GPRM: A HIGH PERFORMANCE PROGRAMMING FRAMEWORK FOR MANYCORE PROCESSORS

ASHKAN TOUSIMOJARAD

SUBMITTED IN FULFILMENT OF THE REQUIREMENTS FOR THE DEGREE OF
Doctor of Philosophy

SCHOOL OF COMPUTING SCIENCE

COLLEGE OF SCIENCE AND ENGINEERING
UNIVERSITY OF GLASGOW

NOVEMBER 2015

Abstract

Processors with large numbers of cores are becoming commonplace. In order to utilise the available resources in such systems, the programming paradigm has to move towards increased parallelism. However, increased parallelism does not necessarily lead to better performance. Parallel programming models have to provide not only flexible ways of defining parallel tasks, but also efficient methods to manage the created tasks. Moreover, in a general-purpose system, applications residing in the system compete for the shared resources. Thread and task scheduling in such a multiprogrammed multithreaded environment is a significant challenge.

In this thesis, we introduce a new task-based parallel reduction model, called the Glasgow Parallel Reduction Machine (GPRM). Our main objective is to provide high performance while maintaining ease of programming. GPRM supports native parallelism; it provides a modular way of expressing parallel tasks and the communication patterns between them. Compiling a GPRM program results in an Intermediate Representation (IR) containing useful information about tasks, their dependencies, as well as the initial mapping information. This compile-time information helps reduce the overhead of runtime task scheduling and is key to high performance. Generally speaking, the granularity and the number of tasks are major factors in achieving high performance. These factors are even more important in the case of GPRM, as it is highly dependent on tasks, rather than threads.

We use three basic benchmarks to provide a detailed comparison of GPRM with Intel OpenMP, Cilk Plus, and Threading Building Blocks (TBB) on the Intel Xeon Phi, and with GNU OpenMP on the Tilera TILEPro64. GPRM shows superior performance in almost all cases, only by controlling the number of tasks. GPRM also provides a low-overhead mechanism, called "Global Sharing", which improves performance in multiprogramming situations.

We use OpenMP, as the most popular model for shared-memory parallel programming as the main GPRM competitor for solving three well-known problems on both platforms: LU factorisation of Sparse Matrices, Image Convolution, and Linked List Processing. We focus on proposing solutions that best fit into the GPRM's model of execution. GPRM outperforms

OpenMP in all cases on the TILEPro64. On the Xeon Phi, our solution for the LU Factorisation results in notable performance improvement for sparse matrices with large numbers of small blocks. We investigate the overhead of GPRM's task creation and distribution for very short computations using the Image Convolution benchmark. We show that this overhead can be mitigated by combining smaller tasks into larger ones. As a result, GPRM can outperform OpenMP for convolving large 2D matrices on the Xeon Phi. Finally, we demonstrate that our parallel worksharing construct provides an efficient solution for Linked List processing and performs better than OpenMP implementations on the Xeon Phi.

The results are very promising, as they verify that our parallel programming framework for manycore processors is flexible and scalable, and can provide high performance without sacrificing productivity.

Acknowledgements

First and foremost, I would like to thank my PhD supervisor, Dr. Wim Vanderbauwhede for his support, encouragement and guidance over the past four years. I would also like to extend my gratitude to my second supervisor, Dr. William Paul Cockshott for his assistance and advice throughout my PhD study. It has been greatly appreciated and I cannot thank you both enough. In addition, I would like to acknowledge the Scottish Informatics and Computer Science Alliance (SICSA) for funding this research.

I would like to express my heartfelt thanks to my great family, my mother, Firoozeh, my father, Khosro, and my sister, Shirin for all their love and encouragement.

I would also like to take this opportunity to thank Dr. Jeremy Singer and Professor Sven-Bodo Scholz, my viva examiners, for their helpful comments and suggestions.

Special thanks to all the staff at the school of computing science for their help and support during these years.

Last but not least, I would like to thank Lily for her kindness, support, and encouragement.

To my family.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

For the past few decades, microprocessor performance scaling was the result of advances in integrated circuit technology, such as integrating more transistors on a single chip and scaling of the processor clock [4] [5]. According to the Moore's law [6], the number of transistors on a chip roughly doubled every eighteen months. The industry trend was to make use of multiple functional units within a single core in parallel, which led to instruction level parallelism.

Superscalars could execute multiple instructions in one clock cycle by simultaneously dispatching them to different functional units on the processor. However, typical instruction streams had a limited amount of potential parallelism. Therefore, the use of superscalar hardware alone was not sufficient for the design of modern systems [7]. Moreover, building superscalar cores with the additional logic to find parallel instructions dynamically became really expensive [8].

A more recent approach is *hardware multithreading*, where multiple hardware threads run simultaneously on a single core. An example of this design is *Simultaneous Multi Threading (SMT)* [9], where multiple streams send instructions into a superscalar scheduler [10].

Today, extracting better performance from uniprocessors, even with better fabrication technologies or the use of pipelined architectures, has come to an end. Instead, the performance growth comes from an increase in the number of processing elements on a single die. There are many reasons behind the current shift away from uniprocessor designs, one of which is the power consumption that could be saved by eliminating the need to increase the clock speed.

1.1 The Shift Towards Chip Multiprocessors (CMPs)

Extracting instruction parallelism in the superscalars became more and more complex and costly. Moreover, compared to a wide-issue superscalar, a machine with simpler cores could better utilise the silicon resources.

There were also some physical facts involved. Power dissipation would become a significant problem in higher frequencies; also no more power saving could be achieved by voltage scaling, as it had stopped due to the high leakage. The clock speed could not be increased as before without overheating, and finally, the cooling technologies would not scale up fast enough to cope with the power requirements [11].

Furthermore, wire delays limited the improvement of instruction throughput and the scaling of memory-oriented structures such as caches and register files. Smaller transistors were faster and required less power, but wires did not scale in the same way. As a result, communication delays became significant for global signals, and therefore centralised structures and global interconnects were no longer efficient [12]. A modular design was needed.

The "Free Lunch" for hardware designers and software developers was over [13]. It was time to decentralise the micro-architecture and move towards CMP and multicore processor designs [1] [11] with simpler and more power efficient cores.

Another trend in parallel computing was the rise of using GPUs [2] for computationally intensive problems. However, we continue our discussion by focusing only on multicore processors.

Instead of scaling the clock frequency, computer scientists and engineers started to increase the number of cores in each generation and designed simpler but more power efficient cores. Using a CMP with two cores for example, could result in the same or better throughput with half of the clock speed of a uniprocessor for server-oriented workloads. The processing time for each request is doubled, but note that the request processing time is mostly limited by the memory or disk rather than the processor. Since for a two-way CMP, two requests could be resolved simultaneously, if there is no significant memory contention, the throughput will be a little better or at least the same. More importantly, at the system level, with a lower clock rate, the system can be designed with an almost linear reduction in operating voltage. Power is proportional to the square of the voltage; therefore, the power required for achieving the same performance is a quarter of the uniprocessor power for each core, and half of it for the whole CMP. This is however not the whole story about the power saving, as static power dissipation and minimum voltage levels required by transistors could be the

[1] The term *multicore processor* is used for a processor having multiple computing units. It is an example of a CMP with the placement of several independent execution cores with all execution resources onto a single processor chip.

[2] Graphics Processing Unit

limiting factors [8].

It is also beneficial for the throughput-oriented workloads to use CMPs with multiple smaller cores rather than a few larger ones. Superscalar-related hardware is wasted for such applications, because of the several memory stalls and the limited instruction-level parallelism. Consider a typical server that receives hundreds of requests at once. By replacing the large cores with multiple smaller ones, although the processing time on each core becomes larger, there is enough work to keep all cores busy at the same time [8].

As stated above, on-chip wires do not scale in the same way as transistors do. In a digital system, most of the power is dissipated in wires. Also, the time spent on global communication relative to the local processing time is significant. Therefore, the performance and power efficiency of most on-chip systems today is constrained by their interconnection. Network-on-Chip (NoC) [14] has emerged as a promising solution to overcome such communication challenges in modern manycore processors.

Today's multicore and manycore chips do multiple jobs at a time to provide better performance and power efficiency. Moving toward manycore platforms [15] could provide the capability to handle terabytes of data. Instead of focusing solely on executing individual tasks faster, many more tasks will be executed in parallel across a group of simpler cores. The transition towards these parallel architectures makes today an exciting time to investigate challenges in parallel computing.

1.2 Parallel Computing

Processors with large numbers of cores are becoming commonplace. In order to take advantage of the available resources in such systems, programmers need to know about parallel computing. We start by describing two classic performance models: Amdahl's law and Gustafson's law.

Amdahl's law [16] specifies the expected speedup after parallelisation on N processors, considering P as the parallelised portion of the program:

$$Speedup(P, N) = \frac{1}{(1 - P) + P/N} \tag{1.1}$$

Amdahl's law implies that if P is small, optimisations will have little effect, and also that the maximum speedup cannot go beyond $1/(1 - P)$.

By adding a hardware model to the Amdahl's law, its variations for symmetric, asymmetric, and dynamic multicores are introduced in [17].

While Amdahl's law considers a fixed problem size to be solved on a changeable parallel machine, Gustafson's law [18] indicates that the problem sizes change to exploit the capabilities of new computers, hence on a more powerful machine, bigger problems could be solved in the same time. Gustafson's law states that when the problem size grows, the parallel fraction of the problem scales much faster than the serial fraction. If that is the case, the serial portion becomes less significant compared to the parallel portion and speedup grows by increasing the number of processors (cores). Both of these laws are correct. It only depends on whether the goal is to solve an existing problem faster or to solve a bigger problem in the same time.

1.2.1 From Threads to Tasks

A program in its execution is called a *Process*. Whenever a process is created by the Operating System (OS), resources such as registers and pages of memory get allocated to that process. *Thread*s of a process share such resources, including the address space. Each thread, only needs a few dedicated resources, such as a program counter and a region of memory (stack) to keep track of its own data. Threads can run simultaneously on a single or multiple cores/processors, and work concurrently in order to execute a program [19]. Parallel programming on shared memory multicore processors has been historically focused on thread-based parallelism.

Taking the abstraction one level further, from *Thread*s to *Task*s is becoming more important as the number of cores grows in modern processors. A task is any form of finite computation (a unit of work) that can be run in parallel with other tasks, if their data dependencies allow [3] [20]. Tasks are more lightweight and hence can be used to express parallelism at a finer granularity. Threads, on the other hand, need their own stack [20]. Moreover, scheduling both tasks on threads and threads on cores imposes increasingly larger overhead to a system with many cores. Furthermore, without a cost model, it is very difficult, if not impossible, for the programmers to determine the optimal number of threads for an application. The most promising solution is to divide the program into tasks and rely on the runtime system to schedule these tasks and achieve the expected speedup [21].

The concept of *task* already exists in many parallel programming models. Compared to pure data-parallel approaches, task-based programming offers more flexibility in many situations [3]. Programmers express parallelism by defining tasks in their applications and runtime libraries schedule the tasks on threads. However, still in many task-based parallel programming models, choosing the right number of threads is key to performance. Hence, the onus is on the programmer to decide not only about the granularity/number of tasks, but also about the optimal number of threads in order to obtain good performance.

1.3 Thesis Statement

The use of manycore platforms is still limited due to the difficulties associated with programming them.

My thesis attempts to answer the question of whether a task-based parallel programming model using a coordination language with implicit parallel semantics is a suitable approach to manycore programming in terms of performance and productivity.

I assert that The Glasgow Parallel Reduction Machine (GPRM) parallel programming framework can provide performance at the same level or better than the state-of-the-art parallel programming approaches, while not increasing the burden of application development for shared memory manycore processors.

In this work, I will explore popular approaches for parallel programming targeting shared memory manycore platforms. The trade-off between high performance and productivity (ease of programming) should be taken into account. This would help to identify the pros and cons of the existing parallel programming models, and thus would be useful for proposing new techniques. More importantly, this work should also investigate whether GPRM can utilise both task and data parallelism efficiently on manycore processors and hence improve the performance.

Our parallel programming approach (GPRM) uses partial evaluation in order to infer a task graph at compile time. Managing the created tasks at runtime is challenging. The main contribution of this work is the development of an efficient runtime system for such a task-based parallel programming approach. A major contribution is adding an efficient stealing mechanism that matches the execution model of the new framework. Adding support for loop-level parallelism as well as designing a high level coordination language are other contributions. None of these steps could be finalised without measuring their impact on performance. I also looked at the important factors in concurrent running of applications on manycore processors, known as multiprogramming, and implemented a low overhead mechanism in GPRM runtime system for such situations.

OpenMP is chosen as the main competitor of GPRM. Solving problems in GPRM and comparing the results with the well-known solutions implemented in OpenMP provides a better view on when to choose GPRM over OpenMP and how to utilise its features.

1.4 Thesis Organisation

The remainder of this thesis is organised as follows:

Chapter 2: Background

In the second chapter, we mainly talk about designing parallel programs and the well-known parallel programming paradigms and models. We also cover task scheduling and load balancing topics. At the end of this chapter, we review some of the related work on multicore and manycore processors, with regards to the architecture, programming, and performance evaluation.

Chapter 3: Hardware Platforms

This chapter begins with an introduction to the classification of parallel platforms followed by a discussion on memory organisations. We then focus on shared memory multicore and manycores, specifically on the two platforms that are used for the purposes of this study: the Tilera TILEPro64 and the Intel Xeon Phi. We have published a paper on cache-aware parallel programming on the TILEPro64:

[Paper 1] Tousimojarad A, Vanderbauwhede W. Cache-aware parallel programming for manycore processors. In: Highly Efficient Accelerators and Reconfigurable Technologies (HEART2013); 2013.

Chapter 4: Task-based Parallel Models for Shared Memory Programming

The fourth chapter starts with a rather detailed introduction on three selected programming models: OpenMP, Cilk Plus and TBB. Three "Basic Benchmarks" are introduced in this chapter to examine the performance characteristics of these programming models on the Xeon Phi. We then run the three benchmarks together to see how these models perform in a multiprogramming environment. We continue this chapter by focusing on multiprogramming and "information sharing" (between the programs running concurrently) using OpenMP-based benchmarks on the TILEPro64 . The content of this chapter is published in the following papers:

[Paper 2] Tousimojarad A, Vanderbauwhede W. An Efficient Thread Mapping Strategy for Multiprogramming on Manycore Processors. In: Parallel Computing: Accelerating Computational Science and Engineering (CSE). vol. 25 of Advances in Parallel Computing. IOS Press; 2014. p. 63–71

[Paper 3] Tousimojarad A, Vanderbauwhede W. Comparison of Three Popular Parallel Programming Models on the Intel Xeon Phi. In: Euro-Par 2014: Parallel Processing Workshops. Springer; 2014. p. 314–325

Chapter 5: GPRM: The Glasgow Parallel Reduction Machine

This chapter covers details about the design and implementation of the GPRM framework. It covers the front-end language and its compiler, details about the runtime system, model of execution, task scheduling as well as "information sharing" when running multiple GPRM instances concurrently. The following papers are published:

[Paper 4] Tousimojarad A. A parallel task composition approach to manycore programming. PLACES 2013. 2013;29

[Paper 5] Tousimojarad A, Vanderbauwhede W. The Glasgow Parallel Reduction Machine: Programming Shared-memory Many-core Systems using Parallel Task Composition. EPTCS. 2013;137:79–94

[Paper 6] Tousimojarad A, Vanderbauwhede W. Steal Locally, Share Globally. International Journal of Parallel Programming. 2015;43(5):894–917

Chapter 6: Comparison of GPRM with Popular Parallel Programming Models

In this chapter, GPRM is compared with OpenMP on the TILEPro64 and with OpenMP, Cilk Plus, and TBB on the Xeon Phi. Both programming and performance aspects are considered. For the only case that OpenMP has better performance compared to GPRM, a detailed comparison for different cases shows that GPRM performs better on average. The chapter ends with a comparison of GPRM with the three models for two multiprogramming cases on the Xeon Phi. The following paper is published based on the results of this chapter:

[Paper 7] Tousimojarad A, Vanderbauwhede W. Number of Tasks, not Threads, is Key. In: 2015 23rd Euromicro International Conference on Parallel, Distributed, and Network-Based Processing. IEEE; 2015.

Chapter 7: Parallel Lower-Upper Factorisation of Sparse Matrices

In this section, we first introduce parallel loops in GPRM and measure their performance on the TILEPro64. We then propose a solution in GPRM to solve a fundamental linear algebra problem: LU factorisation of sparse matrices. The performance results of the proposed approach is compared with those of a task-based OpenMP approach. The following paper is published based on this work:

[Paper 8] Tousimojarad A, Vanderbauwhede W. A Parallel Task-based Approach to Linear Algebra. In: Parallel and Distributed Computing (ISPDC), 2014 IEEE 13th International Symposium on. IEEE; 2014. p. 59–66

Chapter 8: Parallel Image Convolution

This chapter is focused on the comparison of GPRM with OpenMP for parallel 2D image convolution on both platforms. Two algorithms for solving the problem for a separable kernel are covered. The main focus of this chapter is on optimisation techniques for the Xeon Phi. In addition, task agglomeration is used as a parallelisation technique to improve the performance of GPRM. We have published parts of this work in:

[Paper 9] Chimeh M, Cockshott P, Oehler SB, Tousimojarad A, Xu T. Compiling Vector Pascal to the XeonPhi. Concurrency and Computation: Practice and Experience. 2015; *(The names are ordered alphabetically)*

Chapter 9: Parallel Linked List Processing

In this chapter, we present a detailed comparison between GPRM and OpenMP using a parallel linked list processing benchmark. We discuss that a cutoff-based solution is required for achieving high performance. We then compare the GPRM implementation with two OpenMP implementations: the conventional approach with one task per list element and a cutoff-based approach. We show that GPRM outperforms OpenMP in almost all cases. The following paper is published based on this work:

[Paper 10] Tousimojarad A, Vanderbauwhede W. Efficient Parallel Linked List Processing. Advances in Parallel Computing. IOS Press; 2016.

Chapter 10: Conclusion and Future Work

Lastly, we present the thesis conclusions and suggest some directions for future development of the GPRM framework.

Chapter 2

Background

The emergence of multicore and manycore processors has changed the computing landscape. There are various parallel programming approaches to exploit such architectures. In this chapter, we present an overview of designing parallel programs and introduce the most common parallel programming paradigms. We also review a number of parallel programming models, technologies, and APIs. The studied parallel models have either widespread adoption on shared memory architectures, or similarity with GPRM. We defer the discussion about parallel architectures to the next chapter, which is focused on hardware platforms. In the final section, we present the related work.

2.1 Design of Parallel Programs

Parallel programming comes with its price. Race conditions, thread synchronisation, and load balancing are some of the parallel programming pitfalls. A proper separation of concerns can be helpful in addressing these problems. Foster [32] divides a parallel programming design methodology into four different phases:

2.1.1 Partitioning

In this phase, the designer decides about how to partition the computation, using domain and/or functional decomposition techniques. The problem is decomposed into computational tasks and their associated data. The number of processors (or cores) is ignored at this phase. Therefore, the focus is on how to break down a large problem into the smaller parts and find opportunities for parallelism and how to extract the greatest amount of parallelism from the problem.

Domain Decomposition

The domain decomposition approach is data-centric. Therefore, first the data is decomposed into preferably equal pieces (chunks). The next step is to partition the computation, typically based on its association with each piece of data. If some data is needed to be shared between the tasks, the communication phase would resolve it. An example is decomposition of a 3D grid using one-, two-, or three-dimensional decompositions. Obviously, at the first stage of the design, the 3D decomposition provides more flexibility and more opportunity for parallelism, as a task will be defined for each grid point, rather than for each line or each plane in the 2D and 1D decompositions.

Functional Decomposition

The functional decomposition is computation-centric, meaning that the main focus is on what computation to perform rather than what data to manipulate. Therefore, the application is divided into multiple parts, each of which performs a different function. Historically, this step has been considered as a complementary approach, to be used for better optimisation where looking at the data alone is not enough. An example could be the traversal of a search tree for finding an answer: a task is created for the root, and if the root is neither the answer nor the only node, then for each sub-tree a new task will be created. It is evident in this case that there is no obvious data structure to be decomposed via domain decomposition. Another common form of functional decomposition is pipeline decomposition. In this model, pipelined threads (processes) perform processing stages one by one.

In many cases, a hybrid methodology is used. We will show examples of both domain and functional decomposition in our benchmarks. We sometimes refer to the problem itself as having data and task parallelism respectively.

2.1.2 Communication

The next step is to determine the type of communication between the generated tasks and how to coordinate their execution. Most of the time, the tasks comprising a problem need to exchange data. There are two sub-phases involved. First, designing channels of communication between tasks for data transfer. Second, design of the messages containing the data to be sent over these channels. Since communication corresponds to the data flow between the tasks, specifying the communication requirements is more straightforward where functional decomposition is applied, compared to the situations where domain decomposition is used. Foster [32] classifies communication patterns based on four metrics.

Structured / Unstructured

In *structured* communication, the communication pattern forms a known regular structure, such as tree, while in *unstructured* communication, the tasks can communicate in any arbitrary form.

Static / Dynamic

In *static* communication, the communicating tasks remain the same and do not change over time, while in *dynamic* communication, the communicating tasks and possibly the pattern of communication can change dynamically at runtime.

Local / Global

In *local* communication, a task will communicate mostly with its neighbours, while in global communication, each task is in communication with many other tasks.

Synchronous / Asynchronous

The communication is coordinated between producers of data and its consumers in *synchronous* communication, while in *asynchronous* communication, producers of data do not know when the consumers need the data. So, the consumers should explicitly send a request to the producers.

2.1.3 Agglomeration

In this phase, the cost and performance considerations of the previous steps are taken into account. For example, in this phase one should decide about the granularity of the tasks in order to reduce the overhead. Therefore, this step is all about efficient execution of the tasks on the target platform. Apart from deciding about the number of tasks and hence their granularity, in this phase, one would decide about the replication of tasks and/or data.

Agglomeration does not necessarily mean that the number of tasks should be limited to the number of processing units. The aim of this phase is to provide a reasonable balance between communication cost reduction, preserving flexibility in regards to mapping and scalability, and software engineering cost reduction.

We use the concept of *cutoff* in this dissertation in order to agglomerate the tasks. We define the *cutoff* value for a program as the maximum number of ready-to-run tasks it can have at any point of time during its execution.

2.1.4 Mapping

Mapping to the cores (or processors) is the last step, and can be considered as both assigning the tasks to cores (where to run), and specifying the order of their execution (when to run). The cost of communication and utilisation of the processors (cores) should be considered in this phase. The mapping techniques can be static or dynamic at runtime, e.g. load balancing techniques. The mapping problem, in general, is NP-complete. For the early-stage multicores it was reasonable to rely on the operating system to decide about the mapping, but with the rise of manycore processors, it has become more challenging [33].

If finding an efficient agglomeration and mapping strategies for a problem is not straightforward, then load balancing techniques might be used to determine the best mapping based on the runtime situations.

If a functional decomposition is applied, at this stage a centralised or a decentralised *task scheduling* technique will be used.

We use the above methodology, called *PCAM*, as a powerful reference, as its outcome can cover SPMD [1] cases as well. However, not all the steps can be exactly identified for all programs. For example, for an SPMD program concerns of the mapping phase can be combined with those of the agglomeration one. Another point to note is that the *PCAM* design itself is a parallel process and many of these concerns can be considered simultaneously.

2.2 Parallel Programming Paradigms

A parallel programming paradigm governs the development life cycle of parallel programs, from design to coding, testing and tuning.

Basically, parallel programming models attempt to target a broad range of parallel problems, abstract the details of the underlying architectures, and at the same time provide high performance. For sequential programming, the von Neumann model can be used as an efficient bridge between software and hardware, as the sequential program can be compiled efficiently to the underlying hardware [34]. When it comes to parallel programming, however, two main criteria can be considered for the classification of the paradigms: *interaction* and *decomposition*. It is also important to notice that the following classification applies best to the CPU-based systems. The categories should be modified/extended if heterogeneous architectures (composed of CPUs, GPUS, FPGAs) are targeted.

[1] Single Program, Multiple Data. In SPMD, multiple processing cores execute independent parts of the same program. This is different from SIMD which needs a vector processor.

2.2.1 Interaction

Interaction determines the way the processing elements interact with each other. It is also known as the *information exchange*. Two common forms of interaction in a parallel system are shared memory and message passing. A distributed shared memory (DSM) paradigm has also gained a lot of interest and become popular. An implicit model can be considered as well.

Shared Memory

With the shared memory model, a common address space is accessible to the running tasks. Typically, any task might access any data at any time. Therefore, this approach comes with the cost of controlling the access to the global shared places, such that only one task at a time can access a shared object. In order to implement proper data access ordering and providing mutual exclusion, locks, semaphores, or monitors can be used.

Although without the concept of *data ownership* program development can be facilitated, managing locality and writing deterministic programs using this paradigm is difficult.

Pthreads and OpenMP APIs are mainly used for multithreaded programming on shared memory systems.

Message Passing

Each task in the message passing paradigm has its own local data. The interactions between the tasks are through sending and receiving data packets/messages. These interactions can be synchronous or asynchronous.

One of the most common practices in the message passing paradigm is to create a fixed number of identical tasks at the start of the program. This model of execution, known as SPMD, implies that the same program is executed by each task, but on different data.

MPI is predominantly used for message passing on distributed memory (share nothing) systems.

Partitioned Global Address Space

The shared memory paradigm typically exploits a shared memory architecture with a global address space. Message passing is a *share nothing* approach. It targets systems where each processing element has its own local memory, and typically no knowledge about the others' memories.

In the Partitioned Global Address Space (PGAS) paradigm multiple processes (or threads) share a part of their address space [35]. PGAS attempts to combine data locality (*partitioning*) and performance features of the message passing approach with ease of programming and data referencing simplicity in a *global address space* of the shared memory paradigm.

Unified Parallel C (UPC) [36] and Co-array Fortran (CAF) [37] are examples of SPMD PGAS programming models.

Implicit Parallelism

In this paradigm, the interaction between tasks (processes) is not visible to the programmer. This interaction is determined by either compiler or the runtime system. Mostly, Domain Specific Languages (DSL) implement this paradigm.

2.2.2 Decomposition

We have already discussed decomposition in the Section 2.1.1 of the design of the parallel programs. It concerns the way the executing processes (or threads) are formulated. A natural way to express message passing is called *Task Parallelism*, where the focus is on the processes (or threads) of execution. In contrast, *Data Parallelism* focuses on the data set. The data can be accessible to all tasks which operate on it, or be divided between local memories. In this dissertation, we use systems where the data is accessible to all.

2.3 Parallel Programming Models and Standards

The design of manycore processors is strongly driven by demands for greater performance at reasonable cost. To make effective use of the available parallelism in such systems, the parallel programming model is of great importance.

There are several parallel programming models, runtime libraries, and APIs that help developers to move from sequential to parallel programming. The most common ones are MPI [38] for distributed memory programming, and OpenMP [19] for shared memory programming [39]. Generally, most parallel models are either runtime libraries or language extension.

Our mission is therefore to propose a programming model that can be integrated into existing codes in imperative languages, while offering native parallelism, similar to functional languages. Before going into the details of our approach, the Glasgow Parallel Reduction Machine (GPRM), we would like to briefly review some of the important parallel programming models:

- Task-based: OpenMP 3+, Cilk Plus and Intel Threading Building Blocks (TBB) from Intel, Task Parallel Library (TPL) from Microsoft
- Message passing: Message Passing Interface (MPI)
- PGAS: Unified Parallel C (UPC) and Chapel
- Task-based, with support for task dependencies: StarSs - SMPSs - OmpSs
- Dataflow: OpenStream and Swan
- Functional: Glasgow Parallel Haskell (GpH) and Clojure
- Array-based: Single Assignment C (SAC)
- Targeting heterogeneous systems: Open Computing Language (OpenCL)
- Targeting distributed systems: MapReduce

2.3.1 OpenMP

OpenMP is the de-facto standard for shared memory programming, and is based on a set of compiler directives or pragmas, combined with a programming API to specify parallel regions, data scope, synchronisation, and so on. It also supports runtime configuration through the use of runtime environment variables, e.g. `OMP_NUM_THREADS` to specify the number of threads at runtime. OpenMP is a portable parallel programming approach and is supported on C, C++, and Fortran. It has been historically used for loop-level and regular parallelism through its compiler directives. Since the release of OpenMP 3.0, OpenMP also supports task parallelism [3]. It is now widely used in both task and data parallel scenarios.

A new trend in task-based parallel programming is to add more information to *task*s, such as information about the data regions they access. Since the runtime system can build the dependency graph at runtime based on such information, this model –called the dataflow programming model– relaxes the need for explicit task synchronisation. OpenMP 4.0 has adopted this model [40] in 2013.

Since OpenMP is a language enhancement, every new construct requires compiler support. Therefore, its functionality is not as extensive as library-based models, although at the same time it enjoys broad support from its community.

2.3.2 Cilk Plus

Cilk Plus which is based on the Cilk++ [41] is an extension to C/C++ to provide both task and data parallelism. Cilk++ itself evolved from Cilk, but distanced itself from Cilk in several ways: support for loop, support for C++ language, and offering *Cilk hyperobjects*, constructs designed to solve data race problems. Basically, hyperobjects implement a mechanism to provide coordinated local views of non-local variables in a multithreaded program [42]. The most common type of the hyperobject is a reduce, which has similarities with the `reduction` clause in OpenMP, `parallel_reduce` template function in TBB, and the `combinable` object in Microsoft's PPL.

Cilk Plus has become popular because of its simplicity. It has `_Cilk_spawn` and `_Cilk_sync` keywords to spawn and synchronise the tasks. `_Cilk_for` loop is a parallel replacement for sequential loops in C/C++. Intel Cilk Plus starts a pool of threads in the beginning of the program which is analogous to the GPRM thread pool.

2.3.3 Intel Threading Building Blocks (TBB)

Intel Threading Building Blocks (TBB) is an object-oriented C++ template library offered by Intel for parallel programming [43] [44]. Intel TBB contains several templates for parallel algorithms, such as `parallel_for` and `parallel_reduce`. It also contains useful parallel data structures, such as `concurrent_vector` and `concurrent_queue`. Other important features of the Intel TBB are its scalable memory allocator as well as its primitives for synchronisation and atomic operations.

TBB abstracts the low-level threading details, which is similar to the GPRM design consideration. However, the tasking comes along with an overhead. Conversion of the legacy code to TBB requires restructuring certain parts of the program to fit the TBB templates. Moreover, there is a significant overhead associated with the sequential execution of a TBB program, i.e. with a single thread [39].

A *task* is the central unit of execution in TBB, which is scheduled by the library's runtime engine. One of the advantages of TBB over OpenMP and Cilk Plus is that it does not require specific compiler support. TBB's high level of abstraction and the fact that it is based on runtime libraries make it very similar to GPRM. However, we will see in Chapter 5 how the models of execution for these two are different.

2.3.4 Task Parallel Library (TPL)

Task Parallel Library (TPL) [20], a task-based parallel library for the .NET framework, is the Microsoft approach for parallel programming. TPL is mostly based on generics and delegate

expressions. A delegate expression in C# is the equivalent of a Lambda expression. Along with a parallel `for` construct in the form of a `Parallel.For` method, TPL defines `task` and `future` constructs, where `future` is a task that returns a result.

As in the standard work stealing technique [45], threads store tasks in their local task queue. Whenever its local queue becomes empty, the thread steals work from others. However, the performance of the work stealing algorithms strongly depends on the implementation of their task queues. Microsoft uses *duplicating queues* for this purpose, for which the *Take* operation [2] can either remove an element, or duplicate it in the queue.

2.3.5 Message Passing Interface (MPI)

Message Passing Interface (MPI) [38] is an API specification designed for high performance computing. Since MPI provides a distributed memory model for parallel programming, its main targets have been clusters and multiprocessor machines.

`MPI_Send` and `MPI_Recv` are the two basic routines for sending and receiving messages using MPI.

The MPI implementations are evolving. For example, in MPI-1 there was support for interconnection topology and collective message communication. Messages could contain either primitive or derived data types in packed or unpacked data content. Dynamic process creation, one-sided communication, remote memory access, and parallel I/O are some of the advanced features of MPI-2.

Since there are lots of MPI implementations with emphases on different aspects of high performance computing, Open MPI [46], an MPI-2 implementation, evolved to combine these technologies and resources with the main focus on the components concepts.

MPI is portable, and in general, an MPI program can run on both shared memory and distributed memory systems. However, for performance reasons and due to the distributed nature of the model, there might exist multiple copies of the global data in a shared memory machine, resulting in an increased memory requirement. Message buffers are also considered as the associated overhead of MPI within a shared memory machine [47]. Furthermore, MPI low level programming makes it error-prone and sometimes difficult to debug [48].

2.3.6 Unified Parallel C (UPC)

As stated earlier, Partitioned Global Address Space (PGAS) languages aim to combine the best features of the shared memory programming with those of the message passing model.

[2] *Push* and *Pop* operations behave as usual

In Unified Parallel C (UPC) [36], users can specify the distribution of data in order to utilise data locality. UPC also enhances programmability by providing a global name space to access remote data.

While there is an increasing interest in using PGAS languages such as UPC for high performance computing (HPC), studies have shown that obtaining an equivalent performance to that of MPI programs needs a careful utilisation of the language features [49].

2.3.7 Cascade High Productivity Language (Chapel)

Chapel is an object-oriented PGAS language developed by Cray, with the primary goal of increasing supercomputer productivity. It has support for data, task, and nested parallelism. Chapel aims to fill the gap between HPC programmers who are mostly focused on C/C++ and Fortran and the mainstream programmers who have less experience in parallel programming and using HPC techniques [50].

Using anonymous threads, Chapel provides a high level of abstraction for the programmers. Generality, programmability, separation of algorithm and implementation, and data abstractions are considered to be the most important features of Chapel. Locality control is also one of the Chapel's features.

2.3.8 StarSs, SMPSs, and OmpSs

The Star Superscalar (StarSs) is another task-based parallel programming model, which has multiple variations for different domains, including CellSs for the Cell/B.E. processors, GRIDSs for the Grids, SMPSs for multicore processors, and GPUSs for the GPUs [51] [52] [53].

StarSs uses pragmas, similar to OpenMP; it also has a source to source translator and provides a runtime system to schedule the tasks based on their dependencies. Generally, the runtime systems have two main modules: i) a master thread, which executes the annotated code, generates tasks, and schedules and ii) a number of worker threads for task execution. Different variations have different implementations, though, for example, the runtime system in GPUSs has a master thread, a helper thread and a number of workers. The master thread creates tasks and inserts them into a Task Dependency Graph, the helper thread finds the best device for task execution, and the worker threads (one per GPU, but running on the CPU) perform the data transfers and invoke the tasks on GPUs.

The SMP superscalar (SMPSs) project from the Barcelona Supercomputing Center (BSC) [52] similarly allows programmers to write sequential dependency-aware programs and the promise is that the framework is able to exploit the parallelism by means of an automatic

parallelisation at runtime. The SMPSs program code must be annotated using special pre-processor directives. The SMPSs runtime builds a data dependency graph where each node represents an instance of an annotated function and edges between nodes denote data dependencies.

One of the similarities between the SMPSs and the GPRM runtime system is that they both exploit data locality, based on the information they get from the task dependencies. The difference is that in SMPSs, the user should annotate all the functions, because the goal is to parallelise a sequential program, hence all the dependencies should be specified explicitly (same approach as in OpenMP 4.0). The GPRM design, however, is based on the parallel execution of all tasks, unless they are surrounded by a GPRM `seq`. Therefore, expressing the dependencies is as simple as defining `seq` blocks around the user-defined tasks in the GPC code (similar to a functional language, but with a C++ veneer).

OmpSs [54] is based on both StarSs and OpenMP, witch also supports the use of OpenCL or CUDA kernels. Some of the new features in OpenMP 4.0 have their origin in OmpSs. It has some differences with OpenMP though, such as having a different execution model, extra constructs to define data dependencies [40] and specify heterogeneous architectures [55], and so on.

Referring again to GPRM, OmpSs similarly uses a variation of the thread pool model, hence the threads exist from the beginning until the end of the program execution (which has recently become a norm in other approaches as well). As a result, using a `parallel` construct is deprecated in this model. It also allows for nesting of constructs. The implementation of OmpSs is based on the Mercurium compiler and the Nanos++ runtime library [56] [48].

2.3.9 OpenStream

The dataflow concept attempts to overcome the limitations of the control-flow model by reducing the synchronisation overhead and exploring the maximum parallelism; activities are initiated by the presence of their data [57].

OpenStream extends OpenMP to express dynamic dependant tasks. The dataflow execution model in OpenStream has shown better performance compared to the “task and barrier” models [58]. It is important to note that a dataflow approach such as OpenStream requires explicit task dependencies in order to maintain the correctness of parallel execution, while in a model such as StarSs, data dependencies are inferred at runtime [57].

The OpenStream compiler finds producer-consumer cases when creating tasks. Without any polling, the producers will know when the consumers become ready. Once one of the consumer tasks is ready, its producer adds it to its work stealing core-local ready queue.

2.3.10 Swan

Task dataflow languages facilitate parallel programming using a scheduler that is aware of task dependencies. In such languages, arguments of a task can be labelled with a memory access mode, such as input, output, or input/output. Therefore, the scheduler can dynamically track dependencies and change the order of task execution [59].

Swan is a dataflow driven language that extends a Cilk-like work stealing scheduler by allowing task dependencies between tasks spawned from the same parent. In order to track dependencies, a type of hyperobject [42] called "versioned object" and a ticket-based technique are applied. Versioned objects encapsulate the metadata that is essential for dependency resolution and object renaming (privatisation), and the ticketing system is used to sequence the tasks that operate on the same objects [59].

Swan's scheduler unifies dependency-aware scheduling and work-first scheduling. The unified scheduler could retain the efficiency of the Cilk scheduler when tasks are specified without dependencies. Swan can also achieve ease-of-programming without loss of performance for parallel patterns such as pipeline, as presented in [60].

2.3.11 Glasgow Parallel Haskell

Functional programming languages play an important role in the parallel computing world. Haskell is a pure functional programming language [61]. Glasgow Parallel Haskell (GpH) [62] is non-strict parallel dialect of Haskell with the semi-explicit thread-based parallelism. Glasgow Parallel Haskell provides a sophisticated runtime system for thread management, and has support for both shared memory (via GHC-SMP) and distributed memory architectures (via GUM [63]).

The first design of GpH dates back to 1990, when only two constructs were added to the sequential Haskell: `par` and `seq`. The same constructs are defined in GPRM, but to be used in a restricted subset of C++ (GPC code). The current version has support for the evaluation order, evaluation strategies, and skeletons. The term "non-strict", instead of lazy, means that provided that the sub-expressions are needed by the result of the program, they can be evaluated in parallel. This is another similarity between the the fundamentals of GpH and GPRM.

2.3.12 Clojure

Functional parallel programming is not as simple as sequential programming. In addition to what to compute, the programmer should specify how to coordinate the computation.

Clojure [64] [65] also called a modern Lisp is a general-purpose functional programming language that can run on the Java Virtual Machine (JVM), Common Language Runtime (CLR), and JavaScript engines.

Clojure treats functions as first-class objects, meaning that they can be passed to other functions as arguments. It also provides a set of immutable, persistent data structures. In order to support mutable states, Clojure uses a software transactional memory system.

Clojure's syntax is based on S-expressions, i.e. lists where the first element represents the operation and the other elements the operands. This is similar to the functional intermediate language (GPIR) that the GPC code is compiled to. GPIR itself is further compiled into lists of bytecodes, which the GPRM virtual machine executes.

2.3.13 Single Assignment C (SAC)

Single Assignment C (SAC) [66] aimed to incorporate arrays with the O(1) access time into the functional languages. In terms of programming style SAC is similar to the array programming languages, such as APL. The design of SAC is focused on the numerical computations on multi-dimensional arrays. The key features of SAC beside on focus on arrays, are the abstract view of arrays and its state-free functional semantics.

Unlike other functional languages, data aggregation in SAC is based on stateless arrays, rather that trees or lists. In SAC, functions consume arrays as arguments and produce arrays as results. It also supports a generic programming style allowing the functions to abstract away the size and number of array dimensions [67]. SAC can support a variety of architectures from the same source code.

2.3.14 OpenCL

OpenCL [68] is an open standard for heterogeneous architectures. One of the main OpenCL's objectives is to increase portability across GPUs, multicore processors, and OS software via its abstract memory and execution model; however its performance is not always portable across different platforms. It has been suggested to consider the architectural specifics in the algorithm design, in order to address its performance portability issue. The use of autotuning heuristics could also improve the performance [69].

Although OpenCL is mostly compared against Nvidia's CUDA [70], we do not aim to cover discussions about GPUs in this work. The reason why OpenCL is listed here is firstly because it allows for sharing of workload between CPU and GPU with the same program, and secondly because the Xeon Phi, the studied system has support for the OpenCL programs.

2.3.15 MapReduce

MapReduce [71], which is developed by Google, is very popular for processing large data sets and specially its use on large clusters. The processing consists of defining map and reduce functions. The map function is responsible for processing a large volume of data sets and generating intermediate key-value pairs in parallel. The role of the reduce function it merge all the intermediate values with same intermediate key.

MapReduce can be used to target lots of real world problems, such as the reverse Web-link graphs or count of URL access in the web context. The reason why we have listed MapReduce here is that its specification does not assume a shared or distributed memory model. Although, most of implementations have been on large clusters, there has been still interest in optimising it for multicores [72].

A significant feature is that the partitioning, communication and message passing, and scheduling across different nodes are all handled by the runtime system, and the user has to only worry about expressing the MapReduce semantics. This is again similar to our design decisions regarding GPRM.

2.4 Task Scheduling and Load Balancing

Task Scheduling in a parallel system is the arrangement of tasks of a program in time and space on the available execution resources [73]. Therefore, not only the mapping of all jobs to the processing cores, but also the order of execution is crucial to get good performance. An optimal solution to the task scheduling problem cannot be found in polynomial time, which means it is an NP-hard problem. This motivates researchers to develop heuristics to find near optimal solutions [73].

Generally speaking, If one considers static and dynamic scheduling [74], in static scheduling, the arrangement of jobs is done before the execution of the program begins, i.e. at compile time, while in dynamic scheduling, redistribution of jobs among cores/processors can happen at runtime. The act of transferring jobs from heavily loaded cores/processors to the lightly loaded ones is often referred to as *Load Balancing*.

A dynamic load balancing technique typically consists of the following parts: I) Information policy, II) Transfer policy and III) Placement policy. Information policy is about the amount of load information that is available to the decision makers. Transfer policy specifies the condition for a job transfer. Finally, the Placement policy specifies a core/processor to which the job should be transferred.

Dynamic load balancing can be divided into two major subcategories [75] [76]: I) Work Sharing and II) Work Stealing.

2.4.1 Work Stealing

In a work sharing scheduler, whenever new jobs are created, the scheduler attempts to migrate some of them to other processing elements, in hope of transferring the work to underutilised cores. Conversely, in a work stealing scheme, underutilised processing cores try to "steal" work from others. Migrations occur more frequently in a work sharing scheme, because the jobs are always migrated by a work sharing scheduler, while in a work stealing scheme, migration occurs only when the processing cores have no more work to do.

The idea of work stealing dates back at least as far as the 1980s [77] [78], when researchers indicated the heuristic advantages of work stealing in terms of communication and space. Since then, several variations of such a scheme have been considered.

Rudolph et al. [79] have addressed the problem of frequently migrating tasks in a shared memory parallel architecture, and proposed a distributed load balancing mechanism for such systems. In their scheme, a processor examines the task queue of another random processor, and then they exchange tasks until the sizes of the two task queues become equal. They also considered a multiprogramming situations, where they claimed that if the total number of tasks is much larger than the number of processors, since the tasks of the same program are initially places on the same queue, the load balancing mechanism is able to group them together.

Blumofe and Lisiecki [80] presented an adaptive runtime system for parallel execution of functional Cilk programs on a Network Of Workstations (NOWs). Each Cilk program in such an adaptive environment was able to utilise a changing set of otherwise-idle workstations, dynamically.

2.4.2 Scheduling on Modern Processors

In more recent years, studies have focused on the concept of work/task scheduling on multicore and manycore architectures.

StarPU [81] is a well-known runtime system that targets heterogeneous multicore architectures. It provides a unified view of all computational resources and is capable of transferring input data to accelerators transparently and before tasks start. Moreover, it provides a unified task scheduling scheme over heterogeneous architectures using auto-tuned performance models. It also provides power-based scheduling, static scheduling, or even allows for defining a new scheduling policy.

Sasaki et al. [82] developed a sophisticated scheduling scheme to co-schedule multiprogram workloads, which predicts the applications' scalability dynamically, and allocates the optimal number of cores to applications in order to maximise the system utilisation. They

have proposed a technique called *Core Donation* that maximises the CPU utilisation while keeping the scalability of the programs into account.

In [83], two families of the schedulers are discussed: Breadth-First schedulers and Work-First schedulers. The main focus is on how threads execute the tasks. In 2008, [84] suggested that binding the threads to physical processors, also referred to as *thread affinity* be a part of the OpenMP standard. It has been shown how a simple Round-Robin mapping scheme can improve the performance. Newer versions of OpenMP provide this feature to the users.

The OpenMP *task* directives can be used to define units of independent work as *task*s. However, the scheduling decisions are deferred to the runtime system. Olivier et al. [85] have proposed a hierarchical scheduling strategy for efficient task scheduling on modern multi-socket multicore shared memory systems. They have suggested that complex memory hierarchy on the modern systems, including NUMA characteristics and shared caches need to be considered carefully.

Nanos v4 [86] is an OpenMP runtime library that provides some mechanisms to allow users to choose between different task scheduling policies.

Locality awareness is another important aspect of task scheduling on manycore processors. Yoo in [87] discussed that the complexity of cache hierarchies results in high latency and energy consumption as well as non-uniformity in memory accesses. Multi-level private and/or shared caches in hardware can be utilised with the help of locality-aware scheduling mechanisms. They suggest that taking the underlying cache architecture into consideration is crucial for designing an efficient runtime scheduler. Using high-level information they have shown that the programs can run faster at a lower energy consumption. They have also related stealing mechanisms and load balancing to locality-awareness and have shown how to make a stealing mechanism locality-aware. This can be achieved by preserving the original scheduling scheme and migrating tasks for load balancing.

In this section, we started the discussion about scheduling and reviewed some of the related work in this area. Additionally, a detailed discussion on task scheduling and load balancing in our GPRM framework can be found in Section 5.5.

2.5 Related Work

2.5.1 Architectural Aspects

We review some of the research works in the field of parallel computing and high performance computing (HPC) that utilise the Tilera TILEPro64 or the Intel Xeon Phi. We also include a few interesting works on the architectural aspects of shared memory parallel platforms.

Data locality in Non-Uniform Cache Architecture (NUCA) designs is discussed in [88]. The authors propose an on-line prediction strategy which decides whether to perform a remote access (as in traditional NUCA designs) or to migrate a thread at the instruction level.

Authors of [89] have presented the results of implementing the UPC runtime system on the Tile64 processor, an older version of the TILEPro64. For that purpose, they have used the Berkeley UPC to C translator [90] and the Tilera C compiler. The runtime system is built on top of the Global Address Space Networking (GASNet) communication infrastructure [91], which itself is implemented using either a pthreads based channel or an MPI based one. Amongst the highlighted opportunities for optimisations on manycore systems, such as the efficient use of the iMesh interconnects, we would like to focus on placement: the optimisation of placement strategies for efficient utilisation of the resources. We will show in Chapter 5 that our strategy for GPRM is to resolve the placement issue at the task level rather than at the thread level.

In [92], the NUCA characterisation of the TILEPro64 is explored in details. Based on this characterisation, a home cache aware task scheduling is developed to distribute task data on home caches and minimise home cache access penalties.

Similar work to ours on a sorting algorithm but with different methods and purposes is performed on the TILEPro64 [93]. They have targeted throughput and power efficiency of the radix sort algorithm employing fine-grained control and various optimisation techniques offered by the Tilera Multicore Components (TMC) API.

Podobas et al. [94] have performed a quantitative evaluation of some well-known task-based parallel models on two different platforms, including the TILEPro64 (700MHz version). Although their focus was more on the scalability of the models when the numbers of threads changes, they have highlighted the importance of the cutoff values in multiple scenarios. Their results emphasise the need to concentrate more on load balancing techniques as well as the overhead of runtime systems.

Mirsoleimani et al. [95] have considered the scalability of the Monte Carlo Tree Search (MCTS) algorithm, as an unbalanced and irregular workload on the Xeon Phi. They have found that a thread pool with a work-sharing FIFO provides better performance compared to work stealing models such as TBB and Cilk Plus. The key element of their approach is controlling the grain size (cutoff) through a technique called Grain Size Controlled Parallel MCTS (GSCPM).

Saule et al. [96] have used some micro-benchmarks to measure the Xeon Phi's peak performance. They have evaluated the performance of a number of sparse matrix multiplication kernels on this architecture, and concluded that the memory latency (and not the bandwidth) is the bottleneck for such applications. However, they have presented that in most cases, the achieved performance for sparse kernels on the Xeon Phi is better than that on the studied

NVIDIA GPUs and Intel Xeon processors.

2.5.2 Programming and Performance Aspects

It has been investigated that task distribution and task management overheads are the main bottlenecks in modern runtime systems [97]. Authors in [98] and [99] focused on the efficiency of the runtime systems for fine-grained task scheduling as well as task creation.

The trade-off between task granularity and the number of tasks in OpenMP is covered in [100]. The authors suggest that granularity of tasks should be increased as the number of consumer threads increases. This is to ensure that all threads are kept busy doing some useful work. They have also explored that the number of tasks has an effect on the load balance, which means programmers have to trade-off between the number of tasks and granularity in order to get a fair load balance, hence a good performance.

It has been established in [100–103] that OpenMP performs poorly for large numbers of fine-grained tasks. This indicates that the programmer is responsible to figure out how a problem with specific input parameters would fit a particular platform. As a common solution, programmers use a cutoff value when creating OpenMP tasks to avoid composing fine-grained tasks. firstly, for OpenMP programs both a proper cutoff and the right number of threads are key to good performance. Secondly, finding a proper cutoff value is not straightforward, and sometimes needs a comprehensive analysis of the program. It often depends on the application structure and the input data set [83]. Leaving the decision to the runtime system has been proposed as an alternative. The idea is to aggregate tasks by not creating some of the user specified tasks and instead executing them serially. Adaptive Task Cutoff (ATC) [101] implemented in the Nanos [86] runtime system –a research OpenMP runtime system– is a scheme to modify the cutoff dynamically based on profiling data collected early in the program's execution. This, however, cannot be done without any overhead at all, plus the scheme needs to be extended to find all interesting situations for cutoff.

Muddukrishna et al. [104] provided a comprehensive study of task-based OpenMP programs included in the Barcelona OpenMP Tasks Suite (BOTS) benchmark suite. Their concentration has been on performance profiling of tasks in OpenMP programs, which was missing in the well-known threads-based profilers such as the Intel VTune Amplifier and Vampir [105]. The most relevant part of their work is the use of techniques that help choosing an appropriate cutoff to increase performance. Similar to our previous publication [28], this work significantly contributes to directing programmers' attention to the importance of tasks.

XKaapi [106] is a data-flow programming model, which is compared with Intel OpenMP and Cilk Plus on the Xeon Phi as well as a conventional Intel Xeon platform [98]. The authors compared the overhead of the runtime systems for fine-grained tasks, and similar

to our previous work [29] and others mentioned above concluded that the OpenMP runtime system performs poorly if a very large number of small tasks are created. Then main focus of their work, similar to that of Wool-A –a work stealing library– [99] was to minimise the overhead of task creation and scheduling.

Jin et al. [47] have proposed a hybrid approach for programming high performance computing systems by combining OpenMP and MPI. The motivation behind this work has to be found in the hybrid nature of the petascale and exascale systems themselves, where they consist of distributed memory clusters [107] of shared memory systems. They have studied the multi-zone NAS Parallel Benchmarks (NPB-MZ) [108] [109], and two full applications, OVERFLOW [110], which is a general-purpose Navier-Stokes Computational Fluid Dynamics (CFD) solver, and AKIE [111], which is turbine machinery application used to study three-dimensional flow instability around rotor blades. They have also introduced a method to provide hints (data locality pragmas) to the OpenMP compiler and runtime system in order to exploit data locality on complex systems with hierarchical memory subsystems.

Researchers in [112] [113] have shown success in writing SPMD programs using OpenMP for solving large scale real world problems. The idea is to use the MPI programming style with the use of thread ids and explicit data management. However, such techniques are against the benefits of using OpenMP in the first place, such as high level abstraction and ease of programming. In [112], the authors have demonstrated a high performance CFD flow solver by adapting distributed programming techniques, such as explicit domain decomposition and memory management to a shared memory environment.

Authors of [113] have compared the performance of MPI with three OpenMP programming styles for a subset of the NAS benchmarks on two shared memory machines. In 2003, tasking was not available in OpenMP, therefore, their three approaches were naive, improved, and optimised versions of the loop-level parallelism. They have also presented a path to translate from MPI to an SMPD-style programming with OpenMP. They argue that it is not always obvious that the performance of the loop-level OpenMP programs are better than the MPI implementations on shared memory platforms, especially for the naive OpenMP implementations which lack parallel loop-nest optimisations [114], such as blocking.

Krpic et al. [115] compared the performance per watt ratio –known as the Green HPC research [116]– of OpenMP versus MPI for a common HPC benchmark, matrix multiplication, on shared memory multicore platforms. They discuss that recently, the power consumption has been a major issue in high performance computing, while many programming techniques are oblivious to it. They concluded the performance and more importantly the performance per watt of the MPI approach is better than OpenMP on three different platforms. However, the maximum number of processes/threads used for the comparisons has never been more than 8.

Saule and Catalyurek [117] have compared OpenMP, Cilk Plus, and TBB on the Intel Xeon Phi. They have focused on the scalability of the approaches for single-program graph algorithms. We have added GPRM to the comparison and have also targeted multiprogramming situations.

Callisto [118] is a user-mode shared library for co-scheduling multiple parallel runtime systems. Although there is no need to modify the high-level applications or the OS, it has to be linked with the Callisto-enabled versions of the runtime systems. The current version does not support OpenMP tasks. Furthermore, the authors have used pairs of benchmarks on a 2-socket machine. It needs more investigation to find out whether Callisto can be still effective if more benchmarks are run together, or if one moves from a socket-based machine to a modern architecture such as MIC.

Emani et al. [119] used predictive modelling techniques for OpenMP programs to determine an optimal mapping of a program in the presence of external workload. Dynamic runtime information is combined with the compile-time knowledge of the program to decide about the best adaptive mapping of programs to execution resources. Their purpose is to maximise the performance of a target program with minimum impact of the performance of the workloads.

Varisteas et al. [120] have proposed an adaptive space-sharing scheduler for the Barrelfish operating system to overcome the resource contention between multiple applications running simultaneously in a multiprogrammed system.

Bhadauria and McKee [121] have developed real-time scheduling techniques to improve performance and energy efficiency in multiprogramming environments. Their schedulers are applied to co-schedule PARSEC programs that complement each other regarding shared resource requirements. These resource-aware co-schedulers, known as the HOLISYN (which implement both space- and time-sharing), were able to reduce contention and perform better than other co-schedulers on an 8-core CMP.

In our previous work [23], a thread mapping method based on the system's load information is developed for OpenMP programs. Performance of multiprogram workloads in Linux can be improved by sharing the load information and using it for thread placement. However, for this method to be effective, the optimal number of threads for each single program has to be known to the programmer. Most of time, though, the programs are run with the maximum number of threads, with the expectation of achieving the desired performance, and that is the case we target in this dissertation.

2.6 Summary

In this chapter, we focused on a design methodology for parallel programming as the basis of our work. Foster [32] divides a parallel programming design methodology into four different phases (PCAM): *Partitioning*, *Communication*, *Agglomeration*, and *Mapping*, which can be considered concurrently. Partitioning could be data-centric, or computation-centric where *task*s become important. The second phase is to determine the type of communication between the generated tasks and how to coordinate their execution. Agglomeration is the main focus of this work. In this phase, cost and performance considerations of the previous steps are taken into account. For example, in this phase one should decide about the granularity of the tasks in order to reduce the overhead. Therefore, this step is all about efficient execution of the tasks on the target platform. Mapping to the cores (or processors) is the last step, and can be considered as both assigning the tasks to cores (where to run) and specifying the order of their execution (when to run). The mapping techniques can be static or dynamic at runtime, e.g. load balancing techniques.

We then reviewed the parallel programming paradigms and some of the existing parallel programming models and techniques. Different approaches focus on different issues, such as adding support for heterogeneous systems, simplifying writing of parallel programs, or facilitating parallelisation of complex patterns. It is important to understand the latest trends and techniques, and use them in the design of new models.

Starting from task scheduling techniques, we reviewed some of the related work. We discussed the importance of task-based parallel programming and considered various studies on performance optimisation, both of which will be the focus of the next chapters.

Chapter 3

Hardware Platforms

Parallel architectures extend the concept of conventional computing architecture with a communication architecture. In other words, a parallel computer is composed of processing elements that cooperate and communicate in order to solve a problem fast [122].

Contemporary applications have increased the trend of utilising a large number of processing cores in order to meet their performance goals. Most of these applications need single-chip implementations to satisfy their size and power consumption requirements. Multicore and manycore processors have emerged as promising architectures to benefit from increasing transistor numbers.

The main purpose of this chapter is to give a solid overview of parallel architectures used in this study. Our main targets are single-chip systems. With the growing IC technology, the delay of wires becomes significant in comparison with the gate delays, and thus the cost of communication is much more expensive than the cost of computation. Nonetheless, compared to off-chip, on-chip communication is considerably cheaper [12]. The cost consideration as well as size and power consumption requirements are the driving forces towards single-chip implementations.

In this chapter, we first review a number of fundamental topics regarding the parallel architectures. Afterwards, we provide a brief introduction to some parallel machines without covering them in exhaustive detail, and later, we introduce the Tilera TILEPro64 and the Intel Xeon Phi.

3.1 Flynn's Taxonomy

A coarse way to classify parallel systems is by looking at their control flow and data management. Flynn's taxonomy [123] classifies parallel computers based on the number of their

parallel instruction(control) and data streams [1] into four categories:

1. **Single-Instruction stream, Single-Data stream (SISD):** In this architecture there is one processing element (PE) with access to a single program and data storage. It means in each step, an instruction and its corresponding data are loaded and the instruction is executed. The result is stored back in the data storage. It is the conventional sequential Von Neumann model, known as the scalar processor.

2. **Multiple-Instruction stream, Single-Data stream (MISD):** This model has multiple PEs, each of which with its own private program memory, but there is only one access to a single global data memory. In each step, each PE loads an instruction from its program memory and the same data from the global data memory. The instructions will be executed in parallel using the same data as operand.

3. **Single-Instruction stream, Multiple-Data stream (SIMD):** There is a control processor which fetches and dispatches the instructions. In each step, each PE gets the same instruction form the control processor and loads separate data from (shared or distributed) data memory through a private access. This model includes most array processors and is perfect for applications that have significant amount of data parallelism.

4. **Multiple-Instruction stream, Multiple-Data stream (MIMD):** Each PE has a separate instruction and data access to a (shared or distributed) program and data memory. In every step, each PE performs separate instruction on separate data. Unlike SIMD that PEs work synchronously with each other, here they operate asynchronously. Multicore processors are examples of MIMD computers. Although, they might have SIMD components as well, e.g. the Intel Xeon Phi.

   (a) **Single Program, Multiple Data (SPMD):** With the advent of multiprocessing processors, a multiprogramming context has been added to Flynn's taxonomy. From the programming perspective, MIMD can be further divided into two subcategories: SPMD and MPMD[2]. In SPMD [125], multiple processors execute the same program on different data (at independent points) at the same time. While SIMD requires vector processing units, SPMD tasks can run on general-purpose processors.

   (b) **Multiple Program, Multiple Data (MPMD):** In this variant of the original Flynn's taxonomy, multiple processors simultaneously run multiple independent programs. SPMD is suited for the problems that lend themselves to task (functional) decomposition, rather than domain decomposition [124].

[1] Flynn defined stream as a sequence of items (instruction or data) as executed or operated on by a processor
[2] SPMD and MPMD are sometimes classified as parallel programming models [124]

There is another important class of parallel machines, which is often referred to as SIMT (Single Instruction, Multiple Threads) by GPU vendors [126]. In contrast to SIMD, SIMT applies one instruction to multiple threads, not just multiple data lanes [127]. Although, GPUs were initially designed and used to handle graphics, they are currently a major part of scientific computations. As the number of CPU cores are growing, GPUs and many-core CPUs are converging more. Intel Xeon Phi for example, has powerful Vector Processing Units and enables high peak performance for floating point operations. On the other side, Nvidia's Kepler GK110 introduced dynamic parallelism, where GPUs can generate new work for themselves [128].

The variety of parallel programming models that manycore processors support is one of their advantages over GPUs. Our focus remains on the manycore processors for the rest of this dissertation.

3.2 Memory Organisation

In terms of memory organisation, MIMD, which is the preferred model for general-purpose computing, can be considered from two aspects: the physical memory and the programmer's view of the memory.

From the physical perspective, computers with shared memory are often called multiprocessors, and computers with distributed memory are called multicomputers. A hybrid model can be distinguished as well, which is a virtually shared memory on top of a physically distributed memory.

From the programmer's view, memory organisation can be classified into shared address space and distributed address space. It is possible to provide the programmer with a shared address space, even when the physical memory is distributed.

3.2.1 Distributed Memory

Distributed memory machines consist of a number of processing elements (nodes) and an interconnection network which is capable of transferring data between these nodes. Each node has its own processor, local memory, and sometimes I/O elements. Since data in local memory is private, if a node needs data from the local memory of other nodes, *message passing* has to be performed through the interconnection network.

3.2.2 Shared Memory

Communication in shared memory machines is performed by writing to and reading from shared variables. The global memory usually consists of physically separate modules providing a global address space accessible for all processors. An important issue is to avoid concurrent access to the shared variable, as it would result in race conditions. Ensuring that every processor has fast access to the shared memory is not always easy, and as the number of processors grows in a shared memory machine, it becomes harder. Symmetric Multiprocessors (SMPs) are special form of shared memory machines. In SMPs, access time from any processor to any memory location is uniform [129]. SMPs usually have small number of processors connected via standard bus which also provides access to the global memory. SMP processors mostly come with no private memory, instead, each processor has a private cache subsystem. Since the central bus provides a constant bandwidth shared by all processors, this architecture is not really scalable. As the number of processing elements grows, more access collisions would happen and cause longer effective memory access time. It can be reduced by the use of caches and proper cache coherence protocols. Cache coherent means if one processor updates a location in global memory, all the other processors know about the update. In SMPs, this is accomplished at hardware level. Parallel programs in SMPs are usually associated with threads, which can share data with others via the common address space. Because of the uniform memory latency for all processors, bus-based SMP machines are a class of Uniform Memory Access (UMA) machines.

In some architecture, the access time in not uniform, i.e. the access time to data inside the local memory of the processor is faster than the access time to data in a remote memory. These machines are called Non-Uniform Memory Access (NUMA).

3.2.3 Memory Access Time Reduction

Important improvements in performance have come from the processor cycle time reduction. Also, the capacity of DRAM chips (main memory) has been increased. But memory access time has not been improved like processor cycle time. This growing gap, highlights the importance of a memory organisation with suitable access time in order to get better performance.

Local Caches

Caches are fast and small memory between processors and main memory, used to eliminate expensive main memory access for frequently accessed data. Since it is costly to have large fast memory, several levels of caches are implemented, starting from fast and small level 1

(L1) caches over multiple levels (L2, L3) to slow but large main memory. In SMP computers, shared data may be replicated in different caches in order to reduce corresponding access latencies, but any read access should return the last updated value of the data. Thus, a coherent view becomes significantly important, and a suitable cache coherence protocol must be used to resolve the *cache coherence problem* [129] [130] [3].

3.3 Design Variants of Multicore Machines

From a high-level perspective, three types of architectures as well as some hybrid ones can be distinguished [11]. These are hierarchical design, pipelined design, and network-based design. In a hierarchical design, some cores share the same caches. Caches are in a tree structure with larger ones near the root. The root also has the connection to external memory. The hierarchical design is typically used for SMP systems. Intel Quad-Core Xeon and Quad-Core AMD Opteron are also examples of standard desktop processors used a hierarchical design.

In the pipelined design, like conventional pipelines, there are some stages. The arriving data is processed successively by different cores in pipelined fashion. Since each core executes specific processing stages on each part of data, this architecture is called pipelined. Network processors and graphic processors have stages to be applied on data stream, makes them ideal to apply this architecture. Xelerator X11 network processor is an example of pipelined architecture with 800 processing cores arranged in a linear pipeline.

Probably the most interesting design of multicore machines is the network-based design. In this approach, processing cores and their local caches (or memories) are connected via an interconnection network. Examples of these multicore processors are Tilera Tile Processors [131] and Intel Teraflop processor [132]. The Teraflop architecture realises an 80-core prototype with a 2D mesh interconnection network. It is intended to reach more than 1Teraflops execution rate in less than 100W power consumption. TILEPro64 [4] also uses a 2D mesh of tiles, but has 64 processing tiles. It has six independent networks to isolate the traffic and has 2 modes of communication which will be discussed in the next section.

Providing an efficient interconnection network is one of the most important challenges for integrating more and more processing cores into a single die [133]. Is has to be scalable and capable to provide enough bandwidth for communication between cores. Generally, the performance and power efficiency of on-chip systems is constrained by their interconnection [12]. Since synchronisation over large distances is unmanageable, traditional bus-based

[3] The problem of providing a coherent view of the memory system

[4] Although there are some other products from Tilera corporation referred to as Tile Processors, for the rest of this document, the terms TILEPro64 and Tile Processor are used interchangeably

interconnect causes a bottleneck. Also, fixed point-to-point interconnect cannot be the solution, because as the number of cores becomes bigger, the number of links needed increases exponentially. Network-on-Chip (NoC) has emerged as a promising solution to overcome communication challenges between IP (Intellectual Property) cores [14]. It is based on the GALS [5] concept which means the chip is made up of locally synchronous islands that communicate asynchronously. In contrast to fixed point-to-point interconnection, the connectivity in NoC solution is flexible.

Recently, Tilera corporation is acquired by EZchip. The new multi/manycore processor product for EZchip is called TILE-Mx which provides up to 100 ARM Cortex-A53 64-bit cores. As most of the researchers agree [134] that *power* will be limiting the future of High Performance Computing (HPC), integrating the most power-efficient ARMv8 processors into a single chip could be of interest to many.

In this chapter, we describe two modern manycore architectures used for our experiments. Some important features of both platforms will be covered. At the end of this chapter, we compare the two manycores using GPRM programming models. However, the focus is not yet on the implementation details of GPRM program itself, but on the difference between the two platforms.

3.4 Tilera TILEPro64

The Tilera TILEPro64 Tile Processor [1] is composed of 64 tiles, interconnected via multiple 8×8 mesh networks (see Figure 3.1). Each tile contains a 32-bit integer processor with a three-way VLIW architecture[6], which allows to execute up to 3 operations per cycle. It provides distributed cache-coherent shared memory by default. It has four memory controllers and 16GB of DDR memory, but in order to use the global address space shared among all tiles, addressing is limited to 32-bit, i.e. 4GB. It has per-core L1 data caches of 8KB, L1 instruction caches of 16KB, and L2 caches of 64KB. The union of all L2 caches across the chip comprises the distributed L3 cache. The operating frequency of the cores is 866MHz. Out of 64 tiles, one is used for the PCI communication, and the other 63 tiles are available. TILEPro64 has neither a Vector Processing Unit (VPU), nor a Floating Point Unit (FPU).

The asymmetry in the physical distances between cores results in a design called Non-Uniform Cache Architecture (NUCA), in which a home core (tile) is associated with each memory address, and the access latency to the home core depends on the physical on-die location of the requesting core [135] [88].

[5] Globally Asynchronous Locally Synchronous

[6] In a VLIW architecture, multiple instructions could be executed at the same time (ILP).

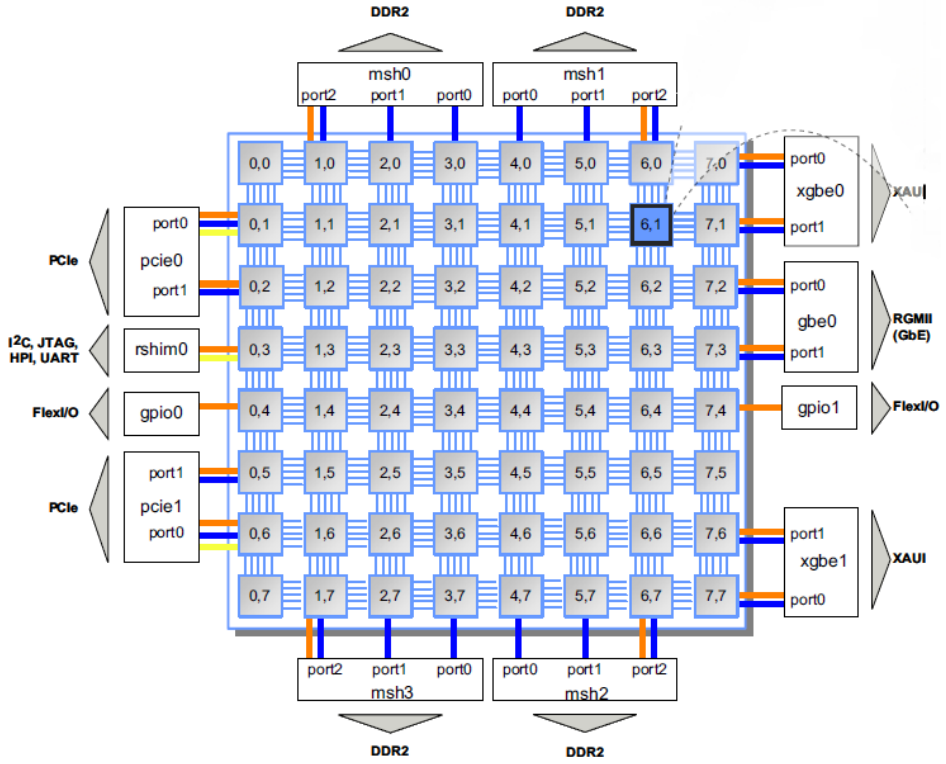


Figure 3.1: Tilera TILEPro64 Architecture (picture borrowed from [1])

The TILEPro64 targets a wide range of compute-intensive applications, such as digital multimedia, wireless infrastructure and advanced networking.

3.4.1 TILEPro64 Architecture

TILEPro64 [131] defines a globally shared, flat 36-bit physical address space and a 32-bit virtual address space. The global address space allows for instructions and data sharing between processes and threads. There is a large difference between the DRAM access time and the speed of the cores, which makes the cache organisation a crucial part of this architecture. Tilera's cache organisation –called Dynamic Distributed Cache (DDC)– is flexible and software configurable. Its aim is to provide a hardware-managed, cache-coherent approach to shared memory. DDC allows a page of the shared memory to be homed on a single tile or hashed across a set of tiles. Other tiles can cache this page remotely.

Each physical memory address in the TILEPro64 is associated with a *home tile*. Cache coherent view of the memory which is key in shared memory programming models is served through the *home tile*. A hardware coherency mechanism is also used to guarantee cache coherence among the tiles. Therefore, it is possible to cache read-write regions of memory in the cache of the tile running the code (local cache). The copy of each cache line can be requested from its *home tile*. If another tile writes new data to the cache line, the *home tile*

is responsible to invalidate all copies, and other tiles have to refetch the newer version. This behaviour makes DDC a dynamic cache organisation. Caching the data by the *home tile* itself is called L3 cache, because the home tile can be thought of as a higher level beyond the L2 cache. In other words, this concept can be thought of as having a virtual L3 cache on top of the actual local L2 caches. If an L2 miss occurs, the request will be first sent to *home tile* rather than directly to the DDR memory. Thus, the distributed L3 cache comprises the union of all L2 caches.

Another feature of DDC, called *hash for home*, is the capability of distributing the *home cache* of memory regions between different tiles at a cache-line granularity. As a result, the potential for hot spots is reduced, and the request traffic will be distributed across the whole chip.

The homing mechanism is basically intended to provide cache coherence, though it can also improve the performance by reducing the read instruction latencies. There are three different classes of homing in the Tile Processor system: I) Local homing, II) Remote homing, III) Hash for home. The local homing strategy homes the entire memory page on the same tile that is accessing the memory. Therefore, on an L2 miss, a request is sent directly to DDR memory. With the remote homing, a different tile than the one accessing the memory is used to home the entire memory page. Therefore, on an L2 miss, a request is first sent to the remote home tile's cache (which can be called the L3 cache). The *hash for home* strategy as described above is a new feature of DDC, which is very similar to remote homing. The only difference is that instead of mapping an entire memory page to a single home tile, it is hashed across different tiles at a cache-line granularity.

The Tilera's version of Symmetric Multiprocessing (SMP) Linux, called Tile Linux, which is based on the standard open-source Linux version 2.6.26, by default sets the home cache for a given page to be *hash for home*. This can be changed by the `ucache_hash` boot option.

3.4.2 TILEPro64 Performance Considerations

The performance of parallel applications on manycores cannot be estimated only based on their time complexity. The memory architecture plays an important role. Moreover, as the number of cores grows, memory contention becomes increasingly significant. We have shown in [22] that how the performance of a regular array-based computation can be improved by the use of the *localisation* technique. The idea is to make sure that the sequential memory accesses are cached in the local cache and accessed by the local thread (the thread tied to the core). The main reason behind our study was to demonstrate that the *hash for home* policy at the cache line granularity is too fine-grained for parallel array computations with lots of sequential memory accesses. The reason is that with this policy, the sequential parts of an array are homed on different tiles, and have different access latencies. A

comprehensive discussion is provided in [93].

Although, our *localisation* technique can be effectively implemented by GPRM, we leave it as a future work, and for the purposes of this work, we use the default hashing policy of the TILEPro64. Also in general, *hash for home* is the preferred option for most of the parallel applications running on the Tilera chip, as it aims to reduce the potential for bottlenecks.

In [22], we have also investigated the effect of other possible options provided by the Tilera hypervisor. The hypervisor configuration file (.hvc) used for this study is as follows [136]:

```
1 options stripe_memory=default
2 options default_shared=0,0
3 device srom/0 srom
4 device pcie/0 pcie
5  dedicated 7,7
6 client vmlinux
7  args $XARGS
```

Listing 3.1: Hypervisor Configuration File for the TILEPro64

Memory pages can be allocated either through a specific memory controller or in striping mode, where each page is striped across all memory controllers in 8KB chunks. With memory striping, Linux will boot up believing it has a single memory controller that is four times larger than any of the actual physical memory controllers. The effect of memory striping is considerable when caching is turned off across the system. However, when caching is enabled, it is mostly transparent to the user.

The default memory striping policy is used, and the last core in the mesh (7,7) is dedicated to the PCI communication between the device and host. Therefore, 63 cores can be used to run the user code.

The most important conclusion of our work was that the native GNU/Linux thread scheduling is not as efficient as expected. By static mapping of threads to cores, high-cost thread migrations do not occur multiple times during the execution time. The result of this work helped us consider a static thread to core mapping inside our framework, and instead focus on the higher level scheduling of tasks on threads.

3.5 Intel Xeon Phi

The Intel Xeon Phi coprocessor 5110P used in this study is an SMP (Symmetric Multiprocessor) on-a-chip which is connected to a host Xeon processor via a PCI Express bus interface. The Intel Many Integrated Core (MIC) architecture used by the Intel Xeon Phi coprocessors

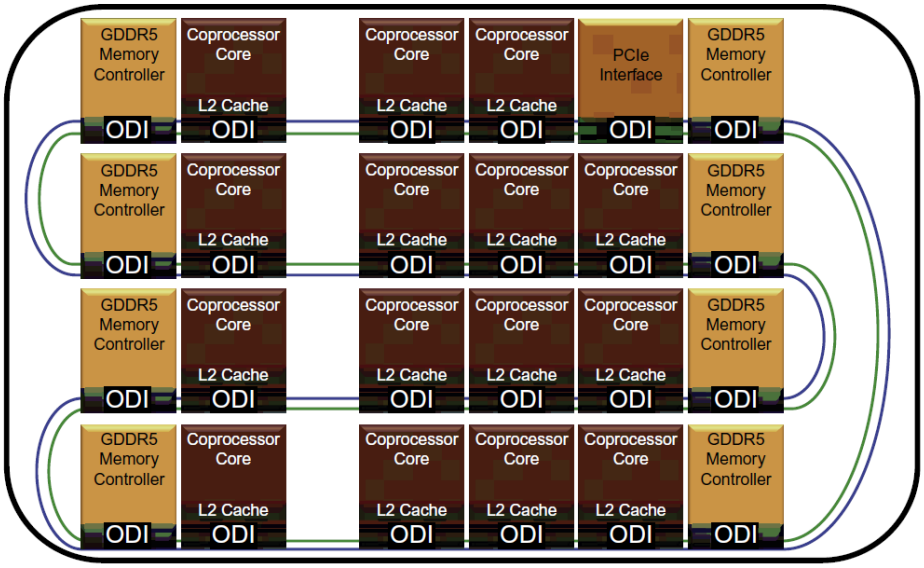


Figure 3.2: Intel Xeon Phi Architecture (picture borrowed from [2])

gives developers the advantage of using standard, existing programming tools and methods. Our Xeon Phi comprises 60 cores (240 logical cores) connected by a bidirectional ring interconnect (see Figure 3.2).

If an application is scalable on the Xeon processors, can make use of vector units, and is able to utilise more memory bandwidth than available with the Xeon processors, then it could be a potential target for the Xeon Phi [2].

3.5.1 Xeon Phi Architecture

The Xeon Phi coprocessor provides four hardware threads sharing the same physical core and its cache subsystem in order to hide the latency inherent in in-order execution. As a result, the use of at least two threads per core is almost always beneficial [2]. The Xeon Phi has eight memory controllers supporting 2 GDDR5 memory channels each. The clock speed of the cores is 1.053GHz. Each core has an associated 512KB L2 cache. Data and instruction L1 caches of 32KB are also integrated on each core. Another important feature of the Xeon Phi is that each core includes a SIMD 512-bit wide VPU (Vector Processing Unit). The VPU can be used to process 16 single-precision or 8 double-precision elements per clock cycle.

3.5.2 Xeon Phi Performance Considerations

Providing four hardware threads (logical cores) sharing the same physical core is known as multithreading in the Xeon Phi. We use the term multithreading [2] here to describe the difference with hyper-threading on the Xeon processors. Throughout this dissertation,

however, multithreading refers to software thread parallelism.

The use of multithreading as a part of the Xeon Phi architecture is crucial to hide latencies of its in-order microarchitecture. Hyper-threading on the Xeon processors, on the other hand, is designed to feed a dynamic execution engine, and depending on the application can be fully ignored without having negative impact of performance. The hardware multithreading on the Xeon Phi should not be ignored similarly.

Generally, the floating-points and memory capabilities the hardware threads offer cannot be achieved with a single thread per physical core. On the other hand, it is also important to note that saturation could happen with even two hardware threads, and as we will see in the following chapters, different applications implemented by different parallelisation approaches experience varying levels of saturation.

It is beneficial to parametrise the number of cores as well as the number of hardware threads per core for applications targeting future manycore architectures.

3.6 Summary

This chapter covered a background on parallel architectures. We started the discussion with the Flynn's taxonomy and continued with a section on memory organisation. We then shortly reviewed a number of multicore machines. We also discussed the architecture of two manycore systems, the TILEPro64 and the Intel Xeon Phi in more detail.

With the increasing number of cores, new programming challenges arise. Understanding the core architectural concepts of a given parallel platform is key to writing correct and efficient parallel programs, regardless of the programming model used. Improving the data locality in manycore architectures is an important factor for achieving high performance. Although there is a lot of fine-grained architecture-specific control that every new platform offer to its users, but in general, to benefit from these features, existing codes have to be changed significantly. Considering such details (e.g. the effect of distributed caches) in the design of runtime systems could help the programmers notably.

It is also important to bear in mind that task and thread scheduling decisions can impose a significant overhead as the number of cores grows. The trade-off is thus to maximise the performance and data locality, while keeping the runtime overhead low. This will be the basis of our discussions in the next chapters.

Chapter 4

Task-based Parallel Models for Shared Memory Programming

In a general-purpose system, applications residing in the system compete for shared resources. Thread and task scheduling in such a multithreaded multiprogramming environment is a significant challenge.

After an introduction to parallel programming models and the concept of task parallelism in the background chapter (Ch. 2), in this chapter we would like to investigate performance characteristics of three popular task-based parallel programming models on a modern many-core system, the Intel Xeon Phi. The main three task-based models that are supported by *icpc* (Intel's C/C++ Compiler) are Intel OpenMP, Intel Cilk Plus, and Intel TBB.

We have used three benchmarks with different features which exercise different aspects of the system performance. Moreover, a multiprogramming scenario is used to compare the behaviours of these models when all three applications reside in the system.

Furthermore, at the end of this chapter we continue the discussion about multiprogramming using examples from our research work on OpenMP applications running on the TILEPro64.

In summary, this chapters reviews our work on other approaches and presents the lessons learnt that helped us design, tune, and improve the GPRM runtime system from its beginning to the present.

4.1 Three Popular Task-based Parallel Models

First, the three chosen models for shared memory programming on the Intel Xeon Phi are described in more detail.

4.1.1 OpenMP

We have introduced OpenMP as the de-facto standard for shared memory programming. OpenMP provides a set of compiler directives, tools, and environment variables which can simplify parallel application development for shared memory architectures with multiple cores. However, the application developers need to be aware of the OpenMP memory model, which provides private and shared data [19]. A notable feature of OpenMP is that it is under active development, and new features are proposed frequently in order to make it more flexible and adaptable to new architectures. One recent example is the support for task dependence since the release of OpenMP 4.0.

Writing of multithreaded programs can become quite complex, without defining certain rules. OpenMP attempts to ease the process by supporting the fork-join model [137]. We have to be clear about the fork-join term, as it can be applied at different level and can imply different meanings. For example, Chapman [19] uses this concept for threads in the earlier versions of OpenMP (before OpenMP 3.0 [1]) as follows: the starting part of the program is executed by a single thread; whenever, a `parallel` construct encounters, a team of threads is created (*fork*); members of the team execute the code collaboratively, and at the end of the construct [2], all team members except the master thread terminate (*join*). The concept of fork-join is used in [126] as a parallel control pattern, where the control flow forks into multiple parallel flows that will join later. In this context –as well as throughout this dissertation–, the focus is on the dynamic task creation and execution in newer versions of OpenMP, which also follow the fork-join pattern using `task` and `task-wait` constructs. It is also important to note that still parallelism happens only in a parallel region.

The Intel OpenMP runtime library (as opposed to the GNU implementation) allocates a task list per thread for every OpenMP team. Whenever a thread creates a task that cannot not be executed immediately, that task is placed into the thread's deque (double-ended queue). A random stealing strategy balances the load [138].

4.1.2 Cilk Plus

Cilk Plus has evolved from Cilk [139], and is an extension to C/C++ with a few additional keywords and an array section notation. It provides very simple but powerful ways of specifying parallelism, as it is integrated into the compiler. It features a fork-join pattern to support irregular patterns and nesting. Cilk Plus provides the `_cilk_spawn` and `_cilk_sync` keywords to spawn and synchronise tasks; `_cilk_for` loop is a parallel replacement for

[1] OpenMP is said to be thread-based before adding the concept of tasks, where all threads have access to the shared memory and worksharing directives (`single`, `for`, `section`, etc.) are used to distribute the work between them [3].

[2] The region enclosed by a parallel construct is called a parallel region.

sequential loops in C/C++. Cilk Plus has a syntactic extension to express fork-join: instead of calling a function, by using `_cilk_spawn` it spawns the function, which means the caller can continue its execution without waiting for the callee to return (fork); `_cilk_sync`, waits for all spawned calls in the current function to join.

The tasks are executed within a work-stealing framework. Every worker thread has deque of tasks. The worker treats its deque as a stack, by pushing and popping tasks at the back of it. Thieves steal from the front of deques [45]. In the Cilk Plus work-stealing framework, thieves steal *continuations*, meaning that the spawned task is immediately started by the spawning thread, and the continuation is left available for stealing. The Cilk Plus scheduling policy provides load balancing close to the optimal [117]. The Intel implementation of Cilk Plus ensures that by running a program on one processor, the same order of operations as the equivalent sequential program is produced [126].

4.1.3 Threading Building Blocks (TBB)

Intel Threading Building Blocks (TBB) is another well-known approach for expressing parallelism [43]. TBB is an object-oriented C++ template library that contains data structures and algorithms to be used in parallel programs.

Parallelism can be expressed in terms of tasks, represented as instances of the `task` class, or concurrent container classes, which allow the access of multiple threads to items of a container.

TBB supports both regular and irregular parallelism, and has direct support for a various parallel patterns, such as task graphs, map, pipelines, etc. TBB abstracts the low-level thread interface. However, conversion of legacy code to TBB requires restructuring certain parts of the program to fit the TBB templates.

TBB uses a library for supporting the fork-join pattern. Similar to Cilk Plus, a common thread pool is shared by all tasks and load balancing is achieved by work-stealing. Each worker thread in TBB has a deque of tasks. Newly spawned tasks are put at the back of the deque, and each worker thread takes the tasks from the back of its deque. If there is no task in the local deque, the worker steals tasks from the front of the victims' deques [140]. But in TBB, thieves steal *children*, meaning that the worker thread spawns a new task and leaves it. It executes the continuation, for example if it is executing a loop, it proceeds to the next iteration and spawns more tasks afterwards, leaves them for stealing, likewise. Furthermore, if it picks a task to run, it would be the last spawned one, as it is the one recently pushed at the back of its deque.

4.2 Comparison on the Xeon Phi: Uniprogramming Workloads

We use three benchmarks, which we refer to as our "Base Benchmarks" in this study. They are used to show that apparently equivalent parallel programming models have different behaviours on modern systems, even for naive algorithms. They are intentionally simple to make it possible to reason about the observed differences between the performance of the selected models.

We compare the speedup results not only for varying number of threads, but also for varying cutoff values. The concept of cutoff has been introduced already [3]. It is worth mentioning that both GCC and ICC control the creation of tasks for OpenMP programs, but as shown in [104], such limitations are not enough for achieving good performance. For example GCC 4.9 cuts off task creation if the number of tasks exceeds $64 \times number\_of\_threads$, or ICC avoids task creation if the task queues are full.

4.2.1 Experimental Setup

All the benchmarks are implemented as C++ programs, and all speedup ratios are computed against the running time of the sequential code implemented in C++. We first compare the results for each single program.

The benchmarks executed natively on the Xeon Phi. For that purpose, the executables are copied to the Xeon Phi, and we connect to it from the host machine using `ssh`. The Intel compiler `icpc (ICC) 14.0.2` is used with the `-O2 -mmic -no-offload` flags for compiling the benchmarks for native execution on the Xeon Phi. The OpenMP programs should be compiled with the `-openmp` flag. The TBB programs need the `-ltbb` flag.

For all approaches some shared libraries must be copied to the Xeon Phi. For the OpenMP applications, the `libiomp5.so` library is required. The `libcilkrts.so.5` is needed for Cilk Plus applications and the `libtbb.so.2` library is required for the TBB programs. The path to these libraries should be set before the execution, for example: `export LD_LIBRARY_PATH=./:$LD_LIBRARY_PATH`.

[3] Although the cutoff value is sometimes equal to the total number of tasks reside in the system, in this dissertation we consider it as the maximum number of ready-to-run tasks that can exist at any point of time in the system

4.2.2 Parallel Fibonacci Benchmark: Fibonacci

We consider a parallel Fibonacci benchmark as the first testcase. The Fibonacci benchmark (calculating the Fibonacci numbers $fib(i) = fib(i-2) + fib(i-1)$ by concurrent recursion) has traditionally been used as a basic example of parallel computing. Although it is not an efficient way of computing Fibonacci numbers, the simple recursive pattern can easily be parallelised and is a good example of creating unbalanced tasks, resulting in load imbalance. In order to achieve desirable performance, a suitable cutoff value for the recursion is crucial. Otherwise, too many fine-grained tasks would impose an unacceptable overhead to the system. The cutoff limits the tree_depth in the recursive algorithm, which results in generating 2^{tree_depth} tasks.

Figure 4.1 shows all the results taken from running this benchmark with the three programming models. Figure 4.1(a) shows the speedup chart for the integer number 47 with 2048 unbalanced tasks at the last level of the Fibonacci heap. Cilk Plus and TBB show similar results. Increasing the number of threads causes visible performance degradation for OpenMP. Setting `KMP_AFFINITY=balanced` results in a negligible improvement of the OpenMP performance.

Figure 4.1(b) shows the importance of a proper cutoff on the performance of this unbalanced problem. Having more tasks (as long as they are not too fine-grained) gives enough opportunities for load balancing.

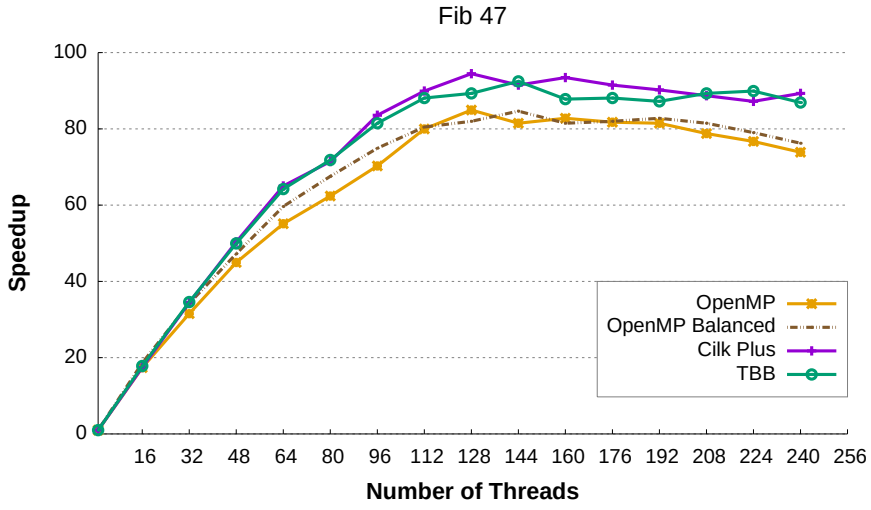
Total CPU Time

For the applications developed for the Intel architectures, Intel VTune Amplifier [141] is considered to be the de-facto performance analyser tool. However, we have to figure out what performance metrics can best describe the differences [142].

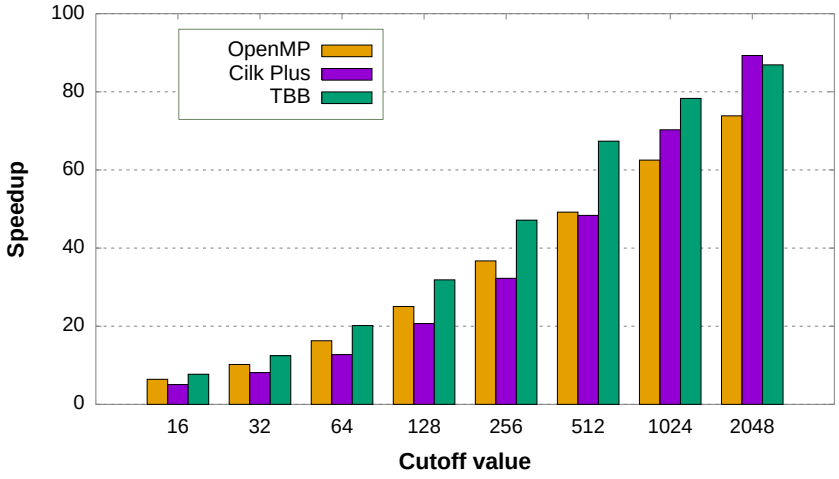
This is a lower-is-better metric that shows the total CPU times consumed in the system from the start until the accomplishment of the job(s). This metric and the detailed breakdown of CPU times are obtained using Intel VTune Amplifier XE 2013 performance analyser [143]. Figures 4.1(d) to 4.1(f) are screenshots taken from the VTune Amplifier when running Fib 47 with cutoff 2048 natively on the Xeon Phi. The x-axis shows the logical cores of the Xeon Phi (240 cores), and the y-axis is the CPU time for each core [4].

For the Fibonacci benchmark, OpenMP consumes the most CPU time, and its performance is bad, the worst amongst the three approaches.

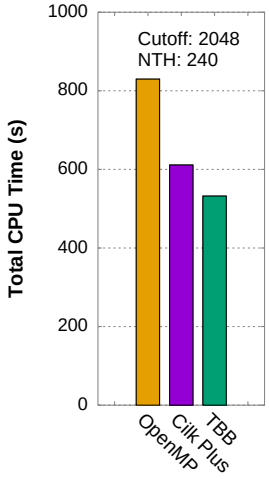
[4]It should be noted that for all experiments, results from the benchmark's kernel are considered in the figures (a) and (b), while in the other results taken from the VTune Amplifier, all information from the start of the application, including its initial phase and the CPU time consumed by the shared libraries is taken into account.



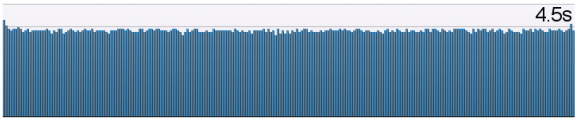
(a) Speedup, cutoff 2048, varying numbers of threads



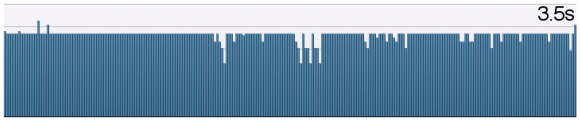
(b) Speedup, 240 threads, varying cutoffs



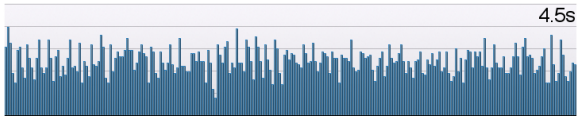
(c) Total CPU Time



(d) OpenMP, CPU balance



(e) Cilk Plus, CPU balance



(f) TBB, CPU balance

Figure 4.1: Parallel Fibonacci benchmark for the integer number 47.
The best performance can be obtained by using Cilk Plus or TBB.
Choosing a proper cutoff value is key to good performance. If there are enough tasks in the system, the load balancing techniques become effective and yield better speedup.
A detailed breakdown of overall CPU time for the case with 240 threads and cutoff value 2048 is illustrated for each approach in the charts (d) to (f). TBB consumes less CPU time in total while providing good performance, and Cilk Plus has the best performance. The y-axis on the (d) to (f) charts is the time per logical core, from 0 to the maximum number specified in seconds.

4.2.3 Parallel Merge Sort Benchmark: MergeSort

Sorting algorithms have attracted a great attention, due to their simple concept but complex optimisation. They are basic blocks of many applications, such as databases, data mining applications, and computer graphics [93]. Furthermore, they are memory-bound, which makes them suitable candidates for investigating the effect of memory organisation on performance. Although based on their time complexity, most of them do not scale linearly with the number of threads, they can still be parallelised easily and effectively.

Merge sort is an efficient divide-and-conquer algorithm and its parallel version is a great example of parallel reduction. The average complexity of its serial version is $O(nlogn)$.

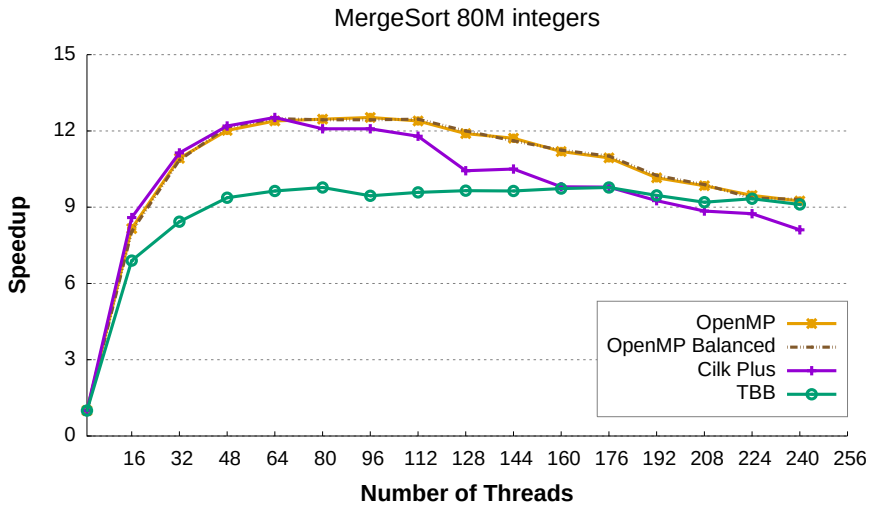
This benchmark sorts an array of 80 million integers using a merge sort algorithm. The ith element of the array is initialised with the number $i*((i\%2)+2)$. The cutoff value determines the point after which the operation should be performed sequentially. For example, cutoff 2048 means that chunks of 1/2048 of the 80M array should be sorted sequentially, in parallel, and afterwards the results will be merged two by two, in parallel to produce the final sorted array.

As shown in Fig. 4.2(a) with larger numbers of threads, there is either no noticeable change (in the case of TBB), or a slowdown (in the case of OpenMP and Cilk Plus). Using thread affinity for OpenMP in this case does not make an appreciable difference.

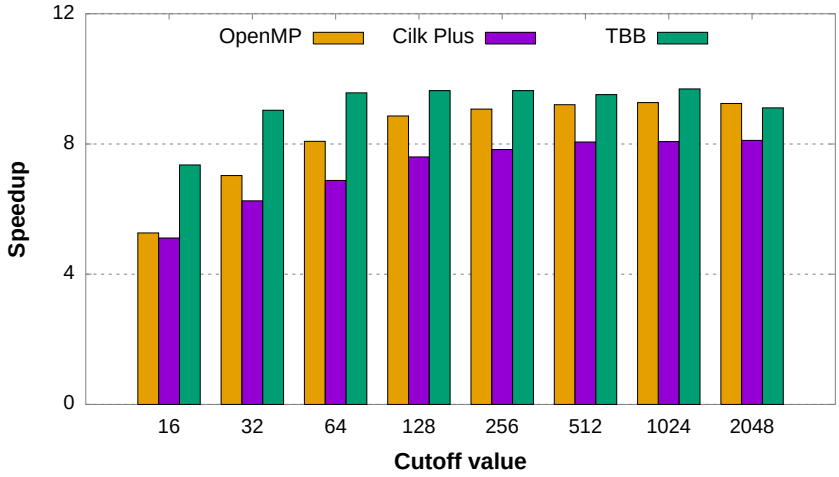
Figures 4.2(c) to 4.2(f) are again based on the results obtained by the VTune Amplifier when running the benchmark with 240 threads and cutoff 2048. Although the TBB performance is not the best, the Total CPU Time it consumes is significantly less than the other two approaches. This is due to its more light-weight runtime library. Since all merges in a branch of the task tree can run on the same core as their children, there would be no need to have balanced load for good performance. In other words, the unbalanced distribution in Fig. 4.2(f) does not imply a poor behaviour of the TBB runtime library.

4.2.4 Parallel Matrix Multiplication Benchmark: MatMul

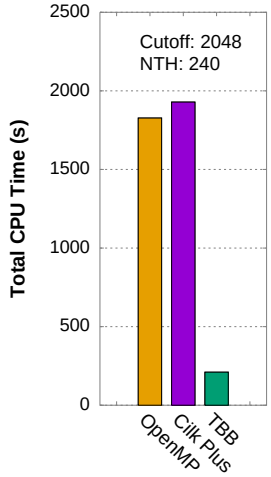
This benchmark performs a naive matrix multiplication by a triple nested loop with ikj loop ordering for caching benefits on square matrices of N×N double-precision floating point numbers. This is a completely data parallel problem which fits very well to OpenMP and its `for` worksharing construct. There is a concept similar to the cutoff in the loop parallelism context to control chunking. It specifies the size of chunk for each thread in a data parallel worksharing scenario. If the cutoff value is assumed as the number of chunks, the chunk (grain) size can be specified for the OpenMP `for` as follows: `#pragma omp for schedule(dynamic, N/cutoff)`. The `dynamic` keyword can be replaced by



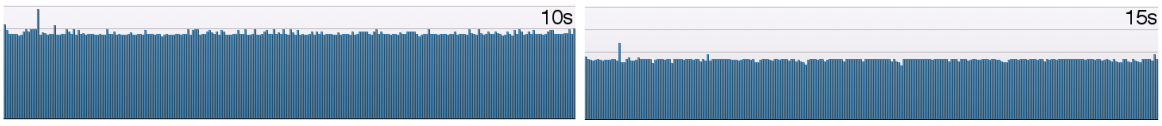
(a) Speedup, cutoff 2048, varying numbers of threads



(b) Speedup, 240 threads, varying cutoffs

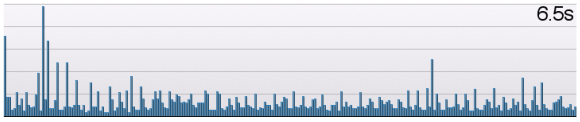


(c) Total CPU Time



(d) OpenMP, CPU balance

(e) Cilk Plus, CPU balance



(f) TBB, CPU balance

Figure 4.2: Parallel MergeSort benchmark for an array of 80 million integers.
This benchmark does not scale well. The best performance, however, can be obtained by using OpenMP or Cilk Plus.
For this memory-intensive benchmark, cutoff values greater than 64 are enough to lead to good performance with as many threads as the number of cores.
TBB consumes significantly less Total CPU Time. With small number of threads, OpenMP and Cilk Plus yield better performance, but finally (with 240 threads) OpenMP and TBB provide slightly better performance.

`static` as well. Grain size in the Cilk Plus is similarly specified via a pragma: `#pragma cilk grainsize = N/cutoff`. Intel TBB has a template function for this purpose, namely `parallel_for`, which can be called with `simple_partitioner()` to control the grain size.

Scheduling Considerations

Before going into details of the results, we would like to focus on some technical considerations:

In order to achieve automatic vectorization on the Xeon Phi, the Intel TBB and OpenMP codes have to be compiled with the `-ansi-alias` flag to resolve the compiler's confusion about the vector dependence.

The `schedule` clause used with OpenMP `for` specifies how iterations of the associated loops are divided (statically/dynamically) into contiguous chunks, and how these chunks are distributed amongst threads of the team. In order to have a better understanding of the relations between the cutoff value (number of the chunks), number of threads, and the thread affinity on the Xeon Phi, consider the following example. Suppose that for the MatMul benchmark, the OpenMP `for` construct with static schedule is used, which means that iterations are divided statically between the execution threads in a round-robin fashion:

`#pragma omp for schedule(static, N/cutoff).`

Runtime of the case(a) on the Xeon Phi is $\approx 3\times$ better than that of the case(b).

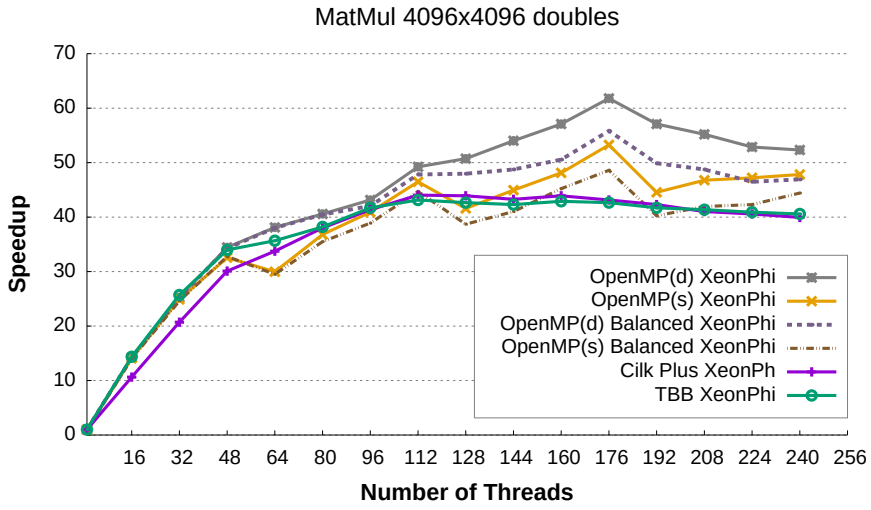
- `omp_set_num_threads(32)`, `cutoff=32`, `KMP_AFFINITY=balanced`

  The threads will be spread across 32 physical cores. With the balanced affinity, they have to be distributed as evenly as possible across the chip, which is one thread per physical core. As a result, every chunk will be run on a separate physical core.

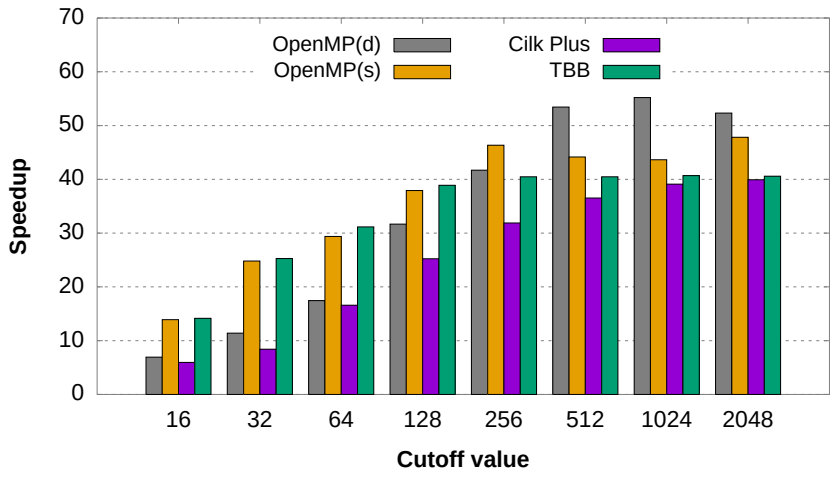
- `omp_set_num_threads(240)`, `cutoff=32`, `KMP_AFFINITY=balanced`

  The threads will be spread across all 60 physical cores. But the work will be distributed between 8 physical cores, which are the first 32 hardware threads. The reason is that with 240 threads, there will be one thread per logical core, and with cutoff 32, every thread with the thread id from 0 to 31 gets a chunk of size N/32.

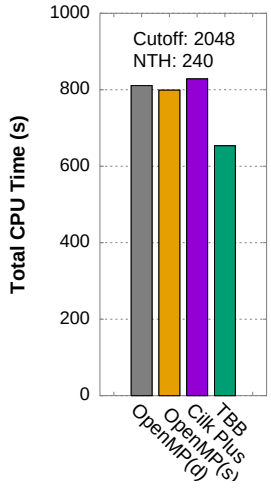
With these considerations, we are ready to run the MatMul benchmark and compare the programming models in a data parallel scenario. The results can be found in Fig 4.3.



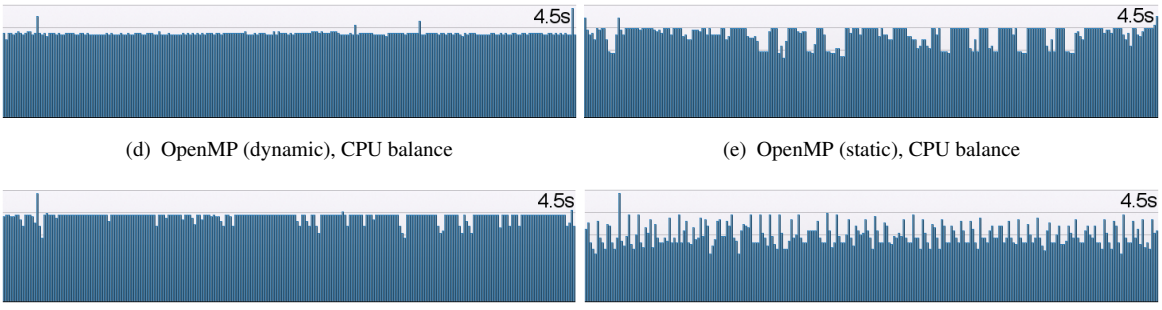
(a) Speedup, cutoff 2048, varying numbers of threads



(b) Speedup, 240 threads, varying cutoffs



(c) Total CPU Time



(d) OpenMP (dynamic), CPU balance

(e) OpenMP (static), CPU balance

(f) Cilk Plus, CPU balance

(g) TBB, CPU balance

Figure 4.3: Parallel MatMul benchmark on a 4096×4096 matrix of double numbers.
The best results can be obtained by using OpenMP approaches.
For the cutoff values greater than 256, OpenMP with dynamic scheduling has the best scaling amongst all.
Again the Total CPU Time of TBB is the least amongst all. There is an evident distinction between the distribution of CPU times in the charts (d) and (e) that shows how OpenMP load balancing, when using dynamic scheduling leads to better performance.

4.2.5 Overhead of the Runtime Libraries

One way to reason about the differences between these parallel programming models is to compare the amount of the Total CPU Time consumed by their runtime libraries. We have therefore summarised the results as the percentage of time spent on the shared libraries in each case.

Table 4.1: Percentage of the Total CPU Time consumed by the runtime libraries

Benchmark	**OpenMP** (`libiomp5.so`)	**Cilk Plus** (`libcilkrts.so.5`)	**TBB** (`libtbb.so.2`)
Fibonacci	50%	16%	5%
MergeSort	78%	81%	3%
MatMul	22% *(Dynamic)* 20% *(Static)*	6%	1%

Table 4.1 gives a better understanding of where the CPU times have been consumed. For instance, in this case [5], for the OpenMP runtime library, the wasted CPU time generally falls into two categories: [6] I) A master thread is executing a serial region, and the slave threads are spinning. II) A thread has finished a parallel region, and is spinning in the barrier waiting for all other threads to reach that synchronisation point.

Although sometimes in solo execution of the programs, these extra CPU cycles have negligible influence on the running time (wall time), we show in Sect. 4.4 how they would affect other programs under multiprogrammed execution.

4.3 Comparison on the Xeon Phi: Multiprogramming Workloads

Applications are shifting towards increased parallelism. Improving the performance of multiple parallel applications running together on the same machine is equally important as improving the performance of a stand-alone application. Thread and task scheduling in such a dynamic environment is a significant challenge, and scheduling algorithms designed for uniprogramming environments are no longer efficient. The problem lies in the assumption that a dedicated set of execution resources are fully available to the program, which is not always the case [144]. Therefore, in the presence of an external workload, such algorithms

[5]The default policy for the OpenMP idle threads depends on the implementation, but in most of the implementations going to sleep after a short period of spinning is the default policy.

[6]In general, there might exist other synchronisation overheads, e.g. waiting to acquire a lock or access to a critical section.

may lead to a significant drop in performance.

We explore how Intel OpenMP, Intel Cilk Plus, and Intel TBB react to a multiprogramming situation on the Xeon Phi. Based on the overhead of the runtime libraries for single program execution (Section 4.2.5), we predict that TBB would perform better than the other two.

4.4 Multiprogramming Benchmark

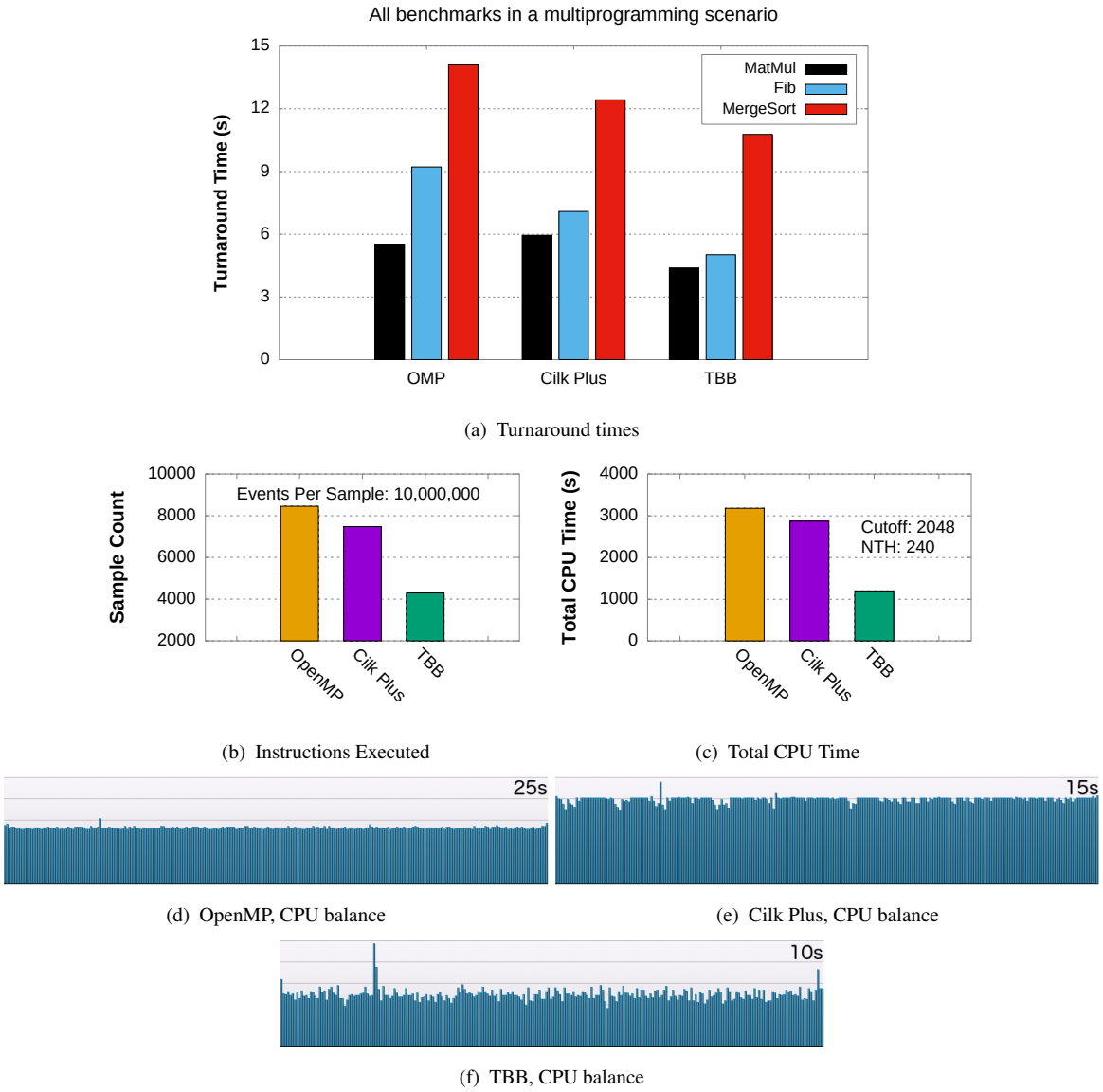
In this section, we consider a multiprogramming scenario to see how the Intel's parallel models behave in a multiprogramming environment. Generally, scheduling policies can be evaluated based on system-oriented or user-oriented criteria [145]. A system-oriented metric is based on the system's perspective and quantifies how effectively and efficiently the system utilises the resources, while the focus of a user-oriented metric is on the behaviour of the system from user's perceptive, e.g. how fast a single program is executed. An example of system-oriented metrics is *throughput*, which is the number of programs completed per unit of time. *Turnaround time* is an example of user-oriented metrics, which is the time between submitting a job and its completion. It is used in this section for the comparison of the models in multiprogramming situations.

The three benchmarks have the same input sizes as the uniprogramming cases in Sect. 4.2 with the cutoff value 2048 and the default number of threads 240 (the same as the number of logical cores in the Xeon Phi). We do not start all of them at the same time. Rather, we want the parallel phases to start almost simultaneously, such that all of the applications' threads compete for the resources. For that purpose, the MergeSort benchmark enters the system first. Two seconds later the MatMul benchmark enters the system, and half a second after that, the Fib benchmark starts [7].

Based on the results obtained from single programs, we expect TBB to perform best because it has the least Total CPU Time in all three benchmarks. It might not affect the runtime of a single program significantly, but when there are multiple programs competing for the resources, the wasted CPU time can play an important role. In other words, CPU time wasted by each program can influence the performance of other programs reside in the system [146].

The results are shown and discussed in Fig. 4.4

[7] The sequential phase of the MergeSort benchmark with the input size 80 million is around 2 seconds, and the initial phase of the MatMul benchmark with the input size 4096×4096 is about half a second.



(a) Turnaround times

(b) Instructions Executed

(c) Total CPU Time

(d) OpenMP, CPU balance

(e) Cilk Plus, CPU balance

(f) TBB, CPU balance

Figure 4.4: A multiprogramming scenario with the three benchmarks
This is what happens when the three benchmarks compete for the resources: (a) shows that the best turnaround times are obtained with TBB. The hardware event, number of Instructions Executed, sampled by the VTune Amplifier in (b), implies a significant difference between TBB and the other two competitors. Results from the Total CPU Time in chart (c) is similar to those in chart (b) and they both show why TBB performs better than OpenMP and Cilk Plus. A detailed breakdown of overall CPU time in the (d) to (f) charts illustrates how OpenMP consumes more CPU time in total, and therefore has the worst performance.

4.5 Information Sharing and Multiprogramming

As we discussed in this chapter, multiprogramming on manycores is challenging. We have shown that even though OpenMP has a competitive performance with TBB for single programs on the Xeon Phi, the performance difference for a multiprogram workload is huge. Improving the performance of OpenMP and exploring the effect of different factors in this case are beyond the scope of this study. Rather, we would like to show an example of one our previous work [8] on the TILEPro64 to highlight the importance of information sharing in a multiprogrammed system.

Sasaki et al. [82] developed a co-scheduling scheme for multiprogramming environments, based on dynamically prediction of the applications' scalability. They also highlighted the fundamental problem that the performance of some applications without linear scalability tend to decrease drastically when more number of cores are allocated to them. Limiting the number of threads is a way to overcome this issue. Therefore, for an application with smaller number of threads than the number of cores, it becomes important to determine where to map those threads. In this section, we review our previous work on multiprogramming to explain how we addressed this issue by sharing global information about the *current CPU load* of the tiles before mapping the threads to them.

4.5.1 Thread Mapping Strategies

We consider four different mapping options for running OpenMP-based benchmarks on the TILEPro64. Except from the first one (decision by the OS), every thread decides about its mapping itself. It first finds a suitable core, maps itself to it and starts doing some work or goes to sleep. In the OpenMP code, it happens after the *parallel* keyword, which is the point where the thread creation happens.

First, let's explain some details about the idle threads in a GNU implementation of OpenMP: if `OMP_WAIT_POLICY` is undefined (similar to our test cases), threads wait actively for a short period before waiting passively without consuming CPU power. In the GNU Offloading and Multi Processing Runtime Library (`libgomp`), this period can be set by the use of `COMP_SPINCOUNT`. If undefined (similar to our test cases), 300,000 spins of the busy-wait loop will be set [147].

Now we are ready to go into the details of the mapping techniques for programs running on the TILEPro64.

[8] This section covers some of my PhD work which was not directly related to GPRM, but helped me in understanding multiprogramming issues and further developing of an "information sharing" mechanism in GPRM

1. **Linux Scheduler:**

   The first option is to leave any scheduling decision to the native Linux scheduler. The Tilera's version of SMP Linux, called *Tile Linux* is based on the standard open-source Linux version 2.6.26. The default scheduling strategy in Linux is a priority-based dynamic scheduling that allows for thread migration to idle cores in order to balance the run-queues.

   Having a single run-queue for all processors in a Symmetric Multiprocessing (SMP) system, and using a single run-queue lock were some of the drawbacks of the Linux 2.4 scheduler. Linux 2.6 implemented a priority-based scheduler known as the O(1) scheduler, which means the time needed to select the appropriate process and map it to a processor is constant. One run-queue data structure per each processor keeps track of all runnable tasks assigned to that processor.

   At each processor, the scheduler picks a task from the highest priority queue. If there are multiple tasks in that queue, they are scheduled in a Round-Robin manner. There is also a mechanism to move tasks from the queues of one processor to those of another. This is done periodically by checking whether the `cpu_load` is imbalanced. In the Linux terminology, `cpu_load` is the average of the current load and the old load. The current load is the number of active tasks in the CPU's run-queue multiplied by `SCHED_LOAD_SCALE`, which is used to increase the resolution of the load [148]. What we will refer to as *load* in this section is the amount of time spent in each processor doing some useful work.

2. **Static Mapping:**

   In the static mapping, OpenMP threads are pinned to the processing cores based on their *thread_id*s in an ordered fashion. The decision is taken at compile time, which would cause an obvious disadvantage: It cannot tune itself with multiprogramming, since every program follows the same rule, and if the number of threads are less than the number of cores, then some cores get no threads at all. It might be discussed why at the first place, the number of threads in each program should be less than the number of cores. As mentioned earlier, the answer to this question can be found in the applications which do not have linear speedup, and after a certain number of threads reach their saturation phase. One example is the Sort program in the Barcelona OpenMP Tasks Suite (BOTS) [149]. Generally, the scalability of some programs tend to saturate at some points, and their performance is degraded by adding more cores [82]. Some programming models such as OpenMP suffer more from this situation, while, as we will show, others such as our own parallel model, GPRM can handle it better. Therefore, the techniques we describe here are more useful for OpenMP programs. However, the result of this work revealed the importance on information

sharing between the programs and was the motivation behind the implementation of the *Global Sharing* feature in GPRM.

3. **Basic Lowest Load (BLL):**

   The Lowest Load mapping technique is presented as two different methods. The first one, assumes the term *load* as an equivalent to a thread. Therefore, if one OpenMP thread is mapped to a core, the core's load becomes 1. We call this method *Basic Lowest Load* (BLL). It fills out the cores of the system in a Round-Robin fashion. This technique is not aware of what is going on inside the system. There are many situations in which some idle cores are ignored, e.x. a short program finishes its execution on them, but the mapper does not use them, because it only points to the next core in its list.

4. **Extended Lowest Load (XLL):**

   The *Extended Lowest Load* (XLL) gets the cores' information from the */proc/stat* file in Linux. The amount of time each core has done different types of work is specified with a number of time units. The time units are expressed in `USER_HZ` or Jiffies, which are typically hundredths of a second. The number of Jiffies in user mode is selected as *load*. In this technique, every OpenMP thread scans the current *load*s of the cores. It then searches for a core with the least change from its old *load* value. The thread maps itself to that core and starts working. In other words, the actual target of this policy is the least busy core. Except from its dynamic awareness of the system, another difference with BLL becomes highlighted when a thread is created but goes instantly to the sleeping mode. XLL automatically finds the sleeping threads since they do not produce any *load*, and hence more threads can be assigned to the corresponding cores, while BLL only counts the number of pinned threads to the core, no matter if they are sleeping or doing some work. The algorithm for XLL methodology is shown in Algorithm 1.

   The proposed methodology requires a globally shared data structure that keeps track of the system's cores. This data structure can be implemented in a runtime system as in our work, or can be embedded in the Linux kernel. It is worth mentioning that this methodology is portable across similar multicore/manycore platforms.

Algorithm 1 The XLL Methodology

```
1:  procedure FINDBESTTARGET
2:      GetTheLock();
3:      for each int i in Cores  do
4:          Scan(CurrentLoad[i]); %Scans from the /proc/stat file
5:          Cores[i].change = CurrentLoad[i] - Cores[i].load + Cores[i].pinned;
6:          Cores[i].load = CurrentLoad[i] + 10; %Creates a better resolution
7:      end for
8:      for each int i in Cores  do
9:          if Cores[i].change <Cores[BestTarget].change then
10:             BestTarget = i; %Finds the least busy core
11:         end if
12:     end for
13:     SetAffinity(BestTarget);
14:     Cores[BestTarget].pinned++; %Increments the number of pinned threads
15:     ReleaseTheLock();
16: end procedure
```

4.5.2 Selected OpenMP Benchmarks for the TILEPro64

We show how different thread mapping strategies can affect the performance of four benchmarks from the Barcelona OpenMP Tasks Suite (BOTS) [149], selected for their different characteristics. The mapping techniques are low-overhead, and can be combined with different cut-off strategies and applied on either *tied* or *untied* tasks.

It is important to note that the applications which do not scale very well are more challenging for parallel computing. Embarrassingly parallel algorithms are easy to parallelise since the tasks are completely (or almost) independent. They can easily run on different processing cores without the need to share data or exchange any information with each other. We have used a benchmark that scales approximately linearly (NQueens), one that does not scale well when the number of threads grows (Strassen), and two others that reach their saturation phases (Sort and Health). The input sets are chosen in such a way that the turnaround times of the programs range from a few seconds to a few tens of seconds. The aim is to show that the overhead of the proposed mapping technique is negligible, even for programs with small turnaround times.

The target platform is the TILEPro64, which runs Tile Linux that is based on the standard open-source Linux version 2.6.26. The C compiler used is the one provided in the Multicore Development Environment (MDE) 3.0 from Tilera Corporation, which is called Tile-cc and

is based on the GCC 4.4.3. The only change made to the BOTS 1.1.2 configuration file is the name of the compiler.

1. **Sort (untied):** Sorts a random permutation of n 32-bit numbers with a fast parallel sorting variation of the ordinary merge sort. First, it divides an array of elements in two halves, sorting each half recursively, and then merging the sorted halves with a parallel divide-and-conquer method rather than the conventional serial merge. Tasks are used for each split and merge. When the array is too small, a serial quick sort is used to increase the task granularity. We have used the default cut-off values (2048) when sorting an array of 50M integers. To avoid the overhead of quick sort, an insertion sort is used for small arrays of 20 elements.

2. **Health (manual-tied):** This program simulates the Columbian Health Care System. Each element in its multilevel lists represents a village with a list of potential patients and one hospital. The status of a patient in the hospital could be waiting, in assessment, in treatment, or waiting for reallocation. Each village is assigned to one task. The probabilities of getting sick, needing a convalescence treatment, or being reallocated to an upper level hospital are considered for the patients. At each time-step, all patients are simulated according to these probabilities. To avoid indeterminism in different levels of the simulation, one seed is used for each village. Therefore, all probabilities computed by a single task are identical across different executions and are independent of all other tasks. three different input sizes are available in the benchmark suite. We have used them in different scenarios. However, the performance scalability of the single program is presented using the medium-size input.

3. **Strassen (tied):** The Strassen algorithm employs a hierarchical decomposition of a matrix for multiplication of large dense matrices. Decomposition is performed by dividing each dimension of the matrix into two parts of equal size. For each decomposition a task is created. A matrix size of 2048×2048 is used for the purposes of this experiment.

4. **NQueens (manual-untied):** The NQueens benchmark computes all solutions of the n-queens problem, whose aim is to find a placement for *n* queens on an $n \times n$ chessboard such that none of the queens attack any other. It uses a backtracking search algorithm with pruning. A task is created for each step of the solution,and it has an almost linear speed-up.

4.5.3 Results of Multiprogramming using Information Sharing

For multiprogram workloads, we have considered three different scenarios to show how the XLL mapper can result in better performance. The error bars on the first two figures (4.5 and 4.6) show how deterministic the results are, while in the third figure (4.7) the error bars illustrate the minimum and maximum running time amongst the 10 identical programs.

First, we have to show why BLL, which is a simple Round-Robin mapping algorithm is inefficient. For this purpose, we have considered three Health programs, each of which with 32 threads (this is based on the best performance achieved for uniprogramming [23]). Two programs have large inputs and one has a small input. The programs enter the system with the interval of 6 seconds. We have previously discussed why Static mapper cannot handle multiprogramming scenarios. The inefficiency of the BLL is also evident from Figure 4.5.

The first scenario clearly shows that the XLL is the winning policy. The scenario is designed in such a way that the program with the small input data set finishes before the second large program enters the system. In the case of BLL, the threads of the first large program are mapped to the first 32 cores of the system. The threads of the small program are mapped to the last 31 cores plus the first core (there are 63 cores to use). Then the small program finishes and the second large program enters the system, but the BLL cannot use the recently freed cores. Instead, based on the Round-Robin algorithm, the threads of the second large program are mapped to the cores 2 to 33, while most of these cores (except one of them) are already busy serving the first large program.

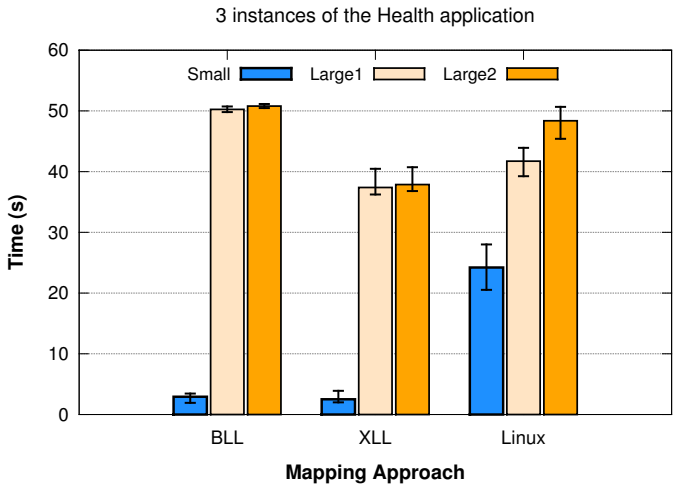


Figure 4.5: The first scenario: The inefficiency of the BLL

The second scenario is to run all four programs selected from the BOTS at the same time. Although they start at the same time, their thread creation time is different. This is due to the fact their initialisation phases and memory allocation times are different. Recall that thread creation happens whenever the execution reaches the *parallel* keyword in the OpenMP code.

According to their uniprogramming performance in [23], the Sort program (50M integers) is limited to 16 threads, the Health program (Medium input) is limited to 32 threads, and both Strassen (2048×2048) and NQueens (15×15) are run with 63 threads. The result is shown in Figure 4.6.

Once again, the XLL results in better performance. The turnaround times for all 4 programs are smaller when the XLL policy is applied.

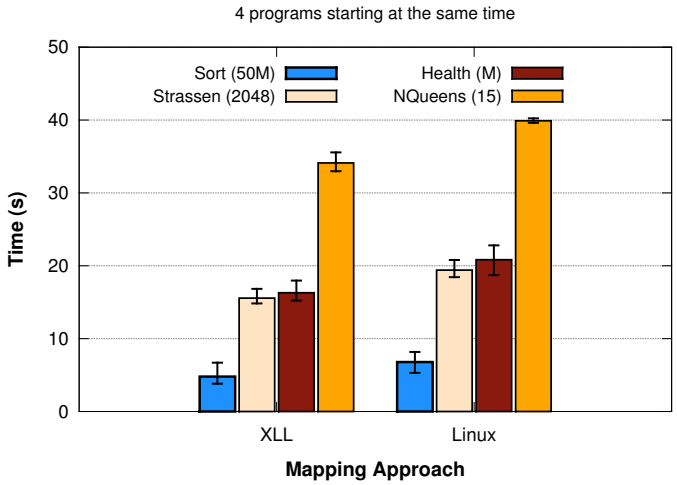


Figure 4.6: The second scenario: Running selected programs as the same time

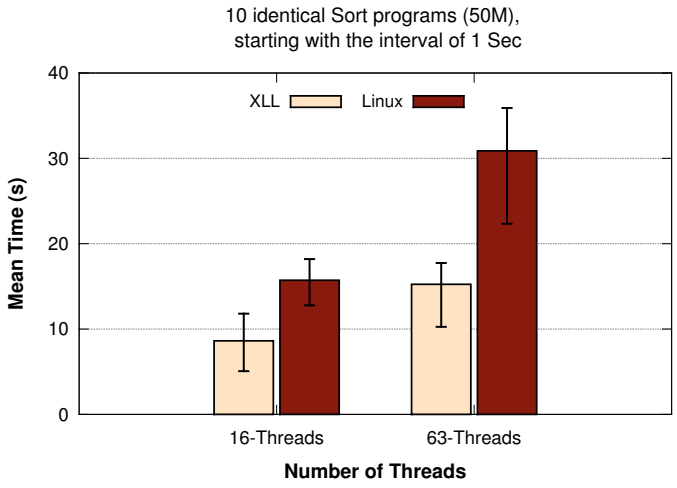


Figure 4.7: The third scenario: Running 10 identical instances of the Sort program

The third scenario gives a better insight on how the XLL mapper outperforms the Linux scheduler when the system is busy. For this scenario, we have used 10 identical instances of the Sort program arriving the system one after the other with the interval of 1 second. The result is depicted in Figure 4.7.

Figure 4.7 shows that the results with both policies are significantly better when each Sort program uses 16 threads rather than 63. It again verifies that increasing the number of threads

does not necessarily result in better performance. It is also evident how much our novel XLL mapping technique can outperform the native Linux scheduler in a multiprogramming environment.

4.6 Summary

We compared some of the performance aspects (in particular speedup, CPU balance, and the Total CPU Time) of three well-known parallel programming approaches, Intel OpenMP, Intel Cilk Plus and Intel TBB, on the Xeon Phi coprocessor. For that purpose, we used three different parallel benchmarks, Fibonacci, Merge Sort and Matrix Multiplication. Each benchmark has distinct characteristics which highlight some pros and cons of the studied approaches in different uniprogramming scenarios. Our multiprogramming scenario on the Xeon Phi was to run all three benchmarks together on the system and observe how the programming models react to this situation.

Based on the results obtained from the uniprogramming scenarios, particularly the Total CPU Time, we predicted that the Intel TBB approach would be more suited to a multiprogramming environment, and our experiment confirmed this. This is just the beginning, and these are our preliminary results. What we learned from these experiments is that keeping the overhead of runtime systems as low as possible in a general-purpose system is crucial, due to its direct impact on multiprogramming performance. The performance results also bold the importance of finding the optimal number of threads, as sometimes increasing the number of threads might lead to performance drop. In the next chapters, we address these problems by designing a new parallel execution model. In GPRM, a big overhead of task scheduling is shifted to the compile time. Moreover, instead of giving attention to both (number of) threads and tasks, the focus is only on the computational tasks.

In addition, since the way Linux deals with multithreaded multiprogramming is sub-optimal, we conclude that there is a need to share additional information between the applications present in the system in order to get better performance. The preliminary experiments on OpenMP applications running on the TILEPro64 were promising. We conclude that information sharing techniques, similar to the technique used in the Extended Lowest Load (XLL) strategy can improve the turnaround times in a multiprogrammed system. This led us to develop this idea inside the GPRM runtime system, denoted by *GPRM Global Sharing*.

Chapter 5

GPRM: The Glasgow Parallel Reduction Machine

McCool et al. [126] emphasis that none of the most well-known programming languages used today were inherently designed for parallel programming. They also list three desired features for the parallel programming models that intend to enable parallelism: I) Performance, II) Productivity and III) Portability.

It should be possible to predict good performance, tune it, and scale it to larger systems. Productivity is not only about expressiveness and composability, but also about maintainability. Supporting a range of targets and operating systems is another desirable property, known as portability.

The Glasgow Parallel Reduction Machine (GPRM) [26] has its origin in Gannet, a system for designing NoC-based SoCs [150] [151]. Gannet was originally designed as a distributed reconfigurable SoC architecture based on the "processing core as a service" paradigm, i.e. a network of services offered by software or hardware cores. The Gannet SoC performs tasks by executing functional task description programs on the Gannet machine. The Gannet system was based on message passing without shared memory support and the Gannet codebase was intended for a static system with different tasks at each node.

The main contribution of this work is to change the existing Gannet framework [150] to work on shared memory environments. This is mainly achieved by sending data pointers (rather than the data itself) between nodes. Gannet was a static system with different tasks at each node. I had to change the task distribution mechanism such that any task can run on any core. A major contribution is adding an efficient stealing mechanism that matches the execution model of the new framework. Adding support for loop-level parallelism as well as designing a high level coordination language -GPC- are other contributions. Moreover, new APIs, such as parallel loops and parallel lists as well as a new information sharing mechanism have been developed. None of these steps could be finalised without measuring their impact on

performance. Therefore, an incremental iterative approach is used for the development of the framework's components.

GPRM [1] borrows some fundamental concepts such as parallel reduction and intermediate functional representation from Gannet, but has a completely different focus and purpose. The implementation of tiles and their internal states in GPRM has been changed to provide a low-overhead task stealing mechanism for shared memory architectures with per-core caches. GPRM provides a task-based approach to manycore programming by structuring programs into *task code*, written as C++ classes, and *communication code*, written in GPC, a restricted subset of C++. The communication language (GPC) describes how the *task*s interact using a functional language semantics with parallel evaluation, but with a C++ syntactic veneer. What this means is that it is possible to compile task code and GPC communication code with a C++ compiler and get correct functionality, but without the parallelism.

We use a partial evaluator to transform a GPC code with the specified number of tasks (as its static data) into an Intermediate Representation (IR), called GPIR. GPIR (the *task description* code) is based on S-expressions that first appeared in the Lisp programming language [152], e.g. $(S_1\,(S_2\,10)\,20)$ represents a task S_1 taking two arguments, the first argument is the task S_2 which takes as argument the numeric constant 10, and the second argument is the numeric constant 20. GPIR is further compiled into lists of *bytecode*s, which the GPRM virtual machine (runtime system) executes with concurrent evaluation of function arguments. In other words, the GPRM virtual machine is a coarse-grained parallel reduction machine where the methods provided by the *task code* constitute the instructions.

The aim of GPRM is to abstract away all the details of threads, such that the users do not need to decide even about the number of them. This is an important issue, as finding the optimal number of threads has always been the most important concern. We contend that by efficient scheduling of *task*s, the number of tasks should be the only important factor affecting performance, because the *task*s are the actual computations, and the threads are only their substrates. The reason why GPRM is the most suitable model to verify our hypothesis is that it is completely task-centric. There is no extra overhead of thread migration, and the user deals only with *task*s. GPRM task scheduling combines compile-time (source to intermediate representation) and runtime (stealing) techniques to provide better performance. In other words, compile-time decisions form the initial distribution of tasks and the runtime system adjusts dynamically. As a result, some threads can be asleep during the execution, which means that the number of active threads are tuned automatically.

Probably, one similarity between GPRM and MPI is that they both implement a message passing environment. However, apart from the fact that the current version of GPRM only supports shared memory architectures and basically only sends pointers, the difference from

[1] https://www.github.com/wimvanderbauwhede/gprm

the programming perspective is that the users do not deal with the messages in GPRM. Reference packets are generated and dispatched, based on the information obtained from the bytecode. Once the processing results become ready, the callee sends back the results to the caller.

5.1 GPRM Architecture

At the start of the execution, GPRM creates a pool of POSIX threads equal in size to the number of available cores (or hardware threads) in the underlying hardware. Each thread runs a *tile*, which consists of a *task manager* (reduction engine) and a *task kernel* (Fig. 5.1). The *task kernel* is typically a self-contained entity offering a specific functionality to the system, and on its own is not aware of the rest of the system. The task kernel has run-to-completion semantics. The corresponding *task manager* provides an interface between the kernel and other *tile*s in the system.

Since threads in GPRM correspond to execution resources, for each processing core there is a thread with its own *task manager*. The GPRM system is conceptually built as a network of communicating sequential *tile*s that exchange packets to request computations and deliver results. In other words, the combined operations of all reduction engines (*task manager*s) in all threads results in the parallel reduction of the entire program.

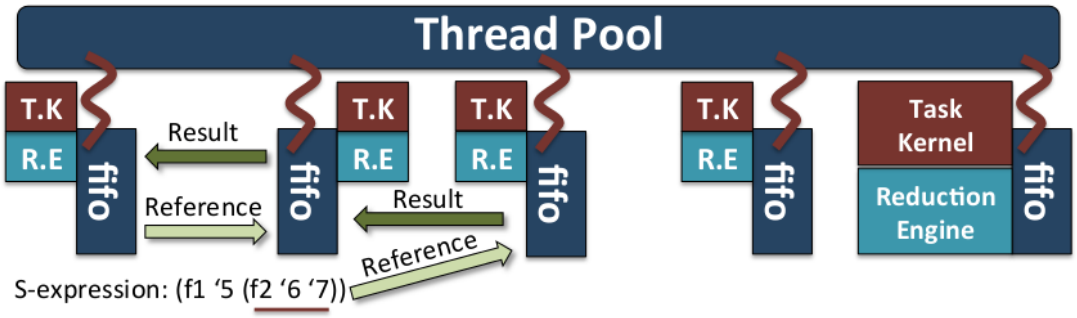


Figure 5.1: GPRM Architecture: Task Kernel (TK), Task Manager or Reduction Engine (RE) and FIFOs

At first glance, the GPRM model may seem static and intolerant to runtime changes. However as stated, we discuss how it can efficiently balance the load at runtime and thrive in dynamic environments. For load balancing, compile-time information about the task dependencies is combined with an efficient task-stealing mechanism. Moreover, by keeping track of the applications that reside simultaneously in the system, GPRM improves the performance of those applications markedly.

5.2 GPC: The GPRM Front-End Language

The language features will be discussed more in detail for the benchmarks that are actually using them in the following chapters. Also, the details of compiling a GPC source code to the Intermediate Representation (IR) source code is not the focus of this work. However, in this section we will express them concisely.

Glasgow Parallel C (GPC) is a subset of C++ in which statements are evaluated in parallel by default. The aim of GPC is to abstract the threading details away from programmers and to provide a consistent development framework for GPRM using C++. A GPC code is structured as classes in the GPRM namespace and tasks are defined as member functions of such classes.

5.2.1 GPC Language Features

GPC introduces a few keywords which do not exist in the C++ standard: the `seq` and `par` block qualifiers and the `par_for` construct that helps parallelise a loop. The `gprm unroll` pragma results in compile-time evaluation of the `for` loop it precedes. In C++, the `for` loop typically has a variable which is updated in every iteration. A GPC `for` loop after a `gprm unroll` pragma allows this, but in a restricted manner which disallows dependencies between different loop iterations and requires the loop boundaries to be constant.

The `seq` and `par` qualifiers are placed before a block of code to denote whether the block should be executed sequentially or in parallel. By default, all statements are evaluated in parallel, so the `par` keyword is optional.

```
1 GPRM::Kernel::Worker wkr;
2
3 void GPRM::ExampleTask::doWork( .... ) {
4 #pragma gprm unroll
5   for(int i=0;i<NT1;i+=1) {
6       wkr.doSomething( ... );
7   }
8 #pragma gprm unroll
9   for(int i=0;i<NT2;i+=1) {
10      wkr.doSomethingElse( ... );
11  }
12 }
```

Listing 5.1: A simple GPC program

A notable property of GPC is that it has serial semantics. That is, without the `seq` and `par` keywords, the same results will be generated when the GPC code is compiled with a C++

compatible compiler. GPC code compiled this way will run serially and not in parallel.

Listing 5.1 shows a very simple GPC program to illustrate the concepts. The C++ class *GPRM::Kernel::Worker* has two methods which are each called in `for` loops with different bounds. The result is that by calling the `doWork()` function member, NT1 *doSomething* tasks and NT2 *doSomethingElse* tasks will be automatically run in parallel.

5.2.2 GPC Compiler

The GPC compiler performs multiple passes and ultimately produces an intermediate representation source file if the compilation is successful. These stages are discussed in [153]: Parsing, Type Checking, Optimisation, and Code Generation. Parsing and Code Generation stages are straightforward. We briefly discuss the other two stages:

Type Checking

The type checking stage involves checking that all identifiers present in expressions have been predefined and enforcing the single assignment rule. Each expression's type is evaluated to verify that it has the correct type with regards to the context it is in.

Optimisations

During the final stages of compilation, the compiler performs optimisations on the GPC Abstract Syntax Tree (AST) before converting it to intermediate representation source code. Sparse Conditional Constant Propagation [154] optimisations are applied to evaluate expressions and eliminate branches. However, to perform these optimisations, the compiler needs to have a representation of the code in Single Static Assignment [155] form (SSA).

SSA requires that every variable is assigned exactly once and is defined before being used. At this point in compilation, the type checker has already checked that these conditions are enforced, so the compiler already contains a representation of the code in SSA form.

Furthermore, the compiler can eliminate every branch and generate an efficient intermediate representation which can be run on the GPRM runtime system.

5.3 GPRM Runtime System

As stated, GPC is compiled to lists of bytecodes representing the expressions to be reduced (i.e., S-expressions), which the GPRM runtime system executes with concurrent evaluation of function arguments (i.e., it is a parallel reduction machine). In the next section (Sect. 5.4),

we describe this flow under the name of "GPRM Model of Parallel Execution".

GPRM is completely task-centric. There is no extra overhead of thread migration, and the user deals only with *task*s. GPRM task scheduling combines compile-time (source to intermediate representation) and runtime (task stealing) techniques to provide better performance; consequently, some threads can be asleep during the execution.

Before going to sleep, threads can steal *task*s from each other when they become idle after finishing their jobs. This can balance the load if there are enough tasks in the pending queues. We will talk about the GPRM task stealing in Section 5.5.

Multiple GPRM runtime systems can adapt themselves to a multiprogramming environment. This will be covered in Section 5.6.

For the rest of this section, we first present the collaboration diagram for the major C++ classes used in the implementation of the GPRM runtime system, and then describe the C++ implementation for the core part of the runtime system for executing tasks: the *Tile*.

5.3.1 Implementation of the Runtime System

GPRM runtime system is implemented in C++. Although more and more support for C++11 has been gradually added to the whole framework, the current system is fully compatible with the C++98.

The collaboration diagram for the `SBA::Runtime` class, generated by Doxygen [156] is shown in Fig. 5.2.

The `Runtime` class has a `System` class as a member. In a multithreaded environment, the `Runtime::run()` member function calls `System::run_th()` (run threaded), which in turn calls `GatewayTile::run_th()`, creates as many `Tile`s as the number of logical cores in the underlying hardware, and then calls `Tile::run_th()` member functions.

In the `Tile::run_th()` member function, a `pthread` is created, and a start_routine named `run_tile_loop(void*)` is called. Inside this start_routine, the `Tile::run()` member function is called. In this member function, the tile waits for packets to arrive on its `Transceiver::RX_Packet_Fifo`. The class `RX_Packet_Fifo` has a member called `wait_for_packets()` for this purpose, which uses the `pthread_cond_wait()` in the background.

On receipt of a packet, the tile sets its `Tile::status` to true and a nested loop begins. The inner loop calls the `TaskManager::run()` with argument 0, which means that the tile is in the *normal execution mode*. After returning from that member function, if the `TaskManager::kernel_status` is "busy", then the `TaskKernel::run()` will be called to execute the actual computation. After that, or if the kernel status is not "busy",

the `Transceiver::run()` will be called, whose job is to transmit packets to other tiles. This is the end of the inner loop. In order to continue the loop, the `Tile::status`, which is the inner loop's condition variable should be true. In order for it to be true one of these three conditions should be met: if the `TaskManager::status` is true, or if either of the `Transceiver::TX_Packet_Fifo` or the `Transceiver::RX_Packet_Fifo` contains a packet [2].

Once the inner loop finishes, meaning that the `Tile::status` is no more equal to true, if task stealing is enabled (`STEAL` macro is defined), which is the case by default, the *stealing mode* begins by calling the `TaskManager::run()` with argument 1. Since the tile has finished its pre-assigned work, this is a one-time chance to look for more jobs from the ready queues of the other tiles. Same steps are repeated and the same three conditions will be `OR`ed at the end of the outer loop. If as a result of stealing, any of the those conditions becomes true, the whole loop starts over in the *normal execution mode*.

5.4 GPRM Model of Parallel Execution

Although this section and the following sections 5.5 and 5.6 describe the GPRM runtime system more in detail, we have separated them to highlight the important specifications of the runtime system. In this section, we cover the whole framework from writing a GPC code to the detailed explanation of task execution. The aim is to demonstrate how the whole GPRM works as a system.

The fundamental difference between GPRM and the competitive task-based models, such as OpenMP and TBB is that GPRM performs partial evaluation of programs with the numbers of tasks held constant. As a consequence, most of the techniques used in the GPRM runtime system, e.g. the stealing mechanism, are rather different from what readers might have in mind, and therefore need more explanation. In the fork-join model, at a *fork* point, new serial control flows are branched from an existing serial control flow. At a *join* point, these control flows can be (possibly selectively) synchronised and merged.

GPRM, on the other hand, analyses all the C++ classes and methods used in the GPC program at compile-time, and maps them to numeric constants. These constants are used in a wrapper function to match the operation from the GPIR code with the actual method call to be executed. This task-specific generated code is combined with the generic GPRM Runtime Library (RTL) and the code for the task classes. The build process is summarised in Fig. 5.3(a). Unlike the fork-join models, GPRM has no keywords for task creation or syn-

[2] A point to consider is that for the transmitter fifo `TX_Packet_Fifo` no lock is needed, as it is only accessed by the tile itself, but since all other tiles can send data to the receiver fifo `RX_Packet_Fifo`, a locking mechanism is required to access it, which has been implemented inside its member functions.

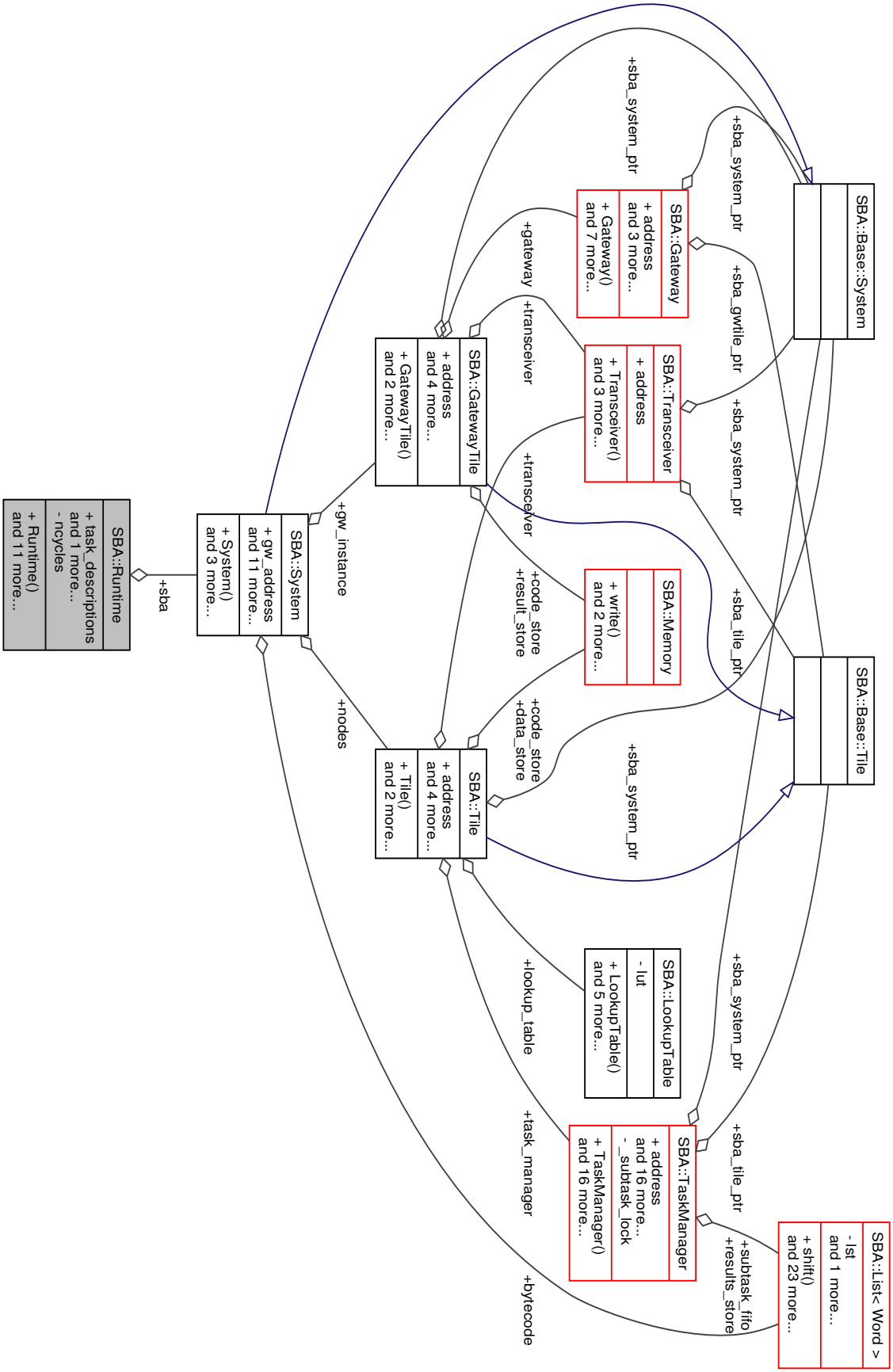


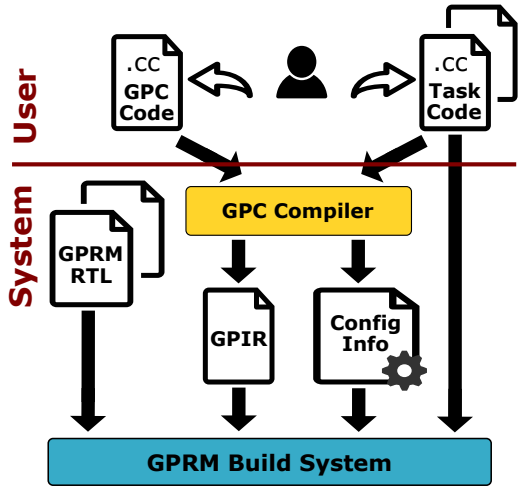
Figure 5.2: Collaboration diagram for `SBA::Runtime`

chronisation, as all the functions in the GPC code are evaluated in parallel, unless otherwise stated (i.e. if a `seq` pragma is used).

When launching GPRM, a pool of POSIX threads (typically equal to the number of available cores) is created before the execution of the actual program starts. The tasks are initially assigned to the threads, based on the indices provided by the GPC compiler into the GPIR code (the indices of the tasks in Fig. 5.3(c)). Because of the restrictions imposed on the GPC language, its abstract syntax is that of a functional language, and hence the indices can be automatically generated based on the dependencies in the call tree; tasks that depend on one another cannot run in parallel and therefore can have similar indices (can be mapped onto the same cores). The parallel execution is achieved by parallel evaluation of the bytecode, as follows:

- Computations are triggered by the arrival of a *reference packet*, a packet which contains a reference to a *task*, i.e. a piece of bytecode representing an S-expression, e.g. the first reference packet for the example in Fig. 5.3(b) will be a reference to the built-in task `seq`, which sequences the operations.
  - Each argument in this S-expression is either a reference or a constant, e.g. both arguments of the task t1 in Fig. 5.3(b) are constants and both arguments of the task t3 are references.
  - References are sent out to other tiles for computation; values are stored. 4 reference packets are shown in Fig. 5.3(c) (curved arrows). As another example, in Fig. 5.3(d), the task with address 1500 sends a reference through the tile's TX FIFO to another tile.
  - The leaf subtasks of the computational tree have either no arguments or constant values as arguments, so no references need to be sent.
- Once all arguments have been evaluated, the reduction engine passes the evaluated arguments of the S-expression to the *task kernel* which performs the actual computation. This is shown as a part of the tile structure in Fig. 5.3(d) with the *Call* and *Return* arrows.
- The result of the computation is returned to the caller, i.e. the sender of the reference packet. The result of the computation from the *task kernel* in Fig. 5.3(d) (the tile structure) is sent to the caller through the tile's TX FIFO.

There are other components in the tile structure in Fig. 5.3(d) that need to be explained. Each *tile* receives packets from others in its *task manager*'s RX FIFO. On the receipt of a *reference packet*, its corresponding *task record* is created. The newly created *task record* is stored in a

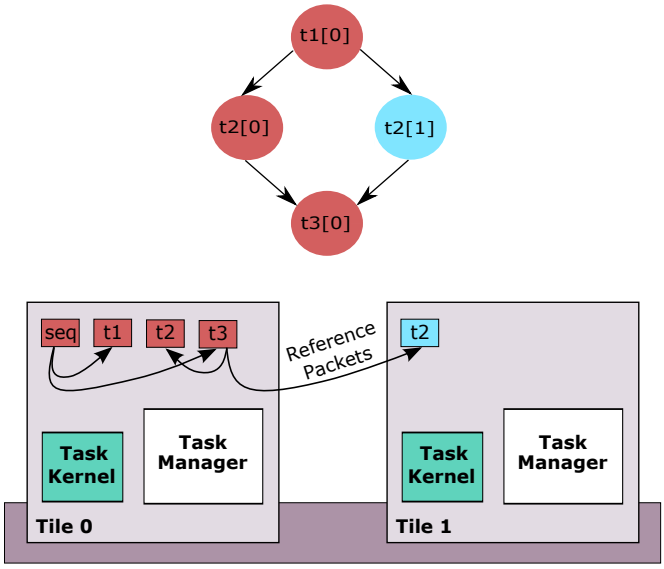


(a) User's view v.s. system's view

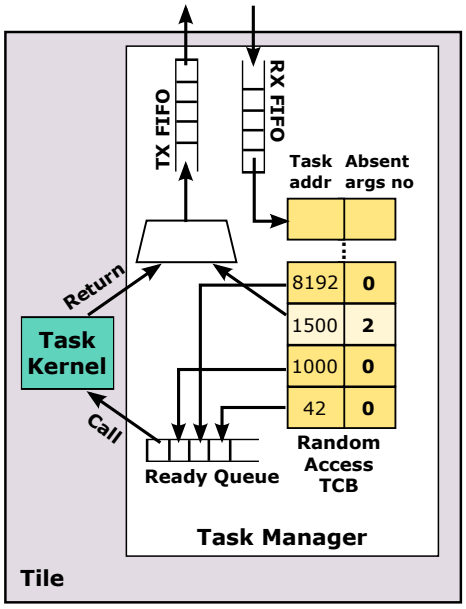
```
int x,y;
GPRM::Kernel::Body B;
#pragma gprm seq
{
  B.t1(x,y); //Init
  B.t3(B.t2(x), B.t2(y));
}
```

```
class Body {
 public:
   int t1(int,int);
   int t2(int);
   int t3(int,int);
};
```

(b) GPC - Task code(.h)



(c) Task dependencies - Mapping to the tiles



(d) Abstract view of tiles

Figure 5.3: (a) Users write GPC and Task codes. Task scheduling is handled by GPRM.
(b) Sample GPC code and the C++ header file for the Task code.
(c) Task dependencies for the example code in (b), and the allocation of the tasks on tiles. 4 reference packets requesting computations are shown.
(d) Internal structure of tiles. If all the arguments of a task in the TCB table become ready, it will be pushed into the *Ready Queue*. Otherwise, references will be sent to the others.

random-access table called *Task Control Block (TCB)* [3], which stores information about the tasks, particularly the number of their absent arguments. If the number of absent arguments is non-zero, references will be sent to other tiles in order to request computations, otherwise, as soon as all arguments of a *task* become ready, it will be pushed into the *ready queue* for computation. Therefore two cases would result in sending a packet to the other tiles: i) a reference packet from a task with absent arguments requesting computations, ii) a data packet containing the pointer to the result of the computation to the caller tile. Therefore, scheduling in our system, similar to other reduction machines (although it has coarser granularity at task level, rather than instruction level) is based on the need for data; this is known as the demand-driven model [157].

5.4.1 Communication Messages

In order to show how the information about the tasks are used by the GPRM runtime system, and also how the communication between the tiles is carried out, consider a user-defined task called `foo` [4]. A micro-program intends to compute `foo(31) + foo(45)`.

The generated task description code (GPIR) is shown in Fig. 5.4 as Test.td. A configuration file (Test.yml) will also be generated, which specifies the *base thread index* for each task [5] as well as the class and methods they are defined in. This information will be converted into constants in the final bytecode.

There is also an index for each task in the GPIR code, the *task index*. That index is the result of the compiler magic. If the program is a series of peer tasks, task index starts from 0 and is increased one by one for generated tasks (So, normally the indices for the `foo` tasks in this example should be 0 and 1, instead of 2000 and 4017); if the program is a tree of tasks, the task indices for the parents are equal to the index of one of their children, and so on.

Therefore, in order to calculate on which thread a task should be run, the *task index* will be added to the *base thread index*. This means that the `add` (+) task should be run on thread_id 89, the first `foo` task on thread_id 2002, and the second `foo` task on thread_id 4019. These numbers are highlighted in the GPRM packets in Fig. 5.4, as they are the addresses the packets use for routing.

Other important fields that are highlighted in Fig. 5.4, are the type of the packets. There are three code packets, which specify the information about the tasks themselves and their arguments, and there is a *reference packet* that triggers the computation. It can be observed that the `add` task has 2 arguments, which are references to other computations, but each `foo`

[3] TCB is called "subtask list" in the previous GPRM papers.

[4] `add` (+) is a built-in task with minimum overhead.

[5] Actually, in the current version of GPRM, all internal control tasks start from base thread index 1, and the base thread index for all other tasks starts from 2. Therefore, number 89 in the example is imaginary.

task has a constant number as its argument, which means that its data is ready.

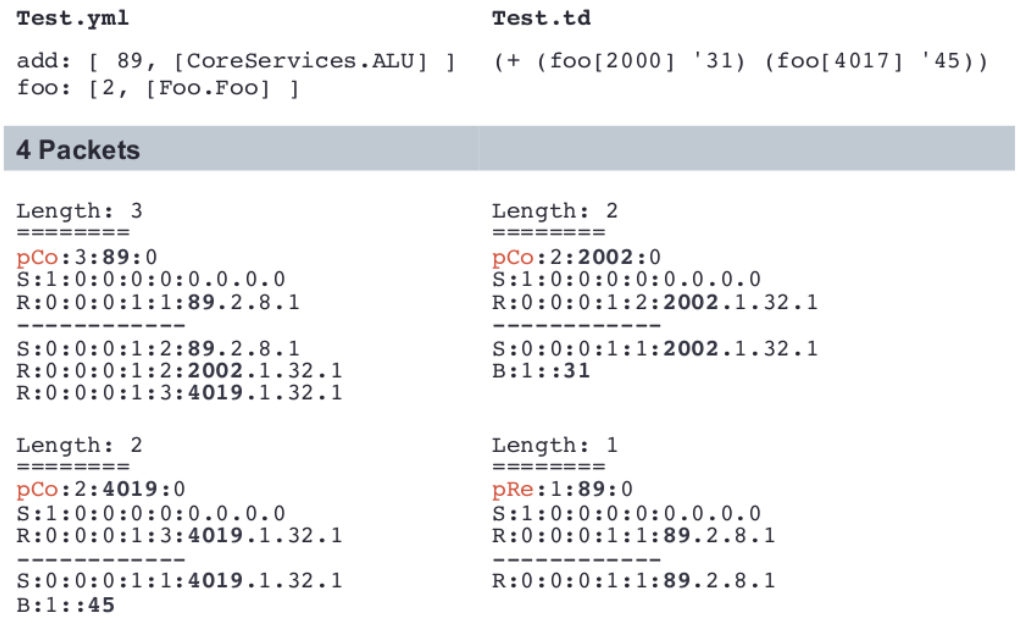


Figure 5.4: GPRM packets at the start of the sample program

The two important features of the GPRM runtime system of specific interest to this work are *Task Stealing* and *Global Sharing*. These features can be enabled via command-line switches when compiling the GPRM runtime system. It that sense they can be considered as runtime support features. We discuss them in the following sections 5.5 and 5.6.

5.5 GPRM Task Stealing

The term "stealing" is conventionally used for *Work-Stealing* inside the fork-join models. The techniques are different from what we call *Task Stealing* in GPRM. In GPRM parlance, *Task Stealing* is the process of stealing *task*s from the *ready* tasks queues of other threads. If enabled, it allows threads to steal *task*s from each other when they become idle after finishing their jobs. This can balance the load if there are enough tasks in the ready queues. We denote GPRM with stealing enabled as *GPRM-Steal* (or *GPRM-S*).

5.5.1 Comparison of the Stealing Strategies

At first sight, the stealing mechanism may seem quite similar to the classical work stealing approaches [139] [75] [144], but there are fundamental differences, due to the nature of our parallel programming model. In a fork-join model, when control flow forks, the master

thread executes one branch and the other branch can be stolen by other threads (thieves). Multiple branches can be generated as the program is executed. This classical approach needs double-ended queues (deques), such that the workers work at the back of their own deques, while thieves can steal from the front of the others' deques. *Steal child* (used by TBB) –the newly created child becomes available to the thieves– and *steal continuation* (used by Cilk Plus) –the continuation of the function that spawned new task becomes available to the thieves– are two variations of the conventional work-stealing approach [126].

In GPRM, C++ methods used in the GPC code are compiled into tasks. At compile-time, the compiler specifies the initial mapping between tasks and threads (even if the creation of a task is conditional, its initial host thread is specified). The parent tasks in the GPRM model are not the same as the parents in the Directed Acyclic Graph (DAG) as shown in Fig. 5.3(c): rather, the parent tasks are the ones that request computations from their children, hence will depend on their children, e.g. in the DAG in Fig. 5.3(c), t3 is the parent of the t2 tasks, following the order of the function calls: `B.t3(B.t2(x), B.t2(y));`.

```
1 void f(int i) {
2  if(i==0) sleep(2);
3  else sleep(1);
4 }
5 ...
6 /* Cilk Plus */
7 for(int i=0; i < N; ++i)
8      cilk_spawn f(i);
9 cilk_sync;
10
11 /* GPRM */
12 #pragma gprm unroll
13  for(int i=0; i < N; ++i)
14       f(i);
```

Listing 5.2: Micro-benchmark to illustrate the differences between the stealing techniques

With this background information, it is more clear what we mean by *task stealing*. Our stealing mechanism is about stealing the individual tasks, rather than the whole branch. In the conventional work-stealing approaches, the stolen branch would create more tasks during the execution of the program, and they would be executed by the thief (unless other workers become free and steal from that thief). The GPRM-specific task stealing mechanism is useful because all the tasks are initially allocated to threads (*tile*s). The stealing mechanism only tunes the initial allocation set by the compiler. Therefore, assuming that all the tasks are exactly the same and the number of them is a multiple of the number of the processing cores in the system, most probably no stealing occurs. In order to illustrate the differences between

the stealing techniques in details, consider the program in Listing 5.2 written in Cilk Plus and GPRM. Rewriting it with other approaches is straightforward.

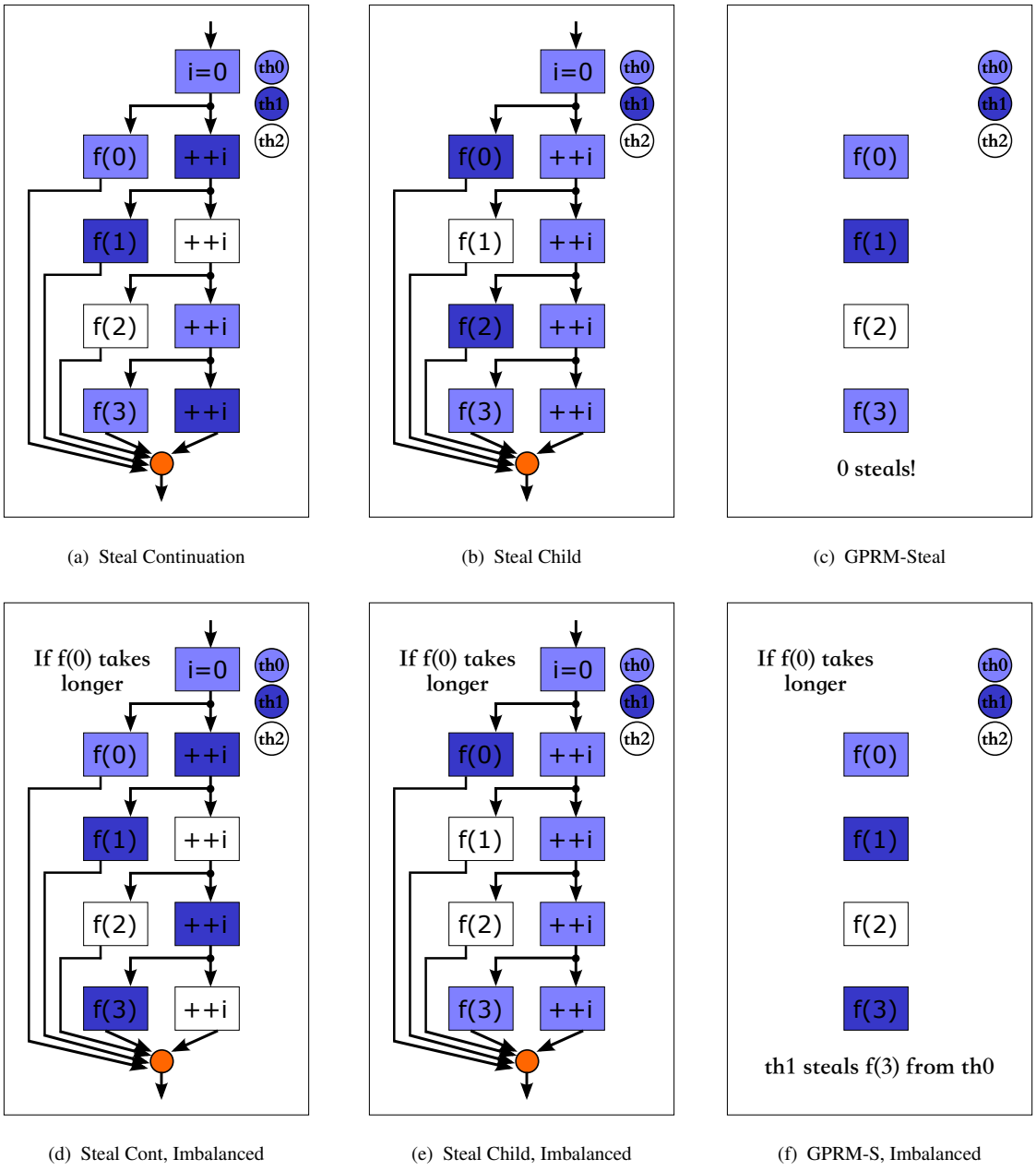
1. Suppose we have only 3 threads in the system and $N = 4$. For the first case, assume that calling the function `f(i)` with different `i`s results in the same runtime:

   - In the *steal continuation* technique shown in Fig. 5.5(a), `th0` (thread0) sets `i=0`, spawns `f(0)`, and immediately start executing `f(0)`, leaving the continuation of the loop available for stealing. `th1` –as an example– steals the continuation, updates `i`, and executes `f(1)`. `th2` could be the next thief that steals the further continuation and executes `f(2)`. Theoretically, `th0` finishes its work before the other threads, hence executes the next iteration and `f(3)`. the last iteration can be stolen by `th1`. Therefore, 3 steals[6] can be considered for this simple case.
   - For the *steal child* technique shown in Fig. 5.5(b), `th0` executes all iterations of the loop, spawns all `f(i)`s, and leaves them available to steal. However, since newly spawned tasks are put at the back of the deque and each worker thread takes the tasks from the back of its own deque (like TBB), therefore after all iterations, `th0` executes `f(3)`. `f(2)` can be stolen by `th1`. There are also 3 steals in this case.
   - Since the GPC compiler unrolls the task creation loop and assigns the tasks to the worker threads, `f(0)` to `f(2)` will be assigned to `th0` to `th2`, and `f(2)` will be assigned to `th0`. Theoretically, no stealing occurs, because if all threads finish their work at the same time, `th0` would execute its next assigned task before others enter their stealing phase and steal it.

2. For the second case, consider the definition of `f()` in Listing 5.2, where executing `f(0)` takes more time:

   - The number of steals for the *steal continuation* becomes 4, as `th1` and `th2` can steal more continuations before `th0` finishes its first job.
   - The number of steals can remain 3 for the *steal child* technique. `th1` steals the first child and `th2` the second. Assuming that `th2` has started executing `f(1)` an epsilon before `th0` reaches `f(3)`, `f(2)` becomes available for `th2` (note that `th1` is still busy executing `f(0)`).
   - In the *GPRM-Steal* technique in Fig. 5.5(f), assuming that `th1` has started its work an epsilon before `th2`, it can steal `f(3)` from the *ready* queue of the busy thread (*tile*), `th0`.

[6] We use the noun "steal" (OED "the act of stealing") rather than "theft"



(a) Steal Continuation (b) Steal Child (c) GPRM-Steal

(d) Steal Cont, Imbalanced (e) Steal Child, Imbalanced (f) GPRM-S, Imbalanced

Figure 5.5: (a) Steal Continuation (used by Cilk Plus), balanced load: 3 steals
(b) Steal Child (used by TBB), balanced load: 3 steals
(c) GPRM with stealing enabled, balanced load : 0 steals
(d) Steal Continuation, imbalanced load: 4 steals
(e) Steal Child, imbalanced load: 3 steals
(f) GPRM-Steal, imbalanced load: 1 steal

Figure 5.5 shows that what we mean by task stealing is actually the minimum number of steals required to balance the load. Even for tiny micro-benchmarks, the techniques as well as the number of steals are quite different.

5.5.2 Implementation of Task Stealing in GPRM

As described above, inside GPRM every thread runs a *tile* with its own *task manager*. Each task manager consists of multiple queues for different purposes, such as exchanging packets (`RX_fifo` and `TX_fifo`) or storing results. As stated, one of these queues is the `ready_fifo`. All tiles are initially in the *non-stealing state*, which means their task managers take tasks from their own `ready_fifo`. If `STEAL` is enabled, then after running the last task in its `ready_fifo`, the task manager searches for jobs in the `ready_fifo`s of other tiles (excluding the control tile). The tile remains in the *stealing state* and repeats this task-stealing process, unless there is no more job to steal after probing all `ready_fifo`s of all tiles once. Then it goes to the *sleeping state*, and waits for a *reference packet* to wake it up and start a new computation. This mechanism is shown in List. 5.3.

```
1 Task TaskManager::taskSteal() {
2  for(int j=tileAddr; j < tileAddr+NTILES; ++j) {
3      int i=(j % NTILES) + 1; // no stealing from tile 0
4      Tile& victim = *(system.tiles[i]);
5      if(victim.TaskManager.ready_fifo.size() != 0) {
6         Task stolen = victim.TaskManager.lockReadyQ(1);
7         // 1 means a thief is locking
8         if(stolen != EMPTY) { // EMPTY is an int number for null tasks
9            ... // registers it to the TCB of the thief
10           return stolen;
11        }
12     }
13 }    // else, loop continues
14 return EMPTY;
15 }
```

Listing 5.3: Task Stealing inside the task manager

The `stolen` task shown in List. 5.3 can be null (`EMPTY`), because we do not lock the queues only to check whether their sizes are greater than 0. This way, we do not interrupt the routine operations of other tiles. However, inside the member function called `lockReadyQ()`, we check if the size is still greater than 0 (or the owner has processed all of its tasks already). If the size of the ready queue is 0, the loop continues in order to find another victim.

The member function `lockReadyQ()` used in both Lists. 5.3 and 5.4 is responsible for locking the `ready_fifo` and returning the top element, if any. As shown in the code, it

gets an argument which specifies whether a thief has locked the FIFO or not. As a result, we can define different actions for the owners and thieves inside this member function. For instance, for research purposes, it is possible to define conditions for task stealing. We can set rules for thieves to steal only from the queues with larger sizes than X. However, for the experiments in this study we did not set any rules.

```
1 void TaskManager::coreControl(bool is_steal_state) {
2 // is_steal_state comes from the Tile class
3 #ifdef STEAL
4  if(is_steal_state) {
5     Task stolen_task = taskSteal(); // defined earlier
6     if(stolen_task != EMPTY) {
7        current_task = stolen_task;
8        status = ready; // the status of the task manager
9     }
10 }
11 else if (status == idle) {
12    Task next = lockReadyQ(0); // 0 means the owner tile is locking
13    if(next != EMPTY) {
14       current_task = next;
15       status = ready;
16    }
17 }
18 #else // STEAL is disabled
19 if(status == idle and ready_fifo.size() > 0) {
20    current_task = ready_fifo.front();
21    ready_fifo.popFront();
22    status = ready;
23 }
24 #endif
25 ...
26 }
```

Listing 5.4: Core Control inside the task manager

Another member function of the class `TaskManager` is called `coreControl`, which is the centre for main operations inside the task manager. As shown in List. 5.4, a macro called `STEAL` is defined to ensure that if task stealing is disabled, no one locks the `ready_fifo`. This way, we can measure the real overhead of stealing, compared to when tiles only take tasks from their own queues. Three cases are shown in this function:

1. If `STEAL` is defined, the tile is either in its *stealing state* or not. If it is, `taskSteal()` will be called, and if successful the `status` will be set to `ready` for processing.

2. The second case is when `STEAL` is defined, and the tile is in the *non-stealing state*. If

the task manager is idle, it will lock its own `ready_fifo` and take a task to process.

3. The third case is when the `STEAL` macro is not defined, and no stealing occurs in the system. Therefore, if the `status` is idle and the `ready_fifo` is not empty, its top element will be processed. There is no need to lock the `ready_fifo`s, as every tile only uses its own queue in this case.

5.6 GPRM Global Sharing

The *Global Sharing* feature in GPRM is intended to be used in dynamic environments, where multiple parallel workloads compete for the resources. The proposed method uses a globally shared data structure that keeps track of thread mapping information. This data structure can be implemented in a runtime system as in our work, or could be embedded in the OS kernel.

Every instance of GPRM (every application) maps this shared data structure to its own memory space, and uses it to share information with others. The information we share, although quite minimal, is crucial to achieve good performance. In GPRM, all sequential tasks run on a specific tile. Generally, sequential tasks are those responsible for initialisation, or the ones that have to run alone after a synchronisation point. In either case, they cannot be stolen, because they are the only ready-to-run tasks existing in the system, and except the tile they are attached to, all other tiles are in the *sleeping state*.

In order to avoid running the sequential parts of different workloads on the same core, the corresponding `tileAddr` is shared between all GPRM applications present in the system. Therefore, every newly arrived GPRM application maps its threads to cores such that its "sequential tile" [7] is pinned to the first available position that is not devoted to sequential tiles of other applications. All other threads of that application will be arranged in order. On the Xeon Phi, where four logical cores form one physical core, the target candidate will be the next physical core.

Although more information could be shared between concurrently present applications, it is important to keep the overhead low. Moreover, even if other information such as the number of active/asleep threads is shared, there is no clue as to whether they would remain the same or not. Thus, such information would have to be shared frequently. We will show that with the small amount of information that we share currently, a noticeable performance improvement can be obtained. Whether or not sharing more information results in better performance remains to be investigated.

[7] The "sequential tile" is responsible for the sequential tasks, but also contributes to the parallel execution, whenever required.

5.7 GPRM Productivity

With increasing complexity of parallel computing, it is crucial to consider high productivity as a key part of the new programming models. It is common to perform field experiments targeting programmers in order to measure productivity [158]. However, since GPRM is still a research framework, we did not have the chance to use similar techniques to measure its productivity. However, we briefly discuss some of its pros and cons compared to OpenMP.

Most of the task-based programming models provide the programmer with some keywords to express parallelism in an imperative language such as C/C++. Pure functional languages on the other hand, have no side effects and allow for safe execution of parallel computations, but compared to mainstream languages such as C++ and Java, none of them have found widespread adoption. Even if a manycore programming language would find wide adoption, it would in the short term obviously be impossible to rewrite the vast amount of single-core legacy code libraries, nor would it be productive.

Using a new model such as GPRM for parallel programming might seem difficult, even though its core is just a library-based extension to C++. At the first glance, an approach like OpenMP is a better option in regards to high productivity and ease of programming. OpenMP and most of similar approaches attempt to add parallelism to the existing sequential codes, while GPC code in GPRM considers parallelism inherently, and the sequential parts of the code should be marked as sequential.

The fact that one can create new tasks dynamically using a fork-join model like in OpenMP is a very useful feature. It provides more flexibility, but at the same time requires more attention (e.g. to avoid uncontrolled recursive task creation). Moreover, joining tasks and placing barriers at correct locations is not as simple as forking them.

We believe that both OpenMP and GPRM have their own complications. Writing OpenMP code is not easy for a newbie. The only straightforward part of parallelisation in OpenMP is the use of parallel loops. In other cases, users would experience difficulties in writing or understanding the OpenMP code. They also need to learn new concepts, such as the *variable capture*, i.e. by default local variables are captured by value (`firstprivate`) and global variables by reference (`shared`).

It is also not easy for a user with no parallel programming experience to write a C++ code with parallel functional semantics in GPC. But in GPRM, data is inherently local to functions, which are GPRM tasks. This helps to reduce the time spent debugging. GPRM facilitates modular design, which is key to improve productivity [159].

We will see that for parallel loops, using OpenMP is much easier than GPRM. In the current version of GPRM, `par_for` constructs should be wrapped by a task. As an option, users can use OpenMP parallel loops inside a GPRM code. However, performance-wise, using

GPRM parallel loops is preferred.

All in all, GPRM has been designed with the highest possible productivity in mind. Parameter tuning is very time-consuming; therefore we will see that it is easier to achieve good performance with GPRM, because there is no need to tune the number of threads.

5.8 Summary

We introduced GPRM, a parallel programming framework for future manycore processors. The aim of GPRM is to achieve high performance by proper assignment of tasks to threads at compile-time and efficient task stealing at runtime, with less user effort (increased productivity) compared to the state-of-the-art approaches. It is also worth mentioning that in terms of portability, cross compiling and running GPRM on the two hardware platforms was as simple as just changing the C++ compiler in the build system. GPRM is supported on most UNIX-based systems.

We believe that a good rule of thumb is to focus on the number of tasks, rather than the number of threads. To convey this message "pay more attention to tasks rather than threads", we have designed GPRM, which implements tasks using a C++-compatible functional task composition language. The GPRM runtime automatically deploys tasks on threads running on each processing core. Performance optimization becomes a simple matter of choosing a proper cutoff value for the number of tasks.

In this chapter, we have also shown how the *PCAM* design methodology [32] can be applied to solve parallel problems in the GPRM model. The *Partitioning* phase is where the programmer thinks about the design of the tasks, by considering a C++ function or an iteration of a loop as a GPRM task. *Communication* is defined in the GPC code and relates the tasks to each other. *Agglomeration* is accomplished by defining the number of tasks. It is also referred to as the cutoff value in our parlance. Finally, the *Mapping* is performed at compile-time using a task description file. Since threads are pinned to cores, mapping of tasks to threads is equivalent to the mapping of tasks to cores. A low-overhead efficient stealing mechanism is applied to balance the tasks at runtime. Also, the exact order of task execution is determined in a demand-driven fashion.

Chapter 6

Comparison of GPRM with Popular Parallel Programming Models

In this chapter, we add GPRM to the comparison between the parallel programming models. Our main objective is to investigate pros and cons of GPRM in comparison with the state-of-the-art models. Moreover, we include the TILEPro64 to compare GPRM with OpenMP more in detail. We have looked into the details of the benchmarks to understand the reasons behind the performance differences and to propose techniques for further improvement.

6.1 Uniprogramming Workloads

We illustrate the results of running the same "Base Benchmarks" as those in Section 4.2 on both the TILEPro64 and the Xeon Phi. Two GPRM approaches are added to the comparison: I) GPRM with no stealing at all and II) GPRM with stealing enabled, referred to as *GPRM-Steal* or *GPRM-S*. GPRM is only compared against OpenMP on the TILEPro64, but the results are shown in the same charts as the Xeon Phi results. This helps to compare the two platforms as well.

Threads and tasks are two completely different concepts. However, in a pure task-based model such as GPRM, parallelisation is only controlled by tasks. Setting the cutoff value less than the number of cores is fairly similar to having a smaller number of threads, because in this situation, tasks are assigned to a fraction of threads and the remaining threads are asleep and will remain so to the end of program. Larger cutoff values lead to the creation of more fine-grained tasks. Suppose that we want to parallelise a simple loop on N elements. In OpenMP, using 100 threads simply means that each thread gets N/100 of the workload (the simplest case with static scheduling). In GPRM for every program, the number of tasks should be specified. For a loop, each chunk corresponds to a task, therefore 100 tasks (i.e.

cutoff=100) on the Xeon Phi for instance, means that each thread gets N/100 of the workload. If we choose a cutoff=480, since there are 240 threads on the Xeon Phi, each thread gets 2 tasks.

Therefore, like before, two different comparisons are shown for every benchmark. The first comparison shows the speedup for varying numbers of threads. For GPRM, considering its thread pool and the above explanation, we only show the results with the default number of threads (as many as the number of cores), which is 63 on the TILEPro64 and 240 on the Xeon Phi.

The second comparison illustrates the speedup with the default number of threads and varying cutoffs. Choosing a small cutoff can restrict parallelism, but choosing a very large cutoff can saturate the system with a massive number of fine-grained tasks. The decision often depends on the input data set [149]. Usually, the cutoff value can be controlled by the user code. Leaving the decision to the runtime system has been proposed as an alternative in Adaptive Task Cutoff (ATC) [101], which is to aggregate tasks by not creating some of the user specified tasks and instead executing them sequentially.

6.1.1 Experimental Setup

All the benchmarks are implemented as C++ programs, and all speedup ratios are computed against the running time of the sequential code implemented in C++ (which means they are directly proportional to the absolute running times). We repeat all the experiments 20 times and use the average results.

On the TILEPro64 all 63 available cores are used. The Tilera's compiler, which is based on `GCC 4.4.3` is called `tile-g++` and is provided by MDE 3.0 from the Tilera Corporation. It The compiler flag `-O2` is specified. The TILEPro64 runs Tile Linux which is based on the standard open-source Linux version 2.6.36. The Intel compiler `icpc (ICC) 14.0.2` is used with `-O2 -mmic -no-offload` flags for compiling the programs for native execution on the Xeon Phi. Unlike the competitive approaches, for the GPRM framework there is no shared library to be copied to the Xeon Phi.

6.1.2 Fibonacci Benchmark

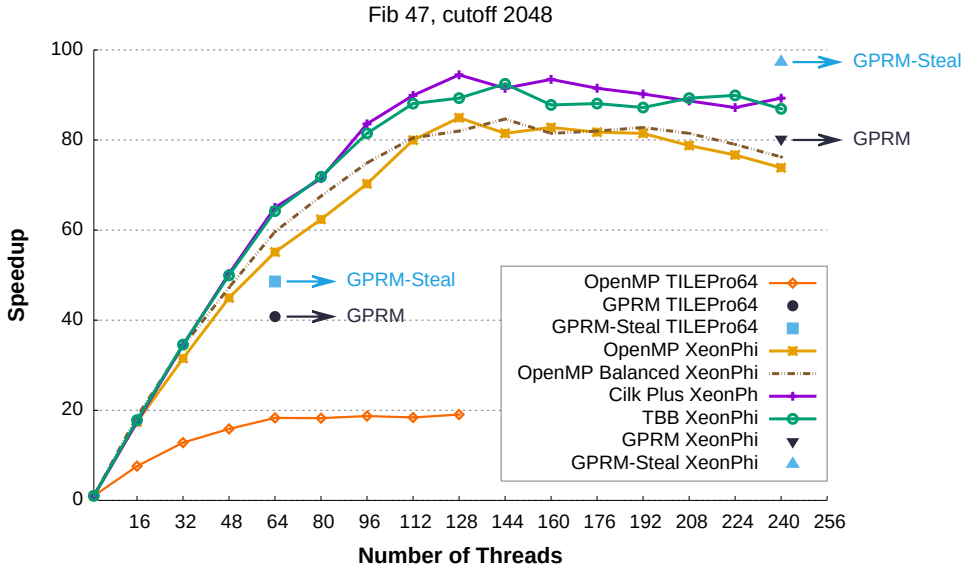
We have already discussed the Fibonacci benchmark in Section 4.2. The code snippet in Listing 6.1 shows the differences between GPRM and other models (e.g. OpenMP) in terms of task creation. We assume that `CUTOFF` is a constant (e.g. 2048). In order to start the computation, one can call `fib(n,1)` from the `main()`.

```
1 int seqFib(int n)
2 {
3   int x, y;
4   if (n < 2) return n;
5   x = fib_seq(n - 1);
6   y = fib_seq(n - 2);
7   return x + y;
8 }
9
10 /* OpenMP */
11 int fib(int n, int c) {
12   int x, y;
13   if (n<2) return n;
14   if ( c < CUTOFF ) {
15       #pragma omp task untied shared(x)
16       x = fib(n - 1, c * 2);
17
18       #pragma omp task untied shared(y)
19       y = fib(n - 2, c * 2);
20
21       #pragma omp taskwait
22   } else {
23       x = seqFib(n - 1);
24       y = seqFib(n - 2);
25   }
26   return x + y;
27 }
28
29 /* GPRM */
30 int fib(int n, int c) {
31   GPRM::Kernel::Fibonacci Fib;
32   if (c < CUTOFF) {
33     /* Add is a built-in task in GPRM for arithmetic addition */
34       return Add(fib(n - 1, c * 2), fib(n - 2, c * 2));
35   } else {
36     /* SeqFib() member-function is implemented the same as seqFib() */
37       return Fib.SeqFib(n);
38   }
39 }
```

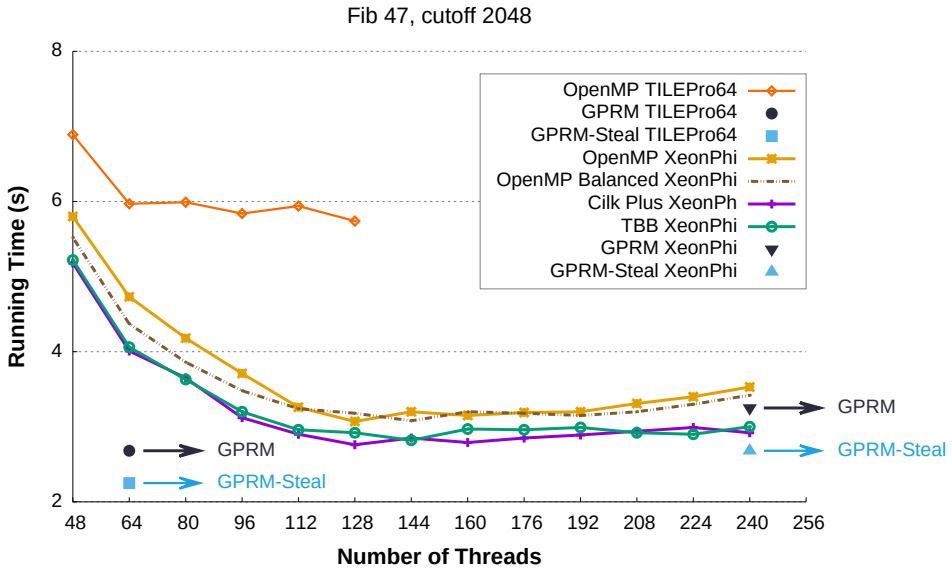
Listing 6.1: Task creation in OpenMP and GPRM for the Fibonacci benchmark

Figure 6.1 shows all the results taken from running this benchmark with different programming models on the TILEPro64 and the Xeon Phi. Figure 6.1(a) shows the speedup chart for $fib(47)$ with 2048 unbalanced tasks at the last level of the Fibonacci heap. In-

creasing the number of threads causes visible performance degradation for OpenMP. Setting `KMP_AFFINITY=balanced` results in a negligible improvement of the OpenMP performance. Cilk Plus and TBB show similar results. Cilk Plus can reach near the GPRM-Steal's performance with 128 threads, but there is no clue for the programmer on how to determine this number before running the experiment. GPRM-Steal has the best performance amongst all on the both platforms.



(a) Speedup



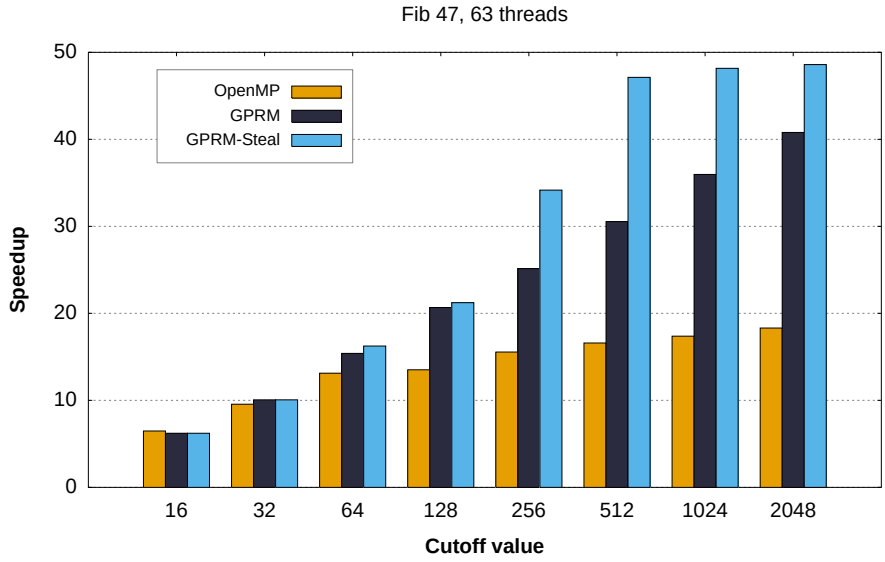
(b) Runtime

Figure 6.1: Parallel Fibonacci benchmark for the integer number 47.

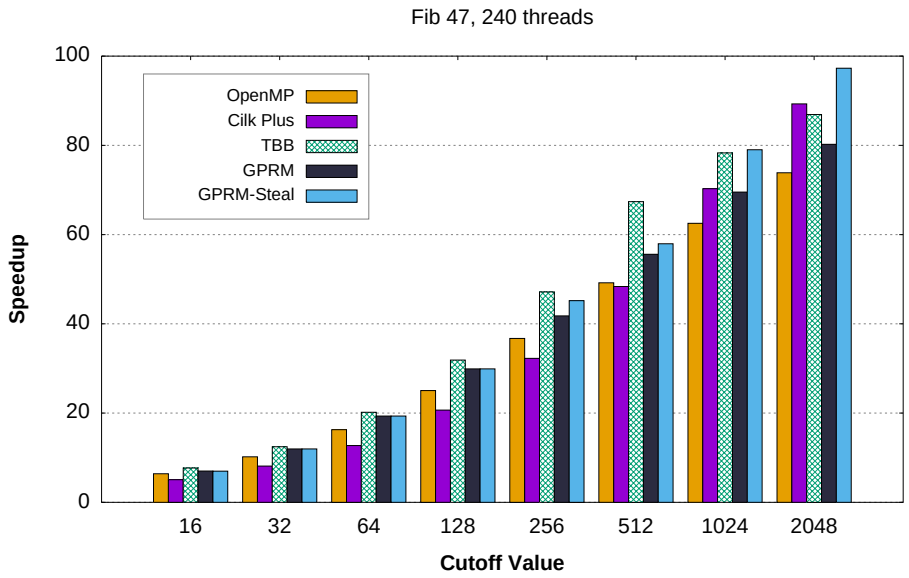
GPRM speedup of 49× on the TILEPro64 can be increased upto 98× on the Xeon Phi. Looking at Fig. 6.1(b), however, reveals that the best running time is obtained on the TILEPro64,

which implies that the serial runtime on the Xeon Phi is much slower than on the TILEPro64 for this benchmark.

We have increased the number of threads for OpenMP on the TILEPro64 to show the effect of oversubscription [160] as well.



(a) TILEPro64, different cutoffs



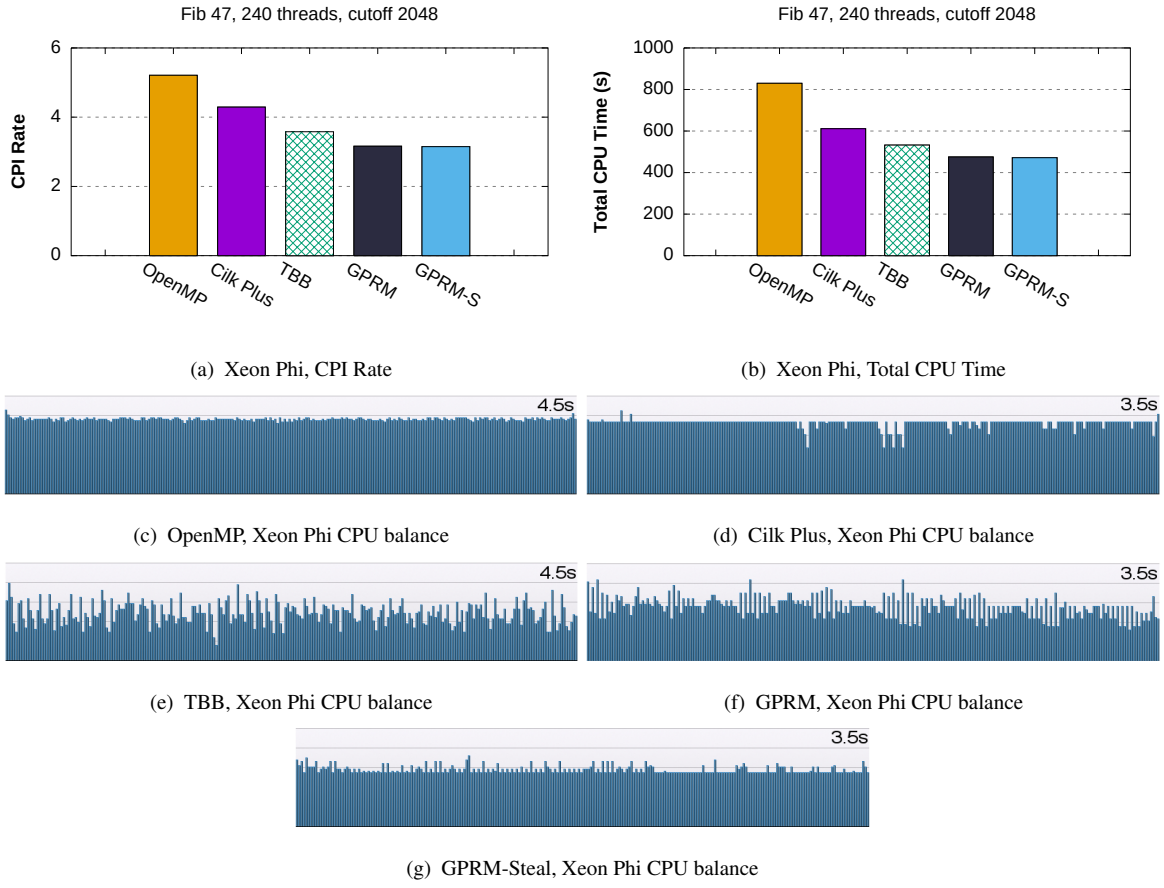
(b) Xeon Phi, different cutoffs

Figure 6.2: Parallel Fibonacci benchmark for the integer number 47.

Figures 6.2(a) and 6.2(b) show that choosing a proper cutoff is key to good performance. It is due to the fact that the tasks are unbalanced, and creating more tasks can result in a more even distribution of them. As we can see, it is easier to predict the proper number

of tasks, rather than finding a good combination of the number of threads and tasks in the system. It seems that because of the frequent thread migrations on the TILEPro64, thread scheduling with OpenMP has more overhead and GPRM outperforms OpenMP significantly on that system. GPRM-Steal improves the performance even more.

We re-emphasise that the benefits of using a proper task cutoff are not only limited to GPRM. The OpenMP speedup of only 1.1× for Fib 47 with 240 threads on the Xeon Phi –similar to the maximum speedup for Fib 38 in [98]– can be improved up to 75× by limiting the number of tasks.



(a) Xeon Phi, CPI Rate

(b) Xeon Phi, Total CPU Time

(c) OpenMP, Xeon Phi CPU balance

(d) Cilk Plus, Xeon Phi CPU balance

(e) TBB, Xeon Phi CPU balance

(f) GPRM, Xeon Phi CPU balance

(g) GPRM-Steal, Xeon Phi CPU balance

Figure 6.3: Parallel Fibonacci benchmark for the integer number 47.

Performance Metrics

The *Clockticks per Instructions Retired* (CPI) event ratio, also known as *Cycles per Instructions*, which is a lower-is-better metric, is one of the basic performance metrics for hardware event-based sampling collection.

Total CPU Time is another lower better metric that shows the total CPU times consumed for running an application[1]. Both of these metrics for the Xeon Phi platform are obtained

[1]Note the log scale on the y-axis of the (f) charts.

using Intel VTune Amplifier XE 2013 performance analyser [141]. Intel TBB and both of the GPRM approaches have a better CPI Rate, while having a considerably better Total CPU Time compared to the other approaches. One reason is that in a model like GPRM, threads go to sleep immediately after finishing their jobs, while e.g. in the Intel OpenMP, they spin-wait for 200ms before going to sleep [161]. Although sometimes in solo execution of the programs, these extra CPU cycles (and generally the overhead of the runtime libraries) have negligible influence on the running time (wall time), they affect other programs under multiprogrammed execution considerably [146] [162].

Figures 6.3(c) to 6.3(g) are screenshots taken from the VTune Amplifier when running Fib 47 with cutoff 2048 natively on the Xeon Phi. The x-axis shows the logical cores of the Xeon Phi (240 cores), and the y-axis is the CPU time, from 0 to the maximum number specified in seconds[2]. The effect of GPRM task stealing is evident by comparing Figs. 6.3(f) and 6.3(g)

Power Consumption

It might be argued that the performance achieved by Cilk Plus with smaller numbers of threads is close to GPRM-Steal with 240 threads, meaning that similar performance can be achieved with half the number of threads, which would be beneficial for power/energy consumption reasons. Using the minimum number of cores to achieve the desired performance can be very important.

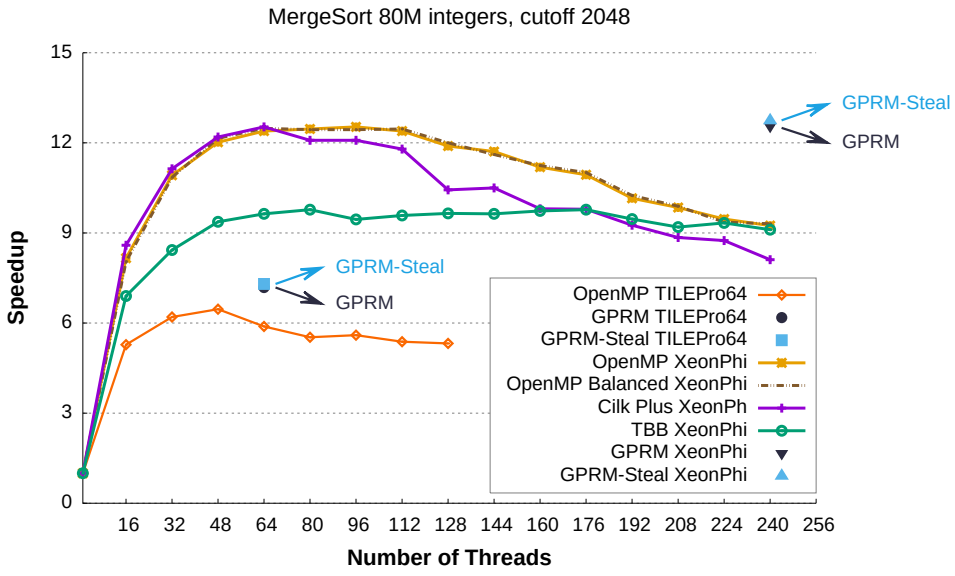
This would be a valid criticism if GPRM threads were making the cores busy by spin-waiting, which is not the case. Instead, the GPRM threads go to sleep if they have no work to do, and hence do not consume CPU time. In order to corroborate this claim, we measured the average power consumption of the best result achieved by each model; all measurements fall in the range of 130-135 Watt.

6.1.3 MergeSort Benchmark

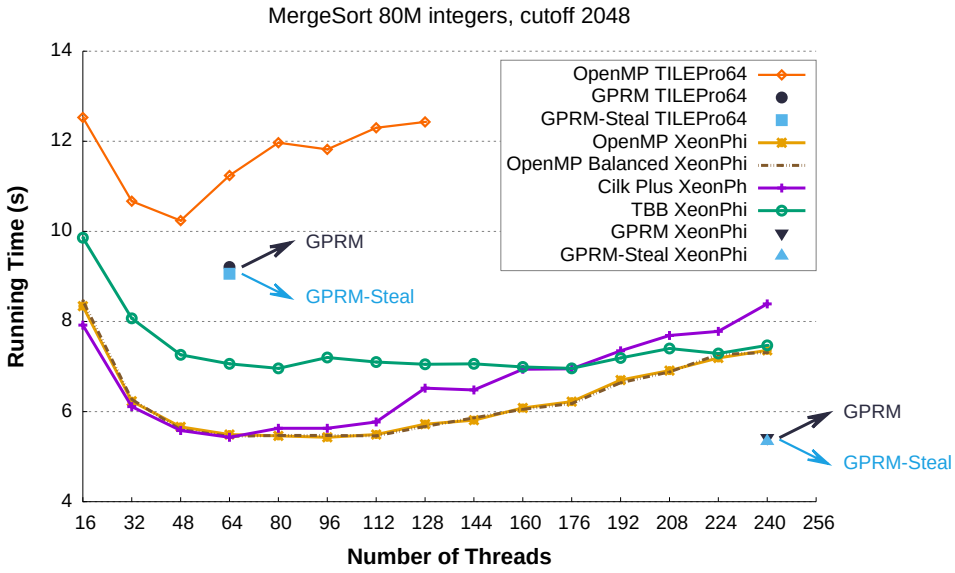
In the Fibonacci benchmark, the parent tasks[3] were lightweight integer additions. But for the MergeSort benchmark, the parent tasks are heavyweight merge operations. Moreover, the children tasks at the leaves –the chunks that need to be sorted sequentially– are almost equal in size. These features make this benchmark distinct from the previous one.

[2]For all experiments, results from the benchmark's kernel are considered in the figures (a) to (d), while in the other results taken from the VTune Amplifier, all information from the start of the application, including its initial phase and the CPU time consumed by the shared libraries is taken into account.

[3]See the GPRM definition of parent tasks in Section 5.5.1



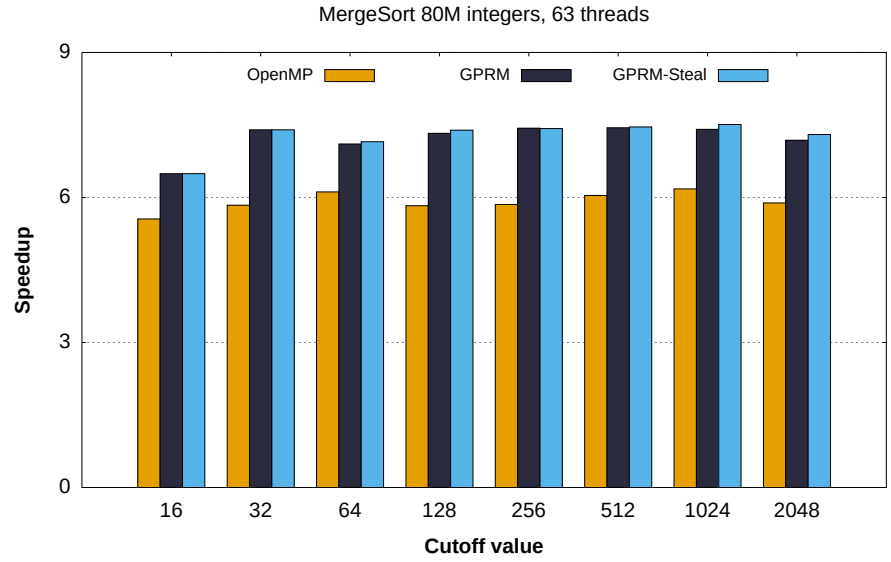
(a) Speedup



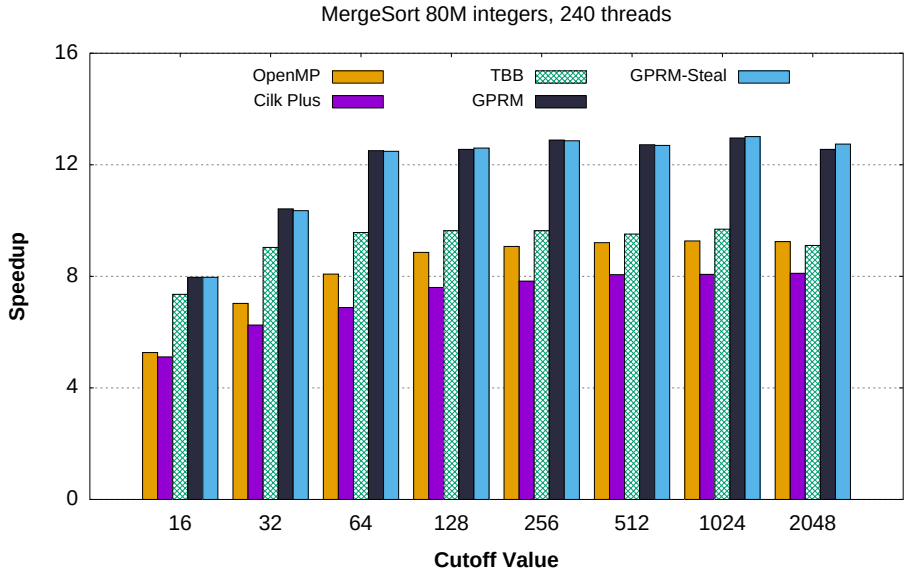
(b) Runtime

Figure 6.4: Parallel MergeSort benchmark for an array of 80 million integers.

As shown in the Fig. 6.4(a), this memory-intensive benchmark does not scale well, as has already been recognised by other authors [3] [94]. However, the larger caches of the Xeon Phi, compared to those of the TILEPro64, can result in better runtime performance. The GPRM approaches have significantly better performance with the default number of threads. On the Xeon Phi, both OpenMP and Cilk Plus perform well with smaller numbers of threads, but their performance drops as the number of threads increases. For the OpenMP and Cilk Plus approaches, more than 75% of the CPU cycles are consumed by their runtime libraries [24]. Using the `balanced` thread affinity for OpenMP has no noticeable effect.



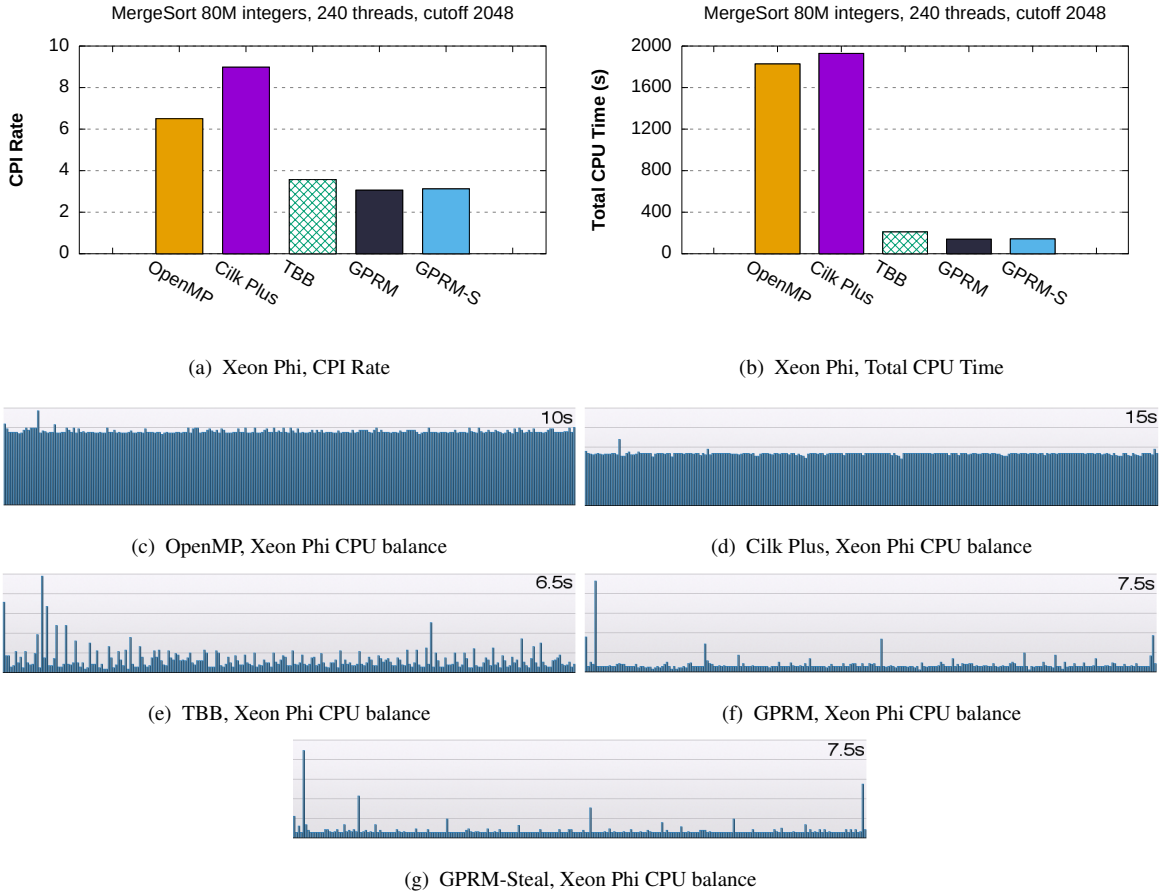
(a) TILEPro64, different cutoffs



(b) Xeon Phi, different cutoffs

Figure 6.5: Parallel MergeSort benchmark for an array of 80 million integers.

Since the child tasks are almost equal in size, cutoff values greater than 64 are enough to reach near-maximal performance with 240 threads, as shown in Fig. 6.4(b). Larger cutoffs, though, do not cause a notable slow down (as long as one does not create a massive number of tasks). Since the amount of work carried out for each task is fairly equal, there is no noticeable difference between the GPRM and GPRM-Steal approaches. It is evident from the speedup charts how regular task-based applications similar to this reduction example are well suited to the GPRM model.



(a) Xeon Phi, CPI Rate

(b) Xeon Phi, Total CPU Time

(c) OpenMP, Xeon Phi CPU balance

(d) Cilk Plus, Xeon Phi CPU balance

(e) TBB, Xeon Phi CPU balance

(f) GPRM, Xeon Phi CPU balance

(g) GPRM-Steal, Xeon Phi CPU balance

Figure 6.6: Parallel MergeSort benchmark for an array of 80 million integers.

```
1 /* TBB */
2 class Sort: public task {
3 public:
4  int* A; int* tmp; int size; int cutoff;
5  Sort(int* _A, int* _tmp, int _size, int _cutoff):
6   A(_A), tmp(_tmp), size(_size), cutoff(_cutoff){}
7  task* execute() {
8   if (cutoff==1) {
9     SeqSort(A, tmp, size);
10  } else {
11    Sort& a = *new(allocate_child()) Sort(A, tmpA, half, cutoff/2);
12    Sort& b = *new(allocate_child()) Sort(B, tmpB, size-half,cutoff/2);
13    set_ref_count(3); spawn(a); spawn_and_wait_for_all(b);
14    Merge& c = *new(allocate_child()) Merge(A, B, tmp, size);
15    set_ref_count(2); spawn_and_wait_for_all(c);
16  }
17  return NULL;}
18 };
```

Listing 6.2: Task creation in TBB for the MergeSort benchmark

```
1 // The other three approaches use the function Sort
2 void Sort(int* A, int* tmp, int size, int cutoff) {
3  int half = size/2;
4  int* tmpA = tmp;
5  int* B = A + half;
6  int* tmpB = tmpA + half;
7
8 /* OpenMP */
9  if (cutoff==1) {
10    SeqSort(A, tmp, size);
11  } else {
12   #pragma omp task
13    Sort(A, tmpA, half, cutoff/2);
14   #pragma omp task
15    Sort(B, tmpB, size-half, cutoff/2);
16   #pragma omp taskwait
17   #pragma omp task
18    Merge(A, B, tmp, size);
19   #pragma omp taskwait
20  }
21
22 /* Cilk Plus */
23  if (cutoff==1) {
24    SeqSort(A, tmp, size);
25  } else {
26    _Cilk_spawn Sort(A, tmpA, half, cutoff/2);
27    _Cilk_spawn Sort(B, tmpB, size-half,cutoff/2);
28    _Cilk_sync;
29    _Cilk_spawn Merge(A, B, tmp, size);
30    _Cilk_sync;
31  }
32
33 /* GPRM */
34  GPRM::Kernel::MergeSort MS;
35  if (cutoff==1) {
36    MS.SeqSort(A, tmp, size);
37  } else {
38    MS.Merge(A, B, tmp, size,
39       Sort(2*i, A, tmpA, half, cutoff/2),
40       Sort(2*i+1, B, tmpB, size-half, cutoff/2));
41  }
42 }
```

Listing 6.3: Task creation in OpenMP, Cilk and GPRM for the MergeSort benchmark

For this benchmark there is a significant difference between the CPI Rate and Total CPU

Time of TBB and GPRM on one side and OpenMP and Cilk Plus on the other side (Fig. 6.6). But these figures do not imply a poor behaviour of the TBB and GPRM runtime systems. Since all merges in a branch of the task tree can run on the same core, one should not expect to see a balanced distribution, and the balanced look in the cases of Cilk Plus and OpenMP, as discussed in Section 4.2.5, is mostly because of the wasted CPU time by their runtime libraries [24].

In order to show how to specify parallelism in each model, the related snippets of the source codes are listed in Listings 6.2 and 6.3. In TBB, class `task` is the base class for tasks. Therefore, in order to define the `Sort` task, we need to derive from `task`. For the other three approaches, we can define a simple Sort function and call it recursively.

6.1.4 MatMul Benchmark

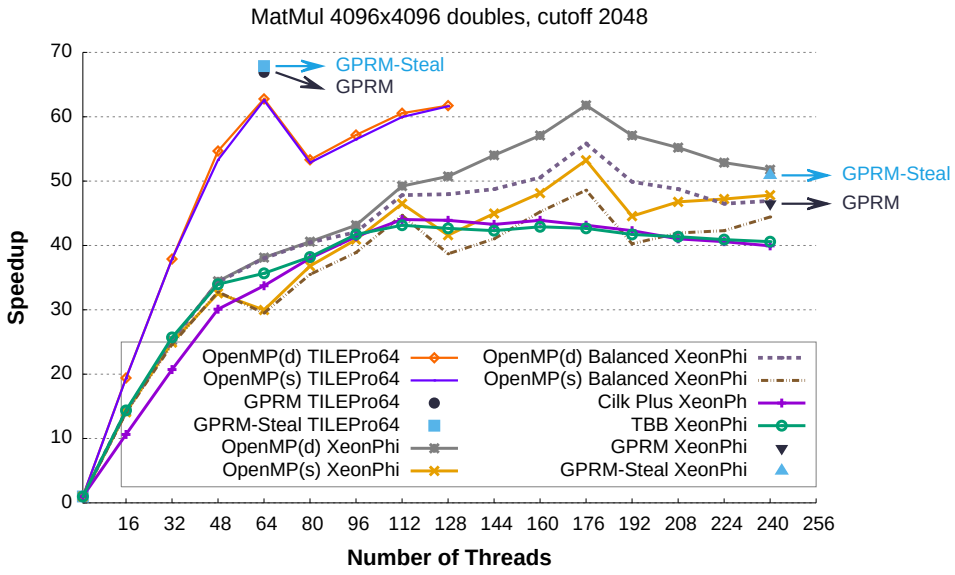
GPRM uses its `par_cont_for` (partial continuous for) worksharing construct, which is a task-based approach to this problem, and distributes the chunks based on their indices amongst the working threads. If the cutoff value is assumed as the number of *task*s in GPRM, the chunk size will be N/cutoff. The implementation of the parallel loops in GPRM will be described in Chapter 7.

```
1 /* OpenMP */
2 #pragma omp for schedule(dynamic, N/cutoff)
3
4 /* Cilk Plus */
5 #pragma cilk grainsize = N/cutoff
6
7 /* TBB */
8 parallel_for(blocked_range<size_t>(0, N, N/cutoff),Body(a,b,c),
      simple_partitioner());
9
10 /* GPRM */
11 par_cont_for(0, N, ind, cutoff, this, &Foo::bar);
```

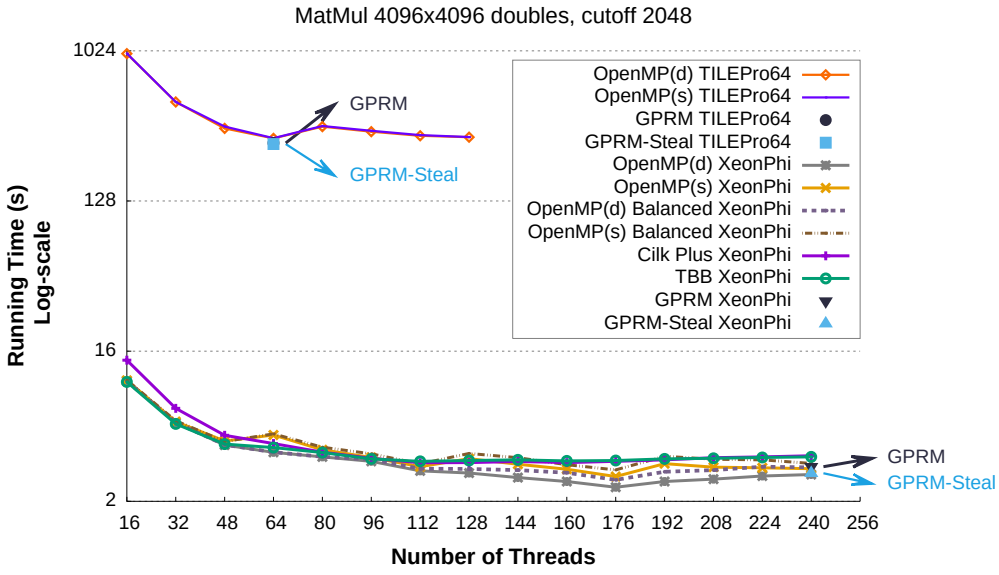
Listing 6.4: Defining the number of chunks (or the chunk size) in different implementations of the MatMul benchmark

The TILEPro64 is a 32-bit architecture without any FPU (Floating Point Unit). There are no vector registers or instructions on this architecture (instead it uses a 32-bit three-way issue scalar VLIW engine), and the size of caches are smaller than those of the Xeon Phi. Therefore, we should expect a huge difference in the performance in this case. In order to achieve automatic vectorization on the Xeon Phi, the Intel TBB and OpenMP codes have to be compiled with the `-ansi-alias` flag.

The `schedule` clause used with OpenMP `for` specifies how iterations of the associated loops are divided (statically or dynamically) into contiguous chunks, and how these chunks are distributed amongst threads of the team. For the MatMul benchmark, we have included both of these OpenMP approaches in the comparison. It is important to note that the dynamic scheduling on the Xeon Phi with cutoff 2048 can improve the performance of OpenMP from 43× for the default case (with no `schedule` clause) to 52×. After these considerations, we are ready to run the MatMul benchmark and compare the platforms as well as the programming models in a data parallel scenario. It is worth noting that with both GPRM approaches we have observed a superlinear speedup on the TILEPro64.



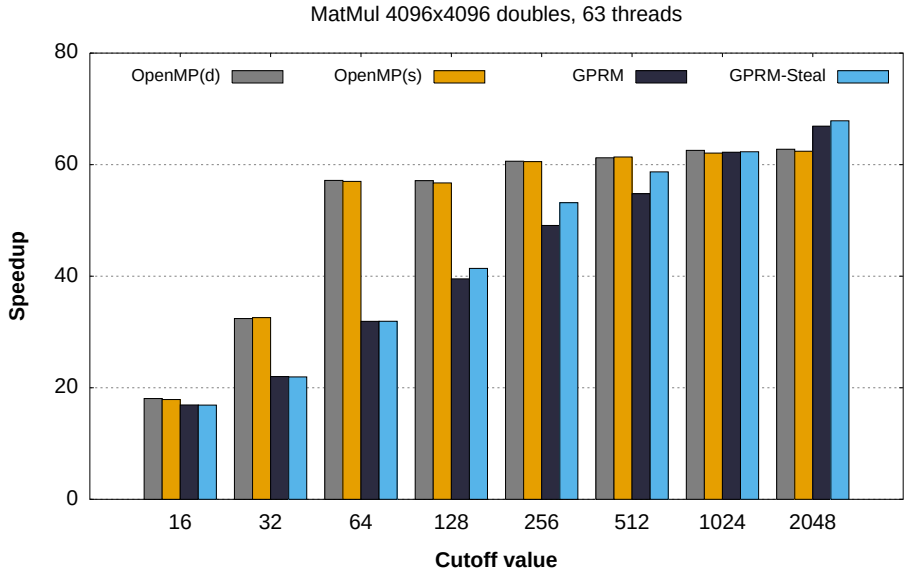
(a) Speedup



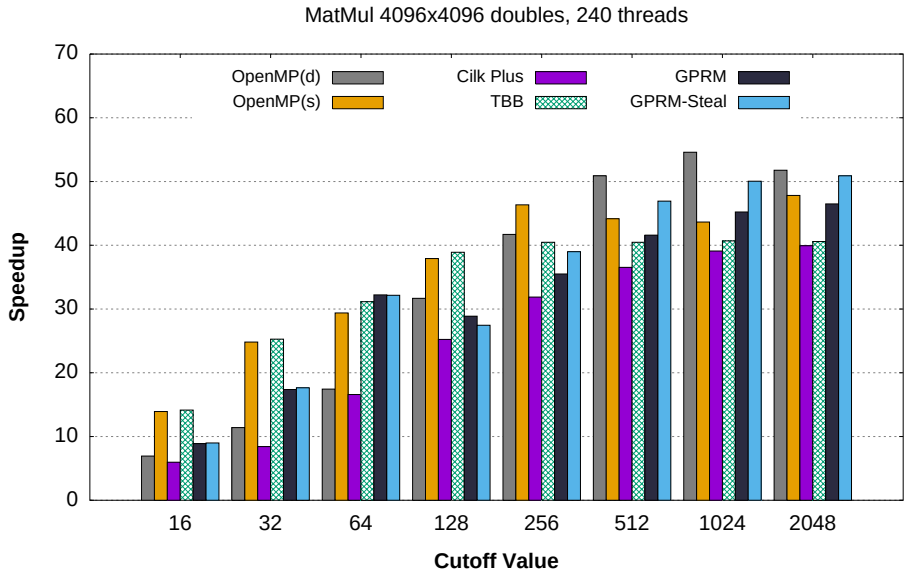
(b) Runtime

Figure 6.7: Parallel MatMul benchmark on a 4096×4096 matrix of double numbers.

Figure 6.7(a) shows that Intel OpenMP with dynamic scheduling has the best scaling amongst all on the Xeon Phi, and both GPRM approaches scale better than TBB and Cilk Plus. On the TILEPro64, the GPRM approaches with the superlinear speedup have better scaling than OpenMP. However, as illustrated in Fig. 6.7(b), there is an enormous difference between the running time on the TILEPro64 and the Xeon Phi[4]. The Xeon Phi is a vector processing machine and can distinguish itself from the TILEPro64 in scenarios like this.



(a) TILEPro64, different cutoffs

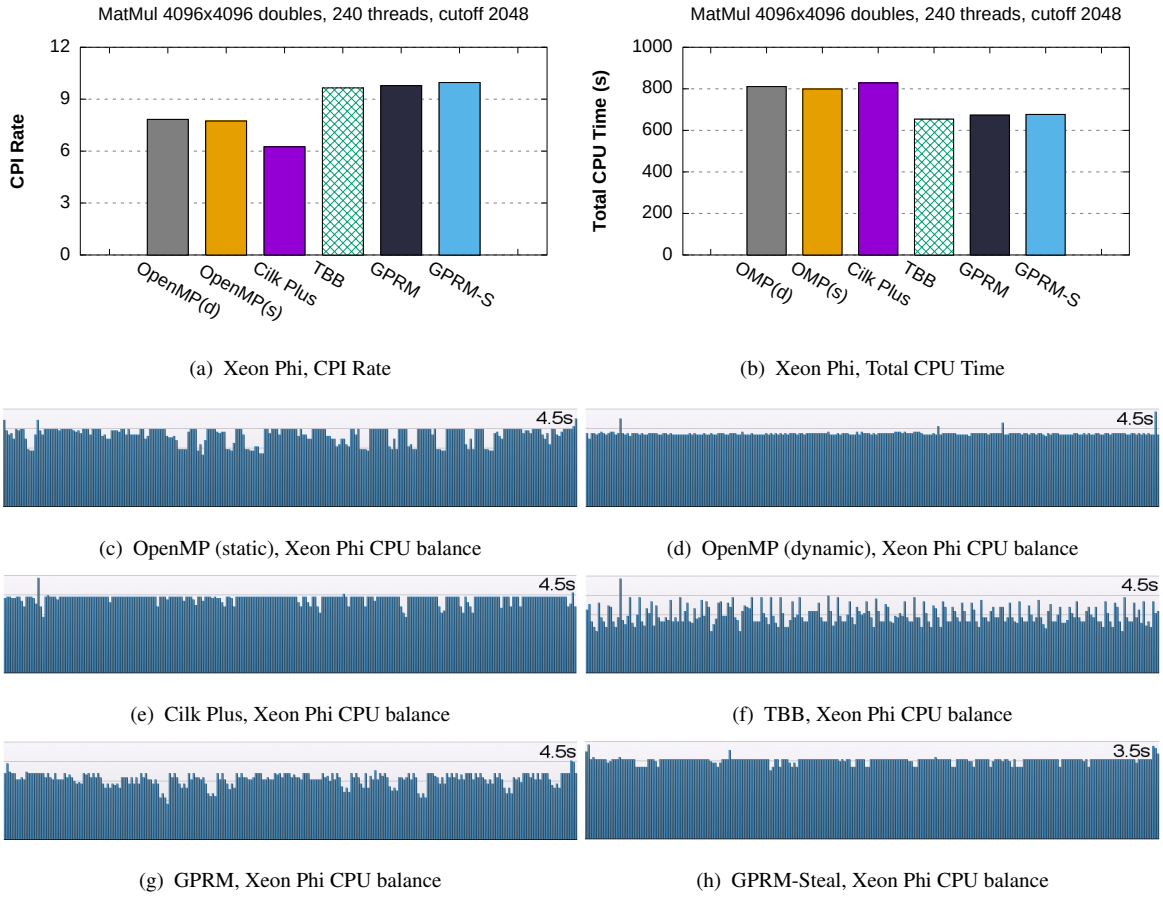


(b) Xeon Phi, different cutoffs,

Figure 6.8: Parallel MatMul benchmark on a 4096×4096 matrix of double numbers.

[4]Note the log scale on the y-axis of Fig. 6.7(b). The best result on the Xeon Phi is approximately 106× faster than on the TILEPro64.

Here, all the tasks are the same, having fairly the same size. To be precise about the results in Fig. 6.8(a), consider that we have 63 threads for GPRM on the TILEPro64 (as many as the number of available cores), but 64 threads for OpenMP. If we wanted to get better results for smaller cutoffs, we had to choose cutoff 63 for GPRM but in order to keep up with the previous experiments, we have used powers of two. 64 threads of OpenMP are time sliced over 63 cores, which results in a good speedup for cutoff 64. However, in the case of GPRM, every thread gets 1 chunk, except one of them which gets 2 chunks. That is why we see a big difference for cutoff 64 between the approaches on the TILEPro64. Instead of choosing better cutoffs for this case (63,126,...), we have increased the cutoff value, and thus creating more tasks has balanced the load distribution. The same reasoning applies to the Xeon Phi. Firstly, 4096 is not a factor of 240 (number of threads). Moreover, cutoff 256 (making 256 tasks) makes 16 cores busier than the others. Although we could choose cutoff 240 to improve the performance, for consistency with other experiments, we have limited ourselves to powers of two. By increasing the cutoff, there will be more tasks and a better distribution, hence better speedup.



(a) Xeon Phi, CPI Rate

(b) Xeon Phi, Total CPU Time

(c) OpenMP (static), Xeon Phi CPU balance

(d) OpenMP (dynamic), Xeon Phi CPU balance

(e) Cilk Plus, Xeon Phi CPU balance

(f) TBB, Xeon Phi CPU balance

(g) GPRM, Xeon Phi CPU balance

(h) GPRM-Steal, Xeon Phi CPU balance

Figure 6.9: Parallel MatMul benchmark on a 4096×4096 matrix of double numbers.

The results of this benchmark on the Xeon Phi raises the question of what causes the high CPI Rate for GPRM and Intel TBB while they run sometimes faster or at least as fast as other implementations? The answer is to be found in the number of executed instructions. When we look at the hardware event *INSTRUCTIONS_EXECUTED* sampled by the VTune Amplifier, then the higher CPI Rate does not necessarily mean degraded performance. Although the CPI Rate is higher for the TBB, GPRM, and GPRM-Steal approaches, the number of *INSTRUCTIONS_EXECUTED* is notably smaller compared to Cilk Plus and OpenMP. For instance, this number in the Cilk Plus approach is almost $2\times$ bigger than that of GPRM.

In the charts in Figures 6.9(c) and 6.9(d), there is an evident distinction between the distribution of CPU times that shows how OpenMP load balancing, when using dynamic scheduling leads to better performance. For a very detailed comparison, other hardware events should be taken into account as well, but we can already reason about the performance only by looking at these few fields.

6.1.5 Detailed Comparison for the MatMul benchmark

In most cases, GPRM [5] was the winning model. However, on the Xeon Phi, the OpenMP implementation of the MatMul benchmark scales up better than the GPRM implementation. We decided to explore this further and therefore, in this subsection we aim to figure out why GPRM could not reach the top performance achieved by OpenMP in this case.

Thread Mapping (Affinity)

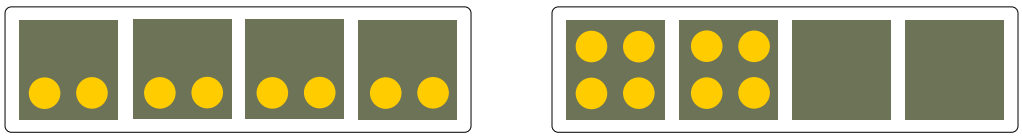
Although it is not obvious to decide about the optimal number of threads for the OpenMP case, we can reason about the performance of GPRM. As stated earlier, similar to [163], we have decided to evenly distribute the threads on the Xeon Phi cores. Our default strategy was to fill two hardware threads from the first to the last physical core, and fill the remaining two hardware threads afterwards.

Obviously, with 240 threads, all the logical cores will be filled, but mapping strategies will ultimately specify which tasks to be run on which cores. The reason is that with pre-assigned tasks to threads in GPRM, the location of a thread will become location where its tasks will be executed on (unless they got stolen at runtime). In some cases, an alternative mapping strategy could improve the performance, e.g. a *compact* approach for filling the logical cores (see Figure 6.10(b)). Finding such cases is not always straightforward. A good solution depends on tasks'communication pattern, whether or not data chunks can fit in an L2 cache, shared by logical cores in a physical core, and so on. For matrix multiplication, since chunks

[5] For the rest of this thesis, by GPRM we mean GPRM with task stealing enabled.

for consecutive threads (according to thread ID) are contiguous, *compact* affinity might be beneficial in many cases, as every 4 consecutive threads can share an L2 cache. However, changing the mapping strategy can be pointless if the chunks could not fit in the caches, because either the input set is large or the cutoff value is small.

GPRM offers a command-line switch to choose between these mapping strategies. By choosing a *compact* affinity for the MatMul benchmark on 4096×4096 matrices, GPRM reaches the top performance achieved by OpenMP in Fig. 6.7(a). Our point is that choosing between a couple of mapping strategy (as in GPRM) in order to get the best performance is much easier than finding an optimal number of threads (as in OpenMP).



(a) Balanced affinity (b) Compact affinity

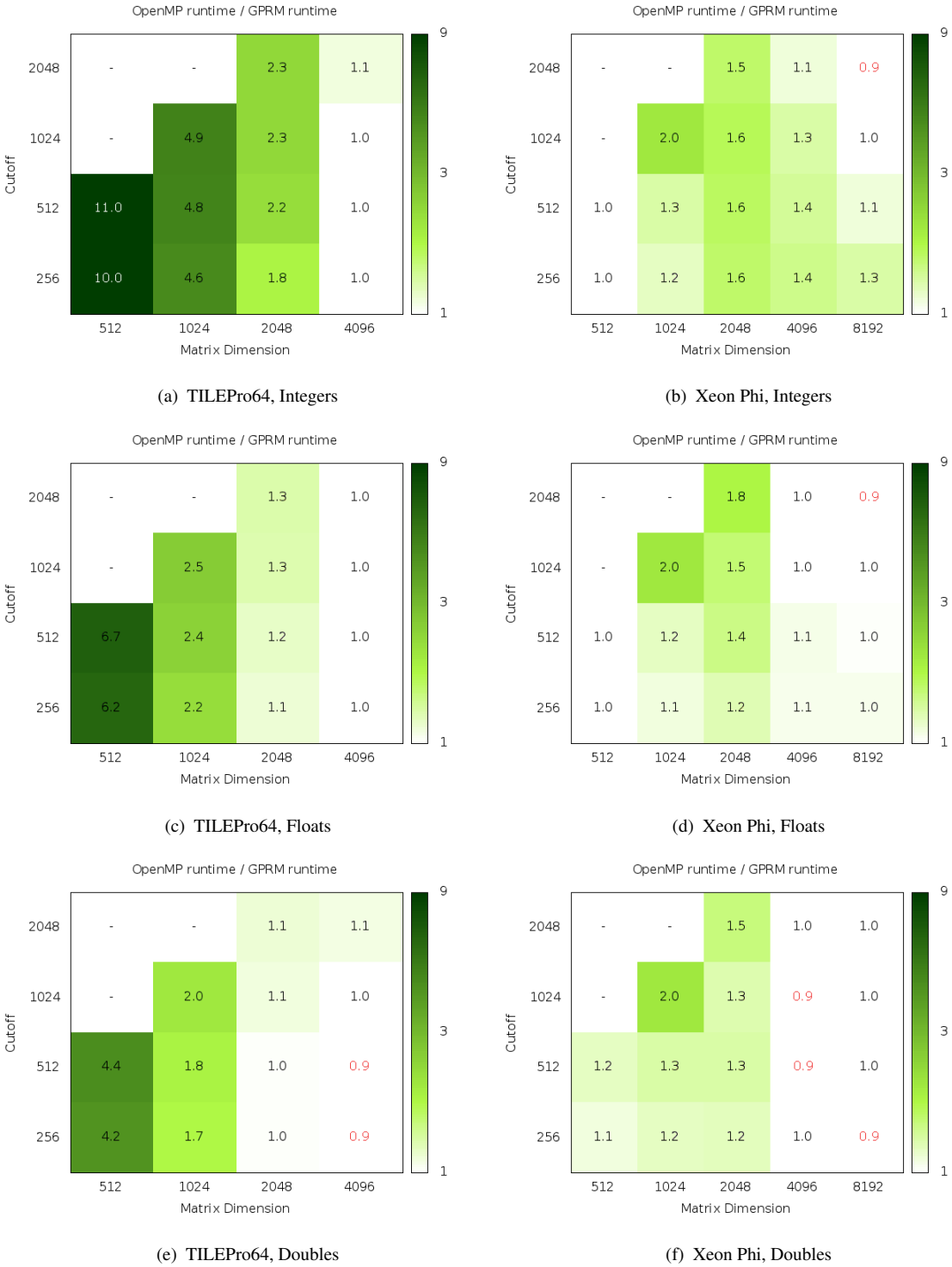
Figure 6.10: Two thread mapping strategies for the Xeon Phi

Relative Performance for Matrix Multiplication

We extend the MatMul benchmark (which is defined for 4096×4096 double matrices) to other square matrices with varying types and sizes of data. Integer, float, and double square matrices with sizes varying from 512 to 4096 are tested. Since matrix multiplication operations on the TILEPro64 are not as fast as on the Xeon Phi, we only include the results for 8096×8096 matrices on the Xeon Phi.

Heat maps are used in Fig. 6.11 to show OpenMP runtime over GPRM runtime ratio with as many threads as the number of cores for both cases. In almost all cases, GPRM outperforms OpenMP. Therefore, even for the matrix multiplication benchmark, GPRM with the default *balanced* affinity can be considered a better approach. Whenever needed though, an alternative mapping strategy could be utilised.

As illustrated in 6.11, for small matrices, hence tiny tasks, GNU OpenMP on the TILEPro64 has very poor performance, compared to GPRM. Intel OpenMP on the Xeon Phi does not suffer from the same issue; however, there are still some cases that its performance is half of the performance of its counterpart, GPRM. These differences for such a simple benchmark are significant, showing how well our pure task-based approach fit into manycore architectures.



(a) TILEPro64, Integers

(b) Xeon Phi, Integers

(c) TILEPro64, Floats

(d) Xeon Phi, Floats

(e) TILEPro64, Doubles

(f) Xeon Phi, Doubles

Figure 6.11: Detailed comparison for the MatMul: *(OpenMP runtime) / (GPRM runtime)*

6.1.6 Choosing a Proper Cutoff

Focusing on GPRM, there are some general rules that can be applied for finding a good cutoff. For example, if the problem is too small, one should consider creating tasks less

than the number of cores to limit the cost of task creation and communication overhead. If the problem is regular, one might consider creating as many tasks as the number of cores in the system to provide the threads with equal workloads. If the tasks are unequal (as in the Fibonacci benchmark), then creating a larger number of tasks and enabling the stealing mechanism can help the runtime system balance the load. If there are children and parent tasks (as in the MergeSort benchmark), GPRM will automatically run the parents on the same core (thread) as one of their children. In such cases, the programmer is only concerned about the number of child tasks. If the number of tasks is not divisible by the number of cores, then creating more tasks along with enabling the stealing mechanism would be useful.

6.2 Multiprogramming Workloads

In this section, we consider two multiprogramming scenarios for our three base benchmarks to see how the programming models behave in multiprogramming environments. The metric used for the comparison is the user-oriented metric *Turnaround Time*.

6.2.1 Case 1: Multiple Instances of a Single Program

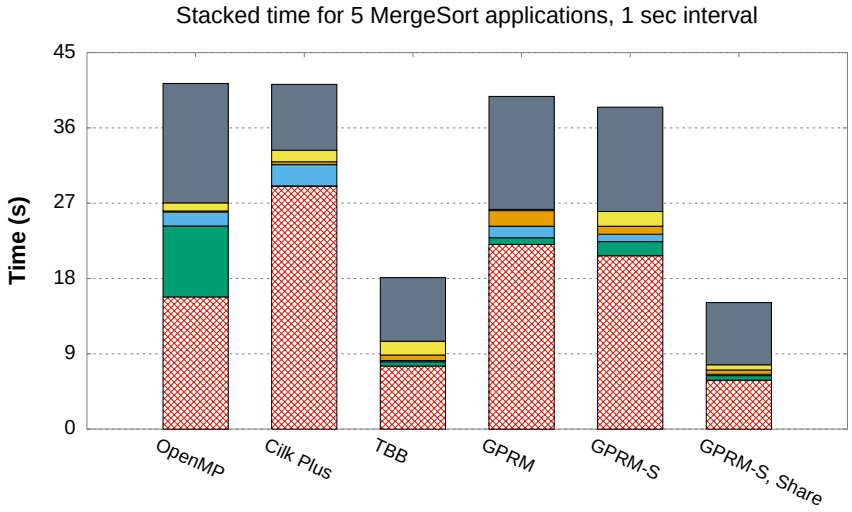
In order to show the effect of *Global Sharing* between different GPRM applications, consider 5 MergeSort applications entering the system with an interval of 1 second. Again, an array of 80M integers with the cutoff 2048 and the default number of threads (240) is considered.

The results are illustrated in Fig. 6.12. The stacked representation is used firstly to illustrate the difference between the kernel times (only the parallel parts) of the applications, e.g. in the OpenMP case, the difference between the best and worst kernel time is around 11.2 seconds, while for the "GPRM-Steal with Global Sharing", the difference is less than 2 seconds.

The second use of the stacked representation is to show all other consumed time in the system at the top of the stacked column chart. This includes the time before job submission (interval) as well as the time spent on initialisations (sequential time). It is evident that for OpenMP or GPRM approaches without Global Sharing, this time is larger, which is an indicator of the amount of overlap between the sequential parts of the applications. It can be seen in Fig. 6.12(h) how the bottleneck is removed with the help of *Global Sharing*.

Beside the *Total CPU Time*, the hardware event *Instructions Executed*, estimated by multiplying sample count by 10M events per sample (obtained by the VTune Amplifier in Fig. 6.12(b)) can be used as another metric for comparison. Based on $(|(V1 - V2)|/((V1 + V2)/2)) * 100$ formula, for the sum of the turnaround times, the difference between "GPRM-Steal with Global Sharing" and TBB as the second best result is around 20%[6].

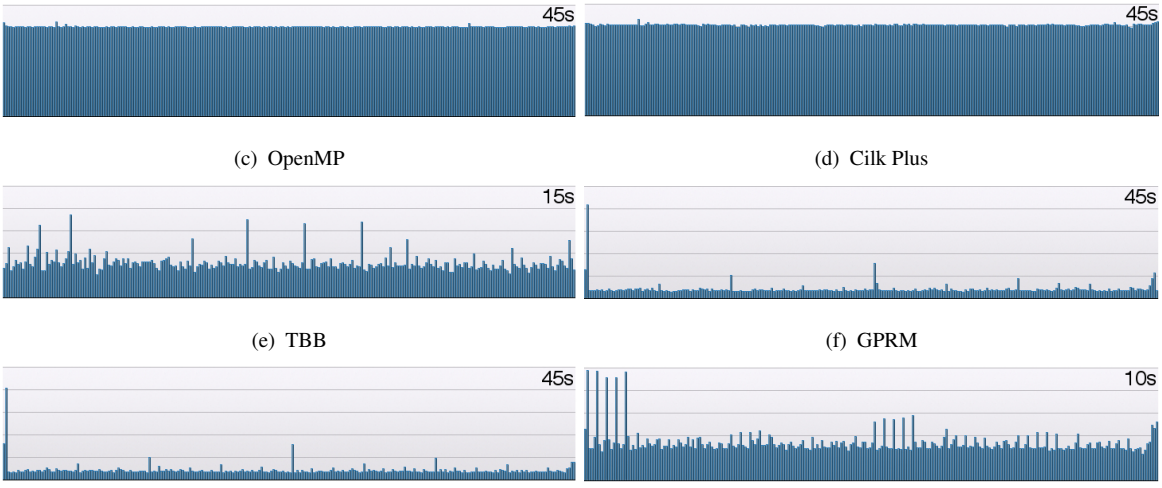
[6]The submission of a job is the start of its initialisation



(a) Stacked Time

Approach	Total CPU Time(s)	Inst. Executed
OpenMP	8.7K	21.2K × 10M
Cilk Plus	8.8K	16.0K × 10M
TBB	1.1K	4.6K × 10M
GPRM	0.9K	3.8K × 10M
GPRM-S	0.9K	3.9K × 10M
GPRM-S, Share	0.8K	3.8K × 10M

(b) Performance Summary



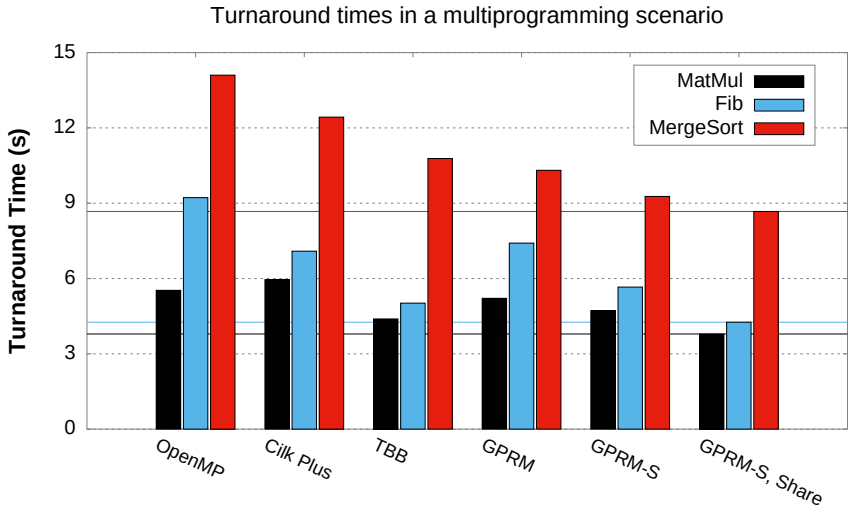
(c) OpenMP (d) Cilk Plus

(e) TBB (f) GPRM

(g) GPRM-Steal (h) GPRM-Steal with Global Sharing

Figure 6.12: Case1: A Multiprogramming case with 5 MergeSort applications with 1 sec interval.

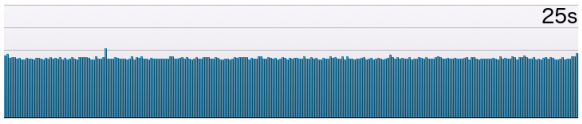
The stacked column chart shows the mean time for each application's kernel, followed by the remaining time spent from the start of the first application to the end of the last one. GPRM-Steal with Global Sharing has the best performance; Cilk Plus has the worst.



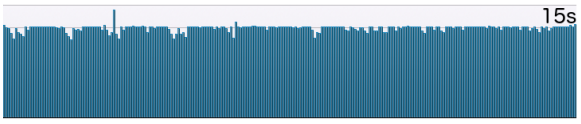
(a) Turnaround times

Approach	Total CPU Time(s)	Inst. Executed
OpenMP	3.2K	8.5K × 10M
Cilk Plus	2.9K	7.5K × 10M
TBB	1.2K	4.3K × 10M
GPRM	1.1K	4.1K × 10M
GPRM-S	1.1K	4.1K × 10M
GPRM-S, Share	1.2K	4.2K × 10M

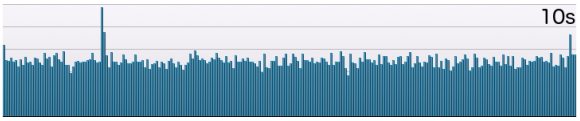
(b) Performance Summary



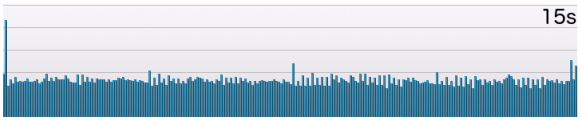
(c) OpenMP



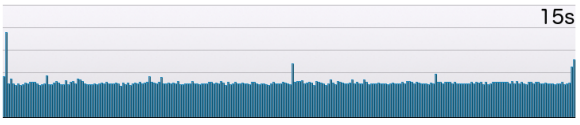
(d) Cilk Plus



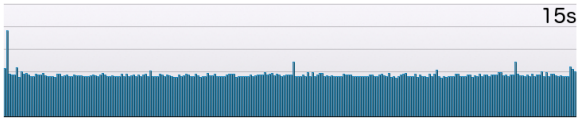
(e) TBB



(f) GPRM



(g) GPRM-Steal



(h) GPRM-Steal with Global Sharing

Figure 6.13: Case2: A multiprogramming scenario with all the three benchmarks.
The best turnaround times are obtained with "GPRM-Steal with Global Sharing". It improves the performance of GPRM by stealing tasks locally inside each application and sharing information globally across multiple applications. OpenMP has the worst performance.

6.2.2 Case 2: Single Instances of Multiple Programs

For this case, we extend the experiment in Section 4.3, by adding the GPRM approaches to the comparison. The three base benchmarks have the same input sizes as the single-program cases with the cutoff value 2048 and the default number of threads 240. We do not start all of them at the same time. Rather, we want the parallel phases to start almost simultaneously, such that the threads of all applications compete for the resources. For that purpose, the MergeSort benchmark enters the system first. Two seconds later the MatMul benchmark enters the system, and half a second after that, the Fib benchmark starts[7]. The results are shown and discussed in Fig. 6.13. For the sum of the turnaround times, the difference between "GPRM-Steal with Global Sharing" and TBB as the second best result is about 17%.

Although the *Total CPU Time* is a key performance metric, it cannot be used solely to interpret the results. A sequential program can have the same value for the *Total CPU Time* as a parallel program. Therefore, it is also important to find out how evenly the tasks are distributed across the system. As we have observed in the multiprogramming cases, compared to other GPRM approaches, the efficiency of the "GPRM-Steal with Global Sharing" comes from its better load balancing. However, the wasted CPU cycles by the runtime libraries, as for OpenMP and Cilk Plus, which can have a significant impact on the results can be detected by *Total CPU Time* and *Instructions Executed*.

6.3 Summary

We have used our "Base Benchmarks": Fibonacci, MergeSort and MatMul again and added GPRM to the performance comparison of the parallel models discussed in Chapter 4. The same benchmarks have also been used to compare GPRM and OpenMP on the TILEPro64.

We have presented a detailed analysis of GPRM's performance. We have also demonstrated the advantages of our task-based parallel programming model over the existing well-known parallel approaches. Traditionally, attention has focused on finding the optimal number of threads in order to achieve desirable performance. In the thread-speedup charts, we used the default number of threads for GPRM and presented its performance with only two points for no-stealing and stealing modes. Performance optimisation in GPRM is a simple matter of choosing a proper cutoff value. In other words, GPRM combines an intuitive task-based approach with excellent performance, without the need to tune the number of threads.

On the TILEPro64, GPRM outperforms OpenMP in all cases. GPRM also achieves top

[7]The sequential phase of the MergeSort benchmark with the input size 80M is around 2 seconds, and the initial phase of the MatMul benchmark with the input size 4096×4096 is about half a second.

performance for 2 out of the 3 uniprogramming test cases on the Xeon Phi, without any tuning. Further investigation on the only case on the Xeon Phi where GPRM was not the best model, matrix multiplication (MatMul), revealed new results: for different integer, float and double matrices GPRM significantly outperforms OpenMP on the TILEPro64, specially for small matrices. On the Xeon Phi, GPRM is again the winning approach in most of the cases. In other cases (such as the one used in the default MatMul configuration), after changing GPRM's thread mapping policy via a command-line switch, it was able to reach the top performance achieved by the optimal number of OpenMP threads.

For multiprogramming on a general-purpose parallel system, we propose the use of GPRM which implements a scheme called "Steal Locally, Share Globally". The idea is to steal tasks locally (from within the same application) only if the initial task assignment is not optimal, and to share the least amount of information about the system's load globally (between different applications). We have shown that our strategy is highly competitive against other approaches, namely OpenMP, Cilk Plus and TBB for all testbenches, and achieves the top performance on the Intel Xeon Phi with 17% to 20% difference with TBB as the second best approach.

Chapter 7

Parallel Lower-Upper Factorisation of Sparse Matrices

OpenMP enjoys wide support from its community and continues to evolve. This makes it a challenging competitor for every new programming model, including GPRM. In this chapter we highlight some of the drawbacks in the OpenMP tasking approach, and propose an alternative solution based on the GPRM programming framework.

We compare the performance of GPRM with that of OpenMP in 2 different scenarios: first a matrix multiplication benchmark[1] which has structured parallelism, and second, a linear algebra problem which fits very well into less structured task-based parallelism.

Lower-Upper factorisation of sparse matrices is a fundamental linear algebra problem. Due to the sparseness of the matrix, conventional worksharing solutions do not result in good performance, since a lot of load imbalance exists. As a well-known testcase, we have used the SparseLU benchmark from the the Barcelona OpenMP Tasks Suite (BOTS) [149].

For the purposes of this chapter, we will show how OpenMP fails to operate as expected for a large number of fine-grained tasks, while GPRM copes with such a situation naturally (more in Section 7.2). Furthermore, we will introduce a hybrid worksharing-tasking approach to avoid creating too many tasks (more in Section 7.3).

7.1 GPRM Parallel Loops

So far, we have only used GPRM parallel loops without discussing them. In this section, we will describe them more in detail.

[1] In this chapter, the matrix multiplication is used to show the effect of creating small tasks (short computations) on the performance of the runtime systems

GPRM is a purely task-based parallel framework. As discussed in Chapter 5, one can create CUTOFF[2] tasks in GPRM, each of which with their own indices. These indices can be then used by a worksharing construct to specify which elements of the loop belongs to which thread. Normally, when the tasks are fairly equal, the best result can be obtained by choosing the cutoff value as the same as the number of threads in GPRM, which is itself as the same as the number of cores. Although in Section 7.3 tasks are not equal, as a solution one can use the GPRM parallel loops to balance the load amongst threads. This solution, as will be shown, works very well when medium size or large sparse matrices are used.

We have created a number of useful parallel loop constructs for use in GPRM. These worksharing constructs corresponds to the `for` worksharing construct in OpenMP, in the sense that they are used to distribute different parts of a work among different threads. However, there is a big difference in how they perform the operation. In OpenMP, the user marks a loop as an OpenMP `for` with a desirable scheduling strategy, and the OpenMP runtime decides which threads should run which part of the loop; in GPRM, multiple instances of the same task –normally as many as the cutoff value– are generated, each with a different index (similar to the `global_id` in OpenCL). Each of these tasks calls the parallel loop passing in their own index to specify which parts of the work should be performed by their host thread.

Figure 7.1: Partitioning a nested $m(3 \times 3)$ or a single $m(9)$ loop amongst $n(4)$ threads. a) Step size of 1, as in the `par_for` and `par_nested_for`, b) Continuous, as in the `par_cont_for`

The `par_for` construct is essentially a sequential loop used for parallelisation of a single loop. It distributes the work in a Round-Robin fashion to the threads. It can also be referred to as a *partial for*, as it is actually a sequential loop that executes only a part of the original loop. A `par_nested_for` treats a nested loop as a single loop and follows the same pattern to distribute the work. Alternatively, the *Continuous* method gives every thread an m/n chunk, and the remainder $m\%n$ is distributed one-by-one to the foremost threads. These methods are shown in Fig 7.1. The need to parallelise nested loops arises often, e.g. in situations where there are variable size loops such as the SparseLU benchmark in Section 7.3.

The `par_for` and `par_nested_for` loops in GPRM are implemented using C++ templates and member-function pointers. The implementation of these worksharing constructs are given in Listing 7.1 and 7.2. They will be our worksharing constructs by default. The parallel loops with *Continuous* partitioning have similar implementations. We denote them as partial continuous loops: `par_cont_for`.

[2]CUTOFF is a constant which defines the number of tasks that can be run simultaneously in the system.

```
template<typename Tclass, typename Param1>
int par_for(int start,int size,int ind, int CUTOFF, Tclass* TC, int (
    Tclass::*work_function)(int,int,int,Param1), Param1 p1) {
 int turn=0;
 for(int i = start; i < size;) {
  if(turn % CUTOFF == ind) {
   (TC->*work_function(i,start,size,p1);
   i = i + CUTOFF;
  }
  else {
   i++;
   turn++;
  }
 }
 return 0;
}
```

Listing 7.1: Implementation of the `par_for`

```
template <typename Tclass, typename Param1>
int par_nested_for(int start1, int size1, int start2, int size2, int
    ind, int CUTOFF, Tclass* TC, int (Tclass::*work_function)(int,int,
    int,int,int,int,Param1), Param1 p1) {
 int turn=0;
 for(int i = start1; i < size1; i++) {
  for(int j = start2; j < size2;) {
   if((turn >= 0) && (turn % CUTOFF == ind)) {
    (TC->*work_function)(i,j,start1,size1,start2,size2,p1);
    j = j + CUTOFF;
    if(j >= size2)  turn = size2 - j + ind;
   }
   else {
    j++;
    turn++;
   }
  }
 }
 return 0;
}
```

Listing 7.2: Implementation of the `par_nested_for`

As we will see in the next sections, since the GPRM `par_nested_for` is implemented with minimum overhead, it is a significantly useful worksharing construct, .

7.2 Matrix Multiplication Micro-benchmark

In this section, we use a naive matrix multiplication algorithm with a triple nested loop as a micro-benchmark to compare OpenMP and GPRM in terms of the overhead of executing simple tasks on the TILEPro64 and find out which cases are problematic. However, for the SparseLU benchmark itself, we consider both platforms. The code is given at Listing 7.3.

As our aim is to use this micro-benchmark to identify the most important barriers on the way, we change the interpretation of the problem to performing multiple jobs. Suppose that the product of an $m \times n$ matrix A and an $n \times p$ matrix B is the $m \times p$ matrix C. We want to parallelise the first loop of the triple nested loop, which loops on m, therefore m becomes the number of jobs for this problem. The size of each job is identified by the sizes of the next two loops in the triple nested loop, i.e. $p * n$. We have chosen $n = p$ to make the problem more regular. We end up with matrices with the following specification: $A : m \times n$, $B : n \times n$, and $C : m \times n$. Due to the poor data locality of this algorithm, one should not expect to see a linear speedup.

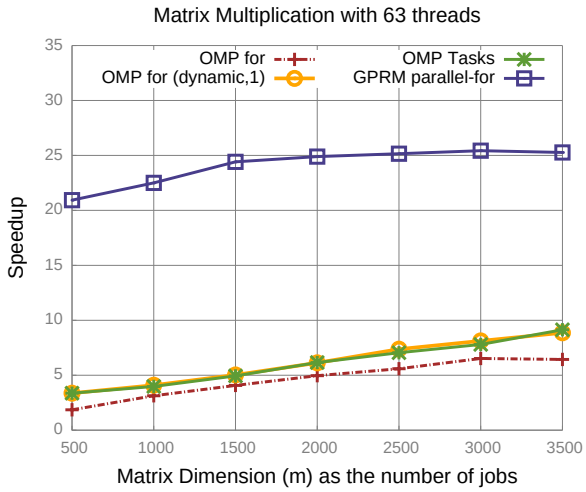
```
1 for (int i = 0; i < m; i++){
2  for(int j = 0; j< p; j++){
3   for(int k = 0; k <n;  k++){
4    C[i*p+j] += A[i*n+k] * B[k*p+j];
5   }
6  }
7 }
```

Listing 7.3: Matrix Multiplication Micro-benchmark

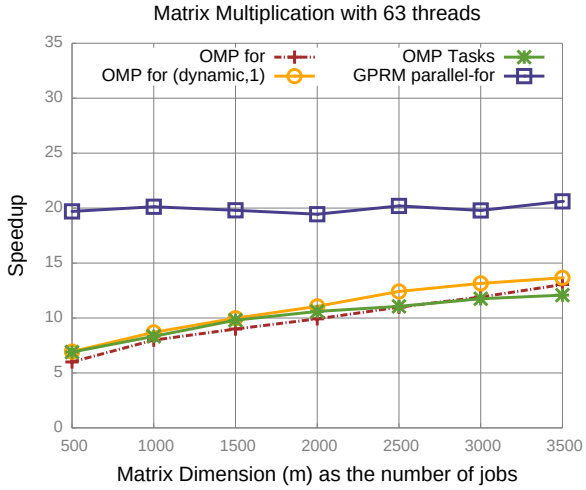
Four approaches are selected for the comparison: I) The OpenMP `for` worksharing construct, II) The OpenMP `for` with *dynamic* schedule and *chunk_size* of 1, III) The OpenMP Tasks, and IV) The GPRM `par_for` construct.

Figure 7.2 shows the speedup for different job sizes over the sequential C++ baseline. GPRM outperforms OpenMP in all cases but especially for the small job case (even then, the job size is still not small enough to show the real overhead of having fine-grained tasks in OpenMP). To our best understanding, the performance difference is due to the overhead of thread scheduling on the TILEPro64, which is more visible in the small job cases with short execution times.

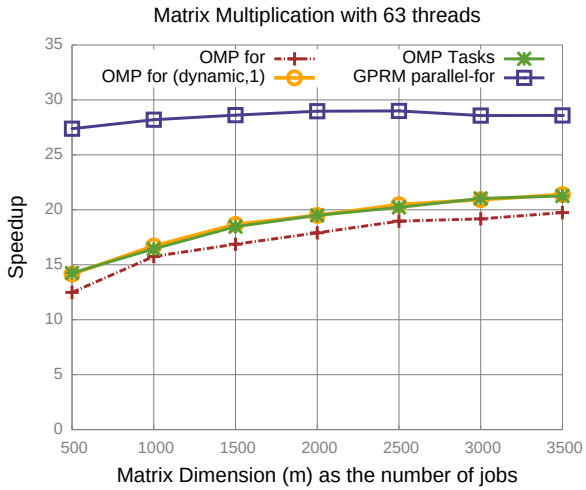
In order to investigate the effect of task granularity on the behaviour of the OpenMP performance, we decrease the size of the tasks even more. We also looked at the influence of the cutoff value on the performance. Since the behaviours of the OpenMP worksharing constructs were fairly similar, only the default `omp for` is used for the next experiment.



(a) 500×500 - Small jobs



(b) 1000×1000 - Medium jobs



(c) 1500×1500 - Large jobs

Figure 7.2: Matrix Multiplication: GPRM `par_for` vs. different OpenMP approaches on the TILEPro64

To improve the behaviour of the tasking approach, we added a cutoff value for the tasks, such that only *m/cutoff* tasks were created. This is similar to sequencing multiple tasks. Fig 7.3 compares the speedup of a tuned version of the OpenMP task-based model with the other alternatives. We believe that the regularity of GPRM in assigning tasks to its lower thread management overhead make it the winner approach.

```
1 for (int i = 0; i < (m/cutoff); i++){
2 #pragma omp task firstprivate(i)
3  for(int t = 0; t < cutoff; t++) {
4   for(int j = 0; j < p; j++){ // p=n
5    for(int k = 0; k < n; k++){
6     C[(i*cutoff+t)*p+j] += A[(i*cutoff+t)*n+k] * B[k*p+j];
7    }
8   }
9  }
10 }
```

Listing 7.4: Matrix Multiplication micro-benchmark with a cutoff value for OpenMP tasks

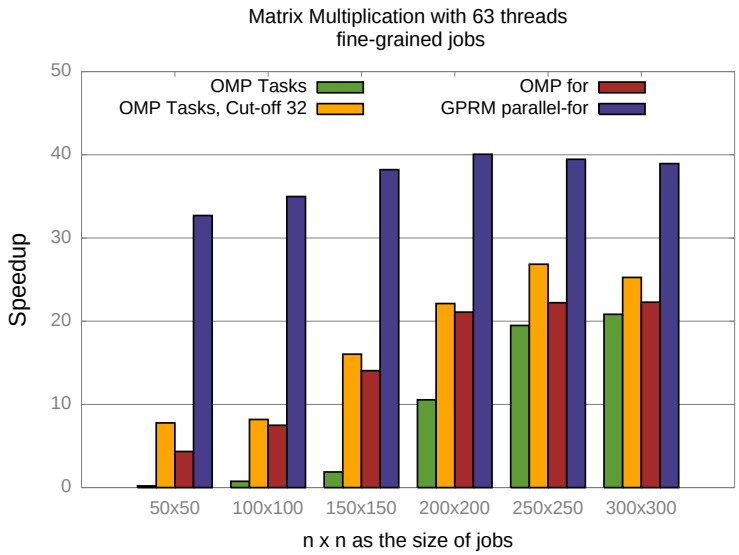


Figure 7.3: Speedup measurement for fine-grained jobs, Number of the jobs: 200,000

Figure 7.3 shows that the poor behaviour of fine-grained tasks can be remedied to a considerable extent by using a proper cutoff value. The OpenMP approaches gradually becomes better when the size of the job is increased. We have chosen the first two cases to show the effect of using a cutoff in more detail, because these cases show degraded performance compared to the sequential implementation if no cutoff is used at all.

Figure 7.4 shows that a good choice of the cutoff value gives the speedup of 38.6× against the case with no cutoff and 7.8× against the sequential version, for the job size of 50×50

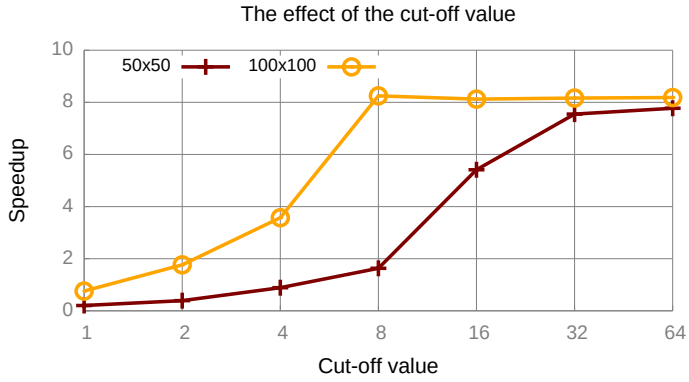


Figure 7.4: Speedup improvement by using a cutoff value for the fine-grained OpenMP tasks. Number of the jobs: 200,000. Size of the jobs: 50×50 - 100×100

with 63 threads. The speedup for the job size of 100×100 is also improved by 10.8× compared to the case with no cutoff and 8.2× compared to the sequential runtime.

7.3 Sparse LU Factorisation

The SparseLU benchmark from the BOTS suite [149], which computes an LU factorisation of sparse matrices is a proper example of matrix operations with load imbalance. In the proposed OpenMP solution [3], a task is created for each non-empty block. The main SparseLU code from [3] (omitting the details of the OpenMP task-based programming, such as dealing with shared and private variables) is copied in Fig 7.5. The number of tasks and the granularity of them depend on the number of non-empty blocks and the size of each block, hence a cutoff value cannot be defined inside the user-written OpenMP code. As has also been discussed in [3], using OpenMP tasks results in better performance compared to using the `for` worksharing construct with dynamic scheduling. Therefore, we use the tasking approach for the comparison.

The GPC code for the SparseLU benchmark implementation can be found in Listing 7.5. The `gprm unroll` pragma results in parallel evaluation of the `for` loop it precedes. By default, expressions in the GPC code are evaluated in parallel. The `seq` pragma forces sequential evaluation of the block it precedes. We have defined different phases of the algorithm as different types of tasks in GPRM.

It is worth mentioning that we have not changed the initialisation phase of generating sparse matrices in the BOTS benchmark suite. Base on the default configuration, the matrices become sparser as the number of blocks increases. For example, in the case of 50×50 blocks, the matrices are 85% sparse, while for the cases with 100×100 blocks, the matrices become 89% sparse.

```
1  int sparseLU() {
2    int ii, jj, kk;
3    #pragma omp parallel
4      #pragma omp single nowait
5        for (kk=0; kk<NB; kk++) {
6          lu0(A[kk][kk]);
7          /* fwd phase */
8          for (jj=kk+1; jj < NB; jj++)
9            if (A[kk][jj] != NULL)
10             /* only create tasks for non-empty blocks */
11             #pragma omp task
12               fwd(A[kk][kk], A[kk][jj]);
13         /* bdiv phase */
14         for (ii=kk+1; ii < NB; ii++)
15           if (A[ii][kk] != NULL)
16             /* only create tasks for non-empty blocks */
17             #pragma omp task
18               bdiv (A[kk][kk], A[ii][kk]);
19         /* wait for previous tasks */
20         #pragma omp taskwait
21         /* bmod phase */
22         for (ii=kk+1; ii < NB; ii++)
23           if (A[ii][kk] != NULL)
24             for (jj=kk+1; jj < NB; jj++)
25               if (A[kk][jj] != NULL)
26                 /* only create tasks for non-empty blocks */
27                 #pragma omp task
28                 {
29                   if (A[ii][jj]==NULL)
30                     A[ii][jj]=allocate_clean_block();
31                   bmod(A[ii][kk], A[kk][jj], A[ii][jj]);
32                 }
33         /* wait for all previous tasks */
34         #pragma omp taskwait
35       }
36 }
```

Figure 7.5: the main code of the SparseLU benchmark, without revealing the OpenMP programming details, borrowed from [3]

To obtain a fair distribution of the matrix elements amongst different threads –bearing in mind the sparseness of the matrix– we have used a `par_nested_for`, because the numbers of iterations are not fixed in this problem.

The loops become smaller as *kk* grows, which means that using a `par_for` would, after a few iterations when $outer\_iters > cutoff$, lead to starvation of some of the threads. By using a `par_nested_for` the threads can get some work as long as the $outer\_iters * inner\_iters > cutoff$. Therefore, in order to implement the *fwd*, *bdiv*, and *bmod* tasks, one can use the GPRM APIs for the parallel loops, as shown in Listing 7.6.

Figure 7.6 shows a sparse matrix of 4000×4000 divided into blocks of varying size. It is again clear that with larger numbers of blocks in each dimension which results in smaller block sizes, the OpenMP performance drops drastically. GPRM can deal with tiny 8×8 blocks 6.2× better than the best result obtained by OpenMP.

```
GPRM::Kernel::SpLU sp;

void fwd_bdiv_tasks (int kk, float** A, const int CUTOFF) {
#pragma gprm unroll
 for (int n = 1; n < (CUTOFF/2); n++) {
  sp.fwd_t(kk, A, n-1, CUTOFF/2);   // fwd task
  sp.bdiv_t(kk, A, n-1, CUTOFF/2); // bdiv task
 }
}

void bmod_tasks (int kk, float** A, const int CUTOFF) {
#pragma gprm unroll
 for(int n = 1; n < CUTOFF; n++) {
  sp.bmod_t(kk, A, n-1, CUTOFF);   // bmod task
 }
}

void GPRM::spluTask::ComputeLU () {
#pragma gprm seq
 { /* GPRM evaluates in parallel unless otherwise stated */
  float** A = init_task();
  for(int kk=0; kk<NB; i++)  { // NB: #Blocks
#pragma gprm seq
   {
    lu0_task(kk, A);
    fwd_bdiv_tasks(kk, A, 63)
    bmod_tasks(kk, A, 63)
   }
  }
 }
 return; }
```

Listing 7.5: SparseLU code in GPRM, Cutoff: 63

Table 7.1 reveals that the best results for the OpenMP approach is not obtained with the default setting, which is as many threads as the number of cores. Besides the large difference in execution times when the block size becomes less than 20×20, there is a significant performance degradation if the number of threads is set to the default value of 63. For example, the execution time becomes 12.25× worse than the best time for the last case. However, it is clear that GPRM reaches its best execution time without the need to tune the number of threads –here, the number of threads and the cutoff value are the same for GPRM. This is due to the fact that instead of creating very small tasks, GPRM offers an efficient way of distributing the work amongst threads.

```
1 int SpLU::fwd_t(int kk, float** A, int ind, int cutoff)
2 {return par_for(kk+1, NB, ind, cutoff, this, &SpLU::fwd_work, A);}
3
4 int SpLU::bdiv_t(int kk, float** A, int ind, int cutoff)
5 {return par_for(kk+1, NB, ind, cutoff, this, &SpLU::bdiv_work, A);}
6
7 int SpLU::bmod_t(int kk, float** A, int ind, int cutoff)
8 {return par_nested_for(kk+1, NB, kk+1, NB, ind, cutoff, this, &SpLU::
    bmod_work, A);}
9
10 /* The fwd_work function here is the same as fwd in the cited paper */
11 int SpLU::fwd_work (int jj, int kk, int NB, float** A) {
12  if(A[(kk-1)*NB+jj] != NULL)
13   fwd(A[(kk-1)*NB+kk-1], A[(kk-1)*NB+jj]);
14  return 0;
15 }
```

Listing 7.6: Implementation of the member functions of the SparseLU class. Work functions can also be defined similar to the phases in [3]

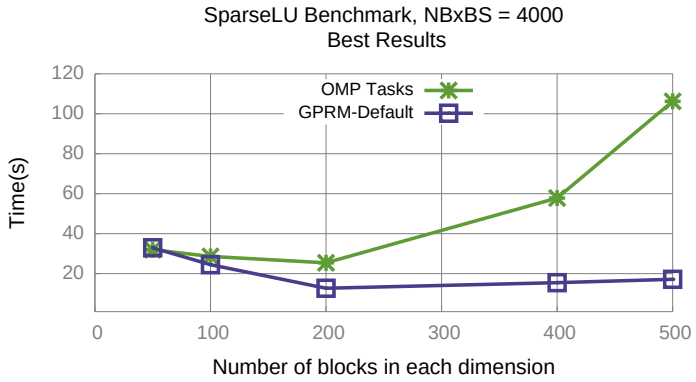


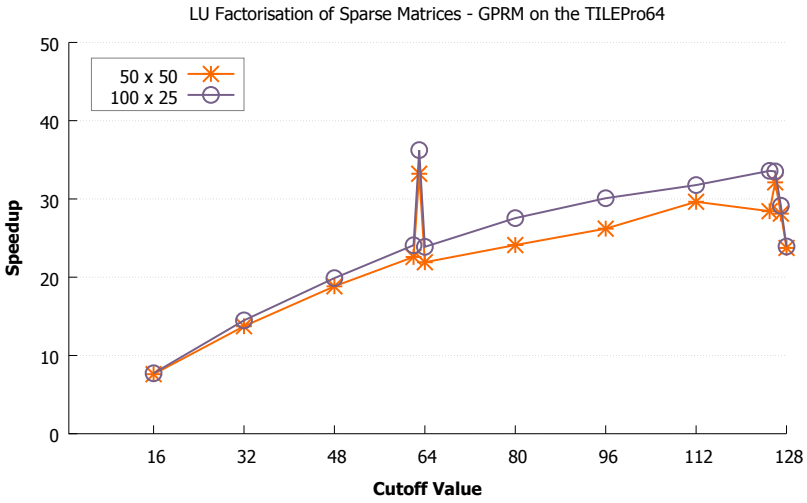
Figure 7.6: Execution time on the TILEPro64 for the sparse matrices of size 4000 with variable block sizes

Table 7.1: Number of threads for the best results obtained by GPRM and OpenMP for the sparse matrices of size 4000 with variable block sizes on the TILEPro64

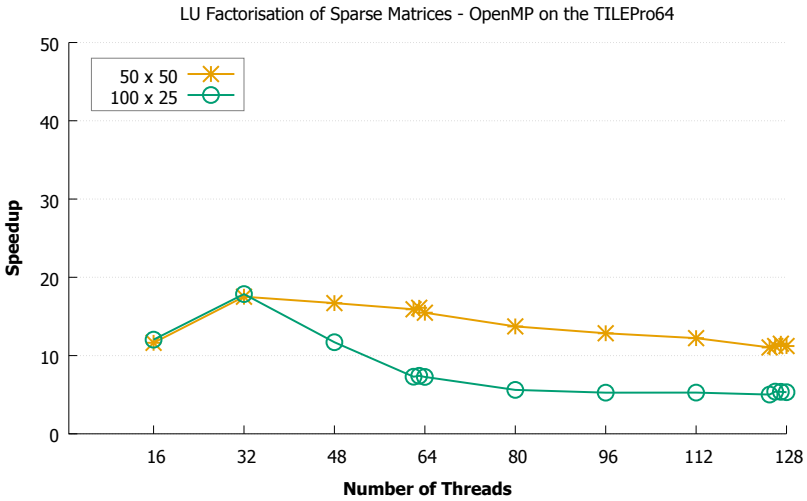
Number of Blocks	50	100	200	400	500
No of threads for OMP tasks	64	63	32	16	8
No of threads for GPRM	63	63	63	63	63

7.4 Results on the TILEPro64

After discussing the implementation details and conducting a preliminary comparison, we conclude that the OpenMP performance comes close to that of GPRM when the number of blocks are smaller, hence the tasks are larger. Therefore, in order to have a meaningful comparison on the TILEPro64, we have chosen a matrix of 2500 elements and compared the performance results for the 50×50 and 100×25 configurations (*no_blocks*×*block_size*).



(a) GPRM Solution: all three cases of 62, 63, and 64 are considered for the cutoff value



(b) OpenMP Solution: all three cases of 62, 63, and 64 are considered for the number of threads

Figure 7.7: SparseLU Factorisation: GPRM vs. OpenMP on the TILEPro64

The speedup charts for the SparseLU benchmark have been shown in Fig 7.7. Since GPRM uses the default number of threads, in order to generate the speedup chart, one should change the number of tasks using the cutoff value. For this purpose, we have increased the cutoff value from 16 to 128 to show how regular our approach is, in the sense that it gets its best

performance with the factors of the number of cores. This is not surprising for this algorithm, since the problem has been partitioned amongst threads regularly, and therefore they can exploit the underlying architecture more efficiently.

Since the solution proposed using OpenMP tasks is different from ours, we simply increased the number of threads in that case (the number of tasks depends on the input set). Increasing the number of threads beyond the number of cores shows the effect of oversubscription.

7.4.1 OpenMP Performance Bottlenecks

We have identified a number of performance bottlenecks when programming with OpenMP. The first is the thread migration overhead. This overhead can often be removed by statically mapping (pinning) the OpenMP threads to the execution cores. Using static thread mapping (pinning) in a platform with per-core caches could be very useful, particularly for load balanced data parallel problems, in which the portion of the work to be done by each thread is fairly equal, and therefore CPU time and local caches can be effectively utilised. Our study [23] shows that for such a platform, static thread mapping is often a good practice in single-program environments, but for multiprogramming environments, in which different programs compete for the resources, it is not always efficient. We refer the readers to [22, 23, 164] for detailed discussions on thread mapping for OpenMP programs.

Another barrier is to find a proper cutoff point to avoid creating overly fine-grained tasks. Programmers have to be very careful about the granularity of the tasks, otherwise the results may be totally unexpected. It has also been observed that there is no guarantee that running an OpenMP program with the default number of threads will result in the best performance.

7.4.2 Comparison of the OpenMP and GPRM Solutions

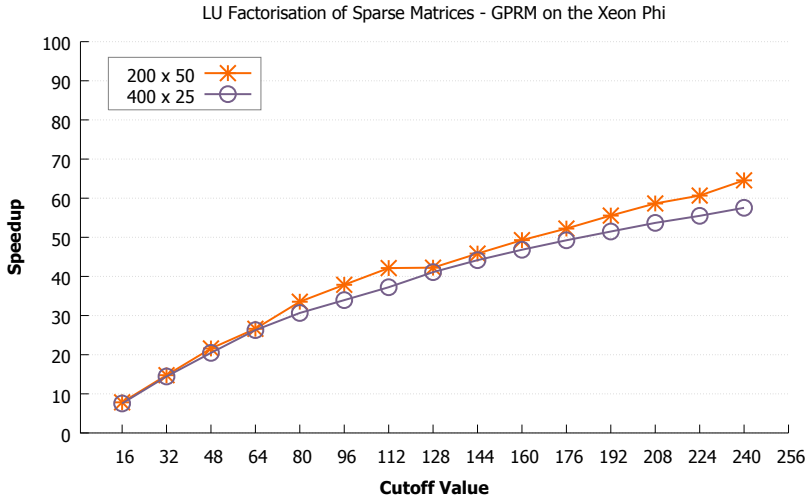
For the SparseLU problem, although creating OpenMP tasks for non-empty blocks is a smart solution, it is not working very well for all matrices. The first reason is that a single thread explores the whole matrix and creates relatively small tasks for non-empty blocks, while in the proposed solution implemented in GPRM, multiple threads look into their portions of the work in parallel. The difference in performance can be noticeable especially in the *bmod* phase with a nested loop. As also reported in [165], combining the OpenMP `for` worksharing construct with tasks, as implemented in `sparselu_for` in the BOTS benchmark suite, is not a viable approach with OpenMP 3.0.

Secondly, in GPRM, every thread has specific work based on the program's task description file. If needed, runtime decisions will be applied to improve the performance, while OpenMP creates the tasks dynamically and all decisions are taken dynamically at runtime,

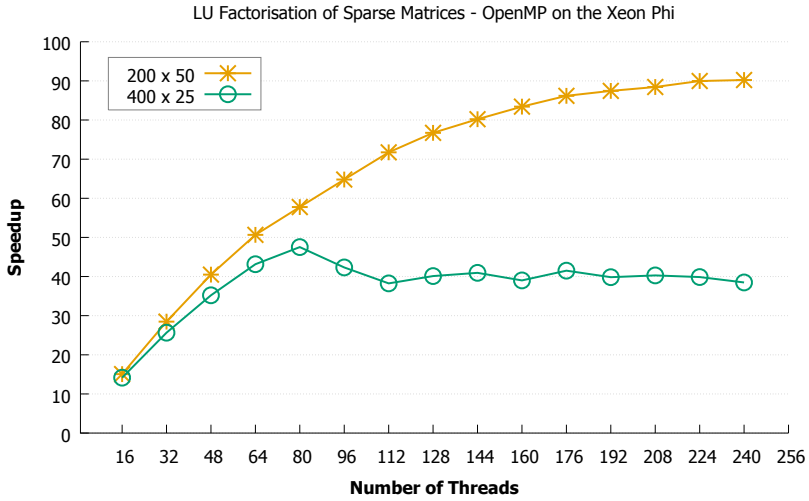
which makes its task management less efficient when the number of tasks becomes larger. Moreover, the overhead of task management becomes significant when the tasks become more fine-grained, as made clear by the Matrix Multiplication micro-benchmark.

In GPRM, using a `par_nested_for` construct in order to partition the matrix in a non-continuous manner results in a good load balance. Furthermore, our hybrid worksharing-tasking technique is pretty much straightforward. There is no pressure on the programmer to worry about private and shared variables, in contrast to OpenMP.

7.5 Results on the Xeon Phi



(a) GPRM Solution



(b) OpenMP Solution

Figure 7.8: SparseLU Factorisation: GPRM vs. OpenMP on the Xeon Phi

We have increased the matrix size on the Xeon Phi to better reflect the differences between

the two models on a system with more (logical) cores than the TILEPro64. A matrix of size 10000 is used for this purpose and we measure the speedup for 200×50 and 400×25 cases.

The results in Figure 7.8 are consistent with those in Figure 7.7 in the sense that OpenMP performs poorly when the number of blocks increases. However, it could outperform GPRM when the block size becomes larger, as in the 200×50 case (1.3×). This is not surprising, because the solution we have proposed in GPRM targets sparser matrices with larger numbers of blocks (and smaller block sizes), and is intended to avoid creating very fine-grained tasks. In other cases, the OpenMP solution proposed in [3] is expected to perform well. As we discussed earlier though, even for the 50×50 case on the TILEPro64, GPRM outperforms OpenMP.

7.6 Summary

In this chapter, we have used a solution implemented in GPRM to distribute small jobs across different threads, without the need to create a task for each of the jobs.

As the main focus of this chapter, we deploy our model to solve a fundamental linear algebra problem, Lower-Upper factorisation of sparse matrices. We first used a matrix multiplication micro-benchmark to highlight the differences between GPRM and OpenMP on the TILEPro64. For the small jobs, GPRM outperformed OpenMP by 2.8× to 11×. For the medium-sized jobs, the speedup improvement ranged from 1.5× to 3.3×. As the jobs get larger the difference became less significant. For the large jobs, the speedup ranged from 1.3× to 2.2×.

We have then used the SparseLU benchmark from the BOTS benchmark suite as a real-world example, and compared the results obtained from our model to those of the OpenMP tasking approach.

We used a sparse matrix of 4000×4000 divided into blocks of varying size on the TILEPro64. We demonstrated that for the larger numbers of blocks, i.e. smaller block sizes, the difference is considerable. The main advantage of GPRM is that it does not need to be tuned in terms of the number of threads. By contrast, tuning is crucial for OpenMP, otherwise for fine-grained tasks a huge drop in performance is inevitable.

We then compared the scalability of the two models on both the TILEPro64 and the Xeon Phi. In the case of GPRM, the number of threads is fixed and the speedup charts are generated by varying the cutoff value, while for the OpenMP approach, the number of tasks depends on the input set, therefore the speedup charts are generated by changing the number of threads. Since the solutions and the varying parameters are different, we have shown the results for each model separately, but we can compare the best results obtained by each.

On the TILEPro64, GPRM scaled 2× better than the best result obtained using OpenMP for both the 50×50 and 100×100 cases. For the default setting (with 63 threads), the speedup improvements were respectively 2.1× and 4.9×.

On the Xeon Phi, for the 200×50 case, OpenMP performed 1.3× better than GPRM, while for the 400×25 The best result for GPRM was 1.2× better than the best result obtained by OpenMP (with 80 threads) and 1.5× better than the default setting (with 240 threads).

We proposed a solution for LU factorisation of sparse matrices, using GPRM parallel constructs, such as its `par_nested_for`. We showed that such a solution can provide superior performance compared to the task-based OpenMP solution when targeting sparse matrices with larger numbers of blocks and smaller block sizes.

Chapter 8

Parallel Image Convolution

Image convolution is widely used for sharpening, blurring and edge detection. In this chapter we review two common algorithms for convolving a 2D image by a separable kernel (filter). We choose the algorithm with better sequential runtime as the baseline for parallelisation. We then compare the parallel performance of the optimised code using OpenMP and GPRM implementations on the both platforms.

We also measure the effects of optimisation techniques (i.e. loop unrolling and SIMD vectorisation) for both algorithms on the Xeon Phi and demonstrate how these optimisations affect sequential and parallel performance.

Apart from comparing the code complexity as well as the performance of OpenMP and GPRM, we investigate the impact of a parallelisation technique, task agglomeration in GPRM.

We have considered a Gaussian separable 5×5 kernel for the purposes of this study. We refer to two implementations of the convolution algorithm as single-pass and two-pass algorithms (implementations). The single-pass algorithm is the general algorithm used for convolution, having a nested loop over the kernel, therefore comprised of four loops to compute the convolution. The two-pass algorithm is specific to separable kernels, and uses a horizontal 1D convolution pass followed by a vertical 1D convolution pass to convolve the image.

In this chapter, we have considered a range of images from 1152×1152 to 8748×8748. There will be 3 planes per image and the benchmark will be run for 1000 times in order to measure a precise running time on both platforms. Therefore, the time for each image should be considered as *RunningTime/1000*.

8.1 Image Convolution Algorithms

Throughput computing applications demand for fast response time while dealing with a large amount of data. Image convolution is one of such throughput computing applications. Con-

volution of an image by a matrix of real numbers can be used to sharpen or smooth an image, depending on the matrix used. If A is an image and K is a convolution matrix, then B, the convolved image is calculated as:

$$B_{y,x} = \sum_i \sum_j A_{y+i,x+j} K_{i,j} \tag{8.1}$$

If **k** is a convolution vector, then the corresponding matrix K is such that $K_{i,j} = \mathbf{k}_i \mathbf{k}_j$

A separable convolution kernel is a vector of real numbers that can be decomposed into horizontal and vertical projections and hence can be applied independently to the rows and columns of the spatial domain to provide filtering [166]. It is a specialisation of the more general convolution, but is algorithmically more efficient to implement.

The convolution algorithm, borrowed from the EU funded Clothes Perception and Manipulation (CloPeMa) project [30] –whose aim is to develop a cloth folding robot using real time stereo vision– only works at the central part of the image that is in sight of multiple cameras, and what happens at the far edges are ignored. Therefore, we can safely ignore the complications at the edges and start the convolution from the pixel for which the kernel can access the required neighbours, i.e. pixel (2,2).

For all tests, separable kernels of width 5 are used. The algorithm uses 3 colour planes and is heavily memory-fetch bound.

8.1.1 Single-pass and Two-pass Algorithms

In order to solve the 2D convolution problem, the simplest approach is to loop over all the image pixels and all the kernel elements in one go. This algorithm is referred as the single-pass algorithm in this study. It uses 4 nested loops, the 2 outer loops on the rows and columns and the image and the 2 inner loops on the rows and columns of the kernel. For a 5×5 kernel, according to Eq. 8.1, it needs 25 multiply-accumulate operations for each pixel.

An alternative comes into play when a separable kernel is used: the two-pass algorithm. As stated, a separable kernel can be decomposed into horizontal and vertical projections and hence can be applied independently to the rows and columns of the spatial domain. For a 5×5 kernel, it reduces the number of multiplications per pixel to 10.

From the algorithmic point of view, the two-pass convolution algorithm should always be the preferred one if the kernel is separable. It has $O(n)$ time complexity, while the complexity of single-pass algorithm is $O(n^2)$, where *n* is the kernel width.

8.2 Results on the TILEPro64

Since the TILEPro64 does not have any FPUs or VPUs, we defer the discussion about the optimisation techniques to the next section where we cover the results on the Xeon Phi. For the purposes of this section, we choose the algorithm with better sequential running time, the two-pass algorithm, as the baseline for the speedup comparisons.

On the TILEPro64, for the range of images we have chosen, the default number of threads provides the best average-case performance for both programming models. Therefore, all the speedup results on this platforms are obtained with 63 threads over the running time for the sequential two-pass algorithm.

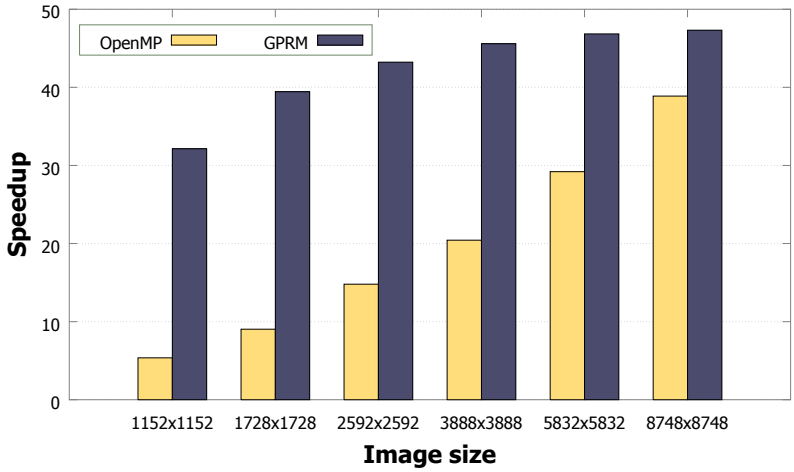


Figure 8.1: The speedup results on the TILEPro64, $\mathbf{R}\times\mathbf{C}$

It can be observed from Figure 8.1 that for the smaller images GPRM outperforms OpenMP significantly. Also in general, the speedup results are improved by increasing the size of the images.

It is possible to inspect the difference between OpenMP and GPRM more in detail. Since GPRM creates tasks and assigns them (initially) to threads at compile time, we can create empty tasks and therefore measure the overhead of distributing the tasks across different threads and the parallel reduction. In other words, it is possible to measure the overhead of communication between tiles in GPRM.

As a solution to mitigate the GPRM overhead, we have used task agglomeration: combining tasks into larger tasks to improve performance [32]. We therefore consider images with the width of 3 times the width of the original images, meaning that each row includes information for all 3 colour planes. This way, we include the 3 colour planes into the parallelisation. Using this technique, the size of tasks in GPRM becomes tripled and the overhead becomes approximately one third. This also improved the OpenMP performance in all cases except

for the largest image. The speedup results for this case, which we call **3R**×**C** is shown in Fig. 8.2.

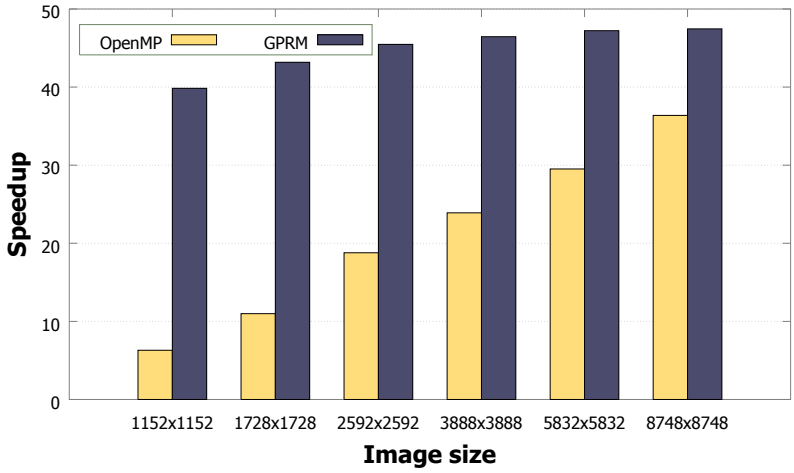


Figure 8.2: The speedup results on the TILEPro64, **3R**×**C**

8.3 Results on the Xeon Phi

We aim to explore a number of optimisation and parallelisation techniques for enhancing the performance of 2D image convolution on the Xeon Phi.

In [30], we have identified that the peak performance for the two-pass algorithm occurs at 100 OpenMP threads. Our initial experiments with GPRM has verified that considering the range of images from 1152×1152 to 8748×8748, 100 could be our magic number for both OpenMP (optimal number of threads) and GPRM (optimal number of tasks) models. It is worth stating that because the convolution time for each image is too short, the communication overhead becomes significant, and using all of the available resources in the Xeon Phi is not advantageous.

8.3.1 From Naive to Parallelised Optimised Code

We have implicitly mentioned the two-pass algorithm as an optimisation for the single-pass algorithm. We discuss in this section and section 8.3.5 that one should be careful about parallel performance prediction based on the sequential runtime of algorithms. For the purposes of this section we have chosen the largest 3 images of our 6 image test cases.

There is a number of other optimisations at different levels that could be considered for image convolution. Nevertheless, we do not consider our final optimised code as a Ninja code, i.e. the best optimised solution. The optimisations listed here can be achieved with a little

programming effort. The resulting optimised code, by definition should have performance comparable to a Ninja implementation, with a little effort of algorithmically improving the naive code (i.e. the compiler-generated code), or by using the latest compiler technology for parallelisation and vectorisation [167]. It has also been reported that the difference between a sequential optimised (similar to our single-pass optimised code) and a sequential Ninja implementation for a 2D convolution algorithm on the Xeon Phi is around 1% [168].

Another point to make is that in this study, we are also concerned about parallelisation techniques. An optimised sequential algorithm that utilises the vector units efficiently is important as the baseline for parallelisation, and that is why we will apply the following optimisations, but the other aspect of this study is to identify the pros and cons of parallel programming models in parallelising the optimised code.

Here, we consider the single-pass algorithm with 4 nested loops as the naive code. It is important to note that since this algorithm convolves image array A to B, at the end of the algorithm we copy back B to A to have the original image convolved. To make sure that the naive code does not utilise automatic vectorisation, the code should be compiled with the `-no-vec` flag (although, the Intel compiler failed to auto-vectorise our naive code).

Opt-1: Single-pass, Unrolled The first optimisation is loop unrolling. An average (among the 3 images) benefit of 2.5× can be obtained by hand unrolling the nested loop over kernel into 25 multiplications. At this stage we change the statement in Eq. 8.2 inside the kernel nested loop into 25 additions in the form of Eq. 8.3.

$$B[pId][i][j] += A[pId][i+kx-2][j+ky-2] * K[kx][ky]; \tag{8.2}$$

$$\begin{aligned} B[pId][i][j] = A[pId][i-2][j-2] * K[0][0] + \\ A[pId][i-2][j-1] * K[0][1] + ... + \\ A[pId][i][j] * K[2][2] + ... + \\ A[pId][i+2][j+2] * K[4][4]; \end{aligned} \tag{8.3}$$

Opt-2: Single-pass, Unrolled, Vectorised After unrolling the kernel loops, only 2 out of 4 initial loops remain. Utilising the compiler technology, we can enforce inner loop vectorisation using `#pragma simd`, which if memory alias or dependence analysis fails,

gives hint to the compiler that the loop is safe to be vectorised [1]. Vectorisation after unrolling gave us an average speedup of 22× over the baseline.

Opt-3: Two-pass, Unrolled The third optimisation is an algorithmic change due to the fact that the kernel is separable, hence instead of 25 multiplication for each pixel (as a result of a 5×5 kernel nested loop), we can use a horizontal 1D convolution followed by a 1D vertical convolution. Therefore, the number of multiplications for each pixel becomes $5 + 5 = 10$. This optimisation is first combined with loop unrolling. Each of the two loops to be unrolled in this case (one in the horizontal pass and the other in the vertical pass) has the size of 5. An average speedup of 5.5× over the baseline can be obtained at this stage.

Opt-4: Two-pass, Unrolled, Vectorised Repeating the second optimisation on the inner loops of the two-pass algorithm (i.e. those over the image columns), we can now get an average of 47.1× performance gain over the baseline, and we have just optimised the sequential code so far.

Par-1: Single-pass, Unrolled, 100 OpenMP Threads OpenMP provides the simplest way of parallelising the outer loop of the single-pass algorithm. We obtained an average of 191.1× speedup over the baseline.

Par-2: Single-pass, Unrolled, Vectorised, 100 OpenMP Threads On top of the previous optimisation, similar to "Opt-2", we have enforced vectorisation on the inner loops over the image columns (for both convolution computation and the copy-back operation). Apart form that, the outer loops over the image rows are parallelised using `#pragma omp parallel for`. An average performance gain of 1268.8× over the baseline has been achieved.

Par-3: Two-pass, Unrolled, 100 OpenMP Threads Parallelised version of the two-pass algorithm provides an average of 393.7× speedup over the baseline. This is almost 2.1× the speedup of the competitive algorithm in "Par-1".

Par-4: Two-pass, Unrolled, Vectorised, 100 OpenMP Threads The best parallelised vectorised approach has the average speedup of 1611.7×. This is only 1.3× the speedup of the competitive algorithm in "Par-2". This shows that the single-pass algorithm can benefit more from vectorisation when parallelised. This is an important finding and we

[1] This always requires extra care, as enforcing SIMD vectorisation while there is vector dependence results in incorrect results

will see in section 8.3.5 that it helps another version of the single-pass algorithm (without copy-back) to outperform the two-pass algorithm with 100 threads.

The speedup results for all the stages from naive to a parallelised optimised code are illustrated in Figure 8.3.

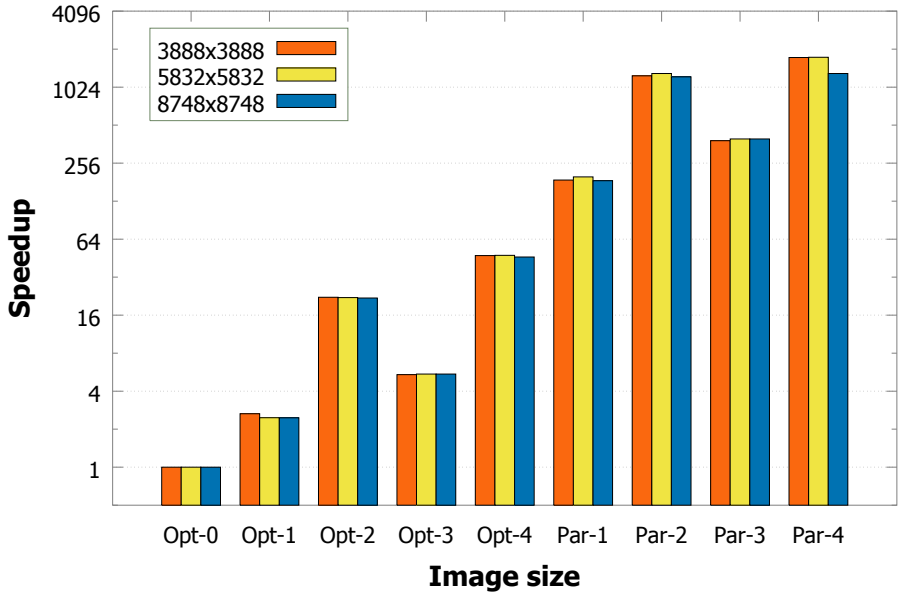


Figure 8.3: From Naive to Parallelised Optimised code on the Xeon Phi

Baseline: single-pass algorithm **with** copy-back to source
Opt-0: Naive, Single-pass, No-vec
Opt-1: Single-pass, Unrolled, No-vec
Opt-2: Single-pass, Unrolled, SIMD
Opt-3: Two-pass, Unrolled, No-vec
Opt-4: Two-pass, Unrolled, SIMD
Par-1 : Single-pass, Unrolled, No-vec, 100 omp threads
Par-2 : Single-pass, Unrolled, SIMD, 100 omp threads
Par-3 : Two-pass, Unrolled, No-vec, 100 omp threads
Par-4 : Two-pass, Unrolled, SIMD, 100 omp threads

8.3.2 OpenMP Implementation Details

An OpenMP implementation of the image convolution algorithm is shown in List. 8.1.

This code corresponds to the last stage of the optimisations, “Par-4”, as it implements the two-pass algorithm with a horizontal pass followed by a vertical pass; the kernel loop is unrolled, `#pragma simd` is used to enforce SIMD vectorisation, and the outer loop is parallelised.

It is worth stating that `#pragma omp parallel for` has an implicit barrier at the end.

```
1 /* 2D convolution on each plane */
2 void twoPassConv(float ***A, float ***B, float *k, int planeId, int
      rows, int cols) {
3  // horizontal pass
4 #pragma omp parallel for
5  for(int i=2; i<rows-2; i++) {
6   #pragma simd
7    for(int j=2; j<cols-2; j++) {
8     B[planeId][i][j] =
9               A[planeId][i][j-2] * k[0] +
10              A[planeId][i][j-1] * k[1] +
11              A[planeId][i][j] * k[2]   +
12              A[planeId][i][j+1] * k[3] +
13              A[planeId][i][j+2] * k[4];
14   }
15 }
16 // vertical pass
17 #pragma omp parallel for
18 for(int i=2; i<rows-2; i++) {
19  #pragma simd
20   for(int j=2; j<cols-2; j++) {
21    A[planeId][i][j] =
22              B[planeId][i-2][j] * k[0] +
23              B[planeId][i-1][j] * k[1] +
24              B[planeId][i][j] * k[2]   +
25              B[planeId][i+1][j] * k[3] +
26              B[planeId][i+2][j] * k[4];
27   }
28 }
29 return;}
30
31 /* calls twoPassConv on each plane */
32 void conv (float ***A, float ***B, float *ker, pimage a) {
33 #pragma novector
34 for (int planeId= 0; planeId< a.planes; planeId++) {
35  twoPassConv(A, B, ker, planeId, a.rows, a.cols);
36 }
37 return;}
```

Listing 8.1: Two-pass Image Convolution Algorithm, OpenMP

8.3.3 GPRM Implementation Details

The GPRM implementation of the two-pass algorithm defines the two phases of the algorithm as two different types of tasks. Since all the tasks defined in the GPC code will be executed in parallel, a `seq` pragma is required to run the two phases sequentially, as shown in Listing 8.3.

```
#include "Conv.h"

void Conv::horizPass(ind, CUTOFF, ...) {
 par_cont_for(2, rows-2, ind, CUTOFF, this, &Conv::horizPassInnerLoop,
    ...);
}

void Conv::vertPass(ind, CUTOFF, ...) {
 par_cont_for(2, rows-2, ind, CUTOFF, this, &Conv::vertPassInnerLoop,
    ...);
}
```

Listing 8.2: Two-pass Image Convolution Algorithm, GPRM Task Code

```
#include "GPRM/Task/ConvTask.h"

void horizontalTasks (const int CUTOFF, ...) {
#pragma gprm unroll
 for (int ind=0; ind < CUTOFF ; ind++) {
   horizPass(ind, CUTOFF, ...); }
}

void verticalTasks ((const int CUTOFF, ...) {
#pragma gprm unroll
 for (int ind=0; ind < CUTOFF ; ind++) {
  vertPass(ind, CUTOFF, ...); }
}

void GPRM::ConvTask::twoPassConv(...) {
#pragma gprm seq
 {
   horizontalTasks(100, ...);
   verticalTasks(100, ...);
 }
}
```

Listing 8.3: Two-pass Image Convolution Algorithm, GPC Code

We specify the number of tasks using a `#pragma gprm unroll` followed by a `for` loop of size `CUTOFF`. Each phase uses a *partial continuous for*, `par_cont_for` [29], in order to parallelise the outer loop over rows (as shown in Listing 8.2), and a `#pragma simd` [2] to help the compiler vectorise the inner loop over columns. As stated in the previous chapters, `par_cont_for` is a sequential *for* loop that works as follows:

In GPRM, multiple instances of the same task are generated (specified by `CUTOFF` in the List. 8.2), each with a different index (similar to the *global_id* in OpenCL). Each of these tasks calls `par_cont_for` passing their own index to specify which parts of the work should be performed by their host thread.

8.3.4 Parallel Performance of the Two-pass Algorithm

The focus of this section is on the parallel performance of the OpenMP and GPRM implementations of the two-pass algorithm.

We start by disabling the vectorisation in the Xeon Phi. The results for the parallelised non-vectorised cases are compared with the vectorised ones in Table 8.1. In order to disable vectorisation for OpenMP and GPRM, the code should be compiled with the `-no-vec` flag.

Table 8.1: Vectorisation effect on the parallel performance (ms) of the two-pass algorithm

Image Size	OpenMP no-vec	GPRM no-vec	OpenMP SIMD	GPRM SIMD
1152x1152	3.9	27.2	0.8 (4.9×)	26.1 (1.0×)
1728x1728	8.5	32.8	2.0 (4.2×)	26.6 (1.2×)
2592x2592	16.7	40.5	4.1 (4.1×)	27.8 (1.5×)
3888x3888	39.9	60.4	8.8 (4.5×)	32.5 (1.9×)
5832x5832	86.7	105.8	19.6 (4.4×)	36.8 (2.9×)
8748x8748	195.4	216.9	59.2 (3.3×)	60.1 (3.6×)

The average speedup obtained through vectorisation for the OpenMP code is about 4.2×. It is important to note that this speedup for the sequential code was almost twice as much (8.6×). Therefore, the reported performance gain is specific to the case with 100 threads and should not be generalised.

It is worth noting that the speedup due to vectorisation in GPRM is much less pronounced, mostly due to the higher overhead of the GPRM runtime for smaller images.

Figure 8.4 shows the speedup of the two-pass algorithm against its optimised sequential implementation (i.e. version "Opt-4"). So far, the algorithm is parallelised over each plane of size R×C, hence **R**×**C** in Fig.8.4. This means for 3 colour planes, the parallelised code

[2]Unlike OpenMP, in this case the use of `#pragma simd` for the innermost loop in the GPRM implementation is optional

will be executed 3 times sequentially [3].

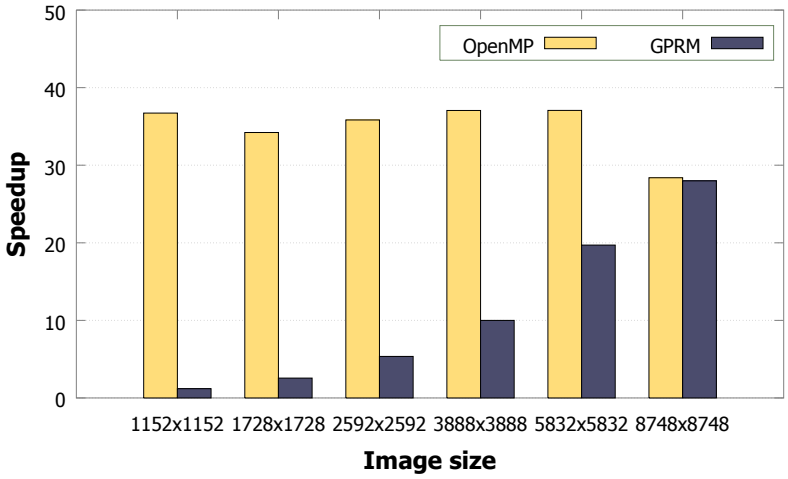


Figure 8.4: Speedup of the Vectorised Two-pass Algorithm, **R**×**C**

Similar to the previous section, it is possible to inspect the difference between OpenMP and GPRM more in detail and measure the overhead of communication between tiles in GPRM. If we deduct this overhead from the running time, we can measure the time spent on the actual computation inside the framework. The GPRM-compute time shown in Table 8.2 is gained by deducting the constant communication overhead of 25.5ms from the total execution time.

Table 8.2: Running time (ms) per image for the two-pass algorithm

Image Size	OpenMP	GPRM-total	GPRM-compute
1152x1152	0.8	26.1	0.6
1728x1728	2.0	26.6	1.1
2592x2592	4.1	27.8	2.3
3888x3888	8.8	32.5	7.0
5832x5832	19.6	36.8	11.3
8748x8748	59.2	60.1	34.6

As a solution to mitigate the GPRM overhead, we have used task agglomeration and considered images with the width of 3 times the width of the original images, meaning that each row includes information for all 3 colour planes. Using this technique, the size of tasks in GPRM becomes tripled and the overhead becomes one third (8.5ms per image). The speedup results for this case, which we call **3R**×**C** is shown in Fig. 8.5. As expected, this technique does not have similar significant impact on the OpenMP performance on the Xeon Phi.

Since the convolution benchmark runs much faster on the Xeon Phi, the small overhead of the GPRM framework becomes significant. Task agglomeration can reduce this overhead and provide a notable performance improvement. However, except for the largest image,

[3] Actually it is 3000 times, considering that we run the code 1000 times

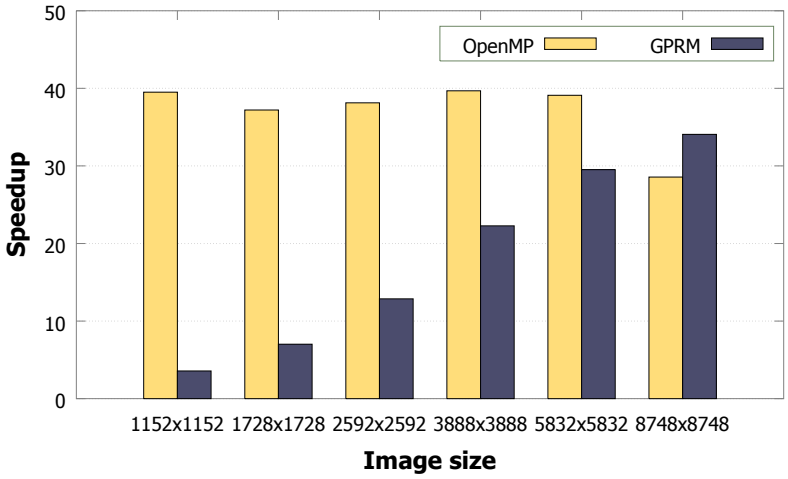


Figure 8.5: Speedup of the Vectorised Two-pass Algorithm, **3R**×**C**

OpenMP has better performance.

8.3.5 Reconsidering the Single-pass Algorithm

In order to compare the single-pass and two-pass algorithms, it is important to note that the two-pass algorithm uses an auxiliary array to store the result of the first pass. In the second pass, it uses the auxiliary array as the source and the original image as the destination, thus at the end of the algorithm, the original image will be replaced by the convolved one. It is convenient that the input and output images can use the same array, but it comes at a price: two assignment operations rather than one for every pixel. In order to have a fair comparison, we expected the same from the single-pass algorithm, i.e. overwriting the original image. This means that although the single-pass algorithm can produce the result on an output image by assigning new values for all the pixels only once, it now needs to copy the convolved values back to the original image.

This copy-back operation constitutes a considerable extra overhead and sometimes is not needed, e.g. when working with the Xeon Phi as a co-processor. Suppose one runs some complex code on the Xeon CPU and offloads the computation of the convolution to the Xeon Phi, e.g. the typical model for an OpenCL program. In that model, there will be host-to-device and device-to-host copy cost. If one copies an image array A to the Xeon Phi, convolves it into an array B and copies that back to the host, there is of course no need to copy on the data back to the original array on the device itself. Consequently, we have also tested the single-pass code without the ultimate "copy back to the original image" operation. We have measured the results again only for the three larger images.

After unrolling the kernel loop(s), for both non-vectorised and vectorised approaches, the results were as expected, i.e. the Two-pass algorithm had much better performance than the

Single-pass algorithm (1.6× to 1.9×).

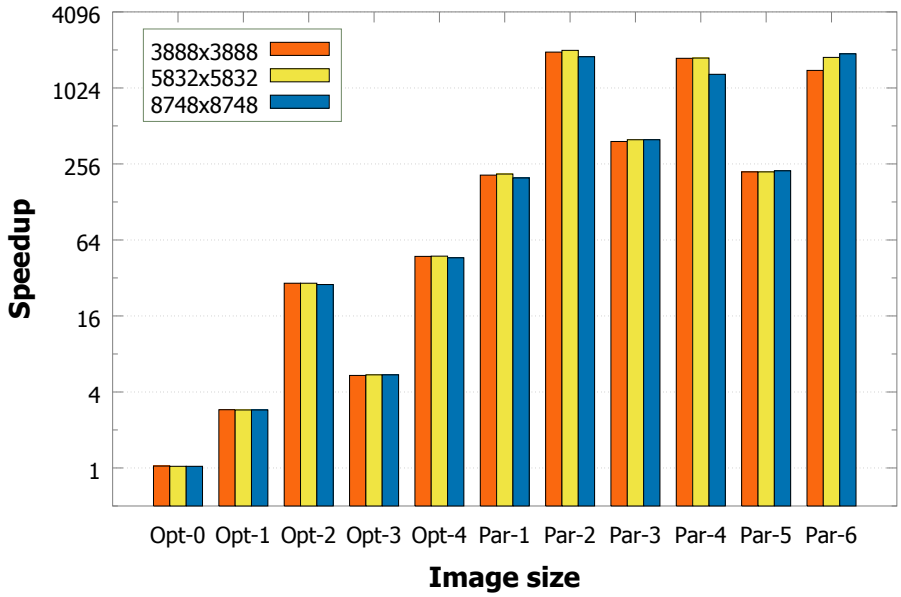


Figure 8.6: From Naive to Parallelised Optimised code on the Xeon Phi

Baseline: single-pass algorithm **without** copy-back to source
Opt-0: Naive, Single-pass, No-vec
Opt-1: Single-pass, Unrolled, No-vec
Opt-2: Single-pass, Unrolled, SIMD
Opt-3: Two-pass, Unrolled, No-vec
Opt-4: Two-pass, Unrolled, SIMD
Par-1 : Single-pass, Unrolled, No-vec, 100 omp threads
Par-2 : Single-pass, Unrolled, SIMD, 100 omp threads
Par-3 : Two-pass, Unrolled, No-vec, 100 omp threads
Par-4 : Two-pass, Unrolled, SIMD, 100 omp threads
Par-5 : Single-pass, Unrolled, No-vec, 100 GPRM tasks, 3R×C
Par-6 : Single-pass, Unrolled, SIMD, 100 GPRM tasks, 3R×C

It is worth mentioning that in some cases, e.g. for 3888×3888 images, the performance of the optimised single-pass algorithm with OpenMP could be improved by up to 15% (10% in average for the largest three images) by tuning the number of threads, e.g. with 120 threads, but since we decided to run experiments with 100 threads or tasks and we do not intend to compare multiple configurations together, we stick to this number.

Figure 8.6 shows that although the average sequential performance of the optimised two-pass code is 1.6× better than the average sequential performance of the optimised single-pass code (without copy-back), the average parallel performance of the optimised single-pass code (using OpenMP) is 1.2× better than that of the optimised two-pass code. The reason

can also be extracted from Fig. 8.6: better utilisation of the vector units by the parallel single-pass code (9.4× its parallel non-vectorised version) compared to the parallel two-pass code (4.1× its parallel non-vectorised version).

Since GPRM had shown a good performance for the largest array when we included parallelisation over planes into the tasks (the **3R**×**C** case), its results has been added to Fig. 8.6. As expected, it produced the best result for the 8748×8748 image, using the optimised single-pass algorithm with no copy-back. Its speedup over the baseline naive code is 1850× with 100 tasks.

As the best result amongst all, we have been able to get up to 1970× (for the 5832×5832 image) speedup over the sequential naive implementation of the algorithm, by only utilising the compiler technology, few algorithmic changes, and parallelisation (using OpenMP). Also, 2160× speedup over the baseline has been observed with 120 OpenMP threads for 5832×5832 matrices with single-pass, no-copy approach.

8.4 Related Work

A similar 5×5 spatial kernel (filter) has been the focus of a number of research papers [168] [167] [30] [169].

Petersen et al. [168] ported a subset of C benchmarks to Haskell and measured their performance on parallel machines, including the Xeon Phi. Considering three classes of naive, optimised, and Ninja C implementations [167], our implementation of the image convolution algorithm is classified as the optimised code, utilising loop unrolling and SIMD vectorisation.

The reported Ninja gap for the Intel Labs Haskell Research Compiler (HRC) for 8192×8192 images on the Xeon Phi using the single-pass algorithm is 3.7× (for 57 threads) [168]. The authors have disabled multithreading on the Xeon Phi, which is essentially different from hyper-threading on the Xeon processors [2].

We explored this further in [30] and figured out that the peak performance can be achieved with 100 threads. We have also reported that the performance gap between the Vector Pascal [30] and an optimised OpenMP implementations of the two-pass algorithm with 100 threads is almost 6.4×.

Authors in [167] also focused on the optimisation techniques for parallel applications, using both advancements in compiler technology and algorithmic techniques to bring down the Ninja performance gap for throughput computing benchmarks, one of which is the single-pass implementation of the convolution algorithm.

Tian et al. [169] focused on efficient utilisation of the SIMD vector units on the Xeon Phi and proposed a number of effective techniques to improve the performance of parallel programs, including a single-pass image convolution. They have reported a speedup of 2000× using their vectorisation techniques along with parallelisation. We observed a speedup of about 1970× (2160× with 120 OpenMP threads) without using any particular vectorisation technique. However, we have also highlighted the importance of the Xeon Phi vector units, specially their impact on parallel performance.

8.5 Summary

In this chapter, we applied GPRM and OpenMP to solve a 2D image convolution problem over a test set of 6 square images, ranging from 1152×1152 to 8748×8748 on both TILEPro64 and Xeon Phi.

For a separable convolution kernel, two different algorithms can be considered: Single-pass, which requires only a single assignment instead of two, but needs an additional copy if the result is required in the original array, and Two-pass, which requires fewer computations and returns the result in the original array.

We used the two-pass algorithm as the baseline on the TILEPro64. GPRM outperformed OpenMP on this platform in all cases. In terms of productivity, GPRM naturally fits algorithms with task (functional) decomposition. It has its own complications though when it comes to domain decomposition, as it requires restructuring certain parts of the program to fit the GPRM structure.

We have explored a number of optimisation and parallelisation techniques on the Xeon Phi which helped us achieve a speedup near 2000× over the baseline, but none of these techniques requires a major rewrite of the original code. The optimisation techniques include loop unrolling, vectorisation, and an algorithmic from single-pass to two-pass or vice versa.

After creating optimised versions of both algorithms on the Xeon Phi, we found that the choice between these algorithm depends on which version of the single-pass algorithm is required: if the result has to be copied back to the original image, then the two-pass algorithm is always better. Otherwise, the single-pass algorithm can provide better parallel performance, even though its sequential performance is still worse. This is due to the fact that the single-pass algorithm can benefit more from vectorisation when parallelised.

Task agglomeration is also used as a parallelisation technique to improve the performance of GPRM on both platforms. The GPRM model has a fixed overhead (tens of milliseconds for hundreds of tasks) due to task creation and communication. Using images with the width of 3× the width of the original images (such that each row contains the information of all 3

colour planes), we reduced the overhead to approximately one third.

In terms of performance on the Xeon Phi, OpenMP is the winning model, except for very large images where GPRM shows better performance after using task agglomeration.

Although we observed an unexpected behaviour of GPRM on the Xeon Phi, we were able to reason about its overhead and decrease it to some extent. Because of the modular design of GPRM, measuring such an overhead is straightforward. For example, in this case, we measured a fixed overhead of 8.5ms for running the benchmark with 200 empty tasks on the Xeon Phi (after agglomeration). It is obvious that this overhead would be dominant for a benchmark that takes couple of milliseconds to run. However, if the tasks are large enough to be worth evaluating in parallel, the same scheduling strategy would result in better performance than OpenMP, as observed in the case of the largest image.

Chapter 9

Parallel Linked List Processing

In this chapter, we continue to highlight the differences between GPRM and the most commonly used parallel programming approach, OpenMP.

OpenMP is a very successful parallel programming API, but efficient parallel traversal of a list (of possibly unknown size) of items linked by pointers is a challenging task: solving the problem with OpenMP worksharing constructs requires either transforming the list into an array for the traversal (e.g. by using the OpenMP `for`), or for all threads to traverse each of the elements and compete to execute them (e.g. by using the OpenMP `single`). Both techniques are inefficient. OpenMP 3.0 allows to addresses the problem using pointer chasing by a master thread and creating a task for each element of the list. These tasks can be processed by any thread in the team.

In this chapter, we propose a more efficient cutoff-based linked list traversal. We compare the performance of this technique in both GPRM and OpenMP implementations with the conventional OpenMP implementation, which we call Task-Per-Element (TPE).

Pointer chasing or list traversal are the names that have been used for this problem [170] [171]. The problem is defined in [170] as traversing a linked list computing a sequence of Fibonacci numbers at each node. As another example, we consider traversing a linked list and sorting a small array of integer numbers (up to a thousand numbers) at each node by a Quicksort algorithm. It is worth mentioning that we use already sorted arrays, which increases the time complexity of the Quicksort algorithm to $O(n^2)$ [172].

Massaioli et al. [173] have investigated the performance of up to 32 OpenMP threads for processing dynamic lists managed by pointers in a simulator of financial markets. They have concluded that if the overhead is not dominant, meaning that the unit of work is not too small, using `omp single nowait` directive appears to be suitable. Also in [170], the performance of the TPE method has been evaluated on a multicore CPU. In this chapter, however, we focus on the challenges arising in manycore systems such as the TILEPro64

with 63 available hardware cores or the Intel Xeon Phi with 240 logical cores. We will demonstrate that when the tasks are tiny and/or the lists are large, the overhead of the TPE method becomes significant.

As always, all the benchmarks are implemented as C++ programs, and all speedup ratios are computed against the running time of the sequential code implemented in C++. The compiler flags and system setup are as described in Section 6.1.1.

9.1 An Efficient Linked List Processing Technique

The agglomeration methods and their role to reduce the overhead of task creation are discussed [83] and [24]. We use this concept again to propose an efficient parallel pointer chasing technique.

To traverse a list (of unknown size) similar to the problem discussed in [3], we propose to limit the number of tasks to a cutoff value. Assuming there are `CUTOFF` chunks of work, the chunk with id `k`, gets the head of list, goes `k` steps further to its starting point, processes the element, and then goes `CUTOFF` steps further to process its next element. It continues to jump `CUTOFF` steps until the end of the list. The implementation of the TPE processing (creating one task per element) of linked lists as well as the implementations of the proposed method in both OpenMP and GPRM is shown in Listing. 9.1.

The GPRM implementation of the `par_list` is similar to a single-threaded version of the OpenMP code (i.e. without pragmas). Wherever the size of the list is known, a GPRM `par_cont_list` with a similar implementation to a `par_cont_for` [29] can also be used.

In order to visualise how the tasks are assigned to the threads, consider a scenario with 3 threads (specified with 3 different colours in Fig. 9.1) and a linked list of 10 elements. However, suppose that the number of elements are not known, therefore we cannot partition the list into continuous chunks.

(a) TPE, A possible assignment of 10 tasks to the threads

(b) OpenMP with cutoff 4, A possible assignment of 4 tasks to the threads

(c) GPRM with cutoff 4, The initial assignment of 4 tasks to threads

Figure 9.1: Task-to-thread assignment in different implementations

```
1 /* TPE (Task-Per-Element)*/
2 #pragma omp parallel
3 {
4  #pragma omp single
5  {
6    p = mylist->begin();
7    while(p!=mylist->end()) {
8     #pragma omp task
9     {
10      process(p);
11    }
12    p++;
13   }//of while
14  }//of single
15 }
16 /* ------------------------------------------------------------------------*/
17 /* OpenMP implementation of the method with cutoff */
18 #pragma omp parallel private(p) //p must be private
19 {
20  #pragma omp single
21  {
22   p = mylist->begin();
23   for(int i=0; i < CUTOFF; i++) {
24    process(p);
25    #pragma omp task //tied
26    {
27     while(p!=mylist->end()) {
28      int j=0;
29      while(j < CUTOFF && p!=mylist->end()) {
30       j++; p++;}
31      if(p!=mylist->end()) {
32       process(p);}
33    }//of while
34   }//of task
35   p++;}
36  }//of single
37 }
38 /* ------------------------------------------------------------------------*/
39 /* GPRM implementation of the method with cutoff */
40 #pragma gprm unroll
41 for(int i=0; i < CUTOFF; i++) {
42  par_list(i, CUTOFF, &MyProcess::process, mylist);}
```

Listing 9.1: TPE vs. cutoff-based linked list processing

For this problem, the GPRM task-to-thread assignment is not fixed at compile time, because

we might not know the number of list elements, and therefore we need to chase the pointers. Nevertheless, we are able to determine that if the list has 10 elements, the GPRM runtime system will assign the tasks to the available 3 threads in the same way as shown in 9.1(c). However, there is no guarantee that the threads execute all of their pre-assigned tasks. For instance, in this scenario, thread C (colour black) executing task 3 (composed of 2 elements) could finish its job faster than thread A (colour red) executing task 1 (composed of 3 elements). Task 4 will be still on the ready task queue of thread A. Therefore, thread C can steal task 4 and start executing it.

9.2 Experiments

As stated earlier, the problem we are targeting is defined as traversing a linked list sorting a small array of integers at each node. The purpose is to parallelise this problem on a manycore system.

We have considered two types of workloads: Balanced and Unbalanced, described in Listing. 9.2. Two different array sizes to sort are chosen: 63 (or 240) and 1000. 63 (or 240) is used to create a completely unbalanced workload on 63 (or 240) cores using the formula in Listing. 9.2. By using `*p % N` as the size of array to be sorted at each node, where `*p` is the linked list node ID, different nodes gets different arrays to sort. On the Xeon Phi for example, the maximum difference with the balanced workload can be seen for the initial configuration with `NTH=240` and `CUTOFF=240`. In this case, `th1` gets all arrays with size 1 to sort, while `th239` gets all arrays with size 239. Obviously, by changing the cutoff value to other numbers, the pattern changes, but still different elements of the list have different array sizes to sort (240 different sizes).

```
1 /* Balanced Workload */
2 //N is the size of the array
3 for(int i=0; i < N; i++) {
4   A->push_back(i);}
5 quickSort(*A, 0, N);
6 /* ------------------------------------------------------------------------*/
7 /* Unbalanced Workload */
8 //p is the pointer to the L-L node ID, thus *p is a unique integer
9 for(int i=0; i < *p % N; i++) {
10  A->push_back(i);}
11 quickSort(*A, 0, *p % N);
```

Listing 9.2: Creation of balanced and unbalanced workloads.

For both balanced and unbalanced workloads, the exact amount of work to be accomplished

by each thread depends on the size of the list, and the cutoff value. Therefore, the balanced workload does not necessarily mean that all threads get exactly the same work, but at least the amount of work to do at each node is the same. For the cutt-off based implementations, instead of finding the optimal cutoff value that has the minimal overhead that leads to a fair distribution of work, we focus on the efficiency of runtime load balancing in GPRM versus OpenMP using different cutoffs.

9.3 Results on the TILEPro64

We first compare the three approaches. For that purpose, we have used the cutoff values 63, and a large power of 2 number (2048) to allow for better distribution of tasks as well as more opportunities for task stealing. Workloads are defined as balanced or unbalanced arrays of size 63 or 1000.

9.3.1 Comparing all Implementations

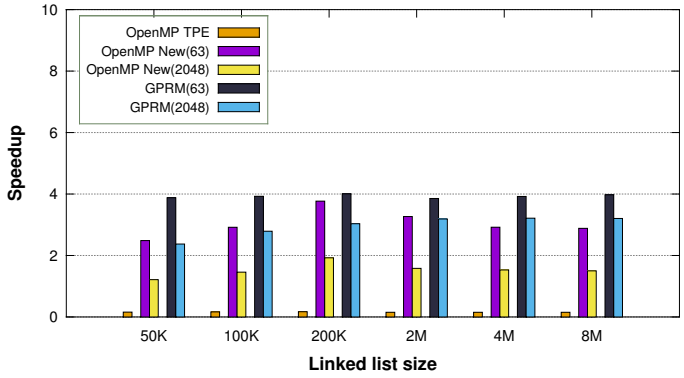
The results of comparing the three approaches on the TILEPro64 are shown in Fig. 9.2. Except from the balanced case with the arrays of size 1000, in all other cases, TPE (Task-Per-Element) performs poorly, and for the arrays of size 63 it even shows a slowdown over the sequential code. It is true that lists with such small arrays cannot be parallelised efficiently, but at least the cutoff-based approaches can provide a speedup of up to $4\times$. For the three smallest lists with the arrays of size 1000, the cutoff-based GPRM solution outperforms the similar OpenMP implementation significantly.

9.3.2 Comparing the Effect of Cutoff in OpenMP v.s. GPRM

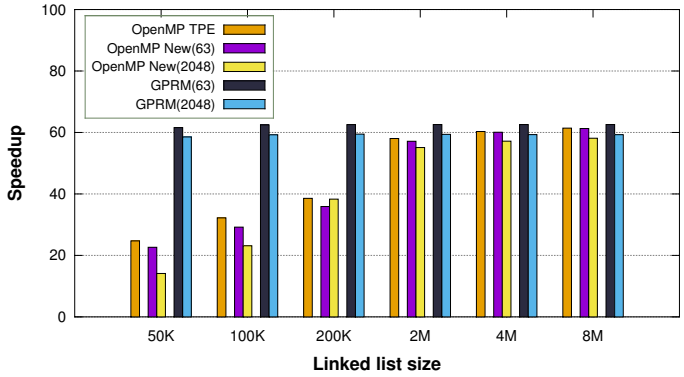
For the next round of comparison, we compare the cutoff-based implementations in GPRM versus OpenMP. For that purpose, we use a medium-sized list to skip the poor performance of the OpenMP version for small lists.

A detailed explanation of what happens inside GPRM is discussed in subsection 9.4.2, where the results on the Xeon Phi are discussed.

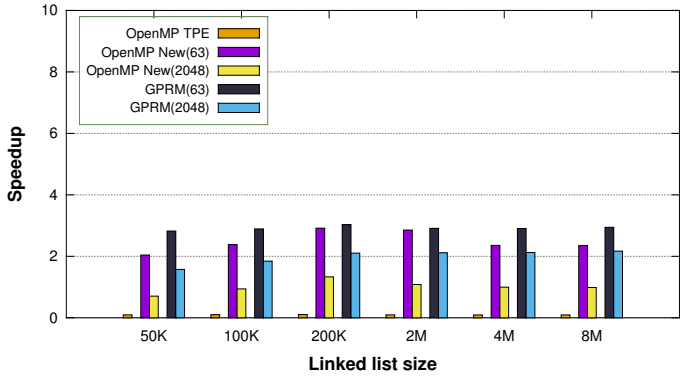
It can be observed from the charts in Figure 9.3 that for the linked list of size 200K on the TILEPro64, GPRM has better performance compared to the cutoff-based OpenMP approach in all cases.



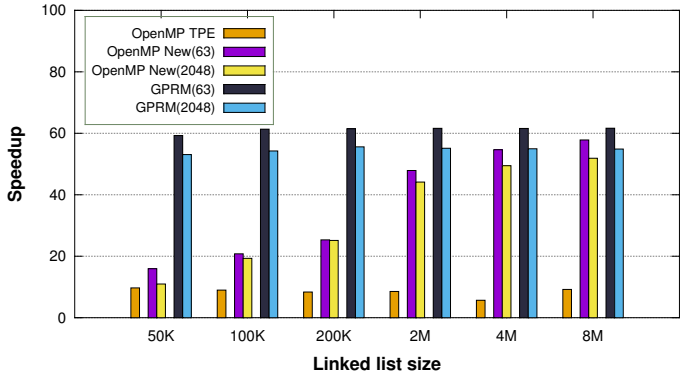
(a) Array of size 63, Balanced - Speedup on the TILEPro64



(b) Array of size 1000, Balanced - Speedup on the TILEPro64

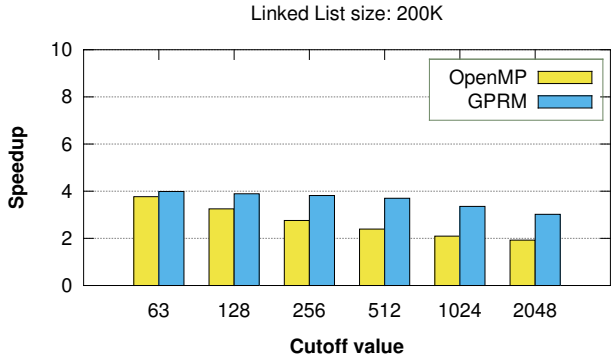


(c) Array of size 63, Unbalanced - Speedup on the TILEPro64

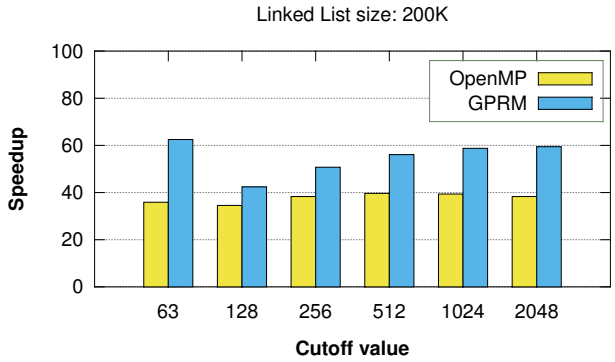


(d) Array of size 1000, Unbalanced - Speedup on the TILEPro64

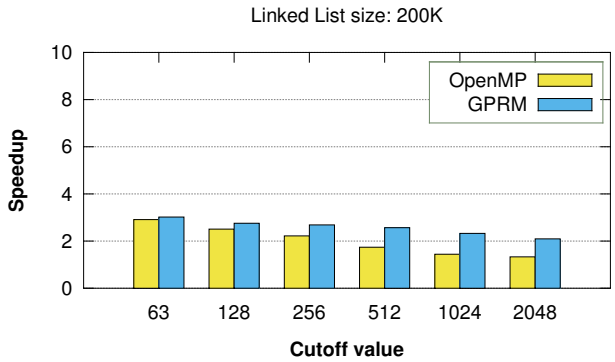
Figure 9.2: Speedup charts on the TILEPro64, cutoffs: 63, 2048



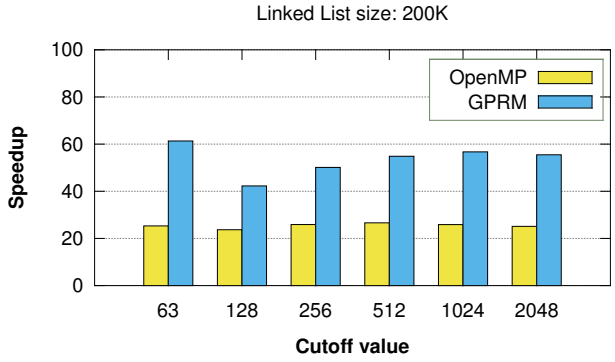
(a) LL size: 200K, Array: 63, Balanced - Varying cutoffs, TILEPro64



(b) LL size: 200K, Array: 1000, Balanced - Varying cutoffs, TILEPro64



(c) LL size: 200K, Array: 63, Unbalanced - Varying cutoffs, TILEPro64



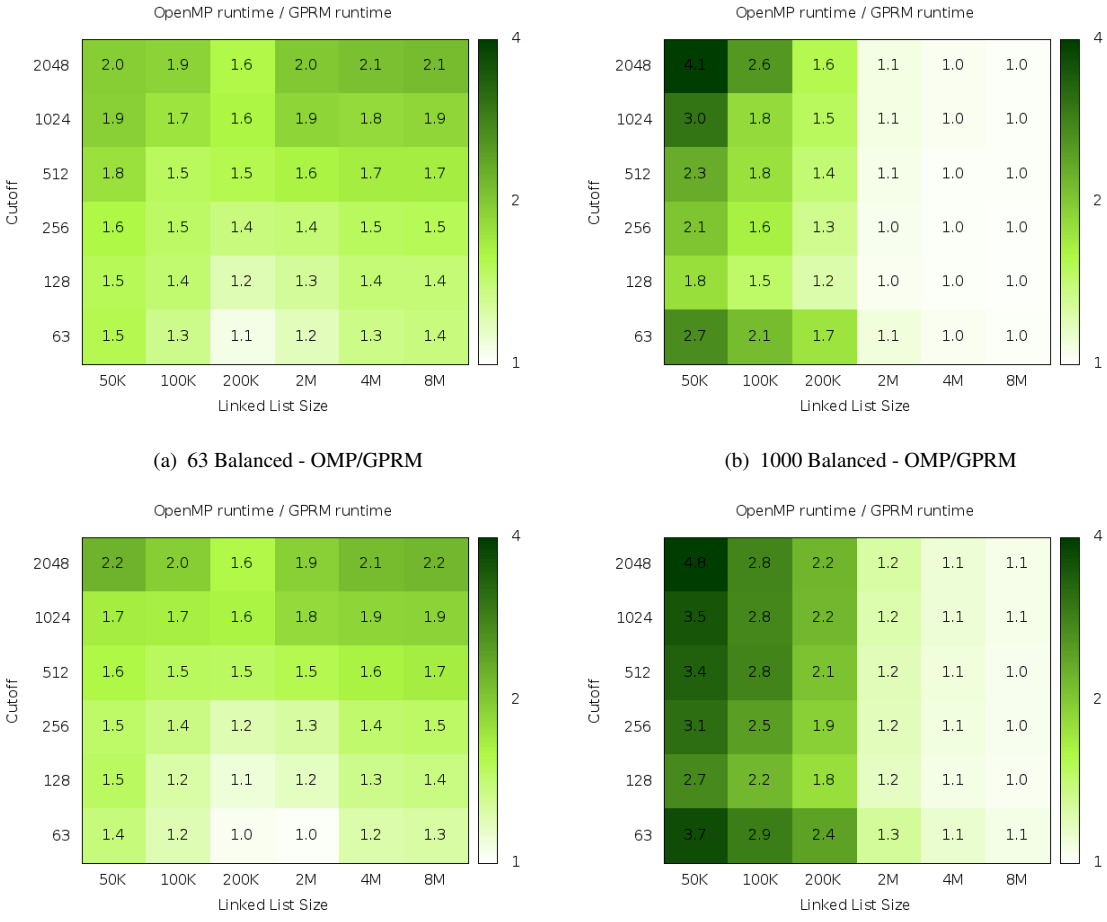
(d) LL size: 200K, Array: 1000, Unbalanced - Varying cutoffs, TILEPro64

Figure 9.3: TILEPro64: cutoff-based implementations of OpenMP v.s. GPRM, LL: 200K

9.3.3 Performance of the Cutoff-based Method in OpenMP v.s. GPRM

For a detailed comparison between GPRM ad OpenMP, we consider heat maps where the value for each cell is calculated as (`OpenMP runtime / GPRM runtime`). This way, we compare the cutoff-based implementations for all lists and all cutoff values.

For the arrays of 63 elements, the difference becomes larger when the cutoff value increases. For the arrays of size 1000, the difference is notable for smaller lists. In both cases, heat maps for the balanced and unbalanced cases have similar patterns.



(a) 63 Balanced - OMP/GPRM

(b) 1000 Balanced - OMP/GPRM

(c) 63 Unbalanced - OMP/GPRM

(d) 1000 unbalanced - OMP/GPRM

Figure 9.4: Performance comparison of the OpenMP and GPRM implementations of the cutoff-based method on the TILEPro64

9.4 Results on the Xeon Phi

For the cutoff-based OpenMP and GPRM implementations, we have used the cutoff 240, and also the cutoff 2048 to allow for better distribution of tasks as well as more opportunities

for task stealing. Although the bests results are not necessarily obtained with these cutoff values (as shown in Fig. 9.6), we aim to show that they are sufficient to outperform the TPE approach for both balanced (with cutoff 240) and unbalanced (with cutoff 2048) workloads.

9.4.1 Comparing all Implementations

The reason why the Task-Per-Element (TPE) approach is inefficient is once again evident from Fig. 9.5. It shows that for small tasks (of size 240), its best performance is about 65% of the peak for balanced workload and about 25% of the peak for the unbalanced one. For the larger tasks (of size 1000) and the linked list sizes greater than 100K, its performance is comparable with the cutoff-based GPRM implementation.

The cutoff-based OpenMP approach has shown unexpected behaviours when the list itself is small. In all cases, the performance for the 50K and 100K lists with cutoff 2048 is poor.

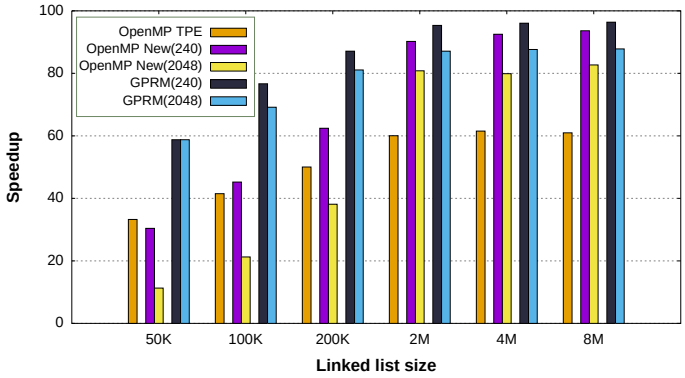
9.4.2 Comparing the Effect of Cutoff in OpenMP v.s. GPRM

Figure 9.6 shows that in all cases, GPRM has a better performance. Below is what happens in the background when using the GPRM approach:

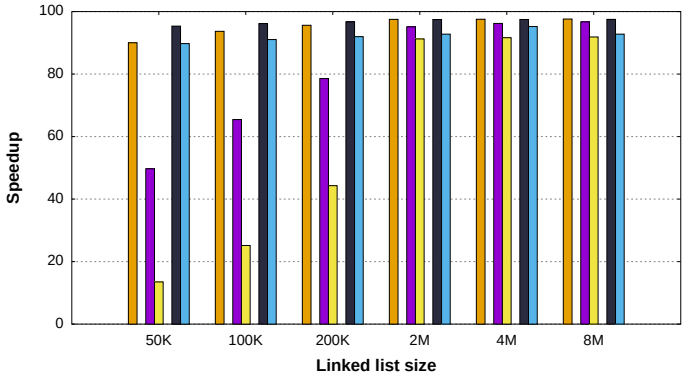
For the balanced workloads, we expect to see the best performance with cutoff 240. Cutoff 256 changes the fair distribution, as 16 threads have to handle the chunks 240 to 256. As the cutoff number becomes larger, more tasks will be created and hence the load can be balanced more efficiently, helping the runtime system to reach near the performance of of the initial case (with cutoff 240).

For the unbalanced workloads, we need to distinguish between the 240 and 1000 cases. In the case of 240, initially the workload is completely unbalanced with no chance of stealing. As the cutoff becomes larger, more opportunities for load balancing become available. Apparently the cutoff of 2048 in this case results in very fine-grained tasks, and imposes a small overhead to the system. As 1000 different array sizes are distributed between only 240 different workers, this situation is different from 240 and the workload is not as imbalanced as we expect. In order to have a similar situation, we have tested the same experiment with a value of 960 (which is a multiple of 240). Therefore, for the cutoff 240, thread `k` receives arrays with sizes `k`, `k+240`, `k+480`, and `k+720`, and the load is again unbalanced. In that case, the cutoff of 2048 has a visible performance improvement over the cutoff of 240 (10-15%) [1].

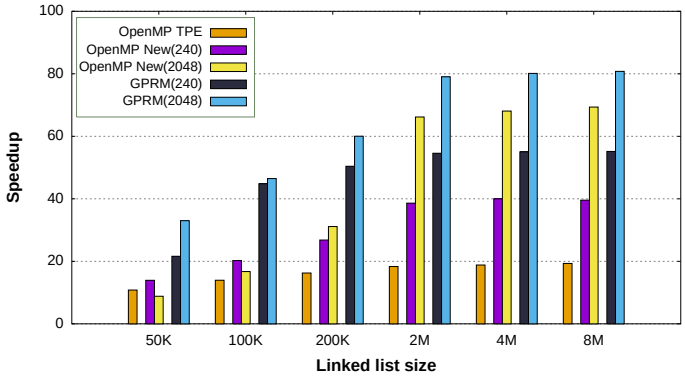
[1] We have intentionally not used the number 960 for the large arrays to show that our results are independent of the regular distribution of tasks on threads



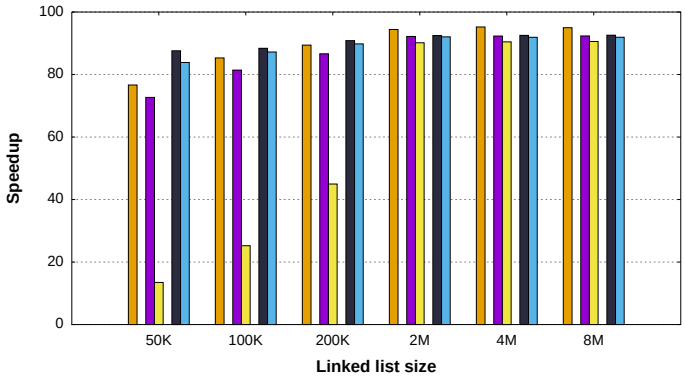
(a) Array of size 240, Balanced - Speedup on the Xeon Phi



(b) Array of size 1000, Balanced - Speedup on the Xeon Phi

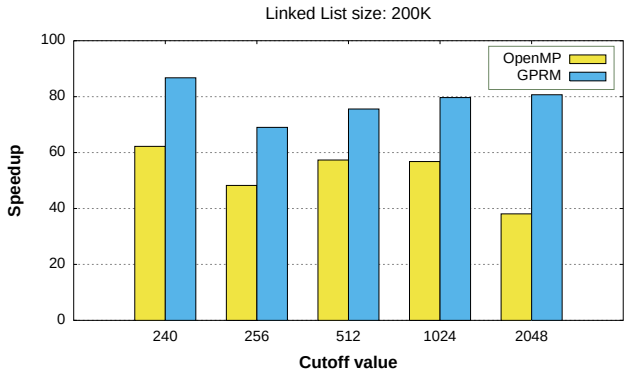


(c) Array of size 240, Unbalanced - Speedup on the Xeon Phi

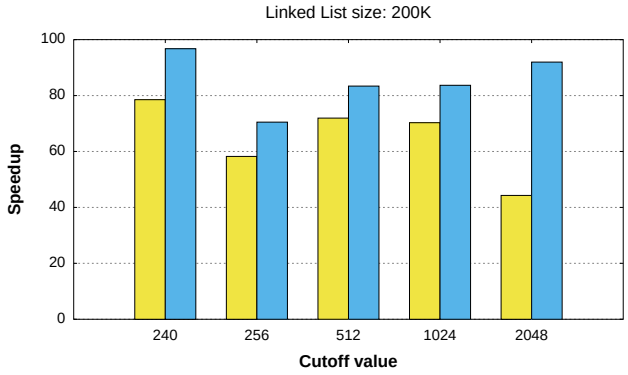


(d) Array of size 1000, Unbalanced - Speedup on the Xeon Phi

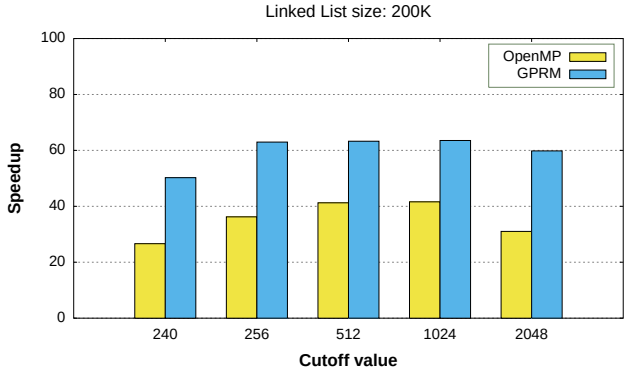
Figure 9.5: Speedup charts on the Xeon Phi, cutoffs: 240, 2048



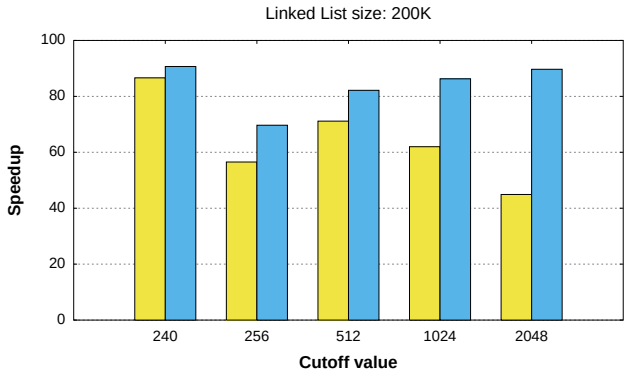
(a) LL size: 200K, Array: 240, Balanced - Varying cutoffs, Xeon Phi



(b) LL size: 200K, Array: 1000, Balanced - Varying cutoffs, Xeon Phi



(c) LL size: 200K, Array: 240, Unbalanced - Varying cutoffs, Xeon Phi



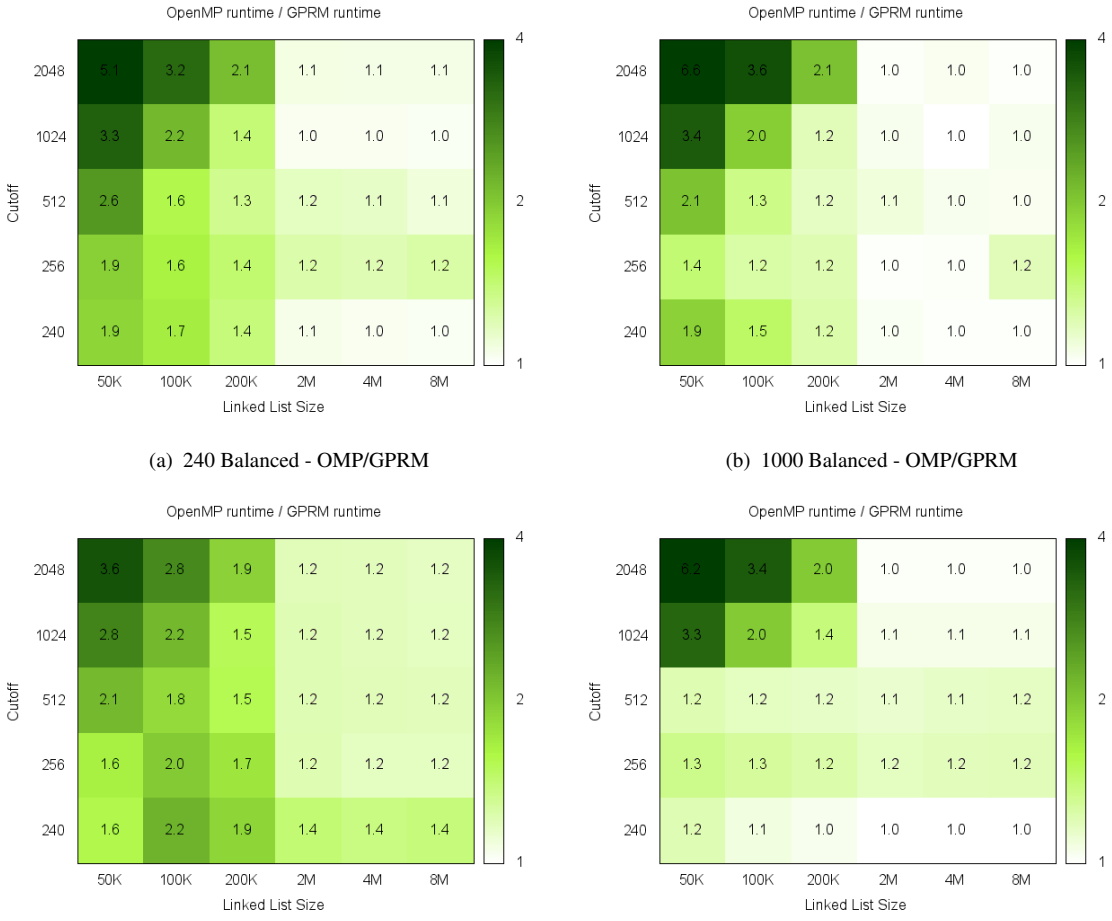
(d) LL size: 200K, Array: 1000, Unbalanced - Varying cutoffs, Xeon Phi

Figure 9.6: Xeon Phi: cutoff-based implementations of OpenMP v.s. GPRM, LL: 200K

9.4.3 Performance of the Cutoff-based Method in OpenMP v.s. GPRM

It can be observed that a large cutoff for small lists in the OpenMP implementation results in poor performance. It is true that the range of optimal cutoff values depends on the task granularity and the input data set [149], but at least for different cases of both balanced/unbalanced workloads here and for cutoff values up to 2048, we did not see a drastic slowdown with GPRM, as opposed to what we observed with OpenMP.

For the three smallest lists, GPRM outperforms OpenMP in almost all cases.



(a) 240 Balanced - OMP/GPRM

(b) 1000 Balanced - OMP/GPRM

(c) 240 Unbalanced - OMP/GPRM

(d) 1000 unbalanced - OMP/GPRM

Figure 9.7: Performance comparison of the OpenMP and GPRM implementations of the cutoff-based method on the Xeon Phi

One reason why the same implementation in OpenMP does not have consistent performance is that in OpenMP the tasks are not pre-assigned to threads, while in GPRM they are assigned to threads almost evenly at compile-time. Another reason is the differences between the stealing mechanisms, which themselves come from the differences between the parallel

execution models.

It is worth mentioning that we have also measured the speedup results for the balanced workloads against the sequential runtime on an E5-2620 (2.00GHz) Xeon processor. The best speedup results for the 240 and 1000 balanced cases using 12 threads (one per logical core) were 6.4× and 5.8× respectively. The parallel performance for these cases could only be improved up to 7.7× and 7.2× using the Xeon Phi (over the sequential runtime on the E5-2620 Xeon system).

9.5 Comparison of the TILEPro64 and the Xeon Phi

We have used the GPRM implementation of the linked list processing algorithm to compare the two platforms. Not only has GPRM the best results amongst the three implementations, but its performance is stable and more predictable. Therefore, it is fair to compare the platforms using GPRM. For that purpose, we have used the ratio of the Xeon Phi runtime over the TILEPro64 runtime for the workload size of 1000 in Table 9.1. The sequential runtime on the TILEPro64 is much better than that on the Xeon Phi [2]. It is important to recall that the Xeon Phi is not intended to target single-threaded scalar code, 32-bit data. The instruction pipelines have in-order superscalar architecture and are designed as two-cycle fully pipelined units. Therefore a hardware thread that is scheduled consecutively will stall in decode for one cycle, and as a result, single-threaded code could only achieve a maximum of 50% core utilisation [2].

Nevertheless, the parallel implementation scales better on the Xeon Phi and as can be observed from Table 9.1, the performance of the Xeon Phi for the parallel implementation comes close to that of the TILEPro64 (and for larger lists it even becomes better). It is important to note that the number of threads in this comparison is 240 for the Xeon Phi, and 63 for the TILEPro64. However, since the numbers of physical cores are close enough, we believe that this is a fair comparison.

Considering that the Xeon Phi has 240 cores, cutoff 512 is a too small for the stealing mechanism to be effective (almost 2 tasks on each logical core); hence for this case, the performance on the TILEPro64 is expected to be better. But all in all, although the sequential time and the speedup for different input sets are different, the best performance we can get for this benchmark from these platforms is almost similar.

[2] 1.6× for the sequential timing is a big difference. For example, for the balanced Linked List of 8M elements, 46520s on the Xeon Phi versus 28460s on the TILEPro64 means the difference of almost 5 hours. The performance difference for the same list after parallelisation with cutoff 2048 becomes 20 seconds.

Table 9.1: Performance comparison using GPRM: (XeonPhi runtime / TILEPro64 runtime)

	Cutoff value	**50K**	**100K**	**200K**	**2M**	**4M**	**8M**
Balanced workload (1000)	*Sequential (1)*	1.63	1.63	1.63	1.63	1.63	1.63
	512	1.12	1.10	1.10	1.13	1.19	1.20
	1024	1.13	1.14	1.15	1.10	1.13	1.09
	2048	1.07	1.06	1.06	1.05	1.02	1.03
Unbalanced workload (1000)	*Sequential (1)*	1.61	1.61	1.61	1.61	1.61	1.61
	512	1.12	1.10	1.07	1.06	1.06	1.05
	1024	1.10	1.07	1.06	1.04	1.03	1.03
	2048	1.02	1.00	0.99	0.96	0.96	0.96

9.6 Summary

We have proposed an efficient cutoff-based parallel linked list traversal method and demonstrated its advantages over the conventional implementation using OpenMP tasks, which we call Task-Per-Element (TPE).

Furthermore, we have shown that our task-based parallel programming model, GPRM, makes it easier to traverse the elements of a linked list in parallel (even if the size of the list changes at runtime). The cutoff-based implementation in GPRM results in superior performance compared to both of the OpenMP implementations (TPE and cutoff-based) in almost all cases, but most dramatically in the case of smaller lists.

The performance of the cutoff-based method in GPRM demonstrates that using a `par_list` construct which controls the number of tasks and determines the initial task to thread mapping, combined with a low-overhead load balancing technique can lead to efficient parallel linked list processing on manycore processors.

At the end of this chapter, we compared the GPRM results on the TILEPro64 with those on the Xeon Phi. Such a comparison showed that huge differences between the sequential timing on the TILEPro64 and the Xeon Phi can be completely removed for the parallel implementations, because of the higher parallelisation opportunities offered by the Xeon Phi and utilised by GPRM.

Chapter 10

Conclusion and Future Work

Use of parallel platforms is growing in every computing domain. In this thesis, we discussed that clock frequency and voltage scaling, complexity of superscalars, power and heat management as well as wire delays are some of the most important reasons behind the shift towards multicore and manycore processors. The "Free Lunch" is over and it is time for software developers to think parallel and learn how to utilise the potential of such architectures. We therefore reviewed some of the most popular models, APIs, and runtime libraries for programming manycore processors. We posit that task-based parallel programming with a higher level of abstraction as compared to thread-based programming is key to high performance.

For the purposes of this study, we considered the Tilera TILEPro64 and the Intel Xeon Phi platforms with 64 and 60 physical cores on a single chip, respectively. The TILEPro64 is designed to support a wide range of compute-intensive applications such as advanced networking applications. The Xeon Phi on the other hand, is a coprocessor that is designed to enhance the performance of the Xeon processors. The suitable applications are those which scale well on the Xeon processors and can utilise vector units and memory bandwidth of the Xeon Phi. The two manycore platforms have similarities such as the number of physical cores, per-core caches, and close clock frequencies, which make them interesting to compare. Compared to the TILEPro64, the Xeon Phi has 8$\times$ larger L2 caches, 4-way multithreading, vector units and floating point units. The TILEPro64 on the other hand, supports L3 caches, has a three-way VLIW architecture, but does not have any VPU or FPU, and we presented that for applications that can make use of such units, the Xeon Phi could be significantly faster, for instance about 100$\times$ for a naive float matrix multiplication benchmark.

We introduced our novel task-based model for programming shared-memory manycore processors, called the Glasgow Parallel Reduction Machine (GPRM). GPRM model is well suited to parallel evaluation, but the tasks should be coarse-grained enough to be worth evaluating in parallel. We also discussed that controlling the granularity of tasks is one of the

key matters for all task-based parallel programming models and is not specific to GPRM.

10.1 Conclusion

Modern parallel systems can provide outstanding performance, but programming them is far from easy. The main contribution of this thesis is the design and development of GPRM as a task-based parallel programming approach, which targets both regular and irregular parallel problems. GPRM fits naturally to the systems with per-core caches, which consequently makes it very promising for manycore processors. Our objective is to provide a low-overhead solution which can be efficient in both uniprogramming and multiprogramming situations on a manycore processor.

Without a precise cost model, the programmer cannot determine the optimal number of threads for a parallel application. The ideal situation for the programmer is to only have to express parallelism, relying on the runtime system to achieve the expected speedup with the default number of threads (as many as the number of cores). We have shown that GPRM delivers this ideal in almost all cases, and where it does not, it comes very close.

The PCAM design methodology, consisting of the Partitioning, Communication, Agglomeration, and Mapping phases was described in the Background chapter. We then showed how the PCAM methodology can be applied in the design of GPRM programs. In the partitioning phase, one writes GPRM task codes to define task units. In the next step, the GPRM programmer defines the communication pattern between the tasks in a GPC code. When writing the GPC code, one is also responsible for deciding a proper cutoff value to control the number of created tasks. We have provided some general rules for finding a good cutoff for GPRM programs at the end of Section 6.1, e.g. if the number of tasks is not divisible by the number of cores, a larger cutoff could be of help in balancing the load. The final phase of mapping is performed by the GPC compiler and tuned by the GPRM scheduler at runtime.

As task-based parallel programming is becoming increasingly popular for programming manycore processors, we started our experiments by comparing OpenMP, Cilk Plus, and TBB together on the Xeon Phi, and highlighting the importance of proper cutoff values. We also illustrated that the overhead of runtime systems could hurt the performance of multiprogram workloads quite significantly. In addition, We showed that sharing mapping information between OpenMP applications in a multiprogramming environment (on the TILEPro64) could improve the turnaround time.

Using three simple benchmarks, Fibonacci, MergeSort, and MatMul, GPRM showed superior performance compared to Cilk Plus and TBB on the Xeon Phi for both uniprogramming and multiprogramming. We have observed in our experiments that, for example, OpenMP

performs well for single programs, but does not offer the best performance for multiprogram workloads. GPRM, on the other hand, combines compile-time information about tasks (through partial evaluation) with an efficient task stealing strategy and a very low-overhead sharing mechanism between different programs in order to achieve high performance. We call this scheme *Steal Locally, Share Globally*. Based on this scheme, within each application, a low-overhead task stealing mechanism based upon the GPRM model of execution is deployed. GPRM threads steal from each other only if their initial task assignment is sub-optimal and load imbalance exists. Additionally, there exists a globally shared data structure to keep track of the thread-to-core mapping information. Every GPRM instance maps this globally shared data structure to its own memory space and uses it to share the mapping information with other GPRM applications, if any. *Steal Locally, Share Globally* scheme's operations take place behind the scenes and are invisible to the programmer. However, both task stealing and global sharing features can be disabled via command-line switches, if need be.

As the most widely used standard for shared-memory programming, OpenMP was chosen as the main competitor of GPRM on the two platforms. For the Fibonacci and MergeSort benchmarks, GPRM outperformed OpenMP notably on both platforms. For the MatMul benchmark, only on the Xeon Phi, OpenMP could outperform GPRM. Considering the 4-way multithreading on the Xeon Phi, however, we found that just by choosing a different thread mapping approach, GPRM (still with the default number of threads) can reach the top performance achieved by the optimal number of OpenMP threads. Furthermore, by extending this benchmark to different integer, float, and double matrices, we observed that in most of the cases, GPRM outperforms OpenMP. The difference was up to $11\times$ on the TILEPro64 and up to $2\times$ on the Xeon Phi. In the multiprogramming experiments, GPRM offered significant improvement over OpenMP.

We continued our detailed comparison with OpenMP using more complex benchmarks: LU factorisation of Sparse Matrices, Image Convolution, and Linked List Processing. Apart from performance comparison, we also focused on GPRM ways of solving these problems, by considering GPRM's execution model as well as its special parallel constructs and APIs.

We concluded that our proposed solution for solving the "LU factorisation of Sparse Matrices" problem is suitable for larger numbers of blocks and smaller block sizes. For the "Image Convolution" benchmark, we highlighted the overhead of task creation and distribution in GPRM for very small tasks. But, we also discussed that using task agglomeration, GPRM can outperform OpenMP for 2D convolution of large images. With the use of "Linked List Processing" benchmark, we showed how linked list processing in GPRM can be easy and efficient. With the help of its `par_list` construct, GPRM can determine the task to thread mapping at compile time and only tune it at runtime.

Throughout the whole thesis, we emphasised the importance of tasks and showed how controlling the granularity and number of them is key to performance. As a result of this study, we encourage the users to avoid saturating the systems with a massive number of fine-grained tasks. Conversely, we would like to stress that the desired performance is achievable by focusing more on the size/number of tasks.

We described the implementation details of the GPRM framework, tested it with some basic examples, and measured its performance for both regular and irregular parallel benchmarks. There are several more complicated parallel patterns that could be used to compare GPRM with other parallel programming approaches. However, based on the evidence demonstrated, we conclude that GPRM is a flexible programming model that could enhance both performance and productivity on manycore processors. We have identified some of its weaknesses and proposed a number of solutions in order to improve its performance. Additionally, nothing stops users from combining GPRM with another model, such as OpenMP whenever needed. In terms of productivity, GPRM's modular structure is of great help for writing complex parallel programs. The fact that a GPC code offers native parallelism is another reason why GPRM could improve productivity. We have also shown that performance tuning in GPRM (due to its high abstraction level) is a matter of dealing with only tasks, and not threads.

We end this dissertation by outlining directions for future work. Our suggestions are based upon the evidence and information gathered during this study. Therefore, the next section can be considered as a complementary section for our final conclusion.

10.2 Future Work

In this section, based on the lessons learnt from the design and implementation of GPRM, we aim to provide suggestions for future improvement and extension.

A very useful part of the future work would be to reduce the overhead of the GPRM runtime system, as a large portion of its code was never written for low overhead, rather for flexibility. In order to make GPRM a successful and comprehensive general-purpose parallel framework, still a lot of work lies ahead of us. There is room for improvement in multiple areas:

10.2.1 Support for other Computing Models

A pleasant feature of GPRM is that it can be integrated into any C++ code and be combined with other parallel programming models. This helps to offload only some parts of computation to GPRM, whenever needed. Also since GPRM is based on a pool of POSIX threads,

it is compatible with approaches like OpenMP. If GPRM threads have no work to do, they simply go to sleep and threads of the other model can be used for parallelisation. Although we have tested GPRM and OpenMP together in some sample benchmarks, achieving high performance for such combinations remains for future work.

The current version of GPRM can be extended to support distributed computing. The only difference should be in sending messages. Instead of passing the pointers around, the system should copy the whole data, if the access request comes from another node. For intra-node communication, still the pointers should be used.

Parallel to this work, other researchers in our group are working on a data-parallel version of the GPRM, for use on GPUs and, eventually, FPGAs. In this way, we aim to create a unified programming framework for heterogeneous manycore systems.

10.2.2 Task Scheduling

The current runtime system is oblivious to the NUMA effect. The techniques used in other parallel programming approaches regarding the NUMA effect can be embedded into the GPRM runtime system. For instance, a locality-aware task scheduling technique is proposed in [174].

Giving priority to tasks can be added as a feature to the GPRM framework. SMPSs [175] uses the `highpriority` clause to indicate that a task has high priority and if there is no data dependency, it must be executed before tasks without high priority. Using similar notation could help the GPRM runtime system to prioritise the tasks.

The next two suggestions are regarding the stealing mechanism. In the current scheme, tasks are stolen from the *Ready Queues*. This could be expanded to support stealing from the *Request Queues*. This way, instead of a single task, a branch of computation will be stolen. However, since GPC compiler does its best to make sure that tasks are evenly distributed, this could only be useful for specific parallel patterns.

A better change to the task stealing mechanism could be stealing multiple tasks at once (instead of just one) until the *Ready Queues* become equal in size.

10.2.3 Oversubscription

In GPRM, oversubscription comes with low overhead, because once the threads are pinned to the processing cores, no further scheduling is performed on them. Two different approaches can be considered here:

Manager/Worker

In this situation, one thread can work as the manager, performing the responsibilities of the *task manager*, and the other one becomes responsible for the *task kernel* jobs. The overhead of thread switching in this case should be investigated. But it is important to recall that in the current version of GPRM, since *task*s are non-preemptive, no decisions can be made about the newly arrived tasks until the execution of the current task finishes.

Therefore, when a task blocks on I/O, the whole *tile* becomes useless, and this could be a flaw in the current system. Although, it can be argued that when the number of cores increases, the tasks become smaller, hence the potential latency associated with them has less impact on overall performance.

Peers

In an alternative approach for solving the above problem, there can be multiple threads on each tile with equivalent access to the task queues. The scheduling of such threads as well as their access to the shared data structures should be managed by the GPRM runtime system. However, in this case the threads do not have pre-defined jobs. The runtime system should be able to switch between them if one blocks for more than a certain threshold.

A hybrid approach could also be considered, where one thread is the manager and others are workers. If a worker thread blocks on a long-running task, another worker thread should be able to execute other tasks. The manager thread would still be responsible for incoming and outgoing messages.

Bibliography

[1] TILERA Corporation. Tile Processor User Architecture Manual UG101; 2011.

[2] Jeffers J, Reinders J. Intel Xeon Phi Coprocessor High Performance Programming. Newnes; 2013.

[3] Ayguadé E, Copty N, Duran A, Hoeflinger J, Lin Y, Massaioli F, et al. The design of openmp tasks. Parallel and Distributed Systems, IEEE Transactions on. 2009;20(3):404–418.

[4] Olukotun K, Nayfeh BA, Hammond L, Wilson K, Chang K. The case for a single-chip multiprocessor. ACM Sigplan Notices. 1996;31(9):2–11.

[5] Agarwal V, Hrishikesh M, Keckler SW, Burger D. Clock rate versus IPC: The end of the road for conventional microarchitectures. vol. 28. ACM; 2000.

[6] Schaller RR. Moore's law: past, present and future. Spectrum, IEEE. 1997;34(6):52–59.

[7] Wall DW. Limits of instruction-level parallelism. vol. 19. ACM; 1991.

[8] Olukotun K, Hammond L. The future of microprocessors. Queue. 2005;3(7):26–29.

[9] Tullsen DM, Eggers SJ, Levy HM. Simultaneous multithreading: Maximizing on-chip parallelism. In: ACM SIGARCH Computer Architecture News. vol. 23. ACM; 1995. p. 392–403.

[10] Marr D. Hyper-Threading Technology in the Netburst® Microarchitecture. 14th Hot Chips. 2002;.

[11] Rauber T, Rünger G. Parallel programming: For multicore and cluster systems. Springer Science & Business Media; 2013.

[12] Bjerregaard T, Mahadevan S. A survey of research and practices of network-on-chip. ACM Computing Surveys (CSUR). 2006;38(1):1.

[13] Sutter H. The free lunch is over: A fundamental turn toward concurrency in software. Dr Dobbs journal. 2005;30(3):202–210.

[14] Benini L, De Micheli G. Networks on chips: a new SoC paradigm. Computer. 2002;35(1):70–78.

[15] Borkar S. Thousand core chips: a technology perspective. In: Proceedings of the 44th annual Design Automation Conference. ACM; 2007. p. 746–749.

[16] Amdahl GM. Validity of the single processor approach to achieving large scale computing capabilities. In: Proceedings of the April 18-20, 1967, spring joint computer conference. ACM; 1967. p. 483–485.

[17] Hill MD, Marty MR. Amdahl's law in the multicore era. Computer. 2008;(7):33–38.

[18] Gustafson JL. Reevaluating Amdahl's law. Communications of the ACM. 1988;31(5):532–533.

[19] Chapman B, Jost G, Van Der Pas R. Using OpenMP: portable shared memory parallel programming. vol. 10. MIT press; 2008.

[20] Leijen D, Schulte W, Burckhardt S. The design of a task parallel library. In: Acm Sigplan Notices. vol. 44. ACM; 2009. p. 227–242.

[21] Tillenius M. SuperGlue: A Shared Memory Framework Using Data Versioning for Dependency-Aware Task-Based Parallelization. SIAM Journal on Scientific Computing. 2015;37(6):C617–C642.

[22] Tousimojarad A, Vanderbauwhede W. Cache-aware parallel programming for manycore processors. In: Highly Efficient Accelerators and Reconfigurable Technologies (HEART2013); 2013. .

[23] Tousimojarad A, Vanderbauwhede W. An Efficient Thread Mapping Strategy for Multiprogramming on Manycore Processors. In: Parallel Computing: Accelerating Computational Science and Engineering (CSE). vol. 25 of Advances in Parallel Computing. IOS Press; 2014. p. 63–71.

[24] Tousimojarad A, Vanderbauwhede W. Comparison of Three Popular Parallel Programming Models on the Intel Xeon Phi. In: Euro-Par 2014: Parallel Processing Workshops. Springer; 2014. p. 314–325.

[25] Tousimojarad A. A parallel task composition approach to manycore programming. PLACES 2013. 2013;29.

[26] Tousimojarad A, Vanderbauwhede W. The Glasgow Parallel Reduction Machine: Programming Shared-memory Many-core Systems using Parallel Task Composition. EPTCS. 2013;137:79–94.

[27] Tousimojarad A, Vanderbauwhede W. Steal Locally, Share Globally. International Journal of Parallel Programming. 2015;43(5):894–917.

[28] Tousimojarad A, Vanderbauwhede W. Number of Tasks, not Threads, is Key. In: 2015 23rd Euromicro International Conference on Parallel, Distributed, and Network-Based Processing. IEEE; 2015. .

[29] Tousimojarad A, Vanderbauwhede W. A Parallel Task-based Approach to Linear Algebra. In: Parallel and Distributed Computing (ISPDC), 2014 IEEE 13th International Symposium on. IEEE; 2014. p. 59–66.

[30] Chimeh M, Cockshott P, Oehler SB, Tousimojarad A, Xu T. Compiling Vector Pascal to the XeonPhi. Concurrency and Computation: Practice and Experience. 2015;.

[31] Tousimojarad A, Vanderbauwhede W. Efficient Parallel Linked List Processing. Advances in Parallel Computing. IOS Press; 2016. .

[32] Foster I. Designing and building parallel programs. Addison Wesley Publishing Company; 1995.

[33] Singh AK, Shafique M, Kumar A, Henkel J. Mapping on multi/many-core systems: survey of current and emerging trends. In: Proceedings of the 50th Annual Design Automation Conference. ACM; 2013. p. 1.

[34] Valiant LG. A bridging model for parallel computation. Communications of the ACM. 1990;33(8):103–111.

[35] Coarfa C, Dotsenko Y, Mellor-Crummey J, Cantonnet F, El-Ghazawi T, Mohanti A, et al. An evaluation of global address space languages: co-array fortran and unified parallel C. In: Proceedings of the tenth ACM SIGPLAN symposium on Principles and practice of parallel programming. ACM; 2005. p. 36–47.

[36] El-Ghazawi TA, Carlson WW, Draper JM. UPC Language Specifications v1. 1.1. October. 2003;200(3):1.

[37] Numrich RW, Reid J. Co-Array Fortran for parallel programming. In: ACM Sigplan Fortran Forum. vol. 17. ACM; 1998. p. 1–31.

[38] Gropp W, Lusk E, Skjellum A. Using MPI: portable parallel programming with the message-passing interface. vol. 1. MIT press; 1999.

[39] Chen LT, Bairagi D. Developing Parallel Programs–A Discussion of Popular Models. Technical report, Oracle Corporation; 2010.

[40] Duran A, Perez JM, Ayguadé E, Badia RM, Labarta J. Extending the OpenMP tasking model to allow dependent tasks. In: OpenMP in a New Era of Parallelism. Springer; 2008. p. 111–122.

[41] Leiserson CE. The Cilk++ concurrency platform. The Journal of Supercomputing. 2010;51(3):244–257.

[42] Frigo M, Halpern P, Leiserson CE, Lewin-Berlin S. Reducers and other Cilk++ hyperobjects. In: Proceedings of the twenty-first annual symposium on Parallelism in algorithms and architectures. ACM; 2009. p. 79–90.

[43] Reinders J. Intel Threading Building Blocks: outfitting C++ for multi-core processor parallelism. O'Reilly Media, Inc.; 2007.

[44] Intel. Threading Building Blocks; 2015. Https://www.threadingbuildingblocks.org/.

[45] Frigo M, Leiserson CE, Randall KH. The implementation of the Cilk-5 multithreaded language. In: ACM Sigplan Notices. vol. 33. ACM; 1998. p. 212–223.

[46] Gabriel E, Fagg GE, Bosilca G, Angskun T, Dongarra JJ, Squyres JM, et al. Open MPI: Goals, concept, and design of a next generation MPI implementation. In: Recent Advances in Parallel Virtual Machine and Message Passing Interface. Springer; 2004. p. 97–104.

[47] Jin H, Jespersen D, Mehrotra P, Biswas R, Huang L, Chapman B. High performance computing using MPI and OpenMP on multi-core parallel systems. Parallel Computing. 2011;37(9):562–575.

[48] Bueno J, Planas J, Duran A, Badia RM, Martorell X, Ayguade E, et al. Productive programming of gpu clusters with ompss. In: Parallel & Distributed Processing Symposium (IPDPS), 2012 IEEE 26th International. IEEE; 2012. p. 557–568.

[49] Jin H, Hood R, Mehrotra P. A practical study of UPC using the NAS Parallel Benchmarks. In: Proceedings of the Third Conference on Partitioned Global Address Space Programing Models. ACM; 2009. p. 8.

[50] Chamberlain BL, Callahan D, Zima HP. Parallel programmability and the chapel language. International Journal of High Performance Computing Applications. 2007;21(3):291–312.

[51] Pérez JM, Bellens P, Badia RM, Labarta J. CellSs: Making it easier to program the Cell Broadband Engine processor. IBM Journal of Research and Development. 2007;51(5):593–604.

[52] Perez JM, Badia RM, Labarta J. A dependency-aware task-based programming environment for multi-core architectures. In: Cluster Computing, 2008 IEEE International Conference on. IEEE; 2008. p. 142–151.

[53] Ayguadé E, Badia RM, Igual FD, Labarta J, Mayo R, Quintana-Ortí ES. An extension of the StarSs programming model for platforms with multiple GPUs. In: Euro-Par 2009 Parallel Processing. Springer; 2009. p. 851–862.

[54] Duran A, Ayguadé E, Badia RM, Labarta J, Martinell L, Martorell X, et al. Ompss: a proposal for programming heterogeneous multi-core architectures. Parallel Processing Letters. 2011;21(02):173–193.

[55] Ayguade E, Badia RM, Cabrera D, Duran A, Gonzalez M, Igual F, et al. A proposal to extend the openmp tasking model for heterogeneous architectures. In: Evolving OpenMP in an Age of Extreme Parallelism. Springer; 2009. p. 154–167.

[56] Balart J, Duran A, Gonzàlez M, Martorell X, Ayguadé E, Labarta J. Nanos mercurium: a research compiler for openmp. In: Proceedings of the European Workshop on OpenMP. vol. 8; 2004. .

[57] Giorgi R, Badia RM, Bodin F, Cohen A, Evripidou P, Faraboschi P, et al. TERAFLUX: Harnessing dataflow in next generation teradevices. Microprocessors and Microsystems. 2014;38(8):976–990.

[58] Pop A, Cohen A. OpenStream: Expressiveness and data-flow compilation of OpenMP streaming programs. ACM Transactions on Architecture and Code Optimization (TACO). 2013;9(4):53.

[59] Vandierendonck H, Tzenakis G, Nikolopoulos DS. A unified scheduler for recursive and task dataflow parallelism. In: Parallel Architectures and Compilation Techniques (PACT), 2011 International Conference on. IEEE; 2011. p. 1–11.

[60] Vandierendonck H, Pratikakis P, Nikolopoulos DS. Parallel Programming of General-Purpose Programs Using Task-Based Programming Models. In: HotPar; 2011. .

[61] Jones SLP. Haskell 98 language and libraries: the revised report. Cambridge University Press; 2003.

[62] Hammond K. Glasgow parallel haskell (gph). In: Encyclopedia of Parallel Computing. Springer; 2011. p. 768–779.

[63] Trinder PW, Hammond K, Mattson Jr JS, Partridge AS, Peyton Jones S. GUM: a portable parallel implementation of Haskell. In: ACM SIGPLAN Notices. vol. 31. ACM; 1996. p. 79–88.

[64] Hickey R. The clojure programming language. In: Proceedings of the 2008 symposium on Dynamic languages. ACM; 2008. p. 1.

[65] Halloway S. Programming Clojure. Pragmatic Bookshelf; 2009.

[66] Scholz SB. Single Assignment C: efficient support for high-level array operations in a functional setting. Journal of functional programming. 2003;13(06):1005–1059.

[67] Grelck C. Single Assignment C (SAC) High Productivity Meets High Performance. In: Central European Functional Programming School. Springer; 2012. p. 207–278.

[68] Stone JE, Gohara D, Shi G. OpenCL: A parallel programming standard for heterogeneous computing systems. Computing in science & engineering. 2010;12(3):66.

[69] Du P, Weber R, Luszczek P, Tomov S, Peterson G, Dongarra J. From CUDA to OpenCL: Towards a performance-portable solution for multi-platform GPU programming. Parallel Computing. 2012;38(8):391–407.

[70] Sanders J, Kandrot E. CUDA by example: an introduction to general-purpose GPU programming. Addison-Wesley Professional; 2010.

[71] Dean J, Ghemawat S. MapReduce: simplified data processing on large clusters. Communications of the ACM. 2008;51(1):107–113.

[72] Mao Y, Morris R, Kaashoek MF. Optimizing MapReduce for multicore architectures. In: Computer Science and Artificial Intelligence Laboratory, Massachusetts Institute of Technology, Tech. Rep. Citeseer; 2010. .

[73] Sinnen O. Task scheduling for parallel systems. vol. 60. Wiley-Interscience; 2007.

[74] Shirazi B, Hurson AR. Special issue on scheduling and load balancing guest editors' introduction. Journal of Parallel and Distributed Computing. 1992;16(4):271–275.

[75] Blumofe RD, Leiserson CE. Scheduling multithreaded computations by work stealing. Journal of the ACM (JACM). 1999;46(5):720–748.

[76] Dinan J, Olivier S, Sabin G, Prins J, Sadayappan P, Tseng CW. Dynamic load balancing of unbalanced computations using message passing. In: Parallel and Distributed Processing Symposium, 2007. IPDPS 2007. IEEE International. IEEE; 2007. p. 1–8.

[77] Burton FW, Sleep MR. Executing functional programs on a virtual tree of processors. In: Proceedings of the 1981 conference on Functional programming languages and computer architecture. ACM; 1981. p. 187–194.

[78] Halstead Jr RH. Implementation of Multilisp: Lisp on a multiprocessor. In: Proceedings of the 1984 ACM Symposium on LISP and functional programming. ACM; 1984. p. 9–17.

[79] Rudolph L, Slivkin-Allalouf M, Upfal E. A simple load balancing scheme for task allocation in parallel machines. In: Proceedings of the third annual ACM symposium on Parallel algorithms and architectures. ACM; 1991. p. 237–245.

[80] Blumofe RD, Lisiecki PA, et al. Adaptive and reliable parallel computing on networks of workstations. In: USENIX 1997 Annual Technical Conference on UNIX and Advanced Computing Systems; 1997. p. 133–147.

[81] Augonnet C, Thibault S, Namyst R, Wacrenier PA. StarPU: a unified platform for task scheduling on heterogeneous multicore architectures. Concurrency and Computation: Practice and Experience. 2011;23(2):187–198.

[82] Sasaki H, Tanimoto T, Inoue K, Nakamura H. Scalability-based manycore partitioning. In: Proceedings of the 21st international conference on Parallel architectures and compilation techniques. ACM; 2012. p. 107–116.

[83] Duran A, Corbalán J, Ayguadé E. Evaluation of OpenMP task scheduling strategies. In: OpenMP in a new era of parallelism. Springer; 2008. p. 100–110.

[84] Zhang G. Extending the OpenMP standard for thread mapping and grouping. In: OpenMP Shared Memory Parallel Programming. Springer; 2008. p. 435–446.

[85] Olivier SL, Prins JF. Comparison of OpenMP 3.0 and other task parallel frameworks on unbalanced task graphs. International Journal of Parallel Programming. 2010;38(5):341–360.

[86] Teruel X, Martorell X, Duran A, Ferrer R, Ayguadé E. Support for OpenMP tasks in Nanos v4. In: Proceedings of the 2007 conference of the center for advanced studies on Collaborative research. IBM Corp.; 2007. p. 256–259.

[87] Yoo RM. Locality-Aware Task Management on Many-Core Processors. Stanford University; 2012.

[88] Shim KS, Lis M, Khan O, Devadas S. Judicious thread migration when accessing distributed shared caches. 2012;.

[89] Serres O, Anbar A, Merchant S, El-Ghazawi T. Experiences with UPC on TILE-64 processor. In: Aerospace Conference, 2011 IEEE. IEEE; 2011. p. 1–9.

[90] Husbands P, Iancu C, Yelick K. A performance analysis of the Berkeley UPC compiler. In: Proceedings of the 17th annual international conference on Supercomputing. ACM; 2003. p. 63–73.

[91] Bonachea D. GASNet Specification, v1. 1. 2002;.

[92] Muddukrishna A, Podobas A, Brorsson M, Vlassov V. Task Scheduling on Manycore Processors with Home Caches. In: Euro-Par 2012: Parallel Processing Workshops. Springer; 2013. p. 357–367.

[93] Morari A, Tumeo A, Villa O, Secchi S, Valero M. Efficient Sorting on the Tilera Manycore Architecture. In: Computer Architecture and High Performance Computing (SBAC-PAD), 2012 IEEE 24th International Symposium on. IEEE; 2012. p. 171–178.

[94] Podobas A, Brorsson M, Faxén KF. A Quantitative Evaluation of popular Task-Centric Programming Models and Libraries. KTH, Software and Computer systems, SCS; 2012. 12:03. QC 20121214.

[95] Mirsoleimani SA, Plaat A, Herik Jvd, Vermaseren J. Scaling Monte Carlo Tree Search on Intel Xeon Phi. arXiv preprint arXiv:150704383. 2015;.

[96] Saule E, Kaya K, Çatalyürek ÜV. Performance evaluation of sparse matrix multiplication kernels on Intel Xeon Phi. In: Parallel Processing and Applied Mathematics. Springer; 2014. p. 559–570.

[97] Podobas A. Improving Performance and Quality-of-Service through the Task-Parallel Model: Optimizations and Future Directions for OpenMP. 2015;.

[98] Lima JV, Broquedis F, Gautier T, Raffin B. Preliminary Experiments with XKaapi on Intel Xeon Phi Coprocessor. In: Computer Architecture and High Performance Computing (SBAC-PAD), 2013 25th International Symposium on. IEEE; 2013. p. 105–112.

[99] Faxén KF. Wool-a work stealing library. ACM SIGARCH Computer Architecture News. 2009;36(5):93–100.

[100] Teruel X, Barton C, Duran A, Martorell X, Ayguadé E, Unnikrishnan P, et al. OpenMP tasking analysis for programmers. In: Proceedings of the 2009 Conference of the Center for Advanced Studies on Collaborative Research. IBM Corp.; 2009. p. 32–42.

[101] Duran A, Corbalán J, Ayguadé E. An adaptive cut-off for task parallelism. In: High Performance Computing, Networking, Storage and Analysis, 2008. SC 2008. International Conference for. IEEE; 2008. p. 1–11.

[102] Podobas A, Brorsson M. A Comparison of some recent Task-based Parallel Programming Models. In: Proceedings of the 3rd Workshop on Programmability Issues for Multi-Core Computers,(MULTIPROG'2010), Jan 2010, Pisa; 2010. .

[103] Leist A, Gilman A. A Comparative Analysis of Parallel Programming Models for C++. In: ICCGI 2014, The Ninth International Multi-Conference on Computing in the Global Information Technology; 2014. p. 121–127.

[104] Muddukrishna A, Jonsson PA, Brorsson M. Characterizing task-based openmp programs. 2015;.

[105] Brunst H, Mohr B. Performance analysis of large-scale OpenMP and hybrid MPI/OpenMP applications with Vampir NG. In: OpenMP Shared Memory Parallel Programming. Springer; 2008. p. 5–14.

[106] Gautier T, Lima JV, Maillard N, Raffin B. Xkaapi: A runtime system for data-flow task programming on heterogeneous architectures. In: Parallel & Distributed Processing (IPDPS), 2013 IEEE 27th International Symposium on. IEEE; 2013. p. 1299–1308.

[107] Buyya R. High performance cluster computing. New Jersey: F'rentice. 1999;.

[108] Bailey DH, Barszcz E, Barton JT, Browning DS, Carter RL, Dagum L, et al. The NAS parallel benchmarks. International Journal of High Performance Computing Applications. 1991;5(3):63–73.

[109] Van der Wijngaart RF, Jin H. Nas parallel benchmarks, multi-zone versions. NASA Ames Research Center, Tech Rep NAS-03-010. 2003;.

[110] Nichols R, Tramel R, Buning P. Solver and turbulence model upgrades to OVERFLOW 2 for unsteady and high-speed applications. AIAA paper. 2006;2824:2006.

[111] Hah C, Wennerstrom A. Three-dimensional flowfields inside a transonic compressor with swept blades. Journal of Turbomachinery. 1991;113(2):241–250.

[112] Berger MJ, Aftosmis MJ, Marshall D, Murman SM. Performance of a new CFD flow solver using a hybrid programming paradigm. Journal of Parallel and Distributed Computing. 2005;65(4):414–423.

[113] Krawezik G. Performance comparison of MPI and three OpenMP programming styles on shared memory multiprocessors. In: Proceedings of the fifteenth annual ACM symposium on Parallel algorithms and architectures. ACM; 2003. p. 118–127.

[114] Wolf ME, Lam MS. A loop transformation theory and an algorithm to maximize parallelism. Parallel and Distributed Systems, IEEE Transactions on. 1991;2(4):452–471.

[115] Krpic Z, Martinovic G, Crnkovic I. Green HPC: MPI vs. OpenMP on a shared memory system. In: MIPRO, 2012 Proceedings of the 35th International Convention. IEEE; 2012. p. 246–250.

[116] Hemmert S. Green hpc: From nice to necessity. Computing in Science & Engineering. 2010;(6):8–10.

[117] Saule E, Catalyurek UV. An early evaluation of the scalability of graph algorithms on the Intel MIC architecture. In: Parallel and Distributed Processing Symposium Workshops & PhD Forum (IPDPSW), 2012 IEEE 26th International. IEEE; 2012. p. 1629–1639.

[118] Harris T, Maas M, Marathe VJ. Callisto: co-scheduling parallel runtime systems. In: Proceedings of the Ninth European Conference on Computer Systems. ACM; 2014. p. 24.

[119] Emani MK, Wang Z, O'Boyle MF. Smart, adaptive mapping of parallelism in the presence of external workload. In: Code Generation and Optimization (CGO), 2013 IEEE/ACM International Symposium on. IEEE; 2013. p. 1–10.

[120] Varisteas G, Brorsson M, Faxen KF. Resource management for task-based parallel programs over a multi-kernel.: BIAS: Barrelfish Inter-core Adaptive Scheduling. In: Proceedings of the 2012 workshop on Runtime Environments, Systems, Layering and Virtualized Environments (RESoLVE12); 2012. p. 32–36.

[121] Bhadauria M, McKee SA. An approach to resource-aware co-scheduling for CMPs. In: Proceedings of the 24th ACM International Conference on Supercomputing. ACM; 2010. p. 189–199.

[122] Almasi GS, Gottlieb A. Highly parallel computing. 1988;.

[123] Flynn MJ. Some computer organizations and their effectiveness. Computers, IEEE Transactions on. 1972;100(9):948–960.

[124] Barney B, et al. Introduction to parallel computing. Lawrence Livermore National Laboratory. 2010;6(13):10.

[125] Darema F. The spmd model: Past, present and future. In: Recent Advances in Parallel Virtual Machine and Message Passing Interface. Springer; 2001. p. 1–1.

[126] McCool M, Reinders J, Robison A. Structured parallel programming: patterns for efficient computation. Elsevier; 2012.

[127] Lindholm E, Nickolls J, Oberman S, Montrym J. NVIDIA Tesla: A unified graphics and computing architecture. IEEE micro. 2008;(2):39–55.

[128] Jones S. Introduction to dynamic parallelism. In: GPU Technology Conference Presentation S. vol. 338; 2012. .

[129] Culler DE, Singh JP, Gupta A. Parallel computer architecture: a hardware/software approach. Gulf Professional Publishing; 1999.

[130] Handy J. The cache memory book. Morgan Kaufmann; 1998.

[131] Bell S, Edwards B, Amann J, Conlin R, Joyce K, Leung V, et al. Tile64-processor: A 64-core soc with mesh interconnect. In: Solid-State Circuits Conference, 2008. ISSCC 2008. Digest of Technical Papers. IEEE International. IEEE; 2008. p. 88–598.

[132] Vangal S, Howard J, Ruhl G, Dighe S, Wilson H, Tschanz J, et al. An 80-tile 1.28 TFLOPS network-on-chip in 65nm CMOS. In: IEEE International Solid-State Circuits Conference, San Fransisco, USA, 2007. IEEE; 2007. p. 98–99.

[133] Held J, Bautista J, Koehl S. From a few cores to many: A tera-scale computing research overview. white paper, Intel. 2006;.

[134] Rajovic N, Rico A, Puzovic N, Adeniyi-Jones C, Ramirez A. Tibidabo: Making the case for an ARM-based HPC system. Future Generation Computer Systems. 2013;DOI: http://dx.doi.org/10.1016/j.future.2013.07.013.

[135] Kim C, Burger D, Keckler SW. An adaptive, non-uniform cache structure for wire-delay dominated on-chip caches. In: Acm Sigplan Notices. vol. 37. ACM; 2002. p. 211–222.

[136] TILERA Corporation. Multicore Development Environment System Programmer's Guide UG209; 2011.

[137] Dennis JB, Van Horn EC. Programming semantics for multiprogrammed computations. Communications of the ACM. 1966;9(3):143–155.

[138] Clet-Ortega J, Carribault P, Pérache M. Evaluation of OpenMP Task Scheduling Algorithms for Large NUMA Architectures. In: Euro-Par 2014 Parallel Processing. Springer; 2014. p. 596–607.

[139] Blumofe RD, Joerg CF, Kuszmaul BC, Leiserson CE, Randall KH, Zhou Y. Cilk: An efficient multithreaded runtime system. Journal of parallel and distributed computing. 1996;37(1):55–69.

[140] Kim W, Voss M. Multicore desktop programming with Intel Threading Building Blocks. IEEE software. 2011;28(1):23–31.

[141] Intel. Software Development Tools: Intel VTune Amplifier XE 2013; 2013. Https://software.intel.com/en-us/intel-vtune-amplifier-xe.

[142] Marowka A. On performance analysis of a multithreaded application parallelized by different programming models using Intel Vtune. In: Parallel Computing Technologies. Springer; 2011. p. 317–331.

[143] Lubin M, McMillan S, Kruse CG, Del Vento D, Montuoro R. Efficient Software Development: 4 Whats New in Intel® Parallel Studio XE 2013 Service Pack. 2013;.

[144] Arora NS, Blumofe RD, Plaxton CG. Thread scheduling for multiprogrammed multiprocessors. Theory of Computing Systems. 2001;34(2):115–144.

[145] Stallings W. Operating Systems, Internals and Design Principles, 7th Edition. Pearson: Prentice Hall; 2012.

[146] Tucker A. Efficient scheduling on multiprogrammed shared-memory multiprocessors. Stanford University; 1994.

[147] Free Software Foundation I. GNU Offloading and Multi Processing Runtime Library; 2015. Https://gcc.gnu.org/onlinedocs/libgomp.pdf.

[148] Aas J. Understanding the Linux 2.6.8.1 CPU scheduler. Retrieved Oct. 2005;16:1–38.

[149] Duran A, Teruel X, Ferrer R, Martorell X, Ayguade E. Barcelona openmp tasks suite: A set of benchmarks targeting the exploitation of task parallelism in openmp. In: Parallel Processing, 2009. ICPP'09. International Conference on. IEEE; 2009. p. 124–131.

[150] Vanderbauwhede W. The Gannet Service-based SoC: A Service-level Reconfigurable Architecture. In: Adaptive Hardware and Systems, 2006. AHS 2006. First NASA/ESA Conference on. IEEE; 2006. p. 255–261.

[151] Vanderbauwhede W. Gannet: a functional task description language for a service-based SoC architecture. In: Proc. 7th Symposium on Trends in Functional Programming (TFP06). Citeseer; 2006. .

[152] McCarthy J. LISP 1.5 programmer's manual. MIT press; 1965.

[153] Meikleham R. Design and compilation of a C-like front end language for the GPRM [Honours Dissertation]. School of Computing Science, University of Glasgow; 2015.

[154] Wegman MN, Zadeck FK. Constant propagation with conditional branches. ACM Transactions on Programming Languages and Systems (TOPLAS). 1991;13(2):181–210.

[155] Brandis MM, Mössenböck H. Single-pass generation of static single-assignment form for structured languages. ACM Transactions on Programming Languages and Systems (TOPLAS). 1994;16(6):1684–1698.

[156] van Heesch D. Doxygen: Source code documentation generator tool. URL: http://www doxygen org. 2008;.

[157] Veen AH. Dataflow machine architecture. ACM Computing Surveys (CSUR). 1986;18(4):365–396.

[158] Ebcioglu K, Sarkar V, El-Ghazawi T, Urbanic J, Center P. An experiment in measuring the productivity of three parallel programming languages. In: Workshop on Productivity and Performance in High-End Computing (P-PHEC); 2006. .

[159] Hughes J. Why functional programming matters. The computer journal. 1989;32(2):98–107.

[160] Iancu C, Hofmeyr S, Blagojević F, Zheng Y. Oversubscription on multicore processors. In: Parallel & Distributed Processing (IPDPS), 2010 IEEE International Symposium on. IEEE; 2010. p. 1–11.

[161] Hofmeyr S, Iancu C, Blagojević F. Load balancing on speed. In: ACM Sigplan Notices. vol. 45. ACM; 2010. p. 147–158.

[162] Yan J, He J, Han W, Chen W, Zheng W. How OpenMP applications get more benefit from many-core era. In: Beyond Loop Level Parallelism in OpenMP: Accelerators, Tasking and More. Springer; 2010. p. 83–95.

[163] Maillard N. A Runtime System for Data-Flow Task Programming on Multicore Architectures with Accelerators. Grenoble University; 2014.

[164] Mazouz A, Touati S, Barthou D. Performance evaluation and analysis of thread pinning strategies on multi-core platforms: Case study of spec omp applications on Intel architectures. In: High Performance Computing and Simulation (HPCS), 2011 International Conference on. IEEE; 2011. p. 273–279.

[165] Olivier SL, Porterfield AK, Wheeler KB, Prins JF. Scheduling task parallelism on multi-socket multicore systems. In: Proceedings of the 1st International Workshop on Runtime and Operating Systems for Supercomputers. ACM; 2011. p. 49–56.

[166] Smith S. Digital Signal Processing: A Practical Guide for Engineers and Scientists: A Practical Guide for Engineers and Scientists. Newnes; 2013.

[167] Satish N, Kim C, Chhugani J, Saito H, Krishnaiyer R, Smelyanskiy M, et al. Can traditional programming bridge the ninja performance gap for parallel computing applications? Communications of the ACM. 2015;58(5):77–86.

[168] Petersen L, Anderson TA, Liu H, Glew N. Measuring the Haskell gap. In: Proceedings of the 25th symposium on Implementation and Application of Functional Languages. ACM; 2013. p. 61.

[169] Tian X, Saito H, Preis SV, Garcia EN, Kozhukhov SS, Masten M, et al. Effective SIMD Vectorization for Intel Xeon Phi Coprocessors. Scientific Programming. 2015;501:269764.

[170] Mattson T, Meadows L. A Hands-on Introduction to OpenMP. Intel Corporation. 2014;.

[171] Ayguadé E, Copty N, Duran A, Hoeflinger J, Lin Y, Massaioli F, et al. A proposal for task parallelism in OpenMP. In: A Practical Programming Model for the Multi-Core Era. Springer; 2008. p. 1–12.

[172] Cormen TH. Introduction to algorithms. MIT press; 2009.

[173] Massaioli F, Castiglione F, Bernaschi M. OpenMP parallelization of agent-based models. Parallel Computing. 2005;31(10):1066–1081.

[174] Muddukrishna A, Jonsson PA, Vlassov V, Brorsson M. Locality-aware task scheduling and data distribution on numa systems. In: OpenMP in the Era of Low Power Devices and Accelerators. Springer; 2013. p. 156–170.

[175] Barcelona Supercomputing Center. SMP Superscalar (SMPSs) Users Manual, Version 2.2; 2008.