMP Programs. *Comput. J.*, 48:588–601, September 2005.

[140] X. Tian, J. P. Hoeflinger, G. Haab, Y.-K. Chen, M. Girkar, and S. Shah. A compiler for exploiting nested parallelism in OpenMP programs. *Parallel Comput.*, 31(10-12):960–983, 2005.

[141] J. L. Träff. What the parallel-processing community has (failed) to offer the multi/many-core generation. *J. Parallel Distrib. Comput.*, 69(9):807–812, 2009.

[142] Valgrind Developers. About Valgrind [online]. Dec. 23, 2010. URL: `http://valgrind.org/info/about.html` [cited Dec. 23, 2010].

[143] Valgrind Developers. *Valgrind Documentation*. Valgrind, release 3.6.0 edition, October 2010.

[144] R. Vaswani and J. Zahorjan. The Implications of Cache Affinity on Processor Scheduling for Multiprogrammed, Shared Memory Multiprocessors. *SIGOPS Oper. Syst. Rev.*, 25(5):26–40, 1991.

[145] Verifysoft Technology. Measurement of Halstead Metrics with Testwell CMT++ and CMTJava (Complexity Measures Tool) [online]. June 2010. URL: `http://www.verifysoft.com/en_halstead_metrics.html` [cited Dec. 10, 2010].

[146] Verifysoft Technology. Measurement of Lines of Code Metrics with Testwell CMT++ and CMTJava (Complexity Measures Tools) [online]. June 2010. URL: `http://www.verifysoft.com/en_linesofcode_metrics.html` [cited Dec. 10, 2010].

[147] Verifysoft Technology. Measurement of McCabe Metrics with Testwell CMT++ and CMTJava (Code Complexity Measures Tools) [online]. June 2010. URL: `http://www.verifysoft.com/en_mccabe_metrics.html` [cited Dec. 10, 2010].

[148] Virtual Machinery. Sidebar 2 - The Halstead Metrics [online]. 2009. URL: `http://www.virtualmachinery.com/sidebar2.htm` [cited Dec. 10, 2010].

[149] L. Vogel, J. Arthorne, N. Boldt, and M. Dexter. *The Official Eclipse FAQs*. The Eclipse Foundation, September 16, 2010. URL: `http://wiki.eclipse.org/The_Official_Eclipse_FAQs`.

[150] P. Wainwright. Data Display Debugger [online]. Dec., 27 2010. URL: `http://www.gnu.org/software/ddd/` [cited Dec. 27, 2010].

[151] A. H. Watson, T. J. Mccabe, and D. R. Wallace. Special Publication 500-235, Structured Testing: A Software Testing Methodology Using the Cyclomatic Complexity Metric. In *U.S. Department of Commerce/National Institute of Standards and Technology*, 1996.

[152] A. Williams and W. Kempf. *Boost C++ Libraries: Thread*. Boost, August 2010. URL: `http://www.boost.org/doc/libs/1_43_0/doc/html/thread.html`.

[153] A. Youssefi. Synchronization and Pipelining on Multicore: Shaping Parallelism for a New Generation of Processors, 2006.

[154] A. Zeller. *Why Programs Fail, Second Edition: A Guide to Systematic Debugging*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2nd edition, 2009.

[155] B. Zhu, K. Li, and H. Patterson. Avoiding the Disk Bottleneck in the Data Domain Deduplication File System. In *Proceedings of the 6th USENIX Conference on File and Storage Technologies*, FAST'08, pages 18:1–18:14, Berkeley, CA, USA, 2008. USENIX Association.

Appendix A

Relevant Compiler Options

A.1 GNU C/C++ Compiler

A number of important debugging and optimisation compiler flags for C and C++, on i386 and x86-64 based computer architectures, are described below [43].

-fno-builtin Disables the generation of special code to handle certain built-in functions.

-pipe Use operating system pipes instead of temporary files for communication between compilation stages.

-mtune=*cpu-type* Tune the generated code to *cpu-type* without limiting the set of available instructions.

-march=*cpu-type* Generate instructions exclusively for the machine type *cpu-type*.

-mfpmath=*unit* Generate floating point instructions for *unit*, where *unit* is one of:

- '**387**': Use the 387 floating point co-processor present in the majority of CPUs.
- '**sse**': Use the scalar floating point instructions present in the SSE instruction set, which is supported by the Pentium 3 and newer CPUs.
- '**both**': Attempt to utilise both floating point instruction sets.

-mmmx, -msse, -msse2, -msse3, -mssse3, -msse4, -msse4a, -msse4.1, -msse4.2 Enable the use of instructions in the MMX, SSE, SSE2, SSE3, SSSE3, SSE4.x, and SSE4A extended instruction sets.

-O0 Disable optimisations, easing debugging and reducing compilation time (the default).

-O, -O1 Optimise the code by attempting to reduce code size and execution time. This increases memory usage as well as increasing compilation time slightly.

-O2 Optimise the code even further, increasing both the performance of the program and the compilation time (`-O2` includes the optimisations from `-O1`).

-O3 Perform even more optimisations, including those from `-O2`, as well as several more aggressive code optimisations.

-Os Optimise the program for code size, including optimisations from `-O2` that do not increase size, as well as additional code size reduction optimisations.

-ffast-math Enable a number of additional mathematical optimisations that yield faster code, but may result in incorrect output (not included by any of the `-O` flags).

-g Compile the program so that it produces debugging information in the operating system's native format for use by debuggers such as GDB.

-pg Instrument the program to produce profiling information suitable for profile analysis programs such as gprof.

--coverage Compile and link program code which is instrumented for code coverage analysis by programs such as gcov.

-pthread Enable support for multi-threading with the Pthreads library.

-fopenmp Enable OpenMP `#pragma omp` directives in C and C++ and causes the compiler to generate parallel code according to the OpenMP standard (implies `-pthread`).

-floop-parallelize-all Use the Graphite data dependence analysis to identify and parallelise loops that do not contain loop carried dependences.

-ftree-parallelize-loops=*n* Parallelise loops with independent loop iterations such that the iterations are split to run in *n* threads (implies `-pthread`).

-fwhole-program Enable more aggressive optimisations by the interprocedural optimisers.

-flto Enable link-time optimisations. Expands the scope of interprocedural optimisations.

-fprofile-generate, -fprofile-use Compile the program to produce profiling information for use by the PGO feature. PGO is invoked with `-fprofile-use` after executing the program to generate the profiling data.

A.2 Intel C and C++ Compilers

Important debugging and optimisation options for the Intel C and C++ compilers for Linux are described below [72, 73].

-fno-builtin Disable inline expansion of intrinsic functions.

-pipe Use operating system pipes instead of temporary files for communication between compilation stages.

-axSSE2, -axSSE3, -axSSSE3, -axSSE4.1, -axSSE4.2 Enable the use of instructions in the SSE2, SSE3, SSSE3, SSE4.1, and SSE4.2 extended SIMD instruction sets, as well as generating generic IA-32 instructions (use `-xSSE`*`code`* to exclude the generic code).

-O0 Disable optimisations. Makes debugging somewhat easier.

-O1 Optimise the code for improved speed.

-O, -O2 Optimise the code for maximum speed (the default).

-O3 Optimise the code for maximum speed and enable more aggressive optimisations that may not benefit all programs.

-Os Enable speed optimizations that do not increase code size for small speed improvements.

-fast Enable `-O3`, `-ipo`, `-static`, `-xHost`, and additional mathematical optimisations that yield faster code, but affect accuracy.

-xHost Generate instructions for the highest instruction set and processor available on the compilation machine.

-ipo Enable multi-file interprocedural optimisation between files.

-prof-gen, -prof-use Compile the program for generating PGO profiling information. PGO is invoked with `-prof-use` after executing the program to generate the profiling data.

-restrict Enable the **restrict** keyword for disambiguating pointers.

-static Prevent linking with shared libraries.

-g Produce symbolic debug information in the resulting object file.

-guide=*n* Set the level (1–4) of guidance for auto-vectorisation, auto-parallelisation, and data transformation (default is 4).

-p, -pg Compile and link for function profiling with gprof tool.

-tcheck Enable the analysis of threaded applications (requires Intel Thread Checker).

-pthread Enable support for multi-threading with the Pthreads library.

-openmp Enable the compiler to generate multithreaded code based on the OpenMP directives.

-cilk-serialize Run a Cilk program as a C/C++ serialised program.

-parallel Enable the auto-paralleliser to generate multithreaded code for loops that can be safely executed in parallel.

-par-schedule-auto Let the compiler or runtime system determine the scheduling algorithm.

-par-num-threads=*n* Set the number of threads to be used by the auto-paralleliser.

-par-report*n*, -openmp-report*n* Control the auto-paralleliser (*n*=0–3) and OpenMP (*n*=0–2) diagnostic level.

-vec-report*n*, -opt-report *n* Control the amount of vectoriser (*n*=0–5) and optimisation (*n*=0–3) diagnostic information.

Appendix B

Baseline Program Source Code

B.1 Matrix Multiplication

```
#include <cstdlib>
#include <iostream>

static int dimension;

double** matrix_a;      // matrix a
double** matrix_b;      // matrix b
double** matrix_res;    // result matrix

void init_matrices(void) {
    // initialise the matrices with random values
    matrix_a = new double*[dimension];
    matrix_b = new double*[dimension];
    matrix_res = new double*[dimension];
    for (int r = 0; r < dimension; r++) {
        matrix_a[r] = new double[dimension];
        matrix_b[r] = new double[dimension];
        matrix_res[r] = new double[dimension];
        for (int c = 0; c < dimension; c++) {
            matrix_a[r][c] = (double)rand() / RAND_MAX;
            matrix_b[r][c] = (double)rand() / RAND_MAX;
            matrix_res[r][c] = 0.0;
        }
    }
}

void mul_matrices(void) {
    // multiply matrix a and matrix b and store in result matrix
    for (int r = 0; r < dimension; r++) {
        for (int c = 0; c < dimension; c++) {
            double sum = 0.0;
            for (int cc = 0; cc < dimension; cc++) {
                sum = sum + matrix_a[r][cc] * matrix_b[cc][c];
            }
            matrix_res[r][c] = sum;
        }
    }
}

int main(int argc, char** argv) {
    dimension = 512;
```

```
42      if (argc > 1) {
43          dimension = atoi( argv[1] );
44      }
45      std::cout << "Matrixmul" << "\n" << dimension << "␣x␣" << dimension << "\n";
46
47      init_matrices();
48      mul_matrices();
49
50      // ouput full result matrix for correctness checking
51      for (int r = 0; r < dimension; r++) {
52          for (int c = 0; c < dimension; c++)
53              std::cout << matrix_res[r][c] << "\t";
54          std:cout << "\n";
55      }
56
57      // prevent optimisations from eliminating code that modifies the
58      // result matrix when the result is not used (2x2 minimum dimension assumed)
59      std::cout << matrix_res[1][1] << "\n";
60
61      for (int i = 0; i < dimension; ++i) {
62          delete [] matrix_a[i];
63          delete [] matrix_b[i];
64          delete [] matrix_res[i];
65      }
66      delete [] matrix_a;
67      delete [] matrix_b;
68      delete [] matrix_res;
69
70      return 0;
71  }
```

Listing B.1: Classic matrix multiplication program.

B.2 Mandelbrot Set

```
1   #include <cstdio>
2   #include <cstdlib>
3
4   void mandelbrot(char* data, int* nbyte_each_line, int dimension, int width_bytes) {
5       const int iterations = 100;
6       const double limit = 2.0 * 2.0;
7       double Cimag, Creal, Zreal, Zimag;
8
9       for (int y = 0; y < dimension; ++y) {
10          int byte_count = 0;
11          int bit_number = 0;
12          char byte_accumulator = 0;
13          char* pdata = data + (y * width_bytes);
14
15          // scale y to -1.0, 1.0 on the imaginary axis
16          Cimag = ((double)y * 2.0 / dimension) - 1.0;
17          bool le_limit;
18
19          for (int x = 0; x < dimension; ++x) {
20              // scale x to -2.0, 1.0 on the real axis
21              Creal = ((double)x * 3.0 / dimension) - 2.0;
22              Zreal = Creal;
23              Zimag = Cimag;
24
25              le_limit = true;
26              for (int i = 0; i < iterations && le_limit; ++i) {
27                  double Zrealtemp = Zreal;
```

```
28                  Zreal = (Zreal * Zreal) - (Zimag * Zimag) + Creal;
29                  Zimag = 2.0 * Zrealtemp * Zimag + Cimag;
30
31                  le_limit = ((Zreal * Zreal) + (Zimag * Zimag) <= limit);
32              }
33
34              byte_accumulator = (byte_accumulator * 2) | (le_limit);
35              if (++bit_number == 8) {
36                  pdata[ byte_count++ ] = byte_accumulator;
37                  bit_number = byte_accumulator = 0;
38              }
39          }
40
41          if (bit_number != 0) {
42              byte_accumulator <<= (8 - (dimension & 7));
43              pdata[ byte_count++ ] = byte_accumulator;
44          }
45          nbyte_each_line[y] = byte_count;
46      }
47  }
48
49  int main(int argc, char** argv) {
50      int dimension = 200;
51      if (argc == 2) {
52          dimension = atoi( argv[1] );
53      }
54      printf("P4\n%d␣%d\n", dimension, dimension);
55
56      int width_bytes = (dimension/8) +1;
57      char *data = (char*)malloc( width_bytes * dimension * sizeof(char) );
58      int* nbyte_each_line = (int*)calloc( dimension, sizeof(int) );
59
60      mandelbrot(data, nbyte_each_line, dimension, width_bytes);
61
62      char* pdata = data;
63      for (int y = 0; y < dimension; y++) {
64          fwrite( pdata, nbyte_each_line[y], 1, stdout);
65          pdata += width_bytes;
66      }
67
68      free(data);
69      free(nbyte_each_line);
70
71      return 0;
72  }
```

Listing B.2: Mandelbrot Set program.

B.3 Deduplication Kernel

```
1  #include "util.h"
2  #include "dedupdef.h"
3  #include "encoder.h"
4  #include "debug.h"
5  #include "hashtable.h"
6  #include "config.h"
7  #include "hashtable_private.h"
8  #include "rabin.h"
9  #include "queue.h"
10 #include "binheap.h"
11 #include "tree.h"
12
```

```
13 #define INT64(x) ((unsigned long long)(x))
14 #define MSB64 INT64(0x8000000000000000ULL)
15 #define INITIAL_SIZE 4096
16
17 extern config * conf;
18
19 /* The pipeline model for Encode is:
20  *    DataProcess->FindAllAnchor->ChunkProcess->Compress->SendBlock
21  * Each stage has basically three steps:
22  * 1. fetch a group of items from the queue
23  * 2. process the items
24  * 3. put them in the queue for the next stage.
25  */
26
27 //We perform global anchoring in the first stage and refine the anchoring
28 //in the second stage. This array keeps track of the number of chunks in
29 //a coarse chunk.
30 u_int32 chunks_per_anchor[QUEUE_SIZE];
31 struct queue *anchor_que, *send_que; //The queues between the pipeline stages
32 int filecount = 0, chunkcount = 0;
33 send_buf_item * headitem; //header
34 int rf_win;
35 int rf_win_dataprocess;
36
37 static int write_file(int fd, u_char type, int seq_count, u_long len, u_char * content) {
38   if (xwrite(fd, &type, sizeof(type)) < 0){
39     perror("xwrite:");
40     EXIT_TRACE("xwrite␣type␣fails\n");
41     return -1;
42   }
43   if (xwrite(fd, &seq_count, sizeof(seq_count)) < 0)
44     EXIT_TRACE("xwrite␣content␣fails\n");
45   if (xwrite(fd, &len, sizeof(len)) < 0)
46     EXIT_TRACE("xwrite␣content␣fails\n");
47   if (xwrite(fd, content, len) < 0)
48     EXIT_TRACE("xwrite␣content␣fails\n");
49   return 0;
50 }
51
52 void sub_Compress(send_buf_item * item) {
53     send_body * body = (send_body*)item->str;
54     u_long len;
55     byte * pstr = NULL;
56
57     //compress the item
58     if (compress_way == COMPRESS_GZIP) {
59       unsigned long len_32;
60       len = body->len + (body->len >> 12) + (body->len >> 14) + 11;
61       if(len >> 32) {
62         perror("compress");
63         EXIT_TRACE("compress()␣failed\n");
64       }
65       len_32 = len & 0xFFFFFFFF;
66       pstr = (byte *)malloc(len);
67       if(pstr == NULL)
68         EXIT_TRACE("Memory␣allocation␣failed.\n");
69       /* compress the block */
70       if (compress(pstr, &len_32, item->content, body->len) != Z_OK) {
71         perror("compress");
72         EXIT_TRACE("compress()␣failed\n");
73         }
74       len = len_32;
75
76       u_char * key = (u_char *)malloc(SHA1_LEN);
77       if(key == NULL)
78         EXIT_TRACE("Memory␣allocation␣failed.\n");
79       memcpy(key, item->sha1, sizeof(u_char)*SHA1_LEN);
80
81       struct hash_entry * entry;
```

```
82        /* search the cache */
83        if ((entry = hashtable_search(cache, (void *)key)) == NULL) {
84          //if cannot find the entry, error
85          MEM_FREE(key);
86          printf("Error: Compress hash error\n");
87          exit(1);
88        } else {
89          //if cache hit, put the compressed data in the hash
90          struct pContent * value = ((struct pContent *)entry->v);
91          value->len = len;
92          value->content = pstr;
93          value->tag = TAG_DATAREADY;
94          hashtable_change(entry, (void *)value);
95        }
96        body->len = SHA1_LEN;
97        MEM_FREE(item->sha1);
98        MEM_FREE(item->content);
99        item->content = key;
100     }
101     return;
102 }
103
104 send_buf_item * sub_ChunkProcess(data_chunk chunk) {
105   send_buf_item * item;
106   send_body * body = NULL;
107   u_char * key;
108
109   key = (u_char *)malloc(SHA1_LEN);
110   if(key == NULL)
111     EXIT_TRACE("Memory allocation failed.\n");
112
113   Calc_SHA1Sig(chunk.start, chunk.len, key);
114   struct hash_entry * entry;
115   /* search the cache */
116   if ((entry = hashtable_search(cache, (void *)key)) == NULL) {
117     // cache miss: put it in the hashtable and the queue for the compress thread
118     struct pContent * value;
119     value = (struct pContent *)malloc(sizeof(struct pContent));
120     if(value == NULL)
121       EXIT_TRACE("Memory allocation failed.\n");
122     value->len = 0;
123     value->count = 1;
124     value->content = NULL;
125     value->tag = TAG_OCCUPY;
126     if (hashtable_insert(cache, key, value) == 0)
127       EXIT_TRACE("hashtable_insert failed");
128     item = (send_buf_item *)malloc(sizeof(send_buf_item));
129     body = (send_body *)malloc(sizeof(send_body));
130     if(item == NULL || body == NULL)
131       EXIT_TRACE("Memory allocation failed.\n");
132     body->fid = filecount;
133     body->cid = chunk.cid;
134     body->anchorid = chunk.anchorid;
135     body->len = chunk.len;
136     item->content = (u_char * )malloc(body->len + 1);
137     if(item->content == NULL)
138       EXIT_TRACE("Memory allocation failed.\n");
139     memcpy(item->content, chunk.start, body->len);
140     item->content[body->len] = 0;
141     item->sha1 = (u_char *)malloc(SHA1_LEN);
142     if(item->sha1 == NULL)
143       EXIT_TRACE("Memory allocation failed.\n");
144     memcpy(item->sha1, key, sizeof(u_char)*SHA1_LEN);
145     item->type = TYPE_COMPRESS;
146     item->str = (u_char *)body;
147   } else {
148     // cache hit: put the item into the queue for the write thread
149     struct pContent * value = ((struct pContent *)entry->v);
150     value->count += 1;
```

```
151       hashtable_change(entry, (void *)value);
152       item = (send_buf_item *)malloc(sizeof(send_buf_item));
153       body = (send_body *)malloc(sizeof(send_body));
154       if(item == NULL || body == NULL)
155         EXIT_TRACE("Memory allocation failed.\n");
156       body->fid = filecount;
157       body->cid = chunk.cid;
158       body->anchorid = chunk.anchorid;
159       body->len = SHA1_LEN;
160       item->content = key;
161       item->sha1 = NULL;
162       item->type = TYPE_FINGERPRINT;
163       item->str = (u_char *)body;
164     }
165     if (chunk.start) MEM_FREE(chunk.start);
166     return item;
167   }
168
169   /*
170    * Integrate all computationally intensive pipeline
171    * stages to improve cache efficiency.
172    */
173   void * SerialIntegratedPipeline(int tid) {
174     const int qid = tid / MAX_THREADS_PER_QUEUE;
175     data_chunk * fetchbuf[MAX_PER_FETCH];
176     int fetch_count = 0;
177     int fetch_start = 0;
178
179     u32int * rabintab = (u32int *)malloc(256*sizeof rabintab[0]);
180     u32int * rabinwintab = (u32int *)malloc(256*sizeof rabintab[0]);
181     if(rabintab == NULL || rabinwintab == NULL)
182       EXIT_TRACE("Memory allocation failed.\n");
183
184     data_chunk * tmpbuf;
185     int tmpsend_count = 0;
186     send_buf_item * senditem;
187     send_buf_item * tmpsendbuf[ITEM_PER_INSERT];
188       while (1) {
189         data_chunk item;
190         //if no item for process, get a group of items from the pipeline
191         if (fetch_count == fetch_start) {
192           int r = dequeue(&anchor_que[qid], &fetch_count, (void **)fetchbuf);
193           if (r < 0)
194             break;
195           fetch_start = 0;
196         }
197
198         if (fetch_start < fetch_count) {
199             //get one item
200             item.start = fetchbuf[fetch_start]->start;
201             item.len = fetchbuf[fetch_start]->len;
202             item.anchorid = fetchbuf[fetch_start]->anchorid;
203             MEM_FREE(fetchbuf[fetch_start]);
204             fetch_start = (fetch_start + 1)%MAX_PER_FETCH;
205
206             rabininit(rf_win, rabintab, rabinwintab);
207
208             u_char * p;
209             u_int32 chcount = 0;
210             u_char * anchor = item.start;
211             p = item.start;
212             int n = MAX_RABIN_CHUNK_SIZE;
213             while(p < item.start+item.len) {
214               if (item.len + item.start - p < n)
215                 n = item.len +item.start -p;
216               //find next anchor
217               p = p + rabinseg(p, n, rf_win, rabintab, rabinwintab);
218               //insert into anchor queue: (anchor, p-src+1)
219               tmpbuf = (data_chunk *)malloc(sizeof(data_chunk));
```

```
220             if(tmpbuf == NULL)
221               EXIT_TRACE("Memory allocation failed.\n");
222             tmpbuf->start = (u_char *)malloc(p - anchor + 1);
223             if(tmpbuf->start == NULL)
224               EXIT_TRACE("Memory allocation failed.\n");
225             tmpbuf->len = p-anchor;
226             tmpbuf->anchorid = item.anchorid;
227             tmpbuf->cid = chcount;
228
229             chcount++;
230             chunks_per_anchor[item.anchorid] = chcount;
231             memcpy(tmpbuf->start, anchor, p-anchor);
232             tmpbuf->start[p-anchor] = 0;
233             senditem = sub_ChunkProcess(*tmpbuf);
234             MEM_FREE(tmpbuf);
235
236             if (senditem->type == TYPE_COMPRESS)
237               sub_Compress(senditem);
238             tmpsendbuf[tmpsend_count] = senditem;
239             tmpsend_count ++;
240             if (tmpsend_count >= ITEM_PER_INSERT)
241               enqueue(&send_que[qid], &tmpsend_count, (void **)tmpsendbuf);
242             anchor = p;
243           }
244
245           //insert the remaining item to the anchor queue
246           if (item.start + item.len - anchor > 0) {
247             tmpbuf = (data_chunk*)malloc(sizeof(data_chunk));
248             if(tmpbuf == NULL)
249               EXIT_TRACE("Memory allocation failed.\n");
250             tmpbuf->start = (u_char *)malloc(item.start + item.len - anchor + 1);
251             if(tmpbuf->start == NULL)
252               EXIT_TRACE("Memory allocation failed.\n");
253             tmpbuf->len = item.start + item.len -anchor;
254             tmpbuf->cid = chcount;
255             tmpbuf->anchorid = item.anchorid;
256             chcount ++;
257             chunks_per_anchor[item.anchorid] = chcount;
258             memcpy(tmpbuf->start, anchor, item.start + item.len -anchor);
259             tmpbuf->start[item.start + item.len -anchor] = 0;
260
261             senditem = sub_ChunkProcess(*tmpbuf);
262             if (senditem->type == TYPE_COMPRESS)
263               sub_Compress(senditem);
264             tmpsendbuf[tmpsend_count] = senditem;
265             tmpsend_count ++;
266             if (tmpsend_count >= ITEM_PER_INSERT)
267               enqueue(&send_que[qid], &tmpsend_count, (void **)tmpsendbuf);
268           }
269           MEM_FREE(item.start);
270         }
271     }
272     if (tmpsend_count > 0)
273       enqueue(&send_que[qid], &tmpsend_count, (void **)tmpsendbuf);
274
275     //count the number of compress threads that have finished
276     queue_signal_terminate(&send_que[qid]);
277     MEM_FREE(rabintab);
278     MEM_FREE(rabinwintab);
279     return NULL;
280 }
281
282 /*
283  * read file and send it to FindAllAnchor thread
284  */
285 void * DataProcess(int nqueues, int fd){
286   int qid = 0;
287   u_long tmp;
288   u_char * p;
```

```
289   u_char * anchor;
290   u_char * src = (u_char *)malloc(MAXBUF*2);
291   u_char * left = (u_char *)malloc(MAXBUF);
292   u_char * newb = (u_char *) malloc(MAXBUF);
293   if(src == NULL || left == NULL || newb == NULL)
294     EXIT_TRACE("Memory allocation failed.\n");
295   u_long srclen = 0;
296   int left_bytes = 0;
297   char more = 0;
298   data_chunk * tmpbuf[ANCHOR_DATA_PER_INSERT];
299   int tmp_count = 0;
300   int anchorcount = 0;
301
302   u32int * rabintab = (u32int *)malloc(256*sizeof rabintab[0]);
303   u32int * rabinwintab = (u32int *)malloc(256*sizeof rabintab[0]);
304   if(rabintab == NULL || rabinwintab == NULL)
305     EXIT_TRACE("Memory allocation failed.\n");
306   rf_win_dataprocess = 0;
307   rabininit(rf_win_dataprocess, rabintab, rabinwintab);
308   u_long n = MAX_RABIN_CHUNK_SIZE;
309
310   //read from the file
311   while ((srclen = read(fd, newb, MAXBUF)) >= 0) {
312     if (srclen) more = 1;
313     else {
314       if (!more) break;
315       more =0;
316     }
317     memset(src, 0, sizeof(u_char)*MAXBUF);
318     if (left_bytes > 0){
319       memcpy(src, left, left_bytes* sizeof(u_char));
320       memcpy(src+left_bytes, newb, srclen *sizeof(u_char));
321       srclen+= left_bytes;
322       left_bytes = 0;
323     } else
324       memcpy(src, newb, srclen * sizeof(u_char));
325     tmp = 0;
326     p = src;
327
328     while (tmp < srclen) {
329       anchor = src + tmp;
330       p = anchor + ANCHOR_JUMP;
331       if (tmp + ANCHOR_JUMP >= srclen) {
332         if (!more)
333           p = src + srclen;
334         else {
335           //move p to the next 00 point
336           n = MAX_RABIN_CHUNK_SIZE;
337           memcpy(left, src+tmp, (srclen - tmp) * sizeof(u_char));
338           left_bytes= srclen -tmp;
339           break;
340         }
341       } else {
342         if (srclen - tmp < n)
343           n = srclen - tmp;
344         p = p + rabinseg(p, n, rf_win_dataprocess, rabintab, rabinwintab);
345       }
346
347       tmpbuf[tmp_count] = (data_chunk *)malloc(sizeof(data_chunk));
348       if(tmpbuf[tmp_count] == NULL)
349         EXIT_TRACE("Memory allocation failed.\n");
350       tmpbuf[tmp_count]->start = (u_char *)malloc(p - anchor + 1);
351       if(tmpbuf[tmp_count]->start == NULL)
352         EXIT_TRACE("Memory allocation failed.\n");
353       tmpbuf[tmp_count]->len = p-anchor;
354       tmpbuf[tmp_count]->anchorid = anchorcount;
355       anchorcount ++;
356       memcpy(tmpbuf[tmp_count]->start, anchor, p-anchor);
357       tmpbuf[tmp_count]->start[p-anchor] = 0;
```

```
358         tmp_count ++;
359
360         //send a group of items into the next queue in round-robin fashion
361         if (tmp_count >= ANCHOR_DATA_PER_INSERT) {
362           enqueue(&anchor_que[qid], &tmp_count, (void **)tmpbuf);
363           qid = (qid+1) % nqueues;
364         }
365         tmp += p - anchor;//ANCHOR_JUMP;
366       }
367     }
368     if (tmp_count >= 0) {
369       enqueue(&anchor_que[qid], &tmp_count, (void **)tmpbuf);
370       qid = (qid+1) % nqueues;
371     }
372     //terminate all output queues
373     for(int i=0; i<nqueues; i++)
374       queue_signal_terminate(&anchor_que[i]);
375     MEM_FREE(rabintab);
376     MEM_FREE(rabinwintab);
377     MEM_FREE(src);
378     MEM_FREE(left);
379     MEM_FREE(newb);
380     return 0;
381   }
382
383   /*
384    * write blocks to the file
385    */
386   void *
387   SendBlock(int nqueues, config * conf) {
388     //NOTE: We *must* start with the first queue in order to get the header first
389     int qid = 0;
390     int fd = 0;
391     struct hash_entry * entry;
392
393     if (conf->method == METHOD_STDOUT)
394       fd = STDOUT_FILENO;
395     else {
396       fd = open(conf->outfile, O_CREAT|O_TRUNC|O_WRONLY|O_TRUNC);
397       if (fd < 0) {
398           perror("SendBlock open");
399           return NULL;
400       }
401       fchmod(fd, S_IRGRP | S_IWUSR | S_IRUSR | S_IROTH);
402     }
403
404     send_buf_item * fetchbuf[ITEM_PER_FETCH];
405     int fetch_count = 0, fetch_start = 0;
406     send_buf_item * item;
407
408     SearchTree T;
409     T = TreeMakeEmpty(NULL);
410     Position pos;
411     struct tree_element * tele;
412     struct heap_element * hele;
413
414     u_int32 reassemble_count = 0, seq_count = 0, anchor_count = 0;
415     send_body * body = NULL;
416     send_head * head = NULL;
417
418     head = (send_head *)headitem->str;
419     if (xwrite(fd, &headitem->type, sizeof(headitem->type)) < 0){
420       perror("xwrite:");
421       EXIT_TRACE("xwrite type fails\n");
422       return NULL;
423     }
424     int checkbit = CHECKBIT;
425     if (xwrite(fd, &checkbit, sizeof(int)) < 0)
426       EXIT_TRACE("xwrite head fails\n");
```

```
427   if (xwrite(fd, head, sizeof(send_head)) < 0)
428     EXIT_TRACE("xwrite head fails\n");
429   MEM_FREE(head);
430
431   while(1) {
432     //get a group of items
433     if (fetch_count == fetch_start) {
434       //process queues in round-robin fashion
435       int r, i = 0;
436       do {
437         r = dequeue(&send_que[qid], &fetch_count, (void **)fetchbuf);
438         qid = (qid+1) % nqueues;
439         i++;
440       } while(r<0 && i<nqueues);
441       if (r<0)
442         break;
443       fetch_start = 0;
444     }
445     item = fetchbuf[fetch_start];
446     fetch_start ++;
447     if (item == NULL) break;
448
449     switch (item->type) {
450     case TYPE_FINGERPRINT:
451     case TYPE_COMPRESS:
452       //process one item
453       body = (send_body *)item->str;
454       if (body->cid == reassemble_count && body->anchorid == anchor_count) {
455         //the item is the next block to write to file, write it.
456         if ((entry = hashtable_search(cache, (void *)item->content)) != NULL) {
457           struct pContent * value = ((struct pContent *)entry->v);
458           if (value->tag == TAG_WRITTEN) {
459             //if the data has been written, just write SHA-1
460             write_file(fd, TYPE_FINGERPRINT, seq_count, body->len, item->content);
461             MEM_FREE(item->sha1);
462             MEM_FREE(item->content);
463             MEM_FREE(item);
464           } else {
465             //if the data has not been written, write the compressed data
466             if (value->tag == TAG_DATAREADY) {
467               write_file(fd, TYPE_COMPRESS, seq_count, value->len, value->content);
468               value->len = seq_count;
469               value->tag = TAG_WRITTEN;
470               hashtable_change(entry, (void *)value);
471             } else {
472               printf("Error: Illegal tag\n");
473             }
474             MEM_FREE(item->sha1);
475             MEM_FREE(item->content);
476             MEM_FREE(item);
477           }
478         } else
479           printf("Error: Cannot find entry\n");
480         MEM_FREE(body);
481         reassemble_count ++;
482         if (reassemble_count == chunks_per_anchor[anchor_count]) {
483           reassemble_count = 0;
484           anchor_count ++;
485         }
486         seq_count ++;
487         //check whether there are more data in order in the queue that
488         // can be written to the file
489         pos = TreeFindMin(T);
490         if (pos != NULL && (pos->Element)->aid == anchor_count) {
491           tele = pos->Element;
492           hele = FindMin(tele->queue);
493           while (hele!=NULL && hele->cid==reassemble_count && tele->aid==anchor_count) {
494             if ((entry = hashtable_search(cache, (void *)hele->content)) != NULL) {
495               struct pContent * value = ((struct pContent *)entry->v);
```

```
496                 if (value->tag == TAG_WRITTEN) {
497                   write_file(fd, TYPE_FINGERPRINT, seq_count, hele->len, hele->content);
498                 } else {
499                   if (value->tag == TAG_DATAREADY) {
500                     write_file(fd, TYPE_COMPRESS, seq_count, value->len, value->content);
501                     value->len = seq_count;
502                     value->tag = TAG_WRITTEN;
503                     MEM_FREE(value->content);
504                     hashtable_change(entry, (void *)value);
505                   } else
506                     printf("Error: Illegal tag\n");
507                 }
508               } else
509                 printf("Error: Cannot find entry\n");
510               MEM_FREE(body);
511               if (hele->content) MEM_FREE(hele->content);
512               MEM_FREE(hele);
513               seq_count ++;
514               DeleteMin(tele->queue);
515
516               reassemble_count ++;
517               if (reassemble_count == chunks_per_anchor[anchor_count]) {
518                 reassemble_count = 0;
519                 anchor_count ++;
520               }
521               if (IsEmpty(tele->queue)) {
522                 T = TreeDelete(tele, T);
523                 pos = TreeFindMin(T);
524                 if (pos == NULL) break;
525                 tele = pos->Element;
526               }
527               hele = FindMin(tele->queue);
528             }
529           }
530         } else {
531           // the item is not the next block to write, put it in a queue.
532           struct heap_element* p;
533           p = (struct heap_element*)malloc(sizeof(struct heap_element));
534           if(p == NULL)
535             EXIT_TRACE("Memory allocation failed.\n");
536
537           p->content = item->content;
538           p->cid = body->cid;
539           p->len = body->len;
540           p->type = item->type;
541           MEM_FREE(item->sha1);
542           MEM_FREE(item);
543           pos = TreeFind(body->anchorid, T);
544           if (pos == NULL) {
545             struct tree_element* tree;
546             tree = (struct tree_element*)malloc(sizeof(struct tree_element));
547             if(tree == NULL)
548               EXIT_TRACE("Memory allocation failed.\n");
549             tree->aid = body->anchorid;
550             tree->queue = Initialize(INITIAL_SIZE);
551             Insert(p, tree->queue);
552             T = TreeInsert(tree, T);
553           } else
554             Insert(p, pos->Element->queue);
555         }
556         MEM_FREE(body);
557         break;
558       case TYPE_HEAD:
559         break;
560       case TYPE_FINISH:
561         break;
562       }
563       MEM_FREE(item);
564     }
```

```
565
566   //write the blocks left in the queue to the file
567   pos = TreeFindMin(T);
568   if (pos != NULL) {
569     tele = pos->Element;
570     hele = FindMin(tele->queue);
571     while (hele!= NULL) {
572       if ((entry = hashtable_search(cache, (void *)hele->content)) != NULL) {
573         struct pContent * value = ((struct pContent *)entry->v);
574         if (value->tag == TAG_WRITTEN) {
575           write_file(fd, TYPE_FINGERPRINT, seq_count, hele->len, hele->content);
576           MEM_FREE(hele->content);
577           MEM_FREE(hele);
578         } else {
579           if (value->tag == TAG_DATAREADY) {
580             write_file(fd, TYPE_COMPRESS, seq_count, value->len, value->content);
581             value->len = seq_count;
582             value->tag = TAG_WRITTEN;
583             hashtable_change(entry, (void *)value);
584             MEM_FREE(value->content);
585           } else
586             printf("Error: Illegal tag\n");
587         }
588       } else
589         printf("Error: Cannot find entry\n");
590       seq_count ++;
591       DeleteMin(tele->queue);
592       if (IsEmpty(tele->queue)) {
593         T = TreeDelete(tele, T);
594         pos = TreeFindMin(T);
595         if (pos == NULL) break;
596         tele = pos->Element;
597       }
598       hele = FindMin(tele->queue);
599     }
600   }
601   u_char type = TYPE_FINISH;
602   if (xwrite(fd, &type, sizeof(type)) < 0){
603     perror("xwrite:");
604     EXIT_TRACE("xwrite type fails\n");
605     return NULL;
606   }
607   close(fd);
608   return NULL;
609 }
610
611 /*-------------------------------------------------------------------------*/
612 /* Encode
613  * Compress an input stream
614  * Arguments:
615  *   conf:    Configuration parameters
616  */
617 void Encode(config * conf) {
618   int32 fd;
619   struct stat filestat;
620   //queue allocation & initialization
621   const int nqueues = (conf->nthreads / MAX_THREADS_PER_QUEUE) +
622                       ((conf->nthreads % MAX_THREADS_PER_QUEUE != 0) ? 1 : 0);
623   anchor_que = (struct queue *)malloc(sizeof(struct queue) * nqueues);
624   send_que = (struct queue *)malloc(sizeof(struct queue) * nqueues);
625   if( (anchor_que == NULL) || (send_que == NULL)) {
626     printf("Out of memory\n");
627     exit(1);
628   }
629   int threads_per_queue;
630   for(int i=0; i<nqueues; i++) {
631     if (i < nqueues -1 || conf->nthreads %MAX_THREADS_PER_QUEUE == 0) {
632       //all but last queue
633       threads_per_queue = MAX_THREADS_PER_QUEUE;
```

```
634       } else //remaining threads work on last queue
635         threads_per_queue = conf->nthreads %MAX_THREADS_PER_QUEUE;
636       //call queue_init with threads_per_queue
637       queue_init(&anchor_que[i], QUEUE_SIZE, 1);
638       queue_init(&send_que[i], QUEUE_SIZE, threads_per_queue);
639     }
640     memset(chunks_per_anchor, 0, sizeof(chunks_per_anchor));
641
642     //initialize output file header
643     headitem = (send_buf_item *)malloc(sizeof(send_buf_item));
644     if(headitem == NULL)
645       EXIT_TRACE("Memory allocation failed.\n");
646     headitem->type = TYPE_HEAD;
647     send_head * head = (send_head *)malloc(sizeof(send_head));
648     if(head == NULL)
649       EXIT_TRACE("Memory allocation failed.\n");
650
651     strncpy(head->filename, conf->infile, LEN_FILENAME);
652     filecount ++;
653     head->fid = filecount;
654     chunkcount = 0;
655     headitem->str = (u_char * )head;
656
657      /* src file stat */
658     if (stat(conf->infile, &filestat) < 0)
659       EXIT_TRACE("stat() %s failed: %s\n", conf->infile, strerror(errno));
660     if (!S_ISREG(filestat.st_mode))
661       EXIT_TRACE("not a normal file: %s\n", conf->infile);
662     /* src file open */
663     if((fd = open(conf->infile, O_RDONLY | O_LARGEFILE)) < 0)
664       EXIT_TRACE("%s file open error %s\n", conf->infile, strerror(errno));
665
666     DataProcess(nqueues, fd);
667     //do the processing
668     SerialIntegratedPipeline(0);
669     SendBlock(nqueues, conf);
670
671     for(int i=0; i<nqueues; i++) {
672       queue_destroy(&anchor_que[i]);
673       queue_destroy(&send_que[i]);
674     }
675     /* clean up with the src file */
676     if (conf->infile != NULL)
677       close(fd);
678   }
```

Listing B.3: Deduplication serial encoding pipeline.

Appendix C

Parallelised Program Source Code

The source code files for each implementation of the three programs can be found on my research site located at `http://taaaki.za.net/parsrc.tar.gz`.

C.1 Shared Task Queue for Pthreads

```
1  typedef struct {
2      int start;
3      int end;
4  } thread_data;
5
6  struct worker_node {
7      worker_node* next;
8      worker_node* prev;
9      thread_data* data;
10 };
11
12 class WorkQueue {
13 private:
14     worker_node* head;
15     worker_node* tail;
16     pthread_mutex_t qlock;
17
18 public:
19     WorkQueue() {
20         this->head = NULL;
21         this->tail = NULL;
22         pthread_mutex_init(&(this->qlock), NULL);
23     }
24
25     ~WorkQueue() {
26         thread_data* my_work;
27         while (true) {
28             my_work = pop();
29
30             if (my_work == NULL) {
31                 break;
32             }
33             else {
34                 delete my_work;
35             }
```

```
36          }
37          pthread_mutex_destroy(&(this->qlock));
38      }
39
40      thread_data* pop() {
41          thread_data* res = NULL;
42          pthread_mutex_lock (&(this->qlock));
43          if (this->head != NULL) {
44              worker_node* curr = this->head;
45              res = curr->data;
46              this->head = curr->prev;
47
48              if (this->head != NULL) {
49                  this->head->next = NULL;
50              }
51              else {
52                  this->tail = NULL;
53              }
54              delete curr;
55          }
56          pthread_mutex_unlock (&(this->qlock));
57          return res;
58      }
59
60      void push(thread_data* data) {
61          worker_node* newjob = new worker_node();
62          newjob->prev = NULL;
63          newjob->data = data;
64
65          pthread_mutex_lock (&(this->qlock));
66          newjob->next = this->tail;
67
68          if (this->head == NULL) {
69              this->head = newjob;
70          }
71          else {
72              this->tail->prev = newjob;
73          }
74          this->tail = newjob;
75          pthread_mutex_unlock (&(this->qlock));
76      }
77  };
```

Listing C.1: Shared Task Queue using Pthreads mutex.