SILKROAD: A SYSTEM SUPPORTING DSM AND MULTIPLE PARADIGMS IN CLUSTER COMPUTING

PENG LIANG

NATIONAL UNIVERSITY OF SINGAPORE

2002

Acknowledgments

My heartfelt gratitude goes to my supervisor, Professor Chung Kwong YUEN, for his insightful guidance and patient encouragement through all my years at NUS. His broad and profound knowledge and his modest and kind personal characters influenced me deeply.

I am deeply grateful to the members of the Parallel Processing Lab, Dr. Weng Fai WONG, who gave me many advices, suggestions, and so much help in both theoretical and empirical work, and Dr. Ming Dong FENG, who led me in my study and research in the early years of my life at NUS. They all actually played the role of co-supervisor in different periods.

I also would like to thank Professor Charles E. Leiserson at MIT, from whom I benefited a lot in the discussions regarding Cilk, and Professor Willy Zwaenepoel at Rice University, who gave me good guidance in my study.

Appreciation also goes to the School of Computing at National University of Singapore, that gave me a chance and provided me the resources for my study and research work. Thanks LI Zhao at NUS for his help on some of the theoretical work. Also thank the labmates in Computer Systems Lab (formerly, Parallel Processing Lab) who gave me a lot of help in my study and life at NUS.

I am very grateful to my beloved wife, who supported and helped me in my study and life and stood by me in difficult times. I would also like to thank my parents, who supported and cared about me from a long distance. Their love is a great power in my life.

Contents

List of Tables

List of Figures

Summary

Cluster of PCs is becoming an important platform for parallel computing and a number of parallel runtime systems have been developed for clusters. In cluster computing, programming paradigms are an important high-level issue that defines the way to structure algorithms to run on a parallel system. Parallel applications may be implemented with various paradigms. However, usually a parallel system is based on only one parallel programming paradigm.

This dissertation is about supporting multiple parallel programming paradigms in a cluster computing system by extending the memory consistency model and providing user level shared virtual memory. Based on Cilk, an efficient multithreaded parallel system, the RC_dag memory consistency model is proposed and the SilkRoad software runtime system is developed. An Extended Stealing Based Coherence algorithm is also proposed to maintain the RC_dag consistency and at the same time reduce the network traffic in Cilk/SilkRoad-like multithreaded parallel computing with work-stealing scheduler.

In order to analyze parallel programming paradigms and the relationship between paradigms and memory models, we also develop a formal graph-theoretical paradigm framework. With the support of multiple paradigms and user-level shared virtual memory, programmability of Cilk/SilkRoad is also examined by providing solutions to a set of examples known as Salishan Problems.

Our experimental results show that with the extended consistency model (RC_dag consistency), a wider range of paradigms can be supported by SilkRoad in cluster computing, while at the same time the applications in Cilk package can also run efficiently on SilkRoad in a multithreaded way with the Divide-and-Conquer paradigm.

Chapter 1

Introduction

In the past decade clusters of PCs or Networks of Workstations (NOW) were developed for high performance computing as an alternative low cost parallel computing resource in comparison with parallel machines. Besides off-the-shelf hardware, the availability of standard programming environments (such as MPI [70, 126] and PVM [65]) and utilities have made clusters a practical alternative as a parallel processing platform.

As clusters of PCs/Workstations become widely used platforms for parallel computing, it is desirable to provide more powerful programming environments which can support a wide range of applications efficiently.

In cluster computing, programming paradigms are an important high level issue of structuring algorithms to run on clusters. Parallel applications can be classified into several widely used programming paradigms [75, 39, 59], such as Single Program Multiple Data (SPMD), Divide-and-Conquer, Master/Slave, etc.

At a lower level, Distributed Shared Memories (DSMs) [110, 109, 103] are a widely used approach to enhance cluster computing by enabling users to develop parallel applications for clusters in a style similar to that in physically shared memory systems.

As a middleware for cluster computing, DSMs are built on top of low level network communication layers and at the same time cater for the requirements from the high level programming paradigms, which are affected by the memory model used.

Cilk [44, 50, 34, 112] is a well known parallel runtime system which supports the Divide-and-Conquer programming paradigm efficiently. It is one of several well-known multithreaded programming systems for clusters. It is effective at exploiting dynamic, highly asynchronous parallelism, which is difficult to achieve in the data-parallel or message-passing styles.

1.1 Motivation and Objectives

Many current parallel applications require global shared variables during the computation, and their corresponding paradigms may vary widely. However, normally a parallel system is based on one particular paradigm. Few systems support multiple paradigms efficiently. This prevents parallel systems from supporting a wider range of applications and achieving better applicability .

In order to achieve the multiple parallel programming paradigms, it is desirable to extend an existing parallel system which is based on a particular paradigm, to enable it to support more than one paradigm. We select Cilk as the base system in our work.

Cilk has been proven to be very efficient for fully strict Divide-and-Conquer computation on SMP (symmetric multiprocessor) systems. However, Cilk system initially does not support cluster-wide shared memory for the user and consequently there cannot be globally shared variables in parallel applications for clusters, because they are absent in Cilk's dag-consistency model and are in any case not necessary for the Divide-and-

Conquer paradigm. Besides, Cilk's multithreading and work-stealing policy may result in heavy network traffic because of the large number of threads and frequent thread migration. This can be a problem in cluster environments in some cases especially when the network is relatively slow and shared by multiple applications. Reducing network traffic may also be helpful to the applications sharing the same network.

The objectives of this research include providing a user-level shared virtual memory for using global shared variables, consequently supporting a wider range of paradigms in a cluster computing system, and reducing the network traffic of Cilk-like systems (due to multithreading and working stealing). Besides, paradigms and their relationship with underlying memory models need to be formally analyzed, and this work is helpful to empirical study in supporting multiple paradigms.

1.2 Contributions

This dissertation explores the idea of extending the memory consistency model to provide user-level shared virtual memory and support multiple parallel programming paradigms in a cluster computing system. My main contribution consists of the following:

- The shared memory approach to multiple parallel programming paradigms in software DSM-based systems and the proposal of RC_dag memory consistency model. The RC_dag consistency is the result of the innovations based on Cilk's Location Consistency (LC). The innovations include (1)the extension of Cilk's LC with providing global synchronization and mutual exclusion, and (2)maintaining memory consistency based on thread steal/return operations. It provides programmers a user-level shared memory which is necessary for many parallel

applications.

- An Extended Stealing Based Coherence (ESBC) algorithm to reduce the network traffic in Cilk system and achieve the RC_dag consistency. It reduces the number of messages and transferred data in computation by implementing Cilk's backing store logically.

- The SilkRoad software runtime system, which supports Divide-and-Conquer, Master/Slave, and SPMD paradigms. SilkRoad is a variant of Cilk. It inherits the features of Cilk and runs a wider range of applications that may require shared variables with the paradigms other than Divide-and-Conquer.

- The concept of generic parallel programming paradigm, which is defined based on the execution instance dag of the computation and the underlying memory model. Under this framework, different paradigms are its subsets, and a mixed paradigm is defined to include several existing paradigms. This mixed paradigm is the one implemented in SilkRoad.

1.3 Organization

The rest of this dissertation is organized as follows: Chapter 2 gives a brief review on cluster computing, especially the concerned issues: parallel programming paradigms and DSMs. The Cilk system is also introduced in this chapter as a background of our research work. Chapter 3 discusses the graph theoretical analysis of parallel programming paradigms and explore their relation with memory consistency models. Chapter 4 presents the SilkRoad system, which is developed to support multiple paradigms. To

demonstrate the programmability of Cilk/SilkRoad, the solution to Salishan problems is given in Chapter 4. Chapter 5 discusses the underlying RC_dag memory consistency model in SilkRoad, including its definition, implementation, and theoretical performance analysis. Some experimental results and analysis on the results are given in Chapter 6. Finally, Chapter 7 gives the concluding remarks of this research work as well as the recommendations for future work.

Chapter 2

Literature Review

This chapter carries out a literature review to provide the background and scope of this research work. It begins with a general introduction of cluster computing. The critical review on cluster computing is focused on parallel programming paradigms and distributed shared memories, which are the relevant issues in this dissertation. As an efficient parallel runtime system for cluster computing as well as the base system of our research work, Cilk is also reviewed. At end of this chapter some remarks are presented.

2.1 Cluster Computing

Clusters [108] or network of workstations (NOW) [10, 122, 15, 5] provide low cost and high scalability in parallel computing and recently they have become important alternatives for scientific and engineering computing.

A cluster consists of a collection of interconnected stand-alone computers working together as a single, integrated computing resource. Cluster computing is implemented by connecting available commodity computers with a high speed network to do high

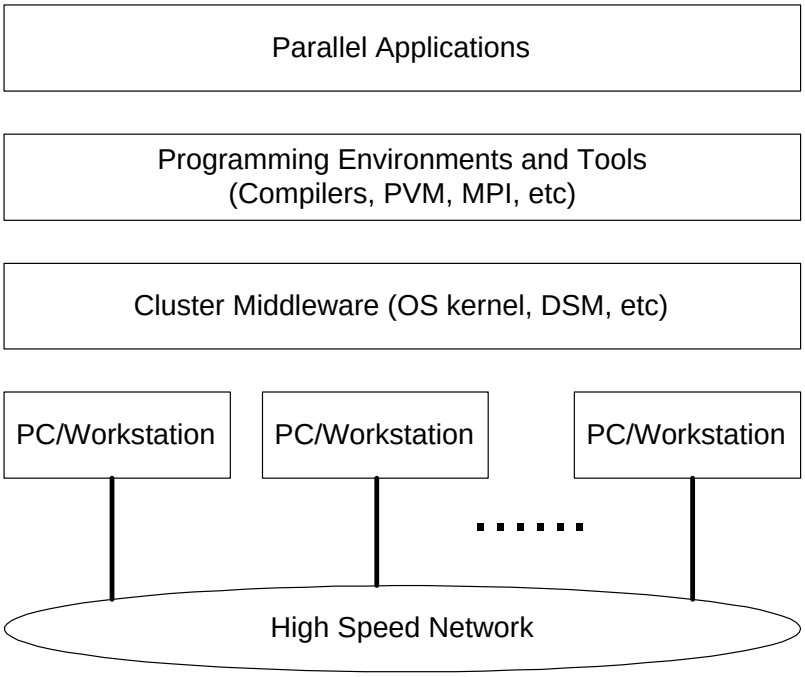


Figure 2.1: The layered view of a typical cluster.

performance computing. Because of its low cost, clustering has been an attractive approach in comparison with the high cost Massive Parallel Processing (MPP). The computer nodes of a cluster can be commodity PCs, SMPs (symmetric multiprocessors), or workstations that are connected via a Local Area Network (LAN). Figure 2.1 shows the layered view of a typical cluster. A typical cluster consists of both low-level components (such as hardware of each single node, network connections), high-level parts (such as runtime library, parallel applications, programming paradigms), and middleware (such as OS kernel, DSMs, single system image, etc.). A LAN based cluster of computers can appear as a single system to users and applications. Such a system can provide a cost-effective way to gain features and benefits that have historically been found only on more expensive centralized shared memory systems.

Besides the cost, the architecture of clusters is also advantageous. In parallel com-

puting architectures, SMPs are an attractive approach. In SMP architecture, multiple symmetric processors all have same access to the shared memory address space. One big advantage of shared memory systems (such as SMPs) is ease of programming. In shared memory systems, programmers do not need to consider how the data are located in memory and accessed by processors. However, these systems are not easy to scale up.

As another alternative, CC-NUMA (Cache Coherent Non-Uniform Memory Access) is more hardware scalable. In CC-NUMA systems, processors have non-uniform access to memory but run single OS. Even though this architecture is scalable, the software/operating system is a limitation to larger scalability. Like SMP, CC-NUMA also suffers from high availability problems.

In comparison, clusters behaves better on these aspects. A cluster can be easily scaled by adding or removing nodes from the network. This also makes clusters widely accepted as a platform for parallel computing.

2.2 Parallel Programming Models and Paradigms

In distributed systems, there are many alternatives for parallel programming models. In terms of the expression of parallelism, they can basically be classified into two categories: implicit and explicit parallel programming models.

In implicit programming models there is no need for the programmers to explicitly specify process creation, task synchronization, and data distribution. Hence, programmers do not specify any parallelism and the programs are parallelized by parallel compiler and the runtime system automatically. The implicit parallel model greatly depends

on parallelizing compilers and runtime systems such as in Jade system [114]. Normally the effectiveness of parallelizing compilers is not very satisfying without any user directions and very few systems achieved implicit parallelism ideally, especially in the cluster environment. A performance analysis on parallelizing compilers was given by Blume et al. [30].

In explicit parallelism, programmers use some special programming language constructs or invok some special functions to express parallelism. Widely used explicit parallelisms include data parallelism, message passing and the shared-memory model.

In the data parallel model, same instruction or piece of code is executed on different processors but on different data sets. In systems such as in High Performance Fortran (HPF) [88], the programmer explicitly allocates data, but there is no explicit synchronization. This model relies much on the form of the data set and it is difficult to realize parallelism with less optimally organized data sets and asynchronous operations.

The message passing model is another widely used programming model. In this model, the programmer explicitly allocates data to the processes and use explicit synchronizations. PVM [65] and MPI [126, 70] are two widely used standard libraries. Message passing systems are more flexible and can be implemented efficiently, but they require programmers to involve in low level message sending and receiving issues and this decreases the programmability.

The shared-memory model assumes that there is a shared memory space to store shared data. Typical examples include Pthreads [76] and OpenMP [104]. It is believed that the shared-memory programming model is easier to use in cluster computing than the message passing model because of the use of a single address space. Unlike in the message passing model, users do not allocate data and communicate explicitly, but they

need to synchronize explicitly. DSM models depend on compilers or system level software/hardware development to provide a shared memory on top of lower level message passing.

All the above programming models have been implemented on clusters at the middleware and programming environment level. Generally, programming models can be implemented with the following approaches:

- Introducing new features into some existing sequential programming languages with the support of pre-processors or extended compilers. Many parallel computing systems employ this approach, because it takes advantage of existing sequential programming languages. For example, C^* [127], $C//$ [134], and Cilk [44] are runtime systems based on the C language.

- Providing libraries for the programs written in a sequential programming language. Some software DSM systems (such as TreadMarks [85]) employ this approach to provide user level libraries for *C* and *Fortran* language so the programs can invoke the provided functions to utilize DSM.

- Using specifically designed parallel or concurrent programming languages. There are a number of examples such as Occam [79], Ada [2], Orca [12], etc.

Parallel programming paradigms are the ways to structure algorithms to run on a parallel system. Different people may have different classification of programming paradigms and there are several widely used programming paradigms into which most of the parallel applications can be classified. The following are popularly used ones [75, 39, 59]:

- Single Program Multiple Data (SPMD)

  SPMD is also called Phase Parallel in some cases. With SPMD, the execution of a parallel program consists of many *super steps*. Each super step has a computation phase and synchronization phase. In computation phase, multiple processes execute the same piece of code in the parallel program, but on different data set. In subsequent synchronization phase, the processes perform synchronization operations (like barrier or blocking communication).

- Divide-and-Conquer

  The Parallel Divide-and-Conquer paradigm uses the same idea as its sequential counterpart in problem solving: a parent process divides its work into two or more independent work pieces and the work pieces are done separately. In parallel computing, the resulted work pieces are done by multiple processors in parallel, and the partial results of the work pieces are merged by their upper level parent process. Usually the dividing and merging procedures are done recursively in parallel programs.

- Master/Slave

  In the Master/Slave paradigm, a master process works as the coordinator and it keeps on producing parallel work pieces and distributes them to slave processes. When the slave processes finish execution, they return their results to the master process and wait for another work piece until all the parallel work pieces have been created and finished.

- Data Pipelining

In the Pipeline paradigm, multiple processes form a virtual pipeline and a continuous data stream is input into the pipeline. In the pipeline, the output data of a process is the input data of the subsequent process. The processes execute at different stages of computation and they are overlapped in order to achieve parallelism. The hardware version of this paradigm is widely used in modern computer processors to improve the processing speed.

- Work Pool

  In this paradigm, a pool is realized as shared data structure in parallel programs to store the work pieces. Processes create work pieces and put them into the work pool. Meanwhile, processes also fetch work pieces from the pool to execute until the work pool is empty. The pool can be considered as a passive Master; also the pipeline can be considered as a distributed pool.

Usually the choice of paradigm is determined by the available parallel computing resources and the type of parallelism inherent in the problem to be solved.

2.3 Software DSMs

Because of the physically distributed memory, programmers have to manage the data transfer between cluster nodes (for example, by using message passing). DSM is an approach to integrate the advantages of SMP and message passing systems. As a cluster middleware, distributed shared memory provides a simple and general programming model for higher level programming environments by enabling shared-variable programming. DSM systems can be implemented at software and/or hardware level. Fig-

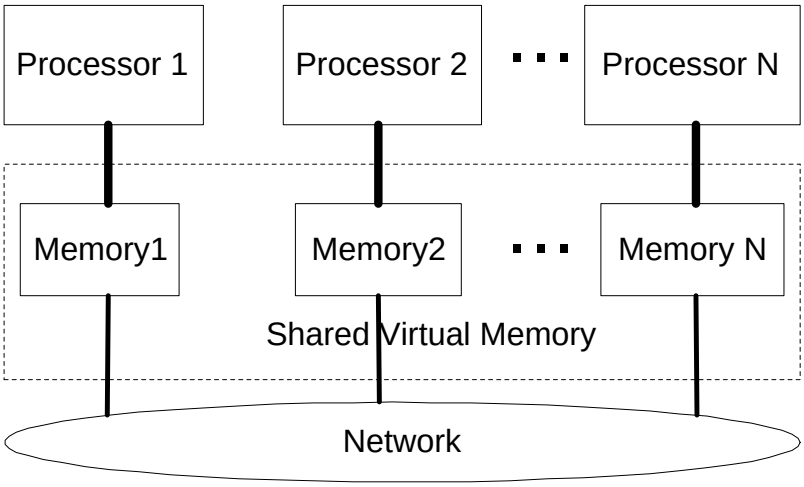


Figure 2.2: Illustration of Distributed Shared Memory.

ure 2.2 illustrates a DSM system consisting of N interconnected nodes, each of which has its own local memory and can see the shared virtual address space (denoted by dotted outline), which consists of memory pieces on each node.

In order to build a shared virtual memory among the cluster nodes, DSM systems must deal with the following problems: mapping the logically shared memory space to the physically distributed memory of each node, keeping the consistency of the data among the cluster nodes, and locating and accessing data from the memory of each node. In the software level implementation of DSMs, mapping the memory space is usually done by mapping some files in to memory. The process of locating and accessing data depends fundamentally on the consistency semantics, i.e. the memory consistency model.

In implementing a software distributed shared memory, the consistency model is critical to the behaviors and performance of the DSM. The original memory consistency model was sequential consistency [90], which was later proven to be too strict and hard

to implement efficiently in distributed environments. Some other relaxed consistency models were proposed to improve the efficiency while keeping the correctness. They will be introduced in following subsections.

Software DSM systems have the following characteristics: They are usually built as a separated layer on top of the communication interface; They take full advantage of the application characteristics; They take virtual pages, objects, and language types as sharing units. As the popularity of cluster computing grows, shared memory system is adopted as one of the approaches to achieve high performance cluster computing.

A number of software level DSMs have been implemented in cluster computing systems. Many of them were implemented as page-based DSMs, such as TreadMarks [85], SHRIMP [23], Millipede [80], CVM [128], Midway [21, 141], JIAJIA [74], ORION [101], etc; some others are object-based DSMs, such as Orca [12], Aurora [96], DOSMOS [38], CRL [83], etc.

There are some other ways to provide shared memory space in parallel programming, such as tuple space. Tuple space is to provide a way to enable different processors to share data in the form of tuples. Tuple space is a place for processors to put and share data by using “in” or “out” operations. This idea has been implemented in Linda [6, 40] and some Linda-based systems such as BaLinda [139, 140].

2.3.1 Cache Coherence Protocols

In a parallel and distributed computing environment such as clusters, there can be multiple copies of data in local memory space/cache of each processor. This raises the coherence problem, which is to ensure that no processor reads data from an obsolete copy.

Usually there are two alternative mechanisms to address this problem: *write-invalidate* and *write-update* [52]. In write-invalidate, when a datum is written, the writer processor sends invalidation messages to the other processors which may have copies of this datum, so subsequent accesses to this datum by processors other than the writer will ask the writer processor for the most up-to-date value of the datum. In write-update, the writer processor sends the new value to every other processor to update their local copies of the datum.

Each protocol has pros and cons. Write-update helps reduce average read latency but results in more inter-processor communication, while write-invalidation avoids the retrieval of information that might never be used and hence reduces the number of communicating messages but the read latency is higher. In design, a trade-off must be achieved according to the performance of the interconnection network.

2.3.2 Memory Consistency Models

The memory consistency model has a significant influence on the behavior and system performance of clusters. Generally, the memory consistency model specifies what event orderings are legal when several processes are accessing a common set of locations [66]. In other words, memory consistency models determine the value that may be returned by read operations in a sequence of parallel read and write operations.

The ultimate goal is to make systems behave like sequential machines, therefore the early choice was *sequential consistency*, which was defined by Lamport [90] as follows:

Definition 2.3.1 *A system is sequentially consistent if the result of any execution is the same as if the operations of all the processors were executed in some sequential order,*

and the operations of each individual processor appear in this sequence in the order specified by its program.

Unfortunately, sequential consistency imposes very strict ordering on memory access operations, so it can not be ideally optimized for high performance. Hence some other relaxed memory consistency models were developed in order to achieve significant performance improvements in parallel programming. The various memory consistency models are briefly introduced in the following:

1. Sequential Consistency

   Sequential Consistency (SC) says that all processors observe the same order of read and write operations of each processors on the memory. It was implemented in some early DSM systems, such as IVY [93] and Mirage [58]. Since SC precludes many potential optimizations, it is difficult to implement efficiently in loosely-coupled distributed systems.

2. Processor Consistency

   Goodman introduces *Processor Consistency* [68] in order to relax Sequential Consistency. In *Processor Consistency*, two processors may observe different orders of memory operations, so it is weaker than *Sequential Consistency*, but the order of each processor's memory operation is maintained.

3. Weak Consistency

   Dubois et al. proposes an even weaker memory consistency model, the *Weak Consistency* [51]. In *Weak Consistency*, the ordinary memory accesses are separated from the synchronization memory accesses and the memory is consistent

only on synchronization accesses.

4. Release Consistency

   In *Release Consistency (RC)* [66], synchronization accesses are further divided into *acquire* and *release*. Those memory accesses that need to be protected are performed within *acquire-release* pairs. Ordinary accesses wait until all the prior *acquire* operations complete; *release* operations also must complete for all previous ordinary accesses to become visible to other processors.

5. Entry Consistency

   Entry Consistency (EC) [19] was first introduced and implemented in Midway system [20]. It requires explicit associations of shared data with synchronization variables. On an acquire, only the data associated with the synchronization variables is guaranteed to be consistent.

6. Scope Consistency

   Scope Consistency (ScC) [78] provides a bridge between RC and EC. It uses a concept called consistency scope to implicitly establish the relationship between data and synchronization events, thus realizing a consistency model that is more relaxed than RC, without the explicit data specification of EC.

The weaker memory consistency models are proposed in order to improve the performance of clusters with DSM systems. In the meantime, programmers must be aware of the synchronization operations when using the weaker memory models.

Usually the available memory consistency models are provided by the parallel computing systems, but sometimes an application may also require a particular memory

consistency because of the problem nature. Generally, stronger consistency models simplify programming work but increase the memory access latency, while weaker consistency models improve the performance but usually require programmers to insert the relevant synchronization constructs for memory access operations.

2.3.3 Lazy Release Consistency

The Release Consistency memory model guarantees memory consistency only at synchronization points. A synchronization is represented by *acquire* or *release* operations. RC allows the notification of changes to shared memory to be deferred until the time of synchronization.

RC can be further classified into *Lazy Release Consistency (LRC)* [84, 86] and *Eager Release Consistency* [41] depending on when the modifications of memory pages are propagated.

According to TreadMarks [132], the information of changes to the shared memory can be passed from the lock releaser to the subsequent lock acquirer at either of the following two moments: when the lock is released, and when it is acquired. In the *eager* release consistency, the lock releaser notifies all processes of the modifications to shared memory pages, because the next acquirer is unknown at release time. With *lazy* release consistency, the acquirer of the lock gets the information of the changes to shared memory only when it receives the lock from the releaser, and the other processes are not aware of the information.

LRC is a refinement of *RC* and it has been implemented in the TreadMarks DSM system [85] developed at Rice university. The main idea of *LRC* in TreadMarks is

that the modifications of the pages (or *diffs*) in the shared address space are propagated only when the requirement of the diffs comes from a remote processor. The delay of propagation of *diffs* is to avoid transferring unnecessary data between processors. In TreadMarks, *LRC* does not make the modifications (which are made after a lock acquisition) visible to all processors at the time of a *release*. Instead, only the processor that acquires the same lock will get the diffs from the previous lock acquirer. Besides, TreadMarks also employs a *multiple-reader multiple-writer* protocol with some adaptive policies to help keep the coherence [9, 49, 45, 8]. Some of the coherence protocols are also widely adopted and discussed in many other research work [125, 67, 3].

Though the memory consistency models are rather mature, in the aspect of theoretical performance model and the scalability of software DSMs, there is still unexplored terrain.

2.3.4 Performance Considerations of DSMs

By relaxing the memory consistency model away from sequential consistency, software implemented DSMs can improve the performance with some advanced mechanisms, such as multiple-writer, delayed propagation, etc. Reducing the network traffic also helps improve the efficiency of processors and hence improve the computation/communication ratio.

Since the network communication is the main overhead of software DSMs in cluster computing, the performance of DSM greatly depends on the latency of the underlying network connection. Other considerations include page size, coherence protocol, granularity, address space organization, etc.

There has been a lot of work done on performance analysis of DSM systems [77, 1, 135, 54, 138, 24, 120, 133, 94], but they are basically based on experimental or empirical results of benchmarking without theoretically predictable performance models.

2.4 Introduction to Cilk

In this section, we introduce Cilk, a multithreaded parallel programming language and run-time system on which our work is based. Cilk's language features, scheduling policy, memory model theory, and the analytical performance model will be introduced.

2.4.1 Cilk Language

Cilk[1] is an algorithmic multithreaded language. "The philosophy behind Cilk is that a programmer should concentrate on structuring his program to expose parallelism and exploit locality, leaving the runtime system with the responsibility of scheduling the computation to run efficiently on a given platform. Cilk's runtime system takes care of details like load balancing and communication protocols. Unlike other multithreaded languages, however, Cilk is algorithmic in that the runtime system's scheduler guarantees provably efficient and predictable performance." [44, 50, 112]

The Cilk language is based on *ANSI C*. The basic Cilk language consists of C and some additional keywords indicating parallelism and synchronization. These keywords are: `spawn, sync, cilk, inlet, abort,` etc.

When a Cilk program is being executed, it keeps on creating ***threads*** in order to

[1]The latest version is Cilk 5.3, which is available on the Cilk website. Unless otherwise stated, in the following context, Cilk means the Cilk-NOW (also called distributed Cilk), the version for network of workstations.

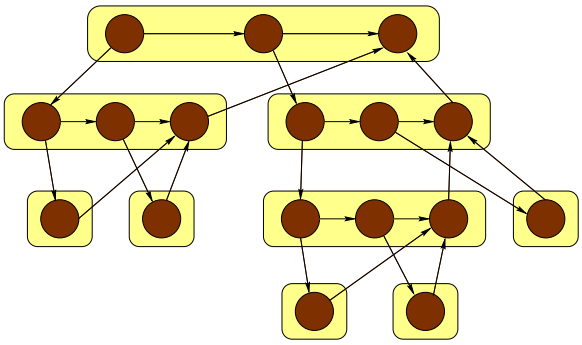


Figure 2.3: In Cilk, the procedure instances can be viewed as a spawn tree and the parallel control flow of the Cilk threads can be viewed as a dag.

explore parallelism. In Cilk terminology, a thread is a maximal sequence of instructions that ends with a `spawn`, `sync`, or `return` (either explicit or implicit) statement. A ***procedure*** in a Cilk program can be broken into a sequence of threads. The creation of threads is accomplished by the `spawn` keyword in Cilk programs. At runtime, the created threads can further "spawn" other threads, and this "spawn" relationship structures the procedures as a rooted ***spawn tree*** with their threads dag embedded, which is illustrated by Figure 2.3. In Figure 2.3, the rounded rectangles indicate procedures and the circles indicate threads. A downward edge indicates the spawning of a subprocedure. A horizontal edge indicates the continuation to a successor thread. An upward edge indicates the returning of a value to a parent procedure. All the three types of edges are dependencies which constrain the order in which threads may be scheduled.

We see that the parallel control flow of the Cilk program can be viewed as a *directed acyclic graph*, or *dag*. *Dag* is an important theoretical basis of Cilk, which will be discussed in later sections.

Cilk programs are pre-compiled to *C* programs before they are executed. To explore the power of local processors and at the same time enable the parallelism, Cilk proce-

dures can be executed in fast and slow style, corresponding to local running and remote stealing respectively (work stealing is introduced in the following subsection). When there are no steal requests, procedures are executed in a fast style, which is comparable to normal *C* procedure execution. In the case of stealing, slow style is used in order to pass additional information to support parallel execution.

The basic parallel programming paradigm of Cilk is Divide-and-Conquer. By using the Divide-and-Conquer strategy, a Cilk program separates a problem into smaller problems by recursively spawning threads which are assigned smaller computation tasks.

2.4.2 The Work Stealing Scheduler

In parallel computing, scheduling is critical to the efficiency of the whole system. Different scheduling policies may result in quite different performance.

Generally, it is hard to achieve pre-scheduled load balancing for the Divide-and-Conquer paradigm because of its dynamism. For dynamic parallelism, usually a dynamic scheduling policy is adopted. There has been a lot of work done on dynamic load balancing [102, 25, 115, 18, 136] and scheduling policies [47, 26, 99, 100, 89, 27, 48, 26, 99, 136] for parallel systems, and Cilk is the one using work stealing [37, 36, 35] and thread migration [129, 82, 119].

In the Cilk runtime system, a ***work-stealing*** based randomized scheduling policy is employed [36, 63, 34]. During the execution, when a processor runs out of work, it will actively "steal" work from other busy processors by randomly choosing a "victim" processor.

The spawn tree is explored in a depth-first manner. In implementation, the proce-

dures are managed by using a double ended queue (*deque*). The bottom of the *deque* can be pushed in or popped out, while the top can only be popped out. When a child procedure is spawned, the local variables of the parent are saved on the bottom of the *deque* and the processor begins to execute the child procedure. When the child procedure returns, the bottom of the *deque* is popped and the parent resumes. On the other hand, if there is a steal from another processor, the top-most procedure in the *deque* is popped out and sent to the stealing node. This is to make sure that the stealing node steals the shallowest ready thread in the victim's spawn tree in order to steal as much work as possible.

To implement the above work-stealing scheduler efficiently, the *THE* protocol is employed, which uses three atomic shared variables T, H, and E to realize the mutual exclusion on the *deque*. The details of the *THE* protocol can be found in [63]. However, the sharing of information between the source and destination processors of a stolen procedure gives rise to new memory consistency issues.

2.4.3 Memory Consistency Models

Memory consistency models are an important issue to programmers in distributed environment. Cilk people developed a computation-centric theory [62, 61, 60] of memory consistency model for parallel multithreaded computations. Based on the *dag*-consistency model [33, 32], a series of related memory consistency models were developed for the Cilk-like multithreaded computations. In Cilk, the *dag-consistency* is implemented by using the BACKER algorithm (which is introduced in this subsection later).

Computation-Centric Memory Consistency Model

Comparing with the ***processor-centric*** memory models [51, 4, 68, 86, 66, 20, 78], which are expressed in terms of processors acting on memory, the ***computation-centric*** memory model is more focused on the computation itself. The philosophy of computation-centric memory model is to separate the logical dependencies among instructions (the computation) from the way instructions are mapped to processors (the schedule) [62]. This approach leads to defining formal properties of memory models that are implementation independent.

The computation-centric memory model theory is based on the concept of ***computation*** and ***observer function***, which are defined as follows [62, 61]:

Definition 2.4.1 *A **computation** $C = (\mathcal{G}, op)$ is a pair of a finite directed acyclic graph (dag) $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ and a function op: $\mathcal{V} \mapsto \mathcal{O}$, where $\mathcal{V}$ is the set of all nodes in the dag, $\mathcal{E}$ is the set of all edges in the dag, and $\mathcal{O}$ is a set of abstract instructions (such as read and write).*

In computation-centric memory model theory, a ***memory*** is characterized by a set $\mathcal{L}$ of locations and a set $\mathcal{O}$ of abstract instructions. Intuitively, each node $u \in \mathcal{V}$ represents an instance of the instruction $op(u)$, and each edge indicates a dependency between its endpoints. Reads and writes to location l are denoted by $R(l)$ and $W(l)$ respectively. In the *dag* of a computation, if there is a path from node u to node v, then it is said that u ***precedes*** v, which is denoted by $u \preceq v$. To indicate strict precedence, we write $u \prec v$. Two nodes u and v are *incomparable* if $u \not\prec v$ and $v \not\prec u$. An empty element $\perp$ indicates that no write operation has been observed and $\perp \prec u$ for every node u of any computation. Based on the above semantics of computation, an ***observer function***

is defined as follows:

Definition 2.4.2 *An **observer function** for a computation C is a function Φ: $\mathcal{L} \times \mathcal{V} \cup \{\perp\} \mapsto \mathcal{V} \cup \{\perp\}$ satisfying the following properties for all $l \in \mathcal{L}$ and $u \in \mathcal{V} \cup \{\perp\}$:*

1. *If $\Phi(l, u) = v \neq \perp$ then $op_C(v) = W(l)$.*
2. *$u \not\prec \Phi(l, u)$.*
3. *If $u \neq \perp$ and $op_C(u) = W(l)$ then $\Phi(l, u) = u$.*

On the basis of the concepts of computation and observer function, ***memory model*** is defined as a set of pairs of computations and observer functions:

Definition 2.4.3 *A memory model is a set Δ such that $\{(\epsilon, \Phi_\epsilon)\} \subseteq \Delta \subseteq \{(C, \Phi) : \Phi$ is an observer function for $C\}$.*

In computation-centric memory model theory, the strictness of memory models is compared according to the following definition:

Definition 2.4.4 *A **memory model** Δ is **stronger** than a memory model Δ' (or memory model Δ' is **weaker** than memory model Δ) if $\Delta \subseteq \Delta'$.*

It means subset is stronger because the subset allows fewer memory behaviors.

There are also some properties defined in computation-centric theory, i.e. constructibility, completeness, etc, which are fully discussed in [61], [60] and [62].

Dag Consistency and Location Consistency

Initially the *dag* consistency was developed to support the Cilk multithreaded parallel programming, and later it was enlarged to be a family of consistency models, including ***location consistency*** [61] [2]. The memory models can be defined based on the ***topological sorts*** of the *dag* of computation and the ***last writer function***. A ***topological sort*** T of a *dag* graph $\mathcal{G}$ is a total order on the node set $\mathcal{V}$ consistent with the precedence relation. The set of all topological sorts of a *dag* graph $\mathcal{G}$ is denoted by $TS(\mathcal{G})$. The ***last writer function*** is defined as follows:

Definition 2.4.5 *Let C be a computation, and $T \in TS(C)$ be a topological sort of C. The last writer function according to T is $W_T : \mathcal{L} \times \mathcal{V} \cup \{\perp\} \mapsto \mathcal{V} \cup \{\perp\}$ such that for all $l \in \mathcal{L}$ and $u \in \mathcal{V} \cup \{\perp\}$:*

1. *If $W_T(l, u) = v \neq \perp$ then $op_C(v) = W(l)$.*
2. *$W_T(l, u) \preceq_T u$.*
3. *$W_T(l, u) \prec_T v \preceq_T u \Rightarrow op_C(v) \neq W(l)$ for all $v \in \mathcal{V}$.*

The ***last writer function*** is actually an observer function for computation.

Based on the topological sorts and last writer functions, ***sequential consistency*** and ***location consistency*** are defined in computation-centric theory as follows respectively:

Definition 2.4.6 ***Sequential Consistency*** *is the memory model*

$$SC = \{(C, W_T) : T \in TS(C)\}$$

[2] This location consistency is not the model with the same name introduced by Gao and Sarkar [64]. [60] has the detailed justification.

Definition 2.4.7 ***Location Consistency*** *is the memory model*

$$LC = \{(C, \Phi) : \forall l \exists T_l \in TS(C) \forall u, \Phi(l, u) = W_T(l, u)\}$$

According to the above definitions, sequential consistency requires that the topological sort be the same for all locations, while location consistency requires that all writes to the same location behave as if they were serialized, so location consistency is weaker than sequential consistency. In location consistency memory, for all location l there exists a topological sort T_l of computation such that every read operation on location l returns the value of the last write to location l occurring in T_l. The whole *dag* consistency memory model family is fully discussed in [61].

Location consistent shared memory is developed for fully strict multithreaded computations [31], which means in the *dag* of a computation every dependency edge goes from a procedure to either itself or its parent procedure. The computations of Cilk programs are fully strict because the result of a Cilk procedure can only be returned to the procedure that calls it. According to the semantics of the Divide-and-Conquer strategy, a big problem is divided into many independent small problems, so there are no interacting mechanisms for the sibling Cilk threads. However, for those computations which are not fully strict, the semantics of the memory model and computations have to be modified or redefined. This problem will be further addressed in Chapter 3.

The BACKER Algorithm

The BACKER algorithm [33] was proposed to implement the *dag* consistency. With the BACKER algorithm, shared memory locations can have different versions in any of the processor caches and the main memory – *backing store*, which is the home of the data

of memory locations. In order for each processor to access the most up-to-date data of a memory location, the data must be transferred from the *backing store* to the local cache of the processor first.

The BACKER algorithm works as follows: there are three basic operations for processors to operate on shared memory locations: ***fetch, reconcile,*** and ***flush***. A ***fetch*** operation copies a location from the *backing store* to the cache of a processor and marks the cached location as clean, so the processor has the most recent copy of the location. A ***reconcile*** operation copies a dirty location from a processor cache back to the backing store in order to keep the copy at "home" most up-to-date. Meanwhile, the cached location is marked as clean. Lastly, a ***flush*** operation removes a clean location from a processor's cache.

The shared memory is kept coherent by the BACKER by using the above three basic operations: When a processor accesses (read from or write to) a memory location, the operation is performed on the copy in its local cache. If a copy is not present in the local cache, it will fetch from the backing store to get the latest version and then perform the operation. For write operations, the dirty bit will be set. Since the capacity of the cache is limited, sometimes it is necessary to flush some clean locations to make space for the new locations. To remove dirty locations, processors first reconcile and then flush them.

The BACKER algorithm also performs additional reconciles and flushes to enforce location consistency besides the three basic operations. For each edge $u \prec v$ in the *dag* of a computation, if nodes u and v are on different processors p and q respectively, then p will reconcile all its cached locations after executing u but before enabling v, and q will also reconcile and flush its entire cache before executing v.

The BACKER algorithm uses a convenient way to keep the coherence of the copies

in different locations, and it is not so complex to implement. It is similar to "home-based" coherence protocol [45], in which case the home keeps the most recent version of location data and the processors always keep in touch with the home. Actually, the backing store can also be logical: ***reconcile*** just makes the local cache pages "latest" and other remote copies "invalid"; when the other copies are accessed, page misses occur and this will cause data transfer from the processor where the "latest" version is located. This is the approach we have adopted in our work.

2.4.4 The Performance Model

A lot of work has been done on the performance bounds of parallel computing with various methodologies, such as fork/join [97, 89], heterogeneous systems [14], DSM systems [87, 95, 121, 98], multithreaded multiprocessors [43], etc.

Cilk provides users an algorithmic model of application performance to predict the runtime of Cilk programs. The execution time of Cilk programs can be measured in terms of its ***work*** and ***critical path length***. The ***work*** of a computation, denoted T_1, is the number of instructions in the *dag* of the computation, which corresponds to the amount of time required by an one-processor execution. The ***critical path length*** of a computation, denoted T_∞, is the maximum number of instructions on any directed path in the *dag* of the computation, which corresponds to the amount of time required by an infinite-processor execution.

For fully strict multithreaded algorithms running on P processors, Cilk's randomized work stealing scheduler achieves performance close to a lower bound, which is T_1/P or T_∞. Specifically, for any such algorithm and any number P of processors, the

Cilk scheduler executes the algorithm in expected time $O(T_1/P + T_\infty)$ [31, 36].

The above model accounts for various overheads introduced in T_1 and T_∞ by the operating system, shared memory protocol, etc. Observing some factors (i.e. cache size, cache miss service time, etc) which affect running times, Cilk's performance model is further refined [32, 61]. With cache size Z which is partitioned into Z/L lines of length L on each processor, ***total work*** $T_1(Z, L)$ is defined as the serial execution time on a machine with a (Z, L) cache, and T_1 is referred as the ***computational work***, which corresponds to the serial execution time if all cache misses take zero time to be serviced. The number of cache misses taken in the serial execution is defined as ***serial cache complexity***, which is denoted by $Q(Z, L)$, so there exists $T_1(Z, L) = T_1 + \mu Q(Z, L)$. Similarly, the critical-path length can also be split into two portions: One is ***total critical-path length*** $T_\infty(Z, L)$, which is the maximum overall directed paths in the *dag* of computation, including the cache misses, to execute along the path by a single processor with a (Z, L) cache. The other is ***computational critical-path length*** T_∞ with zero cache miss cost of the time. The following theorem [61] bounds the parallel execution time of multithreaded Cilk programs:

Theorem 2.4.8 *Consider any fully strict multithreaded computation executed on P processors, each with an LRU cache of height H, using the Cilk work-stealing scheduler in conjunction with the BACKER coherence algorithm. Let μ be the service time for a cache miss that encounters no congestion, and assume that accesses to the main memory are random and independent. Suppose the computation has T_1 computational work, $Q(Z, L)$ serial cache misses, $T_1(Z, L) = T_1 + \mu Q(Z, L)$ total work, and T_∞ critical-path length. Then for any $\epsilon > 0$, the execution time is $O(T_1(Z, L)/P + \mu H T_\infty + \mu \lg P +$*

$\mu H \lg(1/\epsilon))$ *with probability at least* $1 - \epsilon$. *Moreover, the expected execution time is* $O(T_1(Z, L)/P + \mu H T_\infty)$.

Proof: See [61]. □

2.5 Remarks

Cluster computing is a rapidly growing technology for parallel and distributed computing and a number of parallel systems have been developed for clusters. Some cluster computing systems use distributed shared memory as a middleware, and the memory consistency model of DSM acts as an underlying base of the high level programming paradigms.

Even though many parallel systems have been developed for cluster computing, few of them are efficient in multiple paradigms. In addition, parallel programming paradigms are not formally and systematically analyzed.

Cilk is an efficient multithreaded parallel runtime system. In cluster environment, there is no user level shared memory in Cilk, because Cilk was initiated for solving problems by using Divide-and-Conquer with recursion. However, in cluster computing, cluster-wide user level shared virtual memory is necessary in many cases in order to run parallel applications using shared variables with some paradigms other than Divide-and-Conquer. So it is valuable and desirable to develop a runtime system to support a wider range of applications with more paradigms.

Chapter 3

The Mixed Parallel Programming Paradigm

This chapter[1] elaborates on the idea of developing and supporting multiple parallel programming paradigms by using formal methods based on computation-centric theory of memory model [62].

Parallel programming paradigms (or parallel algorithmic paradigms) define ways to structure algorithms to run on a parallel system [75]. Usually a particular parallel system is good at one particular programming paradigms. This limits the generality of parallel runtime systems. So it is desirable to enable a parallel system to support mixed programming paradigms. In this chapter, formal methods are developed to address the problem of supporting multiple paradigms. We show a graph-theoretical way to realize mixed parallel programming paradigms from the viewpoint of memory models.

The execution of both sequential statements and parallel control statements of a par-

[1]The contents of this chapter are partially published in [106].

allel program can be described by an *execution instance dag* $G = (V, E)$, where the node set V represents the execution of a *parallel task*, forking of tasks, or synchronization, and the edge set E represents the data dependencies between the nodes.

Given that the execution instance dags of different paradigms have different features, we define a general parallel programming paradigm with common attributes of dags and then define specific paradigms (such as SPMD, Divide-and-Conquer, Master/Slave, etc) with their special features on the dags. Under this graph-theoretical framework of paradigms, we show that SPMD, Divide-and-Conquer, and Master/Slave are all subsets of the general paradigm. We further extend Divide-and-Conquer to the *mixed paradigm* and this mixed paradigm is a super-set of Divide-and-Conquer, SPMD, and Master/Slave.

We also observe that there are some relationships among the strictness of computation, paradigms, and the memory consistency models. According to computation-centric memory model theory, Cilk with LC memory consistency model is adequate for the *fully strict* multi-threaded computation under the Divide-and-Conquer paradigm. Based on this, the memory model is extended with RC_dag consistency (see details in Chapter 5), and the resulted SilkRoad is capable of supporting a wider range of computation and more paradigms besides Divide-and-Conquer. This implies that under the graph-theoretical framework of parallel programming paradigm, by extending the underlying memory consistency model, it is possible to support less strict computation and more paradigms.

The remainder of this chapter is organized as follows: Section 3.1 proposes the graph theory for parallel programming paradigms, then some widely used parallel paradigms are formally defined in Section 3.2. In Section 3.3, we define the mixed paradigm

and analyze its relationship with parallel computation. Some related work is introduced in Section 3.4. Finally, Section 3.5 gives a summary for this chapter.

3.1 Graph Theory of Parallel Programming Paradigm

There are many different definitions for parallel programming paradigm. The parallel programming paradigm we discuss here is the algorithmic paradigm. A programming paradigm is a class of algorithms that solve different problems but have the same control structure [72]. In some cases, it is also called parallel programming model.

As suggested by Ian Foster [59], the design of parallel algorithms consists of four distinct stages: (1)Partitioning; (2) Communication; (3)Agglomeration; and (4)Mapping. The agglomeration and mapping are lower level concrete implementation considerations (they have, for example, a close relationship with load balancing), while the partitioning and communication depend on abstract semantics of the higher level programming paradigms. That means the paradigms should define the means of partitioning and communication of the tasks.

We try to characterize parallel programming paradigms from the viewpoint of program execution. We say a graph-view of an execution of a parallel program is an *execution instance*. Note that the execution instance dag defined in this chapter is different from the dag defined in computation-centric memory model theory, in that the set of nodes in the latter is defined in a finer granularity (it represents the instances of instructions like read and write operations on memory) and does not include the parallel control constructs. Before we formally define execution instance, some notations and items used in this chapter are introduced first.

In a parallel program which implements a parallel algorithm, the statements can be divided into two classes: the sequential computing statements (including assignments, selection statements, repetition statements, function calls, etc) and the parallel control statements which deal with parallel control issues such as decomposition and synchronization. Many paradigms require some parallel control statements at the programming level to explicitly express parallelism.

In various parallel control statements, "fork/join" is a widely used for dynamic parallelism. In fork/join parallelism, the sequential computation is split by "fork"ing some tasks and the results are then "join"ed, so that there should be some forking and joining statements acting as parallel control statements.

"Barrier" and "lock" are two other widely used parallel control mechanisms for global synchronization and mutual exclusion. In following discussion we use these parallel control statements (i.e. fork, join, barrier, and lock) to express the decomposition of a sequential computation as well as the communication in programs.

To simplify the analysis of various parallel paradigms, it is assumed that parallel programs with a certain kind of paradigm can be written in a base parallel language L, which consists of a set of necessary sequential statements plus a set of parallel control statements.The EBNF syntax of statements in L can be demonstrated as follows (SS is for sequential statements; PS is for parallel control statements; SL is for statement list; token **id** represents a variable; and token **expr** represents an expression):

SS ::= [ ]

| **id** = **expr**

| **if** E **then** SL [**else** SL] **end**

| **for** $Name$ ' := ' E **to** E [by E] **do** SL **end**

| **return** E

PS ::= [ ]

| **fork** SL | **join** | **lock (E)** | **unlock (E)** | **barrier**

| **forall** $Name'$:=' E **to** E $[by\ E]$**do** SL **end**

SL ::= $(SS \mid PS)\{;\ SL\}$

Hence the execution of the parallel program is separated into many "small executions" by performing parallel control operations specified by the parallel control statements in the program written in L. Each small execution is a parallel task, which is defined as follows:

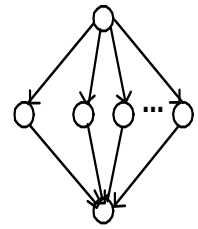
Definition 3.1.1 *A (parallel) task is an execution between two consecutive parallel control operations on a processor during the running of a parallel program written in base parallel language L.*

From the definition we know that a parallel task is a dynamic concept. It is the basic unit of continuous execution without being interrupted by parallelizing constructs on a processor.

To clarify the above definition, Figure 3.1(a) shows the core pseudo code of sequential matrix multiplication (A, B, R are matrices and $R = A \times B$), and Figure 3.1(b) shows its parallelized counterpart. In Figure 3.1(b), the data is (implicitly) divided into several parts (i ranges from $start\_Pi$ to end_Pi) to be processed in parallel. There is also a global synchronization (the *barrier* statement) in order to join all of the partial results. In this example, each sub-computation between the start point and the joining point can be treated as a parallel task. Similarly, In Figure 3.2, the pseudo code of calculating Fibonacci numbers is shown (both the sequential and the parallel version). In

```
for i := 0 to matrix_size do
  for j := 0 to matrix_size do
    for k := 0 to matrix_size do
      R( i, j ) += A( i,k ) x B( k,j );
    end
  end
end
```
(a)

```
for i := start_Pi to end_Pi do
  for j := 0 to matrix_size do
    for k := 0 to matrix_size do
      R( i , j ) += A( i ,k ) x B( k, j );
    end
  end
end
barrier;
```
(b)



(c)

Figure 3.1: Demonstration of a parallel matrix multiplication program ($R = A \times B$) and its execution instance dag.

the parallel code, "fork" and "join" are used to specify the explicit decomposing and synchronization of the tasks, while no explicit fork is used in Figure 3.1.

Like in the graph analysis of sequential programs, there may also be precedence constraints between the parallel tasks. Some tasks cannot be executed until some other tasks has been finished. In our notations, $\prec$ is used to describe this relationship. For tasks u and v, if u must be executed before v and there is no other tasks in between, then we say $u \prec v$, which means there is an edge e from u to v and it is denoted as $e : (u, v)$. If $u \prec v$, we say u is the immediate predecessor of v, or $pred(v) = u$ and v is the immediate successor of u, or $succ(u) = v$. This precedence constraints also apply to the parallel control operations.

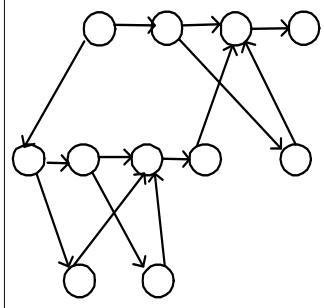
Definition 3.1.2 *An execution instance of a parallel program is a directed acyclic graph (dag) $G = (V, E)$, where V is a set of vertices representing tasks and parallel control operations, E is a set of edges representing precedence constraints between tasks and parallel control operations. V and E are defined as follows:*

```
Fib (n)
   if n < 2 then return n;
    else
         x = Fib (n-1);
         y = Fib (n-2);
         return (x + y);
   end
```

(a)

```
Fib (n)
   if n < 2   then return n;
   else
     x =  fork  Fib (n-1);
     y =  fork  Fib (n-2);
     join ;
     return (x + y);
   end
```

(b)



(c)

Figure 3.2: Demonstration of a program calculating Fibonacci numbers and its execute instance dag

- $V = V_T \bigcup V_F \bigcup V_S$, *where*
  - $V_T = \{v | v$ *is a node representing an execution of a task* $\}$,
  - $V_F = \{v | v$ *is a node and outgoing degree*$(v) > 1\}$, *and*
  - $V_S = \{v |$*incoming degree*$(v) > 1\}$.
- $e = (u, v) \in E$ *iff* $u \prec v$. *Furthermore,* $E = E_D \bigcup E_S$, *where*
  - $E_D = \{(u, v) | u \in V_F\}$ *and*
  - $E_S = \{(u, v) | u \in V_S \vee v \in V_S\} \bigcup \{(u, v) | u \in V_T \wedge v \in V_T\}$.

The vertices in G are divided into three classes: task nodes V_T, forking nodes V_F, and synchronization nodes V_S. Specifically, the forking nodes represent the behavior of splitting computation into many tasks under a paradigm. This can be done implicitly or explicitly, statically or dynamically, depending on the semantics of the different paradigms. An explicit way is using the fork parallel control statement. For static splitting, we have $|V_F| = 1$, which means the splitting is done at the beginning of the compu-

tation by the forking node (the start node v_0). For those paradigms which split computation dynamically, we have $|V_F| > 1$, which means besides the v_0, there are some other splittings during the computation. The outgoing edges from the nodes in V_F are then defined as distributing edges E_D, which represents the behavior of distributing the tasks to the available processors. Lastly, V_S is abstracted from the synchronization/joining of tasks. Intuitively they are the joining of partial results or barrier-like synchronization between the tasks. It actually shows the data dependence in parallel computation. V_S can represent either partial or global synchronization. The edges connecting to or from V_S are also classified into *synchronization edges* E_S. E_S also includes the *mutual edges* which represent the synchronization between two individual tasks (intuitively, this is abstracted from "lock" and "unlock" operations in computation).

To further illustrate the definition of execution instance dag, Figure 3.1(c) and Figure 3.2(c) show the parallel execution instance dags corresponding to the parallelized program in Figure 3.1(b) and Figure 3.2(b) respectively.

Paradigms differ in partitioning computation, mapping tasks to processors, and defining the synchronization manners of tasks. So their execution instance dags also differ from each other. The implementation of a paradigm depends on the runtime system with the scheduling and load balancing policies. The details of implementation are out of the scope of our discussion and we only care about the semantics of the abstract logical execution of computation.

With memory model definition in the computation-centric theory [62] (see Chapter 2) and the definition of execution instance dag, now parallel programming paradigm can be defined as a set of tuples of parallel programs, memory model, and the execution instance dags:

Definition 3.1.3 *A parallel programming paradigm is a set* $\Psi_{gp} = \{(A, \Delta, G) : A$ *is a parallel program and* G *is an execution instance dag of* A *on memory model* $\Delta\}$.

We say Ψ_{gp} is a ***general parallel programming paradigm*** for parallel computing. It defines the general elements in paradigms. However, specific paradigms may also vary in semantics with the underlying runtime system and scheduler. In the following section we define several widely used paradigms and prove that they are subsets of the general paradigm.

3.2 Some Specific Paradigms

In this section, we define and analyze several specific paradigms, i.e. SPMD, Divide-and-Conquer, and Master/Slave and their relationships with the general paradigm.

SPMD

Definition 3.2.1 *The SPMD (Single Program Multiple Data) paradigm* $\Psi_{SPMD} = \{(A, \Delta, G)|$ *where* $G = (V, E)$ *is an execution instance dag of parallel program* A *on memory model* Δ *on* P *processors* $\}$, *the vertices set* V *and edges set* E *are defined as follows:*

- $V = V_T \bigcup V_F \bigcup V_S$, *where*
  - $V_T = \{v|v$ *is a node representing an execution of a task* $\}$,
  - $V_F = \{$ *start node* $v_0\}$, *and*
  - $V_S = \{$ *outgoing degree*$(v) = 0$ *or* $P \wedge$ *incoming degree*$(v) = P\}$.

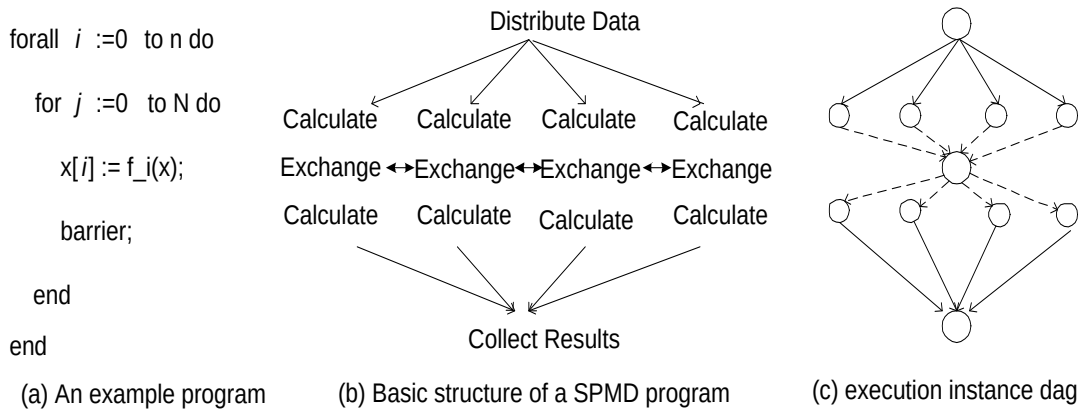


Figure 3.3: The structure and execution instance dag of SPMD programs

- $e = (u, v) \in E$ *iff* $u \prec v$. *Furthermore,* $E = E_D \bigcup E_S$, *where*
  - $E_D = \{(u, v) | u \in V_F\}$ *and*
  - $E_S = \{(u, v) | u \in V_S \vee v \in V_S\}$.

In SPMD the data is statically divided into P partitions at the start node v_0 (where P is the number of processors). The processors then execute the same piece of program code operating on different data sets. During the computing, the sub-computation may need to synchronize at the barrier nodes. The partial results on all processors will be joined together at the final vertex of the execution instance dag. Figure 3.3 shows a sample SPMD program, the structure of typical SPMD programs (figure(b) comes from [39]), and the demonstrative execution instance dag.

In the following, the relationship between the SPMD paradigm and the general paradigm is discussed. First, it is necessary to define the "$\subsetneq$" relation of two graph sets. We say set A is a subset of set B (denoted by $A \subsetneq B$) if for any element $a \in A$, there exists an "equivalent" element $b \in B$, and there exists at lease one element c in B

which has no equivalent elements in A. Here we say element b is "equivalent" to element a means b is a ($b = a$) or b is isomorphic to a. This "equivalent" concept can also be applied to sets of multi-tuples, in which case an element is a multi-tuple. We say a multi-tuple $(a_1, a_2, ..., a_n)$ is equivalent to a multi-tuple $(b_1, b_2, ..., b_n)$ if a_1 is equivalent to b_1, a_2 is equivalent to b_2, ..., and a_n is equivalent to b_n.

Theorem 3.2.2 $\Psi_{SPMD} \subsetneq \Psi_{gp}$.

Proof: $\forall$ a tuple $(A, \Delta, G) \in \Psi_{\text{SPMD}}$, there is an execution instance dag $G_{\text{SPMD}} = (V_{\text{SPMD}}, E_{\text{SPMD}})$. we construct an isomorphic graph $G' = (V', E')$ of G and then prove the equivalent three tuple $(A, \Delta, G') \in \Psi_{gp}$. We first construct an isomorphic graph G' by following steps:

1. Let G' has the same $|V|$ as G_{SPMD}, so V' can also be defined as $V' = V'_T \bigcup V'_F \bigcup V'_S$ similarly where V'_T, V'_F, and V'_S have the same semantics as $V_{T_{\text{SPMD}}}, V_{F_{\text{SPMD}}}$, and $V_{S_{\text{SPMD}}}$ respectively.

2. Uniquely tag all the vertices in V_{SPMD} with 0 and natural numbers, say, tag the start node with 0 and final node with $k - 1$, where $k = |V_{\text{SPMD}}|$.

3. Carry the tags of the nodes in G_{SPMD} over to the nodes in G'.

4. For each node $v_i \in V_{\text{SPMD}}(i = 0, 1, 2, ...k - 1)$, for all edges connecting v and other adjacent nodes in V_{SPMD}, create corresponding edges connecting v' and corresponding nodes in V'.

So we have a graph G' which is isomorphic to G_{SPMD}. Now we prove that the equivalent three tuple $(A, \Delta, G') \in \Psi_{gp}$:

$\forall v' \in V'$, we consider three possibilities:

1. if $v' \in V'_T$, v' represents an execution of a task, which is mapped from the vertices in G_{SPMD}, so $v' \in V_{T_{gp}}$;

2. if $v' \in V'_F$, then we have $v' = v'_0$, so $v' \in V_{F_{gp}}$;

3. if $v' \in V'_S$, according to definition of Ψ_{SPMD}, we have *incoming degree*$(v') = P > 1$, so $v' \in V_{S_{gp}}$.

$\forall e' = (u', v') \in E'$, two possibilities are considered:

1. if $e' \in E'_D$, since $u' \in V_{F_{gp}}$, so $e' \in E_{D_{gp}}$;

2. if $e' \in E'_S$, then $u' \in V_{S_{gp}}$ or $v' \in V_{S_{gp}}$, according to Definition 3.1.2, e' also $\in E_{S_{gp}}$.

So G' is also an execution instance dag and according to Definition 3.1.3, $(A, \Delta, G') \in \Psi_{gp}$. That is to say, for every $(A, \Delta, G_{\text{SPMD}})$ tuple in Ψ_{SPMD}, we can construct a G' such that there is a equivalent three tuple $(A, \Delta, G') \in \Psi_{gp}$.

On the other hand, we can find at least one (A, Δ, G) in Ψ_{gp} such that (A, Δ, G) is not in Ψ_{SPMD}. For example, we can find a G whose $|V_F| > 1$, which means it has some other distributing nodes besides the start node v_0, so this G is not an execution instance of Ψ_{SPMD}, so this tuple $(A, \Delta, G) \notin \Psi_{\text{SPMD}}$. Finally we have $\Psi_{\text{SPMD}} \subsetneq \Psi_{gp}$. $\square$

Divide-and-Conquer

Definition 3.2.3 *The Divide-and-Conquer paradigm* $\Psi_{DC} = \{(A, \Delta, G)|$ *where* $G = (V, E)$ *is an execution instance dag of parallel program* A *on memory model* $\Delta\}$*, and the vertices set* V *and edges set* E *are defined as follows:*

- $V = V_T \bigcup V_F \bigcup V_S$*, where*

- $V_T = \{v | v$ *is a node representing an execution of a task* $\}$,
- $V_F = \{$ *start node* $v_0\} \bigcup \{v |$*incoming degree*$(v) = 1 \wedge$ *outgoing degree*$(v) > 1\}$, *and*
- $V_S = \{v |$*incoming degree*$(v) > 1 \wedge$ *outgoing degree*$(v) = 1\}$.

- $e = (u, v) \in E$ *iff* $u \prec v$. *Furthermore,* $E = E_D \bigcup E_S$, *where*
  - $E_D = \{(u, v) | u \in V_F\}$ *and*
  - $E_S = \{(u, v) | v \in V_S\}$.

Theorem 3.2.4 $\Psi_{DC} \subsetneq \Psi_{gp}$.

Proof: $\forall$ a tuple $(A, \Delta, G) \in \Psi_{\text{DC}}$ where $G_{\text{DC}} = (V_{\text{DC}}, E_{\text{DC}})$, we construct an isomorphic graph $G' = (V', E')$ of G by using the same method of theorem 3.2.2. In the following we then prove $(A, \Delta, G') \in \Psi_{gp}$.

$\forall v' \in V'$, three possibilities are considered:

1. if $v' \in V'_T$, v' represents an execution of a task, which is mapped from the vertices in G_{DC}, so $v' \in V_{T_{gp}}$;
2. if $v' \in V'_F$, then we have *incoming degree*$(v') = 0$ or 1 and *outgoing degree*$(v') > 1$, that is to say v' is the start node v'_0 or a node with *outgoing degree*$(v') > 1$, so $v' \in V_{F_{gp}}$;
3. if $v' \in V'_S$, according to definition of Ψ_{DC}, we have *incoming degree*$(v') > 1$, so $v' \in V_{S_{gp}}$.

$\forall e' = (u', v') \in E'$, two possibilities are considered:

1. if $e' \in E'_D$, since $u' \in V_{F_{gp}}$, so $e' \in E_{D_{gp}}$;

2. if $e' \in E'_S$, then $v' \in V_{S_{gp}}$, according to Definition 3.1.2, e' also $\in E_{S_{gp}}$.

So G' is also an execution instance dag and according to Definition 3.1.3, $(A, \Delta, G' \in \Psi_{gp}$. That is to say, for every $(A, \Delta, G_{\text{DC}})$ tuple in Ψ_{DC}, we can construct a G' such that $(A, \Delta, G') \in \Psi_{gp}$.

On the other hand, we can find at least one (A, Δ, G) in Ψ_{gp} such that (A, Δ, G) is not in Ψ_{DC}. For example, we can find a G whose $|V_F| = 1$, which means it has only one forking node, i.e. the start node v_0, so this execution instance dag G is not an execution instance of Ψ_{DC}, so this tuple $(A, \Delta, G) \notin \Psi_{\text{DC}}$. Finally we have $\Psi_{\text{DC}} \subsetneq \Psi_{gp}$. $\square$

Figure 3.2(c) shows the execution instance dag of program Fib(3).

Master/Slave

Definition 3.2.5 *The Master/Slave paradigm* $\Psi_{MS} = \{(A, \Delta, G)|$ *where* $G = (V, E)$ *is an execution instance dag of parallel program* A *on memory model* $\Delta\}$*, and the vertices set* V *and edges set* E *are defined as follows:*

- $V = V_T \bigcup V_F \bigcup V_S$*, where*
  - $V_T = \{v|v$ *is a node representing an execution of a task* $\}$*,*
  - $V_F = \{$ *start node* $v_0\}$*, and*
  - $V_S = \{v|$*incoming degree*$(v) = 2 \wedge$ *outgoing degree*$(v) = 1\}$.
- $e = (u, v) \in E$ *iff* $u \prec v$*. Furthermore,* $E = E_D \bigcup E_S$*, where*
  - $E_D = \{(u, v)|u \in V_F\}$ *and*

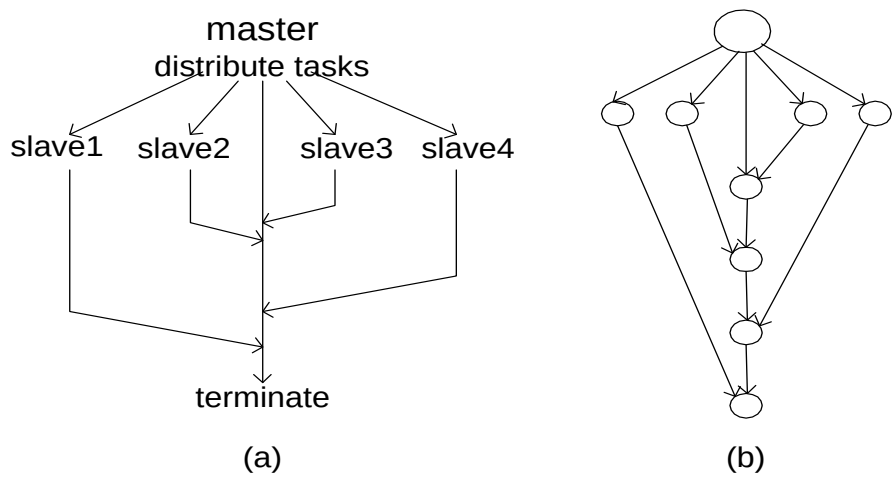


Figure 3.4: The structure and execution instance dag of static Master/Slave programs

– $E_S = \{(u, v) | v \in V_S\}$.

In the Master/Slave structure, master process produces tasks and distributes them to slaves. When slaves finish tasks, they return the results back to the master. Communication only occurs between the master and slaves, so the incoming degree of the synchronization nodes is two (one from master itself and the other from a slave). The tasks are produced statically, so the forking node is the start node, which is in the master process. Static or dynamic load balancing strategy can used. Figure 3.4 shows the structure of static Master/Slave programs and the execution instance dag (where figure(b) comes from [39]).

Theorem 3.2.6 $\Psi_{MS} \subsetneq \Psi_{gp}$.

Proof: $\forall$ a tuple $(A, \Delta, G) \in \Psi_{\text{MS}}$ where $G_{\text{MS}} = (V_{\text{MS}}, E_{\text{MS}})$, we construct an isomorphic graph $G' = (V', E')$ of G by using the same method of theorem 3.2.2. In the following we then prove $(A, \Delta, G') \in \Psi_{gp}$.

$\forall v' \in V'$, we consider three possibilities:

1. if $v' \in V'_T$, v' represents an execution of a task, which is mapped from the vertices in G_{MS}, so $v' \in V_{T_{gp}}$;

2. if $v' \in V'_F$, then $v' = v_0$, so $v' \in V_{F_{gp}}$;

3. if $v' \in V'_S$, according to definition of Ψ_{MS}, we have *incoming degree*$(v') = 2 > 1$, so $v' \in V_{S_{gp}}$.

$\forall e' = (u', v') \in E'$, we consider two possibilities:

1. if $e' \in E'_D$, since $u' \in V_{F_{gp}}$, so $e' \in E_{D_{gp}}$;

2. if $e' \in E'_S$, then $v' \in V_{S_{gp}}$, according to Definition 3.1.2, e' also $\in E_{S_{gp}}$.

So G' is also an execution instance dag and according to Definition 3.1.3, $(A, \Delta, G') \in \Psi_{gp}$. That is to say, for every $(A, \Delta, G_{\text{MS}})$ tuple in Ψ_{MS}, we can construct a G' such that $(A, \Delta, G') \in \Psi_{gp}$.

On the other hand, we can find at least one (A, Δ, G) in Ψ_{gp} such that (A, Δ, G) is not in Ψ_{MS}. For example, we can find a G whose $|V_F| > 1$, which means it has more than one forking node, so this execution instance dag G is not an execution instance of Ψ_{MS}, so this tuple $(A, \Delta, G) \notin \Psi_{\text{MS}}$. Finally we have $\Psi_{\text{MS}} \subsetneq \Psi_{gp}$. $\square$

In Master/Slave, the decomposition is performed statically by the master process before parallel computing begins, the distribution and joining are done by the master process and slave processes are only responsible for computation.

There is no overlapping between Ψ_{SPMD} and Ψ_{DC}, because for the nodes in $V_{S_{\text{SPMD}}}$, their outgoing degree is 0 (for the final node) or the number of processors P, while for the nodes in $V_{S_{\text{DC}}}$, their outgoing degree is 1. Intuitively, the synchronization nodes in SPMD are for global synchronization (such as barrier) and the synchronization nodes

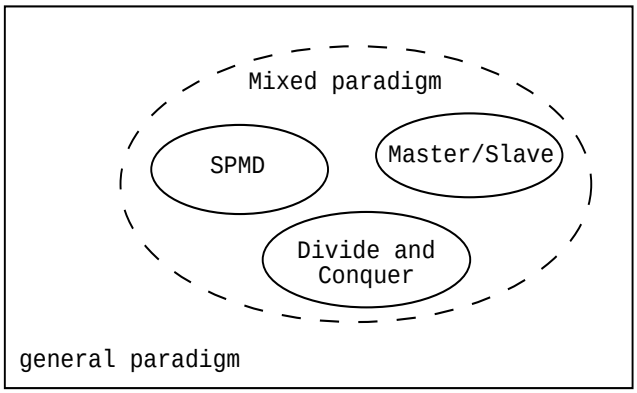


Figure 3.5: The relationship between the discussed parallel programming paradigms.

in Divide-and-Conquer only go to the next task or return to its father task's procedure. The relation between SPMD, Divide-and-Conquer, and Master/Slave can be illustrated by Figure 3.5.

3.3 The Mixed Paradigm

A memory model specifies the values that may be returned by memory with the execution of a memory operation. Different paradigms may have different synchronization features: implicit or explicit, local or global. This results in different memory model requirements. For example, for Divide-and-Conquer, Location Consistency is enough, since no global shared variables and synchronizations required. However, for SPMD or Master-Slave, they may need user level globally shared memory, since the programs may have global shared variables and synchronizations. For a particular parallel programming paradigm, the expression of parallelism is a high level programming issue, and its lower level memory model implementation can vary greatly .

With the above observation, we attempted to enlarge Divide-and-Conquer paradigm

by extending the memory consistency model and providing user level shared memory. We start our discussion from the strictness of parallel computation.

3.3.1 Strictness of Parallel Computation

Our discussion on the strictness of parallel computation is based on the discussion in Blumofe's Ph.D thesis [31]. Blumofe defined the *strictness* condition of multithreaded computation. According to [31], a *strict* multithreaded computation is a computation where every dependency edge goes from a thread to one of its ancestor threads. In *fully strict* computation, the father thread only synchronizes with its child threads and its own parent. This definition is built on multithreaded computing with Divide-and-Conquer, in which case a computation is divided into independent sub-computations and they only return the results to their upper-level computation.

For the parallel programming paradigms other than Divide-and-Conquer, there may be some cases where the sub-computations need to synchronize with each other. If we extend the execution instance dags of Divide-and-Conquer by providing such synchronization nodes or edges, then extended dags also allow synchronizations between the sibling threads. We say the corresponding paradigm is a ***mixed parallel programming paradigm*** and the parallel computation is partially strict computation, which can be defined as follows:

Definition 3.3.1 *For a given fully strict computation $C = (G, op)$, if there exists synchronization edges E'_S and synchronization nodes V'_S such that $G' = G \cup E'_S \cup V'_S$ is a directed acyclic graph, then (G', op) is a partially strict computation.*

We say the dag G' is an extended dag of G and it is an execution instance dag of

a partially strict parallel computation. In the extension of the dag, a global synchronization can be mapped to a synchronization node with some incoming edges and same number of outgoing edges (usually this number equals to the number of processors); a global mutual exclusion can be mapped to a single synchronization edge between two nodes. More details will be illustrated in Chapter 5.

3.3.2 Computation Strictness and Paradigms

The programming paradigm definition also suggests its relationship with the strictness of computation. In this section our discussion is based on the definition of the strictness of computation in [31]. According to [31], computation can be non-strict, strict, or fully strict. In the fully strict computation, every dependency goes from a thread to its parent (direct ancestor). In a strict multithreaded computation, every dependency edge goes from a thread to one of its ancestor threads. LC is suitable for the fully strict computation. By extending the dag with synchronization edges, we relax the strictness of computation and have the *partially strict computation*. In partially strict computation, a thread can synchronize not only with its parents, but also with its siblings. Based on the extended *dag*, a consistency model named RC_dag (see further discussion in Section 5.3) is defined. If the strictness of computation is further relaxed , then we get the non_strict computation, in which case threads can synchronize with each other arbitrarily. The appropriate memory model supporting this computation is SC (Sequential Consistency). The relationship between programming paradigm, strictness of memory model, strictness of computation can be demonstrated by Figure 3.6. This figure shows that when extending the memory consistency model, the corresponding computation is

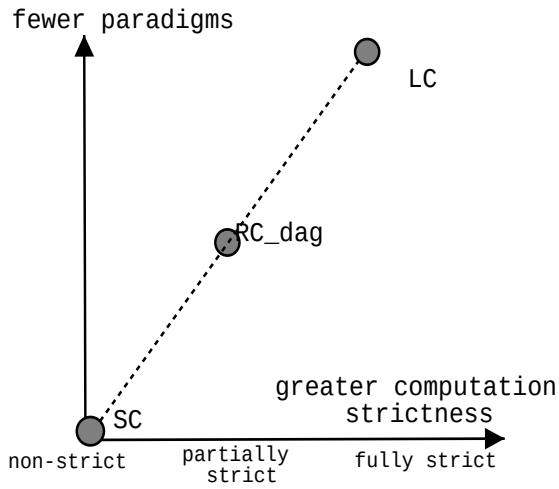


Figure 3.6: The relationship between paradigms, memory models, and computations.

stricter, and it may also support more programming paradigms.

3.3.3 Paradigms and Memory Models

In DSM-based parallel systems, parallel programming paradigms need the underlying support from memory consistency models for certain kinds of parallel computation. Specifically, Divide-and-Conquer paradigm requires the underlying LC for fully strict computation; the needed memory model in the mixed paradigm (see next section) is actually the RC_dag memory model for partially strict computation; and the general paradigm with SC is appropriate for non-strict computation.

3.3.4 The Mixed Paradigm

In this section a mixed parallel programming paradigm is developed on the basis of Divide-and-Conquer and the extension from LC to RC_dag.

Definition 3.3.2 *The mixed paradigm is a set* $\Psi_{MP} = \{(A, \Delta, G)|G$ *is the extended*

execution instance dag of parallel program A on the corresponding extended memory consistency model Δ}.

Theorem 3.3.3 $\Psi_{MP} \supseteq \Psi_{DC}$.

Proof: $\forall(A, \Delta, G_{\text{DC}}) \in \Psi_{\text{DC}}$, Δ is LC consistent memory model, G_{DC} is the execution instance dag of A on Δ. According to definition 3.3.1, $G_{\text{DC}} \subseteq G'$ where G' is an extended dag of G. So there exist a three tuple (A, Δ, G') such that $(A, \Delta, G') \in \Psi_{\text{MP}}$. So we have $\Psi_{\text{MP}} \supseteq \Psi_{\text{DC}}$. $\square$

Theorem 3.3.4 $\Psi_{MP} \supseteq \Psi_{SPMD}$.

Proof: $\forall(A, \Delta, G_{\text{SPMD}}) \in \Psi_{\text{SPMD}}$, G_{SPMD} is the execution instance dag of A. According to definition 3.3.1 and definition 3.2.1, the nodes and edges of G_{SPMD} are subsets of those of G_{MP} (i.e. G') respectively, so $G_{\text{SPMD}} \subseteq G'$ where G' is an extended dag of G. So there exist a three tuple (A, Δ, G') such that $(A, \Delta, G') \in \Psi_{\text{MP}}$. So we have $\Psi_{\text{MP}} \supseteq \Psi_{\text{SPMD}}$. $\square$

Theorem 3.3.5 $\Psi_{MP} \supseteq \Psi_{MS}$.

Proof: $\forall(A, \Delta, G_{\text{MS}}) \in \Psi_{\text{MS}}$, G_{MS} is the execution instance dag of A. According to definition 3.3.1 and definition 3.2.5, the nodes and edges of G_{MS} are subsets of those of G_{MP} (i.e. G') respectively, so $G_{\text{MS}} \subseteq G'$ where G' is an extended dag of G. So there exist a three tuple (A, Δ, G') such that $(A, \Delta, G') \in \Psi_{\text{MP}}$. So we have $\Psi_{\text{MP}} \supseteq \Psi_{\text{MS}}$. $\square$

Intuitively, introducing global barriers makes SPMD possible, and introducing global locks facilitates the Master/Slave paradigm. The mixed paradigm is shown in Figure 3.5 (the dotted circle).

3.4 Related Work

This section covers the related work on following various topics: integrating multiple parallel programming paradigms/models, language support for multiple paradigms, graph analysis of programs or computation, strictness of computation, etc.

There has been some research work on the multiple parallel programming paradigms according to various definitions of paradigm that are different from ours. Millipede virtual parallel machine [81] provides a set of interface to support ParC [17], CParPar [16], and Java [69, 91] for cluster computing, each of which stands for a so called paradigm. Hamelin et.al provided a multi-paradigm object oriented parallel environment [71] supporting both control and data parallelism using both SPMD model and shared virtual memory. Leichl et al. analyzed the multi-parallel parallelism, which was defined by them to be the simultaneous application of both distributed and shared memory parallel processing techniques to a single problem [92] in clusters. Anthony Hey et.al implemented multi-paradigm parallel programming with Occam [73] for transputers by supporting “processor farm”, “geometric array”, and “algorithmic pipe” paradigms. Hansen defined a programming paradigm and a programming methodology for scientific computing based on programming paradigms for multicomputers in [72]. He discussed the following paradigms: pipeline, divide and conquer, parallel monte carlo trials, and parallel cellular automata. Ian Foster discussed the design issues in designing parallel algorithm procedure in [59]. A lot of work has been done on integrating parallel programming paradigms/models, but the paradigms were defined differently and few of them formally analyze the role of memory consistency model in paradigms.

Some related work is focused more on parallel programming languages. Aiken et al.

examined synchronization patterns for different style paradigms and languages in [7]. Mani Chandy and Ian Foster et al. [42] provided an integrated support for task and data parallelism by providing language extensions and compile-time analysis. Their work was based on integrating Fortran M and HPF. Scott et al. discussed multi-model parallel programming in PSYCHE in [118] based on shared memory and message passing models. Rabhi [111] analyzed an approach of providing an intermediate level consisting of common parallel programming paradigms including data parallel, processor networks, etc. Rehg et al. [113] discussed integrated task and data parallel support for dynamic applications. The above work did not discuss the relationship between memory models and paradigms.

There were also a lot of graph analysis work (or dag-based work) on parallel programming and scheduling, but they are not focused on both parallel programming paradigms and memory models. Filho et al. used a graph-theoretic model to analyze shared-memory legality [57]. Their model is based on graph expression of read/write operations on memory locations and shows what operation orders are valid. Although they also use the acyclic dag to analyze memory models, their work is focused on rather basic formal discussion on memory operation orders and programming paradigms are not mentioned. Jeanne Ferrante et al. presented program dependence graph (PDG) in [56]. V. Sarkar et al. [117, 116] studied parallel program graph (PPG) on the basis of control flow graphs and data dependence graphs. T.Ball and S. Horwitz proposed a way to construct control flow from data dependence in [13]. Blieberger et al. [29] analyzed symbolic data flow for tasking programs. Cytron [46] analyzed control and data dependence proposed automatic generation of DAG parallelism from sequential programs. Stoltz and Wolfe [123, 124] studied a sparse data-flow technique for DAG

parallel programs using precedence graph. Thornton et al. gave a graph analysis for run-time minimizing in multi-threaded architectures in [131]. Blelloch et al. studied related parallel algorithms and defined and discussed directed acyclic graphs (dags) in NESL [27, 25, 28]. Alain Darte et al. discussed scheduling and automatic parallelization in [48].

The strictness of computation is discussed in Blumofe's Ph.D thesis [31], where fully strict computation is defined and analyzed in Cilk system. However, it is based on Divide-and-Conquer only and is not related to multiple paradigms.

3.5 Summary

This chapter presents a novel way to view and define parallel programming paradigms by taking the underlying memory models into account as an important factor. The programming paradigms are defined and analyzed in terms of a dag view of the execution. It shows the approach to support multiple paradigms by extending the memory model, and hence support a wider range of parallel computation. A mixed paradigm is proposed based on the extensions of the dag of Divide-and-Conquer programs (based on LC),The mixed paradigm is a super-set of some paradigms including Divide-and-Conquer, SPMD, and Master/Slave. Our mixed paradigm is supported by SilkRoad, which is based on the RC_dag consistency model extended from LC.

Chapter 4

SilkRoad

This chapter describes the SilkRoad system, which is developed to explore the idea of supporting multiple parallel programming paradigms.

SilkRoad is a variant of the Cilk system. It is developed based on Cilk by extending its memory consistency model with RC_dag consistency (see Chapter 5 for details). SilkRoad does not use the backing store as the "home" of the global virtual memory pages for the runtime system. Instead, it introduces the semantics of the Lazy Release Consistency (LRC) [84] to maintain the consistency of the pages on each processor's local memory.

At the programming level, SilkRoad provides a shared memory for users. With the user-level shared virtual memory, there can be shared variables in parallel programs in SilkRoad, and programmers can use cluster-wide global locks for mutual exclusion on shared variables as well as barriers for global synchronization.

The consequence of the extensions is that SilkRoad is able to support both Divide-and-Conquer applications and some other applications that need to use shared variables.

These applications are not *fully strict* computation (see Chapter 3 for detailed discussion) and they cannot be run directly on Cilk.

Cilk is not only a multithreaded parallel runtime system, but also a parallel programming language based on *ANSI C* with some parallel constructs. As a variant of Cilk, SilkRoad inherits the programming features of Cilk while introducing more facilities to users. In this chapter we also shows how the additional facilities in SilkRoad can contribute to programmability and this is illustrated by the solutions to Salishan problems [55].

The rest of this chapter is organized as follows: Section 4.1 introduces the features of SilkRoad. Section 4.2 introduces the programming in SilkRoad, including using lock and barrier mechanisms for global shared variables. In Section 4.3 we show that the added facilities are used in Salishan programs and in the end, Section 4.4 gives a summary for this chapter.

4.1 The Features of SilkRoad

SilkRoad inherits most of the features of Cilk. Its runtime system also keeps Cilk's work stealing policy for load balancing. However, SilkRoad does not use a backing store as the home of the pages in shared memory. Instead, the idea of LRC is introduced to maintain the memory consistency between the distributed nodes of a cluster, and a user level shared memory is provided.

4.1.1 Removing Backing Store

Without the backing store, in SilkRoad, the modifications of the local cached pages are not reconciled with their homes. They are propagated to the next node that needs the fresh contents, possibly after performing work stealing and thread migration. It is similar to the situations in LRC, where the propagation of diffs is delayed until the next remote lock requisition comes. The difference is that in SilkRoad, the transfer of the modifications of the system information is triggered by thread stealings and thread returns, not lock acquisitions and releases (in SilkRoad only the modifications of the user-defined shared data are transferred based on lock or barrier operations).

In SilkRoad, when parent thread and its child threads (assuming that they are initially running at the same node) are separated and running on different nodes because of thread stealing, a child thread will keep its changes of the memory pages locally until thread returning (see Figure 4.1).

Since the memory consistency operations are triggered by thread stealings and returns, we call it stealing-based coherence (SBC). When stealing happens, the corresponding nodes do not "reconcile" its modification to backing store. Instead, they keep them locally. Similarly, when a thread needs a memory page, it "fetches" from the node which is the last modifier. One sequence of removing the backing store is that there is less communication data between the nodes in the cluster. More details of implementation will be provided in Chapter 5.

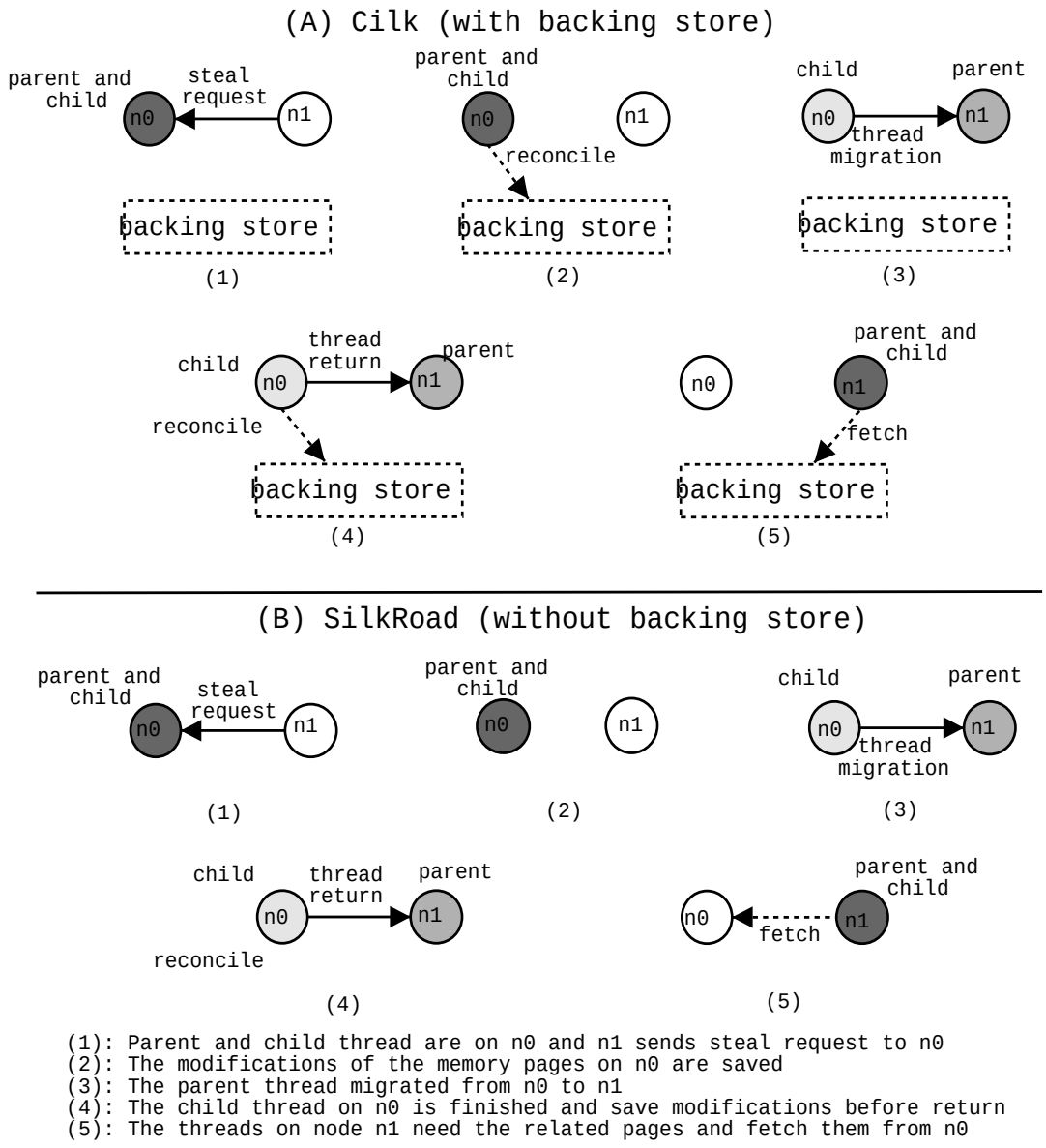


Figure 4.1: A simple illustration of memory consistency in Cilk (figure A) and SilkRoad (figure B) between two nodes (n0 and n1).

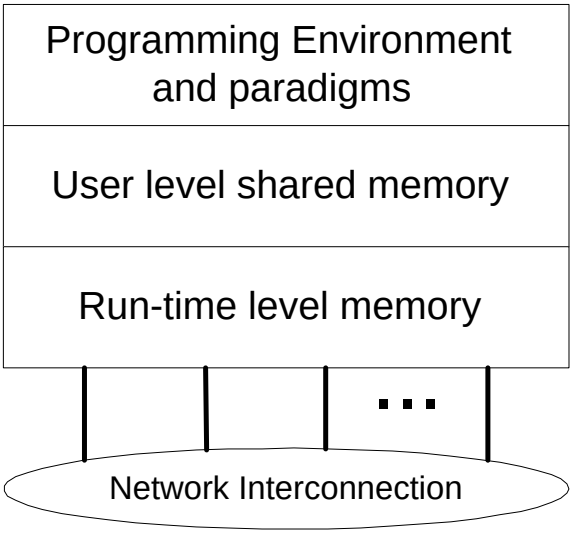


Figure 4.2: The shared memory in SilkRoad consists of user level shared memory and runtime level shared memory.

4.1.2 User Level Shared Memory

As it is mentioned before, SilkRoad provides a shared virtual memory in the cluster scale. Programmers can use shared variables in their parallel applications without consideration of the physical location of the data. Figure 4.2 illustrates the shared memories in SilkRoad.

Unlike in sequential programming, programmers are responsible for dealing with the mutual exclusion issues themselves. This can be done by using SilkRoad's *SR_lock()* and *SR_unlock()* pairs. To perform the global synchronization, *SR_barrier()* function should be called.

4.2 Programming in SilkRoad

The applications executable on SilkRoad are a super-set of those on Cilk. Besides the Divide-and-Conquer programs originally supported by Cilk, some applications in other paradigms with shared variables and global synchronizations also can run on SilkRoad.

4.2.1 Divide-and-Conquer

Programming by using Divide-and-Conquer with recursions in SilkRoad is same as that in Cilk [44].

4.2.2 Locks

In Cilk SMP versions, there are also mutual exclusion provided, which means programmers can use shared variables via $Cilk\_lock$ and $Cilk\_unlock$ function calls. This feature is directly available on physically shared memory machines instead of clusters.

Like many DSM-based cluster computing systems, SilkRoad provides the lock mechanism for global mutual exclusion. Although both SilkRoad's lock and Cilk's lock provide a mechanism to enable users to use shared variables, Cilk's lock is implemented by using OS level locks of SMP machines, while SilkRoad's lock is implemented with cluster wide shared virtual memory. The general format of using locks in SilkRoad is:

```
SR_lock(i);
......   // accessing and operating on the shared variable
SR_unlock(i);
```

where i is the lock number (identification of the lock).

Figure 4.3 demonstrates a typical SilkRoad example program using shared variables. In SilkRoad programs, the memory of the shared variables must be allocated by using dynamic memory allocation (i.e. $SR\_malloc\_shared()$) and be accessed by using pointers.

4.2.3 Barriers

Besides the synchronization between parent thread and child threads (which is inherited from Cilk), global synchronization is also supported in SilkRoad. In order to do global synchronization between all of the threads (this may be required by some programming paradigms such as SPMD), barriers must be used. The general usage of barriers is:

```
......            // doing parallel computing work
SR_barrier(i);
......            // doing parallel computing work
```

where i is the barrier number (the identification of the barrier).

Figure 4.4 demonstrates a typical use of SilkRoad's barrier mechanism.

In SilkRoad, barriers are mainly used for SPMD programs instead of Divide and Conquer with recursions. When programming using SilkRoad barriers, the programmer should be aware of which particular barriers the threads will stop and wait at. In SPMD, the number of threads are usually set equal to the number of the processors in the system and this is the assumption for both programmers and the runtime system. So programmers spawn as many threads as the number of processors, and at the barriers the runtime system is aware of this[1]. For example, Figure 4.4 shows a SPMD style pro-

[1]In chapter 5, the readers will find that in implementation of barrier, the barrier manager will count the number of barrier requests and compare it with the total number of processors. If they are equal, then

```
#include <cilk.h>
#include <cilk-lib.h>

int *n = NULL;

cilk void foo0(){
      SR_lock(1);
      // operations on shared variable n
      SR_unlock(1);
}

cilk int main (void)
{
      n = (int *)SR_malloc_shared(sizeof(int));
      *n = 0;
      spawn foo0();
      spawn foo0();
      sync;
}
```

Figure 4.3: Demonstration of the usage of SilkRoad lock

```
#include <cilk.h>
#include <cilk-lib.h>

cilk void foo0()
{
    // ... computation;
    SR_barrier(0);
    // ... computation;
}

cilk int main (void)
{
    //...
    for (int i = 0; i < Cilk_active_size; ++i)
    {
        spawn foo0();
    }
    sync;
    //...
}
```

Figure 4.4: Demonstration of the usage of SilkRoad barrier

gram. The $Cilk\_active\_size$ is a constant which means the size of the system and it is detected by runtime system. Both the programmer and the runtime system know there are $Cilk\_active\_size$ threads potentially waiting at the global barrier numbered 0, since there are $Cilk\_active\_size$ processors in total and the user spawned $Cilk\_active\_size$ threads.

4.3 SilkRoad Solutions to Salishan Problems

This section explores the programmability of Cilk/SilkRoad at the parallel programming language level. The discussion is based on Cilk/SilkRoad's solutions to the Salishan Problems [55].

The Salishan Problems are a set of four problems proposed at the 1988 Salishan High-Speed Computing Conferences. It was proposed as a standard to compare parallel programming languages. Invited speakers presented solutions to the problems in eight different parallel programming languages (Ada, Occam, Haskel, Id, Sisal, C^*, PCN, and Scheme). Those solutions were edited and published by Feo in [55]. Some other parallel programming languages (such as CC++ [130]) also used Salishan Problems to demonstrate the programmability.

There are several parallel constructs in Cilk/SilkRoad (SilkRoad inherits these features from Cilk), and in solving Salishan problems, the following language features are used:

- ***cilk***: To specify a function/procedure which will be executed as a thread.

it means all the threads have arrived the barrier and the computation can continue

- ***spawn***: The *spawn* keyword is used to create a child thread. After creating child threads, the parent thread and child threads may be executed in parallel. The procedure of spawning a thread in Cilk/SilkRoad is similar to function call in C language, except that the spawned threads can be stolen by other processors.

- ***sync***: The *sync* keyword is used to synchronize parent thread and its child threads, which means the parent thread can not continue unless all its child threads have finished. This is to show the dependencies between threads.

- ***return***: The *return* of a thread can be done explicitly or implicitly. It shows that a thread is over and its partial results are to be returned to its parent thread.

In the following sections, Section 4.3.1 describes the Hamming's problem and discusses the Cilk/SilkRoad's solution; Section 4.3.2 describes the Paraffins Problem and Cilk/SilkRoad's solution; Section 4.3.3 describes the Doctor's Office Problem and then gives the Cilk/SilkRoad's solution; Then Section 4.3.4 describes the Skyline Matrix Solver and shows the Cilk/SilkRoad's solution.

4.3.1 Hamming's Problem (extended)

Problem Description

The Hamming's Problem (extended) is described as follows:

Given a set of primes $a, b, c, ...$ of unknown size and an integer n, output in increasing order and without duplicates all integers of the form $a^i * b^j * c^k ... <= n$. Observe that if r is in the output stream, then, $a * r, b * r, c * r, ... <= n$ are also in the output stream.

The problem tests a language's ability to express recursive stream computations and producer/consumer parallelism, and to support dynamic task creation.

Solution

This problem can easily be expressed in Cilk/SilkRoad program by using Divide-and-Conquer with recursions. The prime numbers are stored in an array and one prime is fetched in each recursion, as illustrated in Figure 4.5.

The program starts from the first prime in the prime array by creating a thread with the index value 0. During the recursion, all possible exponents in a recursion level are explored and then the next level is explored if the condition (the current integer is less than n) is satisfied. The prime array is also passed as a parameter to the thread of next recursion level. The maximum depth of the recursion is determined by the number of primes. The parallelism is explored with the recursion and the threads exhaust possible solutions on different processors in parallel. The satisfying integers are stored in a global shared array (by invoking $save\_result()$ function) and accessing this array needs the acquisition of a lock. Finally the integers in the array will be sorted and printed out after the search has been finished.

4.3.2 Paraffins Problems

Problem Description

The problem is described as follows:

Given an integer n, output the chemical structure of all paraffin molecules for $i <= n$, without repetitions and in order of increasing size. The results should include all

```
cilk void Hamming(int prime_index, int curr_int, int n, int *primes_arr)
{
        int next_prime;

        if (prime_index == NUM_OF_PRIMES)
                return;

        next_prime = primes_arr[prime_index];

        /* try each possible exponent for next prime number */
        while (curr_int <= n)
        {
                if(prime_index+1 < NUM_OF_PRIMES)
                        spawn Hamming(prime_index+1, curr_int, n, primes_arr);
                curr_int = next_prime * curr_int;
                if(curr_int <= n)
                {
                        save_result(curr_int);
                }
        }
        return;
}
```

Figure 4.5: The solution to Hamming's problem.

isomers, but no duplicates. The chemical formula for paraffin molecules is C_iH_{2i+2}. You may choose any representation for the molecules, so long as it clearly distinguishes among isomers. The problem addresses the representation of recursive tree structures, the creations and manipulation of those structures and nested loop parallelism.

Solution

Our solution to this problem is based on the relationship between paraffin molecules and radical molecules (molecules with chemical formula C_iH_{2i+1}), as analyzed in [130]. We generate lists of radicals of size 0 to $n/2$, and generate lists of paraffins of size 1 to n from those radicals. The generation of radicals and paraffins of all sizes can be done in parallel. Since there is no "parallel loop" mechanism in Cilk/SilkRoad, we create as many threads as the number of processors and each thread executes a portion of the work in the loop. The overall program uses an SPMD paradigm and the threads may need to synchronize during the procedure of generating radicals and paraffins. Figure 4.6 and Figure 4.7 illustrate partial code of the idea of generation of paraffins.

In Figure 4.6, the data structure of radicals and the code of the top level of the program are shown. The $number\_of\_trails$ is the maximum value of i in the formula C_iH_{2i+2}. At the top level, each iteration generates a result with the corresponding i. In Figure 4.7, the $generate\_paraffins()$ thread generates the radicals of various sizes and then generates the paraffins. Since generating paraffins is based on generating radicals, there is a barrier between these two steps for global synchronization.

```
typedef struct radical_data{
  unsigned char data[2];
} radical_data;

typedef          struct radical {
  struct radical* subradical[3];
          int kind;
} radical;

typedef          struct radical_array{
  int length;
  radical* element;
} radical_array;

typedef struct radical_array_array{
  int length;
  radical_array* element;
} radical_array_array;

cilk int  cilk_main(int argc, char *argv[])
{
   ...
   for(i=1;i<=number_of_trials;i++)
   {
       /* spawn threads to generate the paraffins */
       for(j = 0; j < Cilk_active_size; ++j)
         spawn  generate_paraffins(&r,&p,maximum_paraffin_size);
      sync;
      ...
      clean(&r,&p); /* clean the results */
   }
   ...
}
```

Figure 4.6: The data structures and top level code of the solution to Paraffins problem.

```
cilk void generate_paraffins(
        radical_array_array* radicals,
        paraffin_array_array* paraffins,
        const int largest_size)
{
  int radical_array_size, a,i,j;
  a=largest_size;
  radical_array_size=a/2+1;
  radicals->length=radical_array_size;
  paraffins->length=largest_size+1;
  ...
   /* initializing the range */
  radical_start = Self * radicals->length / Cilk_active_size;
  radical_end = (Self+1) * radicals->length / Cilk_active_size;
  paraffin_start = Self * paraffins->length / Cilk_active_size;
  paraffin_end = (Self+1) * paraffins->length / Cilk_active_size;

  for(i=radical_start; i<=radical_end-1;i++)
  {
     generate_radicals_of_size(&(radicals->element)[i],i,radicals);
  }
  SR_barrier(0) ;
  for(j=paraffin_start; j<=paraffin_end;j++)
  {
     generate_paraffins_of_size(&(paraffins->element)[j],j,radicals);
  }
}
```

Figure 4.7: Code of the thread generating the radicals and paraffins.

4.3.3 The Doctor's Office

Problem Description

The Doctor's Office Problem is described as follows:

Given a set of patients, a set of doctors, and a receptionist, model the following interactions: initially, all patients are well, and all doctors are in FIFO queue awaiting sick patients. At random times, patients become sick and enter a FIFO queue for treatment by one of the doctors. The receptionist handles the two queues, assigning patients to doctors in a first-in-first-out manner. Once a doctor and patient are paired, the doctor diagnoses the illness and cures the patient in a random amount of time. The patient is then released, and the doctor re-joins the doctor queue to await another patient. The output of the problem is intentionally unspecified. The problem tests the language's ability to program a set of concurrent, asynchronous processes with circular dependencies.

Solution

This problem is actually not a sequential computation being parallelized. It is a simulation of synchronous entities and some parallel constructs are used during the simulation. In this problem, the patients and doctors are defined by arrays of structures and in each structure the status of the patients or doctors is defined, as illustrated by Figure 4.8. The patients array, doctors array, patients queue, and doctors queue are stored in a global shared structure and accessing the data must be exclusive. This can be accomplished by using locks. Figure 4.8 also shows the top level of the program, where a number of threads are spawned to simulate the patients and doctors. The code of patient thread and doctor thread are illustrated in Figure 4.9.

```
typedef struct Patient{
        int cure_time;
        int waiting;
        int patient_id;
} Patient;
typedef struct Doctor{
        int cure_time;
        int busy;
        int doctor_id;
} Doctor;
typedef struct Patient_queue{
        int num;
        int patients[NUM_OF_PATIENTS];
} Patient_queue;
typedef struct Doctor_queue{
        int num;
        int doctors[NUM_OF_DOCTORS];
} Doctor_queue;
struct sharedd {
        Patient             Patients[NUM_OF_PATIENTS];
        Doctor              Doctors[NUM_OF_DOCTORS];
        Patient_queue       PatientQ;
        Doctor_queue        DoctorQ;
} *sharing = NULL;

cilk int cilk_main(int argc, char *argv[])
{
        ...
        for(i = 0; i < NUM_OF_PATIENTS ; i++)
           spawn patient(i);
        for(i = 0; i < NUM_OF_DOCTORS ; i++)
           spawn doctor(i);
        sync;
        ...
}
```

Figure 4.8: Definitions of the data structures and top level code of the solutions to Doctor's Office problem.

```
cilk void patient(int ID)
{
        for(;;)
        {
                being_fine(ID);
                add_to_patient_Q(ID);
        /* try to find a doctor from the doctor queue*/
                if(find_a_doctor(ID))
                {
                        being_cured(ID);
                }
        }
}

cilk void doctor(int ID)
{
        for(;;)
        {
        /* if it is not busy, then join the doctor queue.*/
                if(!sharing->Doctors[ID].busy)
                        add_to_doctor_Q(ID);
                else
                        continue;
        /* try to find a patient from the patient queue*/

                if(find_a_patient(ID))
                {
                        curing_patient(ID);
                }
        }
}
```

Figure 4.9: Patient thread and Doctor thread in the solution to Doctor's Office.

For each patient thread, it basically executes an infinite loop in which the status of the corresponding element in the $Patients$ array is changed accordingly. When a patient becomes sick (after being fine for a random period of time), he joins the patient queue and keeps on checking the doctor queue to find an available doctor. When a doctor is available, both the waiting patient and the doctor quit from their waiting queue and begin the curing procedure (cure time is generated randomly). Each doctor, when he is free, joins the $DoctorQ$ and keeps on checking the $PatientQ$ until he finds one patient and then starts with the curing procedure with the status of busy. After the doctor has finished the curing procedure, his status is changed to be "free" and he joins the waiting queue again.

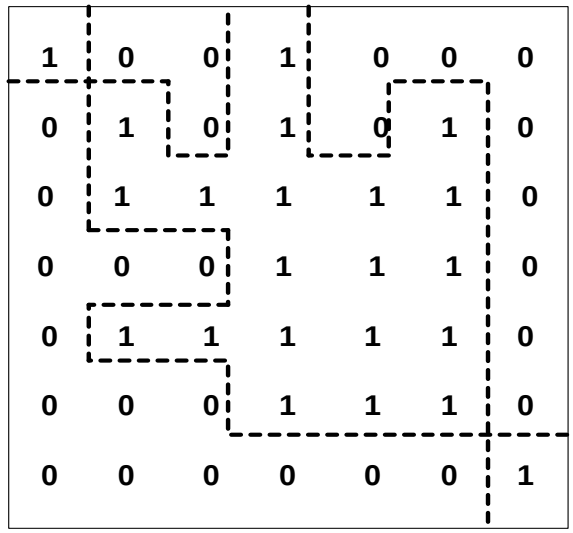
4.3.4 Skyline Matrix Solver

Problem Description

The Skyline Matrix Solver Problems is described as follows:

Solve the system of linear equations $Ax = b$ without pivoting where A is an $n \times n$ skyline matrix. A skyline matrix has nonzero values in row i in column k through i, $1 \leq k \leq i$, and nonzero values in column k through j, $1 \leq k \leq j$. Figure 4.10 shows an example of skyline matrix with the given row and column arrays.

The problem tests the ability to define array structures that include nonessential elements (i.e. the zeros), and given those structures, efficiency of parallel and iterative array computations.



row = {0,1,1,3,1,3,6}
column = {0.1,2,0,2,1,6}

Figure 4.10: An example of sky matrix.

Solution

Our solution makes use of the LU example in Cilk, which performs an LU decomposition of an $n \times n$ matrix without pivoting [32]. A Divide-and-Conquer algorithm is used for the problem and the matrix A and its factors L and U are divided into four parts such that $A = LU$ can be written as

$$\begin{pmatrix} A_{00} & A_{01} \\ A_{10} & A_{11} \end{pmatrix} = \begin{pmatrix} L_{00} & 0 \\ L_{10} & L_{11} \end{pmatrix} \begin{pmatrix} U_{00} & U_{01} \\ 0 & U_{11} \end{pmatrix}$$

According to Cilk [32], the L and U are computed as follows: It recursively factors A_{00} into $L_{00} \cdot U_{00}$. Then U_{01} is calculated in the formula $A_{01} = L_{00} \cdot U_{01}$, while simultaneously L_{10} is solved in $A_{10} = L_{10} \cdot U_{00}$. Finally, it recursively factors the Schur complement $A_{11} = L_{10} \cdot U_{01}$ into $L_{11} \cdot U_{11}$.

Because Cilk/SilkRoad is based on *ANSI C*, the skyline matrix can be store in two

"ragged" arrays: one is for the rows of the lower triangle, the other is for the upper triangle. The whole computation consists of three steps: LU decomposition, forward substitution, and backward substitution. Each step can be divided and parallelized by Divide-and-Conquer with loop ranges adjusted to avoid accessing matrix elements outside of the skyline.

Figure 4.11 shows the partial code of the LU decomposition. The forward substitution and backward substitution steps can also be implemented similarly.

4.4 Summary

SilkRoad provides user level shared memory besides the inherited Cilk features. Programmers are able to define global shared variables like in sequential programs, but they have to control the access to the shared variables to make sure the logic is correct. Mutual exclusion can be ensured by using lock mechanism and global synchronization can be done with barriers. The above features enable SilkRoad to support the applications using shared variables and programed with the paradigms other than Divide-and-Conquer (such as SPMD, Master/Slave). These are achieved by using a underlying memory consistency model extended from LC of Cilk, namely RC_dag model.

In this chapter Salishan problems are also used to examine the programmability of Cilk/ SilkRoad as a parallel programming language. Cilk is good at expressing recursive parallelism and dynamically spawning threads, so it is easy for Cilk/SilkRoad to solve Hamming's problem with Divide-and-Conquer paradigm with recursions. SilkRoad's extension on user level global shared memory makes it possible to replace parallel loops by spawning threads and enabling them to synchronize via global barriers in Paraffin's

```
/* a skyline matrix is represented by two "ragged" arrays: Mr
and Mc. Mr is for the rows of  the lower triangle, Mc is for
the columns of the upper triangle */
cilk void lu(Matrix Mc, Matrix Mr, int num_of_block)
{
        /* check if the matrix is small enough */
        if(num_of_block == 1){
                block_lu(*Mc, *Mr);
                return;
        }
        /* divide the matrix into 4 pieces */
        nb = num_of_block/2;
        Mc00 = &MATRIX(Mc, 0, 0);
        Mr00 = &MATRIX(Mr, 0, 0);
        Mc01 = &MATRIX(Mc, 0, nb);
        Mr01 = &MATRIX(Mr, 0, nb);
        Mc10 = &MATRIX(Mc, nb, 0);
        Mr10 = &MATRIX(Mr, nb, 0);
        Mc11 = &MATRIX(Mc, nb, nb);
        Mr11 = &MATRIX(Mr, nb, nb);
        /* decompose upper left piece */
        spawn lu(Mc00, Mr00 nb);
        sync;
        /* solve for upper right and lower left piece */
        spawn lower_slove(Mc01, Mr01, Mc00, Mr00, nb);
        spawn upper_slove(Mc10, Mr10, Mc00, Mr00, nb);
        sync;
        /* ... */
        /* decompose lower right piece */
        spawn lu(Mc11, Mr11, nb);
        sync;
        return;
}
```

Figure 4.11: The solution to Skyline Matrix Solver problem.

problem with a SPMD style. It also enables users to program asynchronous entities in Doctor's Office problem by allowing them to communicate via mutual exclusion. Since Cilk is based on *ANSI C* language, it can make use of *C*'s "ragged" arrays and pointers to represent sparse matrix in Skyline Matrix Solver problem and solve the problem with Divide-and-Conquer paradigm.

Chapter 5

RC_dag Consistency

In previous chapters, the idea of supporting multiple paradigms by extending the memory consistency model was discussed theoretically, and then we mainly introduced the added features of SilkRoad, which is developed for supporting multiple paradigms. In this chapter, the core part of SilkRoad, i.e. the underlying RC_dag consistency model is discussed, including its formal definition and properties, its design and implementation, and its theoretical performance issues.

The work in this chapter consists of two parts. First, a concept of stealing based coherence is proposed to implement a logical backing store. Second, the dag of computation in Cilk is extended and the concept of RC_dag consistency is proposed. Meanwhile, the stealing based coherence concept is also extended to implement the RC_dag consistency.

The Location Consistency (LC) in Cilk is maintained by the BACKER algorithm (using the backing store) collaborating with work-stealing scheduler. Under this situation, if the number of threads is potentially huge and there is frequent migration, then

there could be considerable network communication. In a cluster environment, observing that the coherence of the data pages can also be maintained logically without the backing store, the dirty cached pages need not to be reconciled back to the *home* node in order to reduce network traffic. In this case, the memory coherence algorithm is tightly coupled with the stealing and return operations.

For Divide-and-Conquer paradigm, there is no need to provide user level shared memory because of its intrinsic nature. However, in cluster computing, shared virtual memory is widely used because it provides a simple and general programming model for programmers. With a user level shared virtual memory, programmers do not need to care about communication issues among processors and they can just assume that there is a shared memory in the distributed environment, like they do programming in physically shared memory systems. Therefore, in order to support more paradigms and run a wider range of applications, providing user level shared virtual memory is a practical approach.

Providing the user level shared memory implies that the execution dag of the computation will be extended with the shared memory operations. Usually some synchronization mechanisms are required to perform the operations on the shared virtual memory. In our work, we mainly consider lock and barrier mechanisms for mutual exclusion and synchronization respectively. A lock (release, acquire) pair can be modeled as a synchronization edge and a global barrier can be modeled as a node with some edges connecting to it in the dag.

Along with the extended dag, the corresponding memory consistency model is also changed, which results in the RC_dag consistency. RC_dag consistency is developed based on the LC consistency under the computation-centric theory. We show

that RC_dag consistency is a more stringent memory consistency model than the LC consistency, but it is weaker than Sequential Consistency (SC). The relationship between LC, RC_dag, SC, and the shared memory M can be described by an *interaction function*. The interaction function defines the way the memory M behaves for the read and write operations. We show that if there exist two nodes in the dag operate on the same memory location and the data is shared via the ways of local synchronization between parent thread and child thread, the memory is LC consistent; if the data is shared via mutual exclusion or global synchronization besides local synchronization, then the memory is RC_dag consistent; if all of the memory locations are protected by mutual exclusion, the memory is actually sequential consistent. Moreover, we show that by extending LC to RC_dag, more paradigms and wider computation are supported.

Meanwhile, an Extended Stealing Based Coherence (ESBC) algorithm is used to implement the RC_dag consistency model. The extension of the algorithm is based on the extended dag and the semantics of the lock and barrier operations on shared memory. We prove that the ESBC function is actually an *observer function* which defines the semantics of the behaviors of the RC_dag memory consistency model.

In implementing RC_dag consistency in SilkRoad with SBC/ESBC algorithm, the semantics of LRC are introduced in order to implement the backing store logically and provide global locks and barriers at the programming level. Here the semantics of LRC means the "lazy" style of not propagating the modifications until they are required. Eager diff creation and lazy diff propagation mechanisms are used and the lazy write notice propagation mechanism is proposed specially for the work stealing and thread migration environment. In lazy write notice propagation, when a thread finishes execution and returns to its parent thread, it postpones sending out the write notice of the dirty

pages until it finally finds out the location of its parent thread, because its parent thread may be stolen and migrating. This helps reduce network communication messages and data.

We also try to theoretically analyze the performance of the SilkRoad system which is built on RC_dag consistent memory model. The analysis is based on Cilk's original performance model, plus the consideration of synchronization overhead (lock acquisition and waiting).

The remainder of this chapter is organized as follows. Section 5.1 discusses the Stealing Based Coherence which is proposed to reduce the network traffic. Section 5.2 introduces the extension of the dag with lock and barrier mechanisms. Section 5.3 defines the RC_dag consistent memory on the basis of the extended dag. Section 5.4 proposes the Extended Stealing Based Coherence algorithm for implementing the RC_dag consistency. The implementation of locks and barriers are introduced in Section 5.5. Section 5.6 theoretically discusses some performance issues. Finally, Section 5.8 gives a summary of this chapter.

5.1 Stealing Based Coherence

The backing store of Cilk is actually a set of "homes" of all locally cached pages on each node, and it is physically distributed on all of the nodes in the cluster. In each "reconcile" operation, modifications of the dirty pages are propagated to the corresponding "home" node and in each "fetch" operation, the whole requested page is transferred from the "home" node to the requesting node. In this situation, with large number of threads and frequent thread stealings and returns, the overhead of resulted network traf-

fic must be considerable.

One of our attempts to improve Cilk is to implement the backing store logically, aiming at reducing the reconciling messages (hence the total number of messages) and transferred data in computation [1]. This is meaningful especially when the network is slow or there are more than one application sharing the network.

5.1.1 SBC Coherence Algorithm

With the work-stealing scheduler, the coherence operations happen with thread migration (stealing and return). This is different from the case in LRC in which the shared data are mainly transferred via the lock release and acquire chains. Therefore, a stealing based coherence (SBC) algorithm is proposed for our situation.

The SBC still uses the basic operations in Cilk: *fetch*, *reconcile*, and *flush* (please refer to chapter 2 for how these operations work). The difference is that in SBC a fetch operation copies the diffs from the node who did the modification, not from the backing store; a reconcile operation just saves the diffs locally and propagates them when required, instead of copying them from local cache to backing store.

When describe how the SBC algorithm works, we try to show that the SBC can also keep the data coherence as BACKER algorithm can. There are two situations need to be considered:

1. thread *i* and thread *j* have data dependency (suppose $i \prec j$) and both of them are located in the same cluster node. In BACKER algorithm, when i finishes, it performs a reconcile operation to put its modified data to the backing store and

[1] The contents of this chapter are partially published in [107, 106].

j performs a fetch operation to get the needed data from the backing store when necessary. In SBC, i also does the reconcilation, but keeps the data locally. So when j needs the data, it is already in the local memory of the node. So SBC is equivalent to BACKER in this case.

2. thread *i* and thread *j* have data dependency (suppose $i \prec j$) but they are located in different cluster nodes because of thread stealing. In BACKER algorithm, the node executing i reconciles its data to the backing store (in implementation, it is logically mapped into a particular node, say node p) when i finishes and before j begins. And j performs a fetch operation to get the data from the backing store (i.e. node p) when necessary. SBC algorithm operates as follows. The node running i (say, node q) also performs reconcile operation, but the modified data are not send to the backing store. Instead, they are kept locally on q node. When j needs the data which was modified by i, the node executing j performs a fetch operation to get the data from the node q. In this case, the node q in SBC takes the role of the node p in BACKER in the above description. That is to say, the coherence of the different copies of data pages can also be maintained logically by the nodes that modify them.

Since there may be multiple copies of data, in implementation, *steal level* is used to identify the "time stamp" of a page. It is defined as the number of successful thread stealings of each node. To keep the consistency of different versions of data, each node has an independent steal level of its own (initiated to be 0) and keeps a record of the latest steal level of the other nodes (initiated to be -1) when it gets diffs from any other nodes (see Figure 5.1). In Figure 5.1, each node (P0, P1, P2) keeps an one dementional

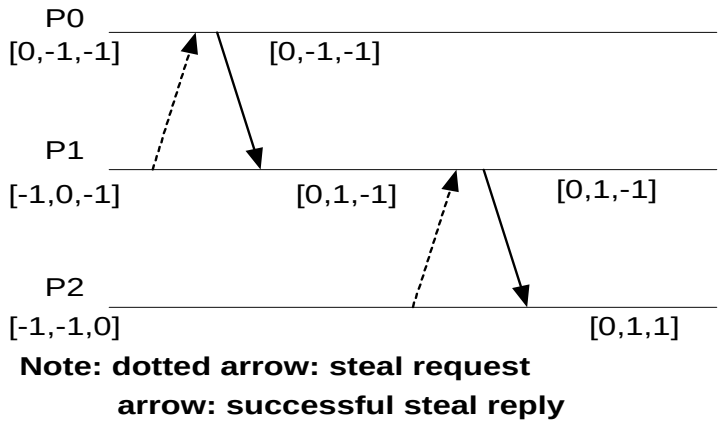


Figure 5.1: The steal level in the implementation of RC_dag.

array of steal levels of all nodes, and they are updated in successful stealing. Dotted arrows are stealing requests and arrows are grants for stealing. When a node fetches diffs of a page from other nodes, it updates (1)its own steal level, (2) the records of the latest steal levels of the other nodes are updated accordingly. Those records shows how up-to-date the copy of the page is on this node and will be used to filter the obsolete diffs in deciding which diffs should be fetched when it needs diffs later. For example, in a three node cluster, if node 2 has the steal level record of [2,3,5] of a pagge, it means that it has the level 2 copy of the page on node 0, level 3 copy of the page on node 1, and level 5 copy of the page on node 2. So later if node 2 performs *fetch* operation to get the data of the page from node 1 and suppose the steal level of the page on node 1 is 5, node 1 should pass node 2 the diffs of the pages ranging from 3 to 5 (not including 3, because node 2 has the level 3 copy already). And after this propagation, the steal level record of this page on node 2 should be update to [2,5,5].

5.1.2 Eager Diff Creation and Lazy Diff Propagation

In SilkRoad, the diffs of the "dirty" pages are created "eagerly" when a thread is stolen or when a child thread returns. In stealing, the victim node saves and keeps the diffs and then sends the stealing reply. In returning, the child thread saves and keeps the diffs of the locally modified pages and then sends write notices to its direct parent thread. When diff request comes, the node will check the steal level of both requester and itself in order to decide which diffs to propagate. Here lazy diff creation may not be appropriate because if threads do not save diffs when returning, the modifications may be lost when the node steals another thread to execute later, in which case its local memory will be refreshed.

5.1.3 Lazy Write Notice Propagation

In the stealing based coherence, we propose lazy write notice propogation, i.e. the propagation of the write notices is delayed until needed. This means when a child thread returns to its parent, it does not send write notices to its parent immediately, because the node where its parent stays may be changing because of thread migration, and in that case its returning request will be forwarded to another node where its parent may be located. To reduce the transferred data size, the write notices will not be sent out until the child finally finds its parent. Figure 5.2 illustrates the procedure of lazy write notice propagation. In Figure 5.2, write notices will not be sent out until the current location of the parent thread is finally found on P3 node (i.e. the thread return message is forwarded to the parent thread).

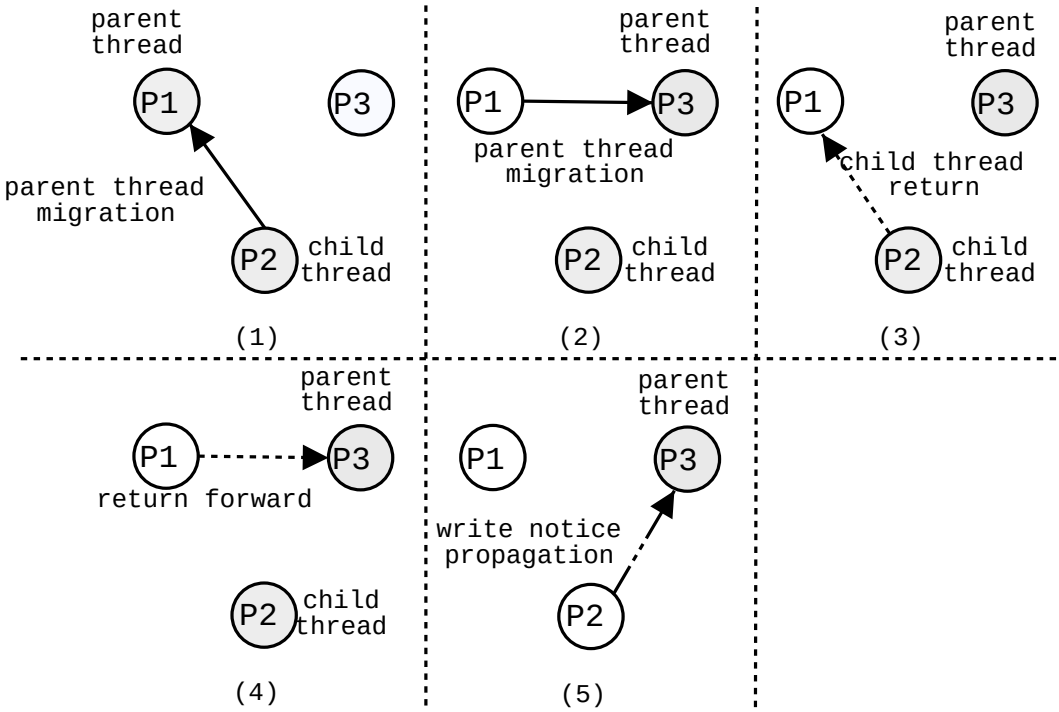


Figure 5.2: Demonstration of lazy write notice propagation.

5.2 Extending the DAG

Our motivation to support more computation by extending the dag of computation can be realized by enclosing mutual exclusion and global synchronization. With these extensions, the computation is not fully strict any more and we say it is partially strict computation (the formal definition is in Chapter 3). The two extensions are introduced in the following.

5.2.1 Mutual Exclusion Extension

We consider the case of locking for the mutual exclusion. A lock pair (release, acquire) is modeled as a synchronization edge in the dag. For example, in Figure 5.3, i, j, and

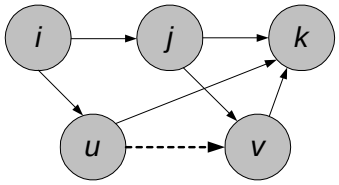


Figure 5.3: In the extended dag, threads can synchronize with their siblings.

k are the procedures in parent thread; u and v are procedures in child threads. If node u releases a lock and later the lock is acquired by node v, then a mutual edge from u to v is defined. This shows first u then v enter the same critical section (for example, they may use the same shared variable in the program), and if u performs a write operation on a memory location and v performs a read operation on the same memory location, then the read value at v depends on the written value at u. The dotted arrow (u, v) is a synchronization edge.

5.2.2 Global Synchronization Extension

At the barrier-like global synchronization point, the partial results of all nodes are exchanged and merged. The shared memory is made consistent for all processors. The global barrier synchronization can be modeled in two ways: (1) Edges only and (2) Nodes and edges, as demonstrated in Figure 5.4, where figure(a) shows that a global synchronization is modeled as a set of intercepted edges and figure (b) shows that a global synchronization is modeled as a synchronization node and the connected edges.

We adopt the second way by modeling barrier as a node and some connected edges, which produces much less edges but only one more node than the first way does.

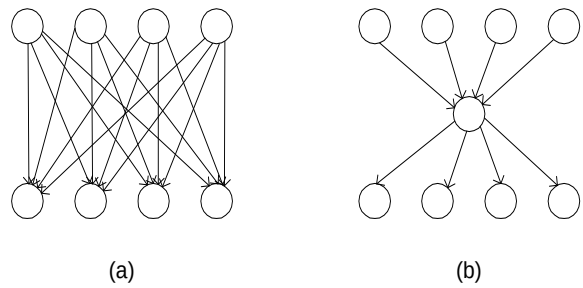


Figure 5.4: Graph modeling of global synchronizations.

5.3 RC_dag Consistent Memory Model

According to computation-centric memory model theory [62] in Cilk, Divide-and-Conquer paradigm has underlying supporting memory model LC for fully strict computation. In this section, RC_dag consistent memory model is proposed in SilkRoad for the partially strict computation to support wider paradigms on the basis of LC.

We first define the ***extension*** of a dag. An ***extension*** of a dag $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ is a dag $\mathcal{G}' = (\mathcal{V}', \mathcal{E}')$ such that $\mathcal{V} \subseteq \mathcal{V}'$ and $\mathcal{E} \subseteq \mathcal{E}'$. For computation $C = (\mathcal{G}, op)$ and $C' = (\mathcal{G}', op')$, if $\mathcal{G}'$ is an extension of $\mathcal{G}$, and op' is the extension of op to $\mathcal{G}'$, then we say C' is an extended computation of C.

Now the last writer function can be defined by introducing additional properties for the extended *dag* on the basis of the definitions in Cilk's theory[61].

Definition 5.3.1 *Let $C = (\mathcal{G}, op)$ be a computation ($\mathcal{G} = (\mathcal{V}, \mathcal{E})$ is a dag), $C' = (\mathcal{G}', op')$ be an extension of C ($\mathcal{G}' = (\mathcal{V}_{C'}, \mathcal{E}_{C'})$, where $\mathcal{V}_{C'} = \mathcal{V} \cup V_S, \mathcal{E}_{C'} = \mathcal{E} \cup E_S$ and V_S is the set of synchronization nodes, E_S is the set of synchronization edges, and $T \in TS(C')$ be a topological sort of C'. Let $\mathcal{L}$ denote the set of memory locations, the last writer function according to T is $W_T : \mathcal{L} \times V_{C'} \cup \{\perp\} \mapsto \mathcal{V}_{C'} \cup \{\perp\}$ such that for all $l \in \mathcal{L}$ and $u \in \mathcal{V}_{C'} \cup \{\perp\}$:*

1. *If* $W_T(l,u) = v \neq \perp$ *then* $op_{C'}(v) = W(l)$.

2. $W_T(l,u) \prec_T u$.

3. $W_T(l,u) \prec_T v \prec_T u \Rightarrow op_{C'}(v) \neq W(l)$ *for all* $v \in \mathcal{V}_{C'}$.

4. *If* $\exists e = (u,v) \in E_S$, *then* $\exists l \in \mathcal{L}, W_T(l,v) = W_T(l,u)$.

5. *If* $\exists u \in V_S$, *then* $W_T(l, succ(u)) = W_T(l,u)$.

Properties 1,2, and 3 were already defined in computation-centric theory. Property 4 in the above definition shows that the *synchronization edges* introduce data dependencies between nodes in the *dag*. Property 5 shows that the immediate successors of a synchronization node get the same values as the synchronization node does.

With the definition of the last writer function, now the corresponding memory consistency model can be defined as follows:

Definition 5.3.2 *RC dag-consistency is the memory model* $RC\_dag = \{(C,\Phi) : \forall l \exists T_l \in TS(C_{\mathcal{G}_S}) \forall u, \Phi(l,u) = W_T(l,u)\}$.

Note that with the extension of dag and computation, the formal definition of RC_dag consistency is the same as the LC except the definition of the last writer function. Actually, with this definition of RC_dag, the observations range from LC to Sequential Consistency (SC) [90], depending on how stringent the last writer function is in setting mutual exclusion regions. If all locations in memory are protected by mutual exclusion (say all location accessing need a lock acquisition), then the memory model equals SC. If no mutual exclusion is used at all, then it is LC. From this point, we can say

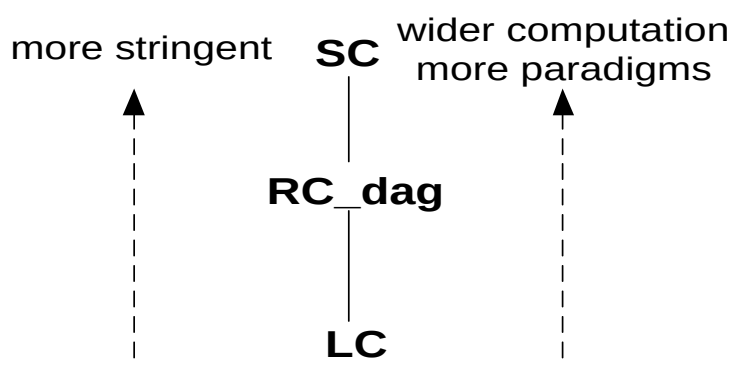


Figure 5.5: The RC_dag consistency is more stringent than LC but weaker than SC.

that RC_dag consistency is stronger than LC. The relationship between SC, LC, and RC_dag can be demonstrated in Figure 5.5.

Formally, we can use an ***interaction function*** to show the different synchronization features between subcomputations in each memory model and the relationship between the memory models. Here a subcomputation is the computation that one processor performs from the time it obtains work to the time it finishes the work or enables a synchronization task.

First, let us see how the interaction function can be defined for the Location Consistency:

Definition 5.3.3 *The interaction function for a computation C is I_C: $\{C_i\} \times \mathcal{L} \mapsto \{C_i\} \times \mathcal{L}$ ($\mathcal{L}$ is the set of all shared memory locations), satisfying the following property for all subcomputations C_i: $\forall l \in \mathcal{L}$, if $op(u,l)$ and $op(v,l)$, then (u,v) is not a synchronization edge in dag (u and v are nodes in dag).*

It means if two nodes operate on the same memory location, then the data value of this location is shared via the ways other than global synchronization between subcom-

putations. This definition is suitable to describe the situation in LC, in that the subcomputations return the results to a higher level computation and there is no extra synchronization protection needed for the memory locations for the Divide-and-Conquer paradigm.

The properties in definition of interaction function can be strengthened as follows:

Definition 5.3.4 *The interaction function for a computation C is I_C: $\{C_i\} \times \mathcal{L} \mapsto \{C_i\} \times \mathcal{L}$, satisfying the following property for all subcomputations C_i: $\exists l \in \mathcal{L}$, that if $op(u, l)$ and $op(v, l)$, then there is edge (u, v) for global synchronization.*

This means that there are certain memory locations that need to be protected by global synchronization. The synchronization edge (u, v) is an extension of the *dag* of LC (see Figure 5.3). With this strengthening, some other paradigms are possibly supported, because it provides global shared variables which are protected by synchronization. The corresponding computation can also be represented by an extended dag with synchronization edges.

The definition of interaction function can be even further strengthened as follows:

Definition 5.3.5 *The interaction function for a computation C is I_C: $\{C_i\} \times \mathcal{L} \mapsto \{C_i\} \times \mathcal{L}$, satisfying the following property for all subcomputations C_i: $\forall l \in \mathcal{L}$, that if $op(u, l)$ and $op(v, l)$, then there is edge (u, v) for global synchronization.*

With this definition, all memory locations need synchronization protection. This is actually the semantics of the Sequential Consistency (SC) [90].

Extending the memory model makes it possible to support wider computation models. On the other hand, SC has been proven to be too hard to be implemented efficiently

in the distributed environment, so there must be a compromise between efficiency and the weakness of memory models. RC_dag aims at supporting more applications without becoming too strong to be implemented inexpensively.

Theorem 5.3.6 *RC_dag memory model is weaker than SC, i.e. $SC \subseteq RC\_dag$.*

Proof: This theorem follows from the definitions of SC and RC_dag. SC requires that the topological sort be the same for all memory locations, while RC_dag allows different topological sorts for different memory locations. $\forall (C, W_T) \in SC$, according to definition of RC_dag, $\exists T_l \in TS(C)$ such that $\forall u, W_T(l, u) = \Phi(l, u)$. So $(C, W_T) \in RC\_dag$. The theorem follows. □

We say RC_dag consistency model is an extension of LC, because RC_dag's computation C_{RC_dag} is an extension of LC's computation C, and Φ_{RC_dag} defines more properties than Φ_{LC}. That is to say, the computation running on LC can also run on RC_dag. However, the reverse is not always true. So we say RC_dag is more stringent than LC.

We define a new relation "$\ll$" for the extended dag, and this relation will be used to prove some properties of the RC_dag consistent memory model.

Definition 5.3.7 *For $\mathcal{G}_S = (\mathcal{V}, \mathcal{E})$ of a computation C, an observation $(C, \Phi_{\mathcal{G}_S})$ induces a relation $\ll$ on C's dag graph's nodes, as follows:*

1. *If $u \prec v$, then $u \ll v$,*
2. *If $u \ll v$ and $v \ll w$, then $u \ll w$.*
3. *If u reads location l, then $\Phi_{\mathcal{G}_S}(l, u) \ll u$.*

The following is the description of the acyclicity of the RC_dag consistency.

Definition 5.3.8 *We say an observation* $(C_{\mathcal{G}_S}, \Phi_S)$ *is* RC_dag *consistent if*

1. $\Phi_S = W_T$.

2. *For any* u *that reads* l, $u \not\ll W_T(l, u)$.

Theorem 5.3.9 *(Acyclicity Theorem) Given an observation* $(C_{\mathcal{G}_S}, W_T)$, $(C_{\mathcal{G}_S}, W_T)$ *is* RC_dag *consistent iff* $\ll$ *is acyclic.*

Proof: $(\Leftarrow)$ Suppose $\ll$ is acyclic, we prove by contradiction that the properties of RC_dag consistency are true. Property 1 is true obviously. Assume property 2 is false, i.e. there is node i reads x such that $i \ll W_T(x, i)$. Let $W_T(x, i) = j$, we have $i \ll j$. However, according to the definition 5.3.7, $W_T(x, i) \ll i$, i.e. $j \ll i$. There is a cycle with i and j.

$(\Rightarrow)$ Assume $(C_{\mathcal{G}_S}, W)$ is RC_dag consistent, we prove $\ll$ is acyclic by contradiction. Assume there is a cycle. If j reads from i, $W_T(x, j) = i$. On the other hand, $j \ll i$ because they are in cycle, thus $j \ll W_T(x, j)$ convicting the property 2 of definition 5.3.8. $\square$

5.4 The Extended Stealing Based Coherence Algorithm

To implement the RC_dag in SilkRoad, we further define an extended stealing based coherence function as follows:

Definition 5.4.1 *An Extended Stealing Based Coherence (ESBC) function is* $\Phi_S : \mathcal{L} \times \mathcal{V} \cup \{\perp\} \mapsto \mathcal{V} \cup \{\perp\}$ *such that*

1. *$\Phi_S(l, u) = u$ if $u \neq \perp$ and u writes on location l;*

2. *If $\Phi_S(l, v) = u \neq \perp$ then u writes on l and:*

   (a) *v reads the value of l written by u through thread stealing/return;*

   (b) *v reads the value of l written by u through mutual exclusion; or*

   (c) *v reads the value of l written by u and there is a global synchronization node $w \in V_S$ such that $u \prec w \prec v$.*

3. *$u \not\prec \Phi_S(l, u)$.*

Properties 2(a) and 2(b) in the above definition are for the case of $(u, v) \in \mathcal{E}$ and $(u, v) \in E_S$ respectively.

Lemma 5.4.2 *Φ_S is an observer function.*

Proof: We compare the above definition of Φ_S with the definition of observer function in Chapter 2. The above property 1 is actually the property 3 of definition 2.4.2. The above property 3 is the same as the property 2 in definition 2.4.2. The above property 2 is the same as the property 1 of definition 2.4.2 and it explains the three possibilities of the value written by u on local l being seen by v: via thread stealing/return, via lock acquisition, or via barrier. So we see that Φ_S satisfies the properties of an observer function. □

Lemma 5.4.3 *Given (C, Φ_S) where Φ_S is an ESBC function, $\mathcal{G}_S = (\mathcal{V} \cup V_S, \mathcal{E} \cup E_S)$ is the dag of C and a write node $u \in \mathcal{V}$ which writes on location l. For any node $v \in \mathcal{V}$ accessing location l, if $u \ll v$, then $\Phi_S(l, u)$ writes on l in the cache before v writes on l.*

Proof: Two cases of (u, v) are considered:

1. If $(u, v) \in \mathcal{E}$, i.e. $u \prec v$, if there is thread stealing/return between u and v, according to property 2 of ESBC, $\Phi_S(l, u)$ will be propagated to v if v accesses location l, so it must happen before v writes on l. If there is no thread stealing/return between u and v, then u and v are on the same processor, the order of their writing on l is according to the topological order, so $\Phi_S(l, u)$ writes before v writes.

2. If $(u, v) \in E_S$, then there must be a $(release, acquire)$ pair between u and v. According to property 3 of ESBC, $\Phi_S(l, u)$ will be propagated to v before v writes on location l. □

Theorem 5.4.4 *The Extended Stealing Based Coherence is RC_dag consistent, i.e. if $\forall \Phi_S$, Φ_S is an ESBC observer function, then we have $(C, \Phi_S) \in RC\_dag$.*

Proof: By contradiction. If the Extended Stealing Based Coherence algorithm is not RC_dag consistent, then there is an observation (C, Φ_S) such that $\ll_{(\mathcal{G}_S, \Phi_S)}$ is cyclic. Suppose $u_1 \ll u_2 \ll u_3 \ll \cdots \ll u_n \ll u_1$ is a circle and u_1 is a write node. By Lemma 5.4.3, u_1 writes on u's local cache before itself does. So we get a contradiction. □

In the following the implementation of the RC_dag memory model in the SilkRoad system is described.

5.5 Implementation of RC_dag

We borrow the semantics of Lazy Release Consistency (LRC) in TreadMarks [84, 85] in implementation with the *extended stealing based coherence algorithm*. In work steal-

ing, as discussed in Section 5.1, the victim node propagates its modifications on its local cache pages to the stealer only when the stealer requests for it. In lock acquisition, the last lock requester does not propagate its modifications in the critical section until the next requester needs them. This section introduces the implementation details of lock and barrier in SilkRoad.

5.5.1 Mutual Exclusion

A straightforward centralized scheme is used in implementing mutual exclusion. For each lock (identified by a lock number), a processor is chosen statically in a round-robin manner to be its manager according to the lock number. To enter the same critical section, different processors must acquire the same lock. To obtain a lock, the requester will send a lock request message to the lock's manager. If no other thread is holding the lock, the manager sends a reply message to the requester granting the lock acquisition request. If the lock is already held by an other thread, the request will be forwarded to the latest requester and the current requester waits in a queue associated with the lock. In this case there is a distributed waiting queue: each requester remembers its direct successor and the lock holder remembers the first waiting node in the queue. The lock holder will send a message to the first waiting thread when it releases the lock. The other waiting nodes remain in the queue. To conform to the messaging convention in Distributed Cilk, we used active messages [53] for message passing.

Eager diff creation and the write invalidation protocol are used to propagate the modifications. User programs have to acquire cluster-wide locks to access the shared variables and release it after finishing using the shared variables. When releasing a lock,

the diffs for the modifications on shared pages during this lock are created and stored. Thus there is a correspondence between diffs and locks. During the next remote lock acquisition, write notices will be sent to the requester. When the requester requires the diffs of a page, only the diffs associated with this lock will be sent out to the requester. So in this way the number of transferred diffs is greatly reduced. The idea of associate diffs with the lock number is similar to scope consistency, which implicitly build a relationship between data and synchronization operations. This makes our implementation different from the one in TreadMarks.

In implementation, the lock acquisition consists of following steps:

1. Save diffs of local dirty pages and set "write-protect" for the pages (the status of the pages will not be changed until the next access).

2. Send the request to the manager of the lock.

3. Wait for the grant message.

4. When the grant message arrives, save the write notices and invalidate corresponding pages according to the write notices. After that, if the memory locations in the invalidated pages are accessed, the saved diffs will be transferred from previous lock holder to the current holder.

The lock release performs the following basic operations:

1. Save diffs of local dirty pages and set "write-protect" for the pages.

2. Create write notices for the dirty pages.

3. Check the waiting queue of the lock. If there is an direct successor, the lock releaser sends the write notices to the successor node.

The mutual exclusion is guaranteed by the lock manager, since every time when there is lock acquire or release operation, the manager will check if the lock is held by anyone else, or check if there is anyone else waiting for the same lock. If a lock is released by one thread and then acquired by another thread, this procedure can be modeled as an synchronization edge according to the description of Section 5.2. Meanwhile, the lock releaser saves the diffs (operation 1 of above lock release description) and if the same location is accessed by the following lock acquire, the diffs will be propagated to the acquirer (operation 4 of above lock acquire description). This satisfies property 4 of definition 5.3.1. Similarly, in barrier operations (see the subsequent subsection), operation 1 save the diffs and they will be propagated to the appropriate nodes according operation 4. And this satisfies property 5 of definition 5.3.1. Properties 1, 2, and 3 of definition 5.3.1 are defined by Cilk and inherited by SilkRoad, they apply to normal thread stealing and returning situations. Lastly, according to definition 5.3.2, we say it implements RC_dag consistency.

5.5.2 Global Synchronization

The barrier global synchronization is implemented by using a central manager. When a node arrives at a barrier, it sends write notices of its dirty pages to the barrier manager and then waits for the manager's reply. The barrier manager assembles the write notices from each node and the forward them to the rest of the nodes when the manager has received the barrier requests from all nodes (it counts the number of the requests of this particular barrier and compare it with the cluster size). Each node then dis-assembles the write notices from the manager and marks the pages invalid accordingly. After

that, they can departure from the barrier point and begin with subsequent computation. During the subsequent computation, if a node needs to get the most up-to-date data of an invalidated page, it sends a message to the modifier directly to get the diffs.

In implementation, each barrier participant performs the following basic operations:

1. Create write notices and save diffs of the dirty pages, and set "write-protect" for the pages.

2. Send barrier request message (including the write notices) to barrier manager node.

3. Wait for the reply message from manager node.

4. When the reply message arrives, invalidate the pages according to the write notices in the reply message. After that, if the memory locations in the invalidated pages are accessed, diffs will be required by the accessing node and propagated from the previous modifying node.

5.5.3 User Shared Memory Allocation

In order to organize the virtual memory efficiently, the user shared memory is differentiated from the system level shared memory. In SilkRoad, to allocate global shared memory in programming, the user need to use "$SR\_malloc\_shared()$" function. Hence the allocated memory space is in another part of the heap (user shared memory space) and consistency of these shared pages are assured by the extension part of RC_dag. Meanwhile, user may also use global lock or barrier to access or synchronize the data on these pages.

5.6 The Theoretical Performance Analysis

In software DSM systems, the execution time of applications usually consists of the following parts: computation time, scheduling overhead, and synchronization overhead. The computation time is the actual time spent in computing, and this is determined by the application itself and the hardware of the nodes. The scheduling overhead is the overhead in distributing computation to each node. In the systems with dynamic scheduling (such as Cilk and SilkRoad), the scheduling overhead exists throughout the execution. The synchronization overhead is the overhead in performing synchronization operations, e.g., lock acquisition for a critical section or barrier synchronization. In RC_dag consistency, the synchronization overhead is tracable because of the semantic property of the Release Consistency: it allows the consistency of updated data to be delayed until *releases* and *acquires* occurs. With the above observation, the total execution time T_P of an application running on a software DSM system with P nodes can be expressed as follows:

$$T_P = T_C + T_S + T\_syn$$

where T_C is the computation time, T_S is the scheduling overhead, and T_syn is the overhead because of global synchronization.

In Cilk, for a multi-threaded computation which has T_1 total work (the execution time on one processor) and T_∞ critical-path length (the execution time on infinite number of processors), the expected execution time on P processor is $O(T_1(Z, L)/P + \mu H T_\infty)$, where Z is cache size, L is the line size of the cache, $H = Z/L$ is the cache height, and μ is the service time for a cache miss without congestion [61, 32]. Actually, this performance model is for LC and it has already included the computation time T_C

and scheduling overhead T_S. Since Cilk does not support global mutual exclusion in cluster computing environments, the T_syn portion does not exist in Cilk. With the extended *dag*, for the partially strict computation, the T_syn portion should be considered in SilkRoad because there may be global synchronization between threads.

For the synchronization overhead T_syn, the following situation is considered: during the computation, there are a large number of locking acquisitions (so that the lock waiting time is not too insignificant to be negligible), and computation time between a lock acquire and release pair is very little (this means that the granularity of the lock should not be large, as a generic suggestion for programming). In this case, the procedure of acquiring a particular lock (identified by a lock number) can be modeled as a $M/M/1$ queue: All the acquirers of the same lock are queuing and waiting to be served (i.e. getting the grant for the lock acquisition). If the average computation time inside the lock (i.e. after getting the grant from the lock manager and before lock release) is denoted by t_l, the lock request rate (i.e. the number of lock requests within a unit time) is λ, the average waiting time of each acquisition on this lock is $T_l = \frac{1}{\frac{1}{t_l}-\lambda}$. If there are totally m requests on this lock, the total waiting time is $m \times T_l$. If there are totally n locks (identified by lock number in program) and for each lock there are $m_1, m_2, ...m_n$ such kind of requests respectively (assuming there are not recursive lockings), then total lock waiting time can be expressed as $T\_syn = \sum_{i=1}^{n} m_i \frac{1}{\frac{1}{t_l}-\lambda}$.

In [61] and [32], it was proven that for any $\epsilon > 0$, with probability at least $1 - \epsilon$, the total number of steal requests and related page transfers is at most $O(HPT_\infty + HP\lg(1/\epsilon))$. This lemma applies to fully strict multithreaded computation with work-stealing scheduler, and the page transfers are resulted from thread stealings and thread returns with the random work stealing scheduler. In SilkRoad, all the page transfers

can be divided into two sets: one is the scheduling transfer caused by the random work stealing scheduler (the above mentioned lemma applies to this situation), the other is caused by the extensions in SilkRoad like global mutual exclusions (its overhead is in T_syn). Based on this observation, we have the following theorem for situations with large number of locks and small lock granularity (refer to Chapter 2 for explanation of some of the terminologies).

Theorem 5.6.1 *Consider any partially strict multithreaded computation executed on P processors, each with an $LRU(Z, L)$-cache of height H, using the work-stealing scheduler (like in Cilk and SilkRoad) in conjunction with the ESBC coherence algorithm. Let μ be the service time for a cache miss that encounters no congestion, and assume that accesses to the main memory are random and independent. Suppose the computation has T_1 computation work, $Q(Z, L)$ serial cache misses, $T_1(Z, L) = T_1 + \mu Q(Z, L)$ total work, and T_∞ critical-path length. Then for any $\epsilon > 0$, the execution time is $O((T_1(Z, L) + \sum_{i=1}^{n} m_i \frac{1}{\frac{1}{t_l} - \lambda})/P + \mu H T_\infty + \mu \lg P + \mu H \lg(1/\epsilon))$ with probability at least $1 - \epsilon$, where n is the total number of locks identified by lock number, $m_i (i = 1, 2, ..., n)$ is the number of requests on lock i in the computation, t_l is the average computation time inside the lock and λ is the frequency of the locks. Moreover, the expected execution time is $O((T_1(Z, L) + (\sum_{i=1}^{n} m_i \frac{1}{\frac{1}{t_l} - \lambda}))/P + \mu H T_\infty)$.*

Proof: As in [61] and [32], we shall also use an accounting argument to bound the running time. During the execution, at each time step, each processor puts a piece of silver into one particular buckets according to its activity at that time step. However, for partially strict multithreaded computation, two more buckets are considered: LOCK and LOCKWAIT. Additionally, unlike in [61], since the ESBC algorithm is not using

backing store as a shared virtual memory for the run-time system, there is a little change with the buckets XFERWAIT.

- **WORK**. A piece of silver is put in this bucket if the processor executes a task. So this bucket contains exactly T_1 dollars, because there are exactly T_1 tasks in the computation.

- **STEAL**. A piece of silver is put in this bucket if the processor sends a steal request. Since there are $O(HPT_\infty + HP\lg(1/\epsilon'))$ steal requests (see Lemma 26 of[61]), there are $O(HPT_\infty + HP\lg(1/\epsilon'))$ pieces of silver in the STEAL bucket. This portion is determined by the random work-stealing scheduler.

- **STEALWAIT**. A piece of silver is put in this bucket if the processor waits for a response to a steal request. According to the **recycling game**[36], if N requests are distributed randomly to P processors for service, with at most P requests outstanding simultaneously, the total time waiting for the requests to complete is $O(N + P\lg P + P\lg(1/\epsilon'))$ with probability at least $1 - \epsilon'$. Since there are $O(HPT_\infty + HP\lg(1/\epsilon'))$ steals, then the total time waiting for steal requests is $O(HPT_\infty + P\lg P + HP\lg(1/\epsilon'))$ with probability at least $1 - \epsilon'$[61]. However, in this case, since there is no reconciling cache to *backing store* for ESBC algorithm, we do not need to account for the time spent in reconciling. With the consideration of the idle steps to avoid too frequent steal requests[61, 32], the total number of pieces of silver in this bucket is $O(\mu HPT_\infty + P\lg P + HP\lg(1/\epsilon'))$.

- **XFER**. If a processor sends a line-transfer request, it puts a piece of silver into this bucket. Even though in ESBC algorithm the request is sent to the last victim

instead of the *backing store*, the number of pieces of silver in this bucket is still $O(\mu Q(Z,L) + \mu HPT_\infty + \mu HP \lg(1/\epsilon'))$[61].

- **XFERWAIT**. If a processor waits for a line transfer to complete, it puts a piece of silver into this bucket. The recycling game shows that there are $O(\mu Q(Z,L) + \mu HPT_\infty + \mu P \lg P + \mu HP \lg(1/\epsilon'))$ pieces of silver in this bucket with probability at least $1 - \epsilon'$.

- **LOCK**. If a processor performs an *acquire* operation, it puts a piece of silver into this bucket. This results from the extended dag for partially strict multithreaded computation. The number of lock acquisitions depends on the application and the scheduler. If n is the total number of locks identified by lock number, $m_i (i = 1, 2, ..., n)$ is the number of requests on lock i in the computation, then the total number of locks in the whole computation is $\sum_{i=1}^{n} m_i$.

- **LOCKWAIT**. According to the analysis above, the lock waiting time can be expressed as $T\_syn = \sum_{i=1}^{n} m_i \frac{1}{\frac{1}{t_l} - \lambda}$.

Now we add up the silver in each bucket and divide by P to get the running time. With probability at least $1 - 2\epsilon'$, the sum of all the pieces of silver in all the buckets is $T_1 + O(\mu Q(Z,L) + \mu HPT_\infty + \mu P \lg P + \mu HP \lg(1/\epsilon') + \sum_{i=1}^{n} m_i \frac{1}{\frac{1}{t_l} - \lambda})$ with probability at lease $1 - 2\epsilon'$. Dividing by P, we can obtain runtime $T_P \leq O((T_1 + \mu Q(Z,L))/P + \mu HT_\infty + \mu \lg P + \mu H \lg(1/\epsilon') + (\sum_{i=1}^{n} m_i \frac{1}{\frac{1}{t_l} - \lambda})/P)$ with probability at lease $1 - 2\epsilon'$. Using the identity $T_1(Z,L) = T_1 + \mu Q(Z,L)$ and substituting $\epsilon = 2\epsilon'$ yields the high-probability bound. The expected bound follows similarly. □

The performance model of distributed shared memory is an interesting problem but there is not much theoretical work on it. Some related work done on this topic include:

Donald Yeung et al. [137, 138] started from the clusters of SMPs with a relatively simple protocol. Their focus is on the performance of the large scale multi-grain system, which consists of clusters of SMP machines (totally hundreds of CPUs or even more) with different levels of memory. They model the performance with the consideration of page fault, network latency, etc, so they analyze the performance at a lower level than we do. Bilas [22] analyzed the performance of shared virtual memory on networks from communication layer, protocol layer, and application layer. The factors that may affect performance are analyzed in detail, but no theoretical performance models are proposed. Our performance model is based on Cilk's initial theoretical model with the consideration of the overhead of shared memory operations.

5.7 Discussion

Cilk is featured by its efficient load balancing and LRC is about distributed shared memory. They seems to be orthogonal, but there exists a cross point, which is the memory consistency model. Memory consistency model is a critical element of DSM. On the other hand, Cilk's efficient load balancing is built on both work-stealing and Divide and Conquer paradigm. Its underlying supporting memory consistency model is LC. In our work, our main target is to achieve the support of multiple paradigms (including Divide and Conquer and SPMD, etc). In order to achieve this goal , we explore the memory model approach. We focus our work on the overlapped part of these two topic and hope to extend the underlying memory model and hence more paradigm can be supported (see Figure 5.6). In addition, LRC's "lazy" idea provides some hints of reducing network traffic for existing system.

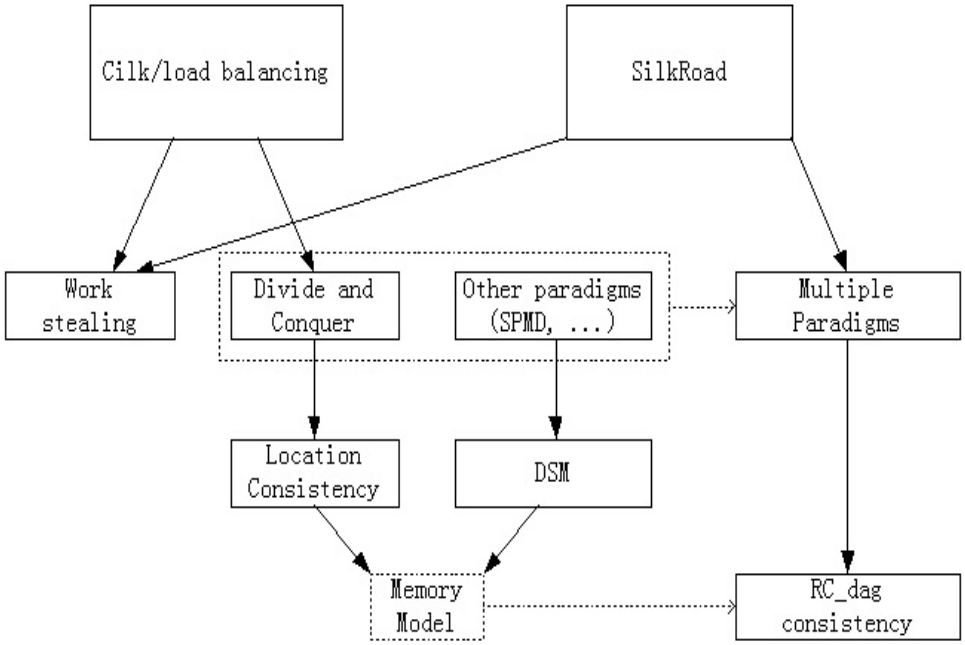


Figure 5.6: The memory model approach to achieve multiple paradigms in SilkRoad.

In our earlier implementation of SilkRoad [105], the barrier mechanism was not implemented and we only introduced the lock mechanism without changing Cilk's backing store and the way to keep memory consistent in Cilk runtime system. So the consistency of the system information was maintained by LC, and the lock was implemented with LRC semantics. In our later work, the barrier was introduced. More importantly, RC_dag is formally analyzed based on the computation-centric memory model theory system, and the later implementation differs from the earlier one in that "lazy" semantics are introduced and the backing store of Cilk are removed. So the similarity between RC_dag and LRC is that the modifications of shared memory pages are propagated in a "lazy" style in both memory models. However, one difference is that the system data modification propagations are triggered by thread stealings or returns in RC_dag, while in LRC all propagations are triggered by lock or barrier operations. So we can also say

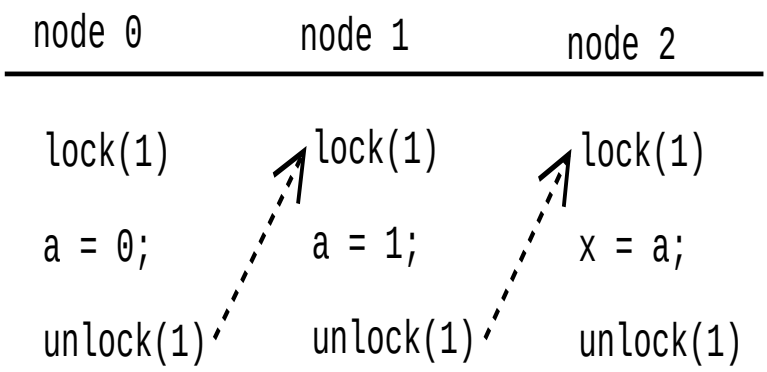


Figure 5.7: A situation that might be affected by interference of lock operations and thread migration

RC_dag is a "hybrid" model in that the page propagations are coupled with either the operations of thread stealings or global synchronizations (i.e. global lock acquisitions and barriers) according to the ESBC introduced in this chapter.

At the programming level, our extensions introduce different semantics from LRC in that more restrictions are applied to the programmers since thread migration and memory consistency operations co-exist. With typical LRC, programmers care about the data sharing via the mechanisms like barrier and lock. In contrast, RC_dag consistency puts more restrictions on programmers: In the programs with locks, the user should reduce the lock granularity and make sure that there is no thread spawning within the locks. For example, Figure 5.7 illustrates a simple situation of using shared variables with locks(the dotted arrows show the lock transferring sequence): the thread on node0 acquires a lock and write a value to the shared variable a which is followed by a lock release; then the thread on node1 also does a write operation on a; lastly the thread on node2 does a read operation on a. There will be no problem if these three pieces code are running on different node under LRC. However, with RC_dag, the scenario

node 0	node 1

`barrier(0);`
`x = a[0][0] + a[0][1];`
`a[0][0] = x;`
`barrier(1);`
	`barrier(0);`
	`y = a[0][0] - a[0][1];`
	`a[0][1] = y;`
	`barrier(1);`

Figure 5.8: A situation that might be affected by interference of barrier operations and thread migration

may be changed if, after lock acquisition, the thread on node 2 spawns another thread and it is migrated to node0 if node0 is free while node2 is heavy loaded. In this case, the read operation (i.e. $x = a$;) will possibly get an obsolete value since node0 is not the last writer of a.

In SPMD programs with global barriers, the user should care about the number of threads and their level in the spawn tree. Users are suggested to spawn as many threads as the number of processors and these threads are leaves in the spawn tree. Otherwise, thread migration might also interfere with memory consistency operations. For example, the pseudo code in Figure 5.8 illustrates this situation: in between two barriers (i.e. $barrier(0)$ and $barrier(1)$), the program calculate the values of the elements in a 2×2 matrix (stored in shared memory): the value of the first column is calculated by summing the values of the elements in the same row of the two columns, and the second column is calculated by subtracting the values of the elements in the same row of the two columns (the code in the figure only shows the calculation of the first elements of the columns). If this is done by two threads on two processors and each calculates one

column of elements, each thread will get the same values of the shared variables when they leave the first barrier (i.e. $barrier(0)$). According to LRC, the written values of write operations (i.e. $a[0][0] = x$ and $a[0][1] = y$) on the shared memory will not be "seen" by each other until they all arrive the barrier (i.e. $barrier(1)$). So those write operation do not affect the other thread's read operation. However, in RC_dag, after they leave barrier 0, if thread 0 spawns some other threads and itself is migrated to node1 because of the load imbalance, the write operations might affect the read operations, depending on the execution speed of the two threads.

So we can see that for the threads that do memory consistency operations (like lock release/acquire and barrier), these restrictions (which do not exist in normal LRC systems) keep them running at the leaf level in a spawn tree so that they are not migrated (according to Cilk's policy, the parent threads are usually stolen so that the stealer can get more work to do), hence avoiding the complicated situation in which memory consistency operations and thread migrations interfere with each other. The thread migration in SilkRoad is random and not predictable, so far we did some feasibility study on the synthesis of these two aspects. It is still a challenging and interesting future research topic to do further exploration.

5.8 Conclusions

One difference between SilkRoad and Cilk is that SilkRoad does not use the backing store to maintain the consistency of the cache pages in order to reduce the network traffic. SilkRoad employs SBC to maintain the coherence at runtime level. Without backing store, the consistency of system data is maintained with the event of thread

stealing and return, which makes it different from LRC. Moreover, RC_dag memory consistency model is built on the basis of LC and it is motivated by the attempt to providing user level shared memory based on LC to support wider range of computation. The semantics of the LRC are introduced and the BACKER algorithm in Cilk is replaced by ESBC algorithm. This shows a way to support more programming paradigms in a parallel system.

Chapter 6

SilkRoad Performance Evaluation

This chapter evaluates the performance of the SilkRoad system. Our experiments are to demonstrate two major aspects of SilkRoad:

- The efficiency and performance of RC_dag.
- The ability to support multiple paradigms.

As it is mentioned in previous chapters, the implementation of RC_dag consistency makes SilkRoad provide a user level shared virtual memory and consequently makes it possible to support the applications programmed in the paradigms other than Divide-and-Conquer. This chapter shows the experimental performance and discussions on the results. The speedup of various applications with different problem sizes will be shown first to demonstrate the overall performance of SilkRoad. We further compare the performance of SilkRoad and Cilk by running some Divide-and-Conquer applications chosen from Cilk's test suite. This is to show the effect of reducing network traffic and the introduced overhead in SilkRoad when providing a global shared memory based

on Cilk's runtime system. The performance of SilkRoad is also compared with that of TreadMarks. This is done by running some non-Divide-and-Conquer applications.

The remainder of this chapter is organized as follows: Section 6.1 introduces the platform used for the performance evaluation. Section 6.2 describes the applications in our test suite and their attributes. In Section 6.3, the experimental results are shown as well as the analysis and discussion on them. The comparisons between SilkRoad, Cilk, and TreadMarks are also shown and analyzed. Finally, Section 6.4 gives a conclusion of this chapter.

6.1 Experimental Platform

The test-bed for our experiments is a 16-node PC cluster. The processor of each node is Intel Pentium-III 500 MHz CPU. The memory size is 256 MB (or 512 MB for the node acting as the NFS/NIS server). Nodes are interconnected with 100Mbps Fast Ethernet network in a star topology through a 100BaseT switch. The operating system of each node is RedHat Linux 6.2 with the kernel of version 2.2.18.

6.2 Test Application Suite

In our experiments, nine applications are used and they are introduced below. Depending on the nature of the applications, different programming paradigms were used. Specifically, the Matrix Multiplication, N queen, LU, and Knary are selected from Cilk's test suite and the Divide-and-Conquer paradigm is used in these programs. The Traveling Sales Problem program is basically written with Master/Slave paradigm, and

it uses shared variables. Embarrassingly Parallel, Red-Black Successive Over Relaxation, Jacobi iteration, and Gaussian Elimination use the SPMD paradigms and also need a global shared memory.

Matrix Multiplication (*Matmul*) Matrix multiplication is a basic application which is widely used in benchmarking. The *Matmul* program multiplies two $n \times n$ matrices A and B and puts the results into another matrix C. In our test suite, the *Matmul* program uses classical algorithm to do the multiplication. It fits into the divide-and-conquer paradigm well: recursively splitting the problem into eight $n/2 \times n/2$ matrix multiplication subproblems and combining the results with one $n \times n$ addition. This program needs the runtime level shared memory support because three matrices are shared among the spawned threads. Neither lock nor barrier is needed however as the basic parallel control constructs suffice.

N Queen Problem (*NQueen*) The objective of the *NQueen* program is to place n queens on an $n \times n$ chess board such that they do not attack each other. The program finds all such configurations for a given chess board size and differs from the original n queens program in Cilk, in which case if one solution is found all the other searching threads are aborted. The SilkRoad program explores the different columns of a row in parallel, using a divide-and-conquer strategy. The chess board is placed in the DSM such that child threads can get the chess board configuration from their parent thread. The data in the chess board must be kept consistent at runtime level. Again the user lock is not necessary in the program.

LU decomposition (*LU*) The LU program performs the Divide-and-Conquer form of a blocked LU decomposition of a dense matrix $(A = LU)$. LU factorization is

the most time consuming step of a common method of solving a system of linear equations. The dense $n \times n$ matrix is divided into an $N \times N$ array of $B \times B$ blocks to exploit temporal locality of sub-matrix elements ($n = NB$). In our experimental program *LU*, the block size is set to be 16.

Knary (*Knary*) *Knary* is a synthetic benchmark in Cilk. Its parameters can be set to produce a variety of values for work and critical path length. The syntax of the command line is $knary < serial >< parallel >< work >< depth >$. With the provided parameters, it generates a tree of depth $< depth >$ and branching factor in which the first $< serial >$ children at every level are executed serially and the remainder $< parallel >$ children are executed in parallel. At each node of the tree, the program runs an empty "for" loop for a number of times.

Traveling Salesman Problem (*TSP*) The *TSP* program solves the traveling salesman problem using a branch and bound algorithm. In this program, a number of workers (i.e., threads) are spawned to explore different paths. The actual number of workers depends on the number of available processors. Unexplored paths are stored in a global priority queue in the user-level shared memory. All workers will retrieve the paths from the priority queue. So it is basically a work-pool paradigm. The bound is also kept in the shared memory, and each thread accesses (i.e., reads or writes) the bound through a lock, in order to ensure the consistency.

Embarrassingly Parallel (*EP*) The Embarrassingly Parallel (*EP*) from NAS benchmark suite [11] accumulates two-dimensional statistics from a large number of Gaussian pseudo-random numbers which are generated according to particular scheme that is well-suited for parallel computation. The computation-communica-

tion ratio of the parallel version is very high and the only communication occurs when summing up a list in the final of the program. The updates to the shared list are protected by a lock.

Red-Black Successive Over-Relaxation (*SOR*) Red-Black Successive Over-Relaxation (*SOR*) is a method of solving partial differential equations. In parallel *SOR*, The red and the black arrays are divided into roughly equal size bands of rows and each of them is distributed to a different processor. It is a typical SPMD style: same code are executed on different matrices and during the computation, communication occurs across the boundary rows between bands. In SilkRoad, the red and black arrays are stored in shared memory and the program uses global barriers to synchronize.

Jacobi iteration (*Jacobi*) Jacobi is a method for solving partial differential equations. The *Jacobi* program iterates over a two-dimensional array. During each iteration, every matrix element is updated to the average of its nearest neighbors (above, below, left, and right). Because of the strong data dependence, it is hard to divide the problem into several smaller independent problems, so we use the SPMD paradigm to solve this problem. The program uses a local array to store the new values computed during each iteration in order to avoid overwriting the old value of the element before it is used by its neighbor. In the parallel version, the two-dimensional array is divided into roughly equal size parts and distributed to each node. Their boundary rows are shared by the neighboring nodes. Barriers are used for synchronization after the calculation and copying data from shared array to local array in each iteration.

Gaussian Elimination (*Gauss*) Gaussian Elimination (*Gauss*) decomposes a square matrix into upper and lower triangular submatrices by repeatedly eliminating the elements of the matrix under diagonal, one column at a time. In this SPMD paradigm, communication occurs after the calculation in each iteration via global barriers.

6.3 Experimental Results and Discussion

6.3.1 Performance Evaluation

The overall performance of SilkRoad programs is listed in Table 6.1. In the following analysis, the speedup is computed by dividing the sequential program's executing time by the corresponding parallel program's executing time. We used the `gcc` compiler (version 2.91.66) to compile all of the application programs.

Applications	serial	2 procs	4 procs	8 procs	16 procs
Matmul($1K \times 1K$)	84.66s	38.41s/2.20	28.14s/3.00	24.73s/3.42	21.27s/3.98
NQueen(13)	76.64s	39.44s/1.94	19.78s/3.87	10.84s/7.07	5.43s/14.11
LU($1K \times 1K$)	83.55s	28.30s/2.95	21.59s/3.87	16.33s/5.11	13.74s/6.08
Knary(0,10,10,7)	31.77s	15.95s/1.99	8.08s/3.93	4.13s/7.69	2.69s/11.81
TSP(19b)	11.54s	6.89s/1.67	5.43s/2.13	3.37s/3.42	4.92s/2.34
EP(2^{24})	22.99s	11.62s/1.98	6.01s/3.83	3.15s/7.30	1.59s/14.46
SOR($2K \times 2K$)	21.69s	11.62s/1.87	6.57s/3.30	3.78s/5.74	2.55s/8.50
Jacobi($1K \times 1K$)	12.06s	6.19s/1.94	3.72s/3.24	2.67s/4.51	1.94s/6.21
Gauss($1K \times 1K$)	23.51s	13.42s/1.74	8.43s/2.79	5.14s/4.57	7.32s/3.21

Table 6.1: Timing/speedup of the SilkRoad applications.

Matmul

In *Matmul*, the Divide-and-Conquer strategy used in the SilkRoad program achieved a good performance speedup. We tested with the matrix size of 1024×1024 and we achieved good speedup and even super-linear speedup. The speedup is 2.20 on 2 processors, 3.00 on 4 processors, 3.42 on 8 processors, and 3.98 on 16 processors. The super-linear speedup on two processors comes from the data locality. In SilkRoad, if all elements of a divided *Matmul* block can fit in the local cache, there are much fewer cache misses in comparison with the sequential program that stores the matrices in the cache in row major order. When the matrices cannot fit into the local cache, thrashing occurs. In the SilkRoad *Matmul* program, the matrices are divided into small blocks until it reaches the size of 16×16 allowing them to fit easily into the local cache. The system overhead reduces the overall speedup a lot in *Matmul*. For example, for the problem size of 1024×1024 on four nodes, the CPU working time takes about 65% of the overall execution time and the rest is taken by system, which spends a lot of time to process the large size of data in the shared memory.

NQueen

We ran the *NQueen* program with the board size of 13. It achieved speedup of 1.94 on 2 processors, 3.87 on 4 processors, 7.07 on 8 processors, and 14.11 on 16 processors. In this program, the chess board is stored in the DSM, but the amount of data (i.e., the current chess board configuration) to be transferred is less than that of *Matmul*. Thus, the parallel execution does not suffer too much from the DSM overhead and it achieved better speedup than *Matmul*. In our experiments the system overhead of

the execution is less than 1% of the total execution time so that the over speedups are good. In comparison with Cilk *NQueen* (shown in Table 6.3), the speedup of SilkRoad *NQueen* is comparable in scale of our experimental cluster.

LU

We ran *LU* program with the matrix size of 1024×1024, and we got speedup 2.95, 3.87, 5.11, and 6.08 on 2, 4, 8, and 16 processors respectively. Like the *Matmul*, by dividing large matrices into the small size blocks (16×16 blocks in this application), data locality was utilized and parallelism was increased. In comparison with Cilk *LU* (as it is shown in Table 6.3), SilkRoad performs close to Cilk. However, when the problem size is large, the system overhead goes up quickly (for example, it takes about 20% of the execution time for 1024×1024 on four nodes in our experiments) and the overall speedup is affected a lot.

Knary

We ran *Knary* program by providing the parameters with the values (0,10,10,7) and achieved the speedup 1.99, 3.93, 7.69, and 11.81 on 2, 4, 8, and 16 processors respectively. In this example, the serial work is specified to be null and hence high parallelism is achieved.

TSP

In *TSP*, the distances of all cities, the current shortest route, the bound of the current shortest route, and a priority queue storing all unexplored routes are held in global

shared memory that is frequently accessed by multiple worker threads via locks. This paradigm is not directly supported in Cilk because it requires user-level shared memory and global mutual exclusion. Currently the SilkRoad *TSP* program with the size of 19b achieves speedup of 1.68 on 2 processors, speedup of 2.13 on 4 processors, 3.42 on 8 processors, and 2.35 on 16 processors respectively for the same problem size. When the number of processors is increased to 16, the execution slows down. This shows that the rapid increase of communication overhead on 16 processors offsets the benefits of parallelism.

EP

For *EP*, with 2^{24} random numbers, we obtained speedup of 1.98, 3.83, 7.30, and 14.46 on 2, 4, 8, and 16 nodes respectively. In this program, the computation to communication ratio is very high so the communication overhead are compensated and good speedups are achieved. This program also can not be directly supported by Cilk because it needs a global lock to access data stored in the shared variable.

SOR

For the typical SPMD style *SOR*, we ran it with the problem size of 80 iterations with 2048×2048 matrix size. On 2 processors the speedup is 1.87. On 4, 8, and 16 processors the speedups are 3.30, 5.74, and 8.51 respectively. This program uses barrier for global synchronization during computation. Usually the barrier operations are considered to be time consuming. In SilkRoad, for example, with problem size of 512×512 running on four processors, the barrier operations (including barrier waiting and mes-

sage processing) take about 9% of the total execution time. The speedups show that the barrier can be efficiently realized in RC_dag consistency.

Jacobi

In another SPMD program *Jacobi* with 100 iterations on the matrix size of 1024×1024, we achieved speedups of 1.94 on 2 processors, 3.24 on 4 processors, 4.52 on 8 processors, and 6.21 on 16 processors. It shows the efficiency of the RC_dag's barrier for global synchronization in SPMD paradigms.

Gauss

Gauss got speedup of 1.75, 2.79, 4.57, and 3.21 on 2, 4, 8, and 16 processors respectively. On larger cluster size (16 nodes), the speedup is decreased down and this is because the overhead of processing barrier write notices (i.e. assembled by the barrier manager and dis-assembled by each node) increases fast when more nodes are involved into the computation. Meanwhile, this problem size may not put enough computation on each node to offset the increase overhead.

We also test SilkRoad's scalability with problem size. We ran the *NQueen* with problem size of 11, 12, 13, and 14. We also ran the *SOR* with matrix size of 512×512, 1024×1024, 2048×1024, and 2048×2048. The results are shown in Table 6.2.

For *SOR* with 512×512 matrix size, SilkRoad's speedup is not as good as those with larger matrix sizes. The speedup even drops when the number of processors is increased to 16. This shows that for small problem size the overhead of the system cannot be offset by the achieved parallelism and larger data size makes the application

Applications	problem size	2 procs	4 procs	8 procs	16 procs
SOR	512×512	1.71	2.76	3.24	2.59
	1024×1024	1.80	2.99	3.53	3.71
	2048×1024	1.79	3.01	3.96	5.09
	2048×2048	1.87	3.30	5.74	8.50
NQueen	11	1.92	3.20	4.69	5.61
	12	1.96	3.76	6.18	7.82
	13	1.94	3.87	7.07	14.11
	14	1.99	3.94	7.60	14.67

Table 6.2: SilkRoad's speedup with different problem sizes.

benefit more from parallelism.

Similarly, *NQueen* also achieved better speedups with larger problem sizes, especially on larger cluster scales. This is mainly because of the relatively small data communication in computation.

6.3.2 Comparing with Cilk

In order to see the effects of extending the memory model of Cilk, the performance of the SilkRoad is compared with that of Cilk in Table 6.3. The applications shown in the table are all using Divide-and-Conquer paradigm. Table 6.4,Table 6.5, and Table 6.6 show that generally RC_dag consistency results in less communication data and messages, since the stealing based lazy style diff propagation policy is employed and hence the modifications of shared pages are not written back to the *backing store* each time when thread stealing and return happen.

For *Matmul*, SilkRoad runs in a speed close to Cilk with smaller cluster scale, even though SilkRoad sends less messages and transferred less data. This is because processing the messages in SilkRoad takes longer time since more data are piggy backed

within the messages. SilkRoad achieves better performance with larger cluster scale. This shows that when the number of processors is increased, the introduction of LRC ideas takes more effect. However, when the cluster scales up, we notice that the transferred data increase faster than the number of messages (Table 6.6 shows SilkRoad *Matmul* transferred more data than Cilk *Matmul*). This is because the number of diffs in the messages increases very fast with the number of processors, so it shows the diffs maintenance policy in SilkRoad can be further optimized.

For *NQueen*, the speed of SilkRoad gets close to that of Cilk when increasing the number of processors. Since the data of *NQueen* is much less than that of *Matmul*, the network communication is less frequent than Cilk even when the number of processors increases. Like in *Matmul* application, SilkRoad transferred less data and messages than Cilk.

For *LU* and *Knary*, SilkRoad also achieved comparable speedup on all of the cluster scales. Generally we can see that with removing the backing store and maintaining the memory consistency with thread migration, SilkRoad performs comparable to Cilk with four of our applications which use Divide-and-Conquer paradigm.

6.3.3 Comparing with TreadMarks

For the applications that are not directly supported by Cilk and are implemented with the paradigms other than Divide-and-Conquer, their performance are compared against TreadMarks (version 1.0.3), a well-known software DSM system. The results are shown in Table 6.7.

For *SOR (2K×2K, 80 iterations)*, SilkRoad got the speedups very close to that of

Applications	Number of processors	SilkRoad	Cilk
Matmul ($1K \times 1K$)	2	38.41s	29.36s
	4	28.14s	26.08s
	8	24.73s	24.39s
	16	21.27s	22.01s
NQueen (13)	2	39.44s	38.32s
	4	19.78s	19.58s
	8	10.14s	9.98s
	16	5.43s	5.65s
LU ($1K \times 1K$)	2	28.30s	28.55s
	4	21.59s	22.64s
	8	16.33s	17.06s
	16	13.74s	15.04s
Knary ($0, 10, 10, 7$)	2	15.95s	15.82s
	4	8.08s	8.05s
	8	4.13s	4.08s
	16	2.69s	2.66s

Table 6.3: Timing of the applications for both SilkRoad and Cilk.

TreadMarks with a small number of processors, but when the number of processors is increased, the speedup of SilkRoad is less than that of TreadMarks. This shows that the barrier implementation in SilkRoad is as efficient as TreadMarks with small scale cluster but less efficient when the cluster scales up. This is mainly because the quickly increased message size in SilkRoad offsets some of the gained performance.

For *TSP* with 19 cities, both TreadMarks and SilkRoad achieved increased speedups

Applications	Transferred data		Number of messages	
	SilkRoad	Cilk	SilkRoad	Cilk
Matmul ($1K \times 1K$)	30.6MB	78.5MB	28,404	98,849
NQueen (14)	132KB	296KB	561	633
LU ($1K \times 1K$)	6.1MB	30.7MB	16,232	39,555
Knary ($0, 10, 10, 7$)	35KB	107KB	375	381

Table 6.4: Messages and transferred data in the execution of SilkRoad and Cilk applications (running on 2 processors).

Applications	Transferred data		Number of messages	
	SilkRoad	Cilk	SilkRoad	Cilk
Matmul ($1K \times 1K$)	94.4MB	160.5MB	95,619	195,169
NQueen (14)	617.6KB	1.9MB	2,793	4,216
LU ($1K \times 1K$)	17.9MB	63.9MB	51,498	116,971
Knary $(0, 10, 10, 7)$	138KB	765KB	1,285	2,493

Table 6.5: Messages and transferred data in the execution of SilkRoad and Cilk applications (running on 4 processors).

Applications	Transferred data		Number of messages	
	SilkRoad	Cilk	SilkRoad	Cilk
Matmul ($1K \times 1K$)	343MB	268MB	207,955	330,193
NQueen (14)	2.77MB	6.94MB	7,877	15,938
LU ($1K \times 1K$)	122.39MB	147.6MB	138,214	319,495
Knary $(0, 10, 10, 7)$	765KB	1.87MB	5207	6460

Table 6.6: Messages and transferred data in the execution of SilkRoad and Cilk applications (running on 8 processors).

with smaller cluster scale, but the speedup slows down with larger cluster scale (sixteen processors). The performance descrease on sixteen nodes shows the implementation of lock needs improvement because the overhead of diff processing (creating diffs, comparing and filtering diffs, and applying diffs) increases fast when the cluster size is big. TreadMarks outperforms SilkRoad, because in *TSP*, the lock for the global queue (storing the partial results) has large granularity and SilkRoad threads spend more time in waiting for the lock. This also shows the implementation of SilkRoad needs further optimization. Moreover, the eager diff creation in SilkRoad also takes more time than TreadMarks in creating diffs that will possibly not be used later.

For *Gauss*, the speedup of both SilkRoad and TreadMarks are increasing on two, four, and eight processors but decreases when the number of processors is increased to sixteen. The SilkRoad performance descrease on sixteen nodes is due to the introduced

Applications	No. of processors	Speedup of SilkRoad	Speedup of TreadMarks
SOR ($2K \times 2K$, 80 iterations)	2	1.87	1.82
	4	3.30	3.49
	8	5.74	6.36
	16	8.50	10.09
TSP (19b)	2	1.67	1.88
	4	2.13	3.60
	8	3.42	4.47
	16	2.34	2.92
Gauss ($1K \times 1K$)	2	1.74	1.85
	4	2.79	3.02
	8	4.57	5.03
	16	3.21	4.89
EP (2^{24})	2	1.98	1.99
	4	3.83	3.99
	8	7.30	7.98
	16	14.46	15.98

Table 6.7: Comparison of speedup for both SilkRoad and TreadMarks applications.

barrier overhead which is big when the cluster size is big. The barrier manager is potentially a performance bottleneck because it needs to assemble the write notices (i.e. the information about which pages have been modified by which node) and broadcasts to each node. Each node dis-assemble the write notices upon receiving from the manager. This assembly (at the manager side) and dis-assembly (at each barrier participant) procedure for each barrier can be time-consuming when the number of processors is large, since the manager needs to assemble the write notices for a lot more times and each time with more source nodes' write notices. Each node also needs to dis-assemble more nodes' write notices. This overhead increases quickly when the number of nodes grows large (greater than eight). Meanwhile, every node sends the barrier request to the manager node and waiting in a queue for the reply. When the number of nodes increased from eight to sixteen, the barrier waiting time increases obviously since there

are eight more requests in the queue. In this program, SilkRoad also achieved speedups close to but a little less than TreadMarks, especially with large cluster scale.

EP also achieved good performance on two, four, eight, and sixteen processors comparing to TreadMarks. This shows that the low communication in Embarrassingly Parallel results in very low communication overhead and high speedup in SilkRoad.

It can be seen that the performance of SilkRoad is not significantly worse than that of TreadMarks. On the other hand, since it is more natural to solve some problems with Divide-and-Conquer paradigm, SilkRoad provides users more choices of paradigms for their parallel programming. In the support of the paradigms other than Divide and Conquer, we notice that some programs (for example, Guass) of the newly supported paradigms (for example, SPMD) get performance decrease when the cluster size is big (i.e. sixteen). The effect of reducing network traffic is not obvious on these applications and this is mainly due to (1)There are very few thread migrations (for example, for four nodes only four threads spawned and they only migrate from the starting node to the rest computing nodes and then back to the starting node when finish) and hence very few operations on backing store. In comparison, Divide and Conquer programs usually produce thousands or more thread migrations, so the effect of removing backing store is obvious; (2)The quickly increased barrier processing overhead (as explained in previous page). On the other hand, we also notice that the TreadMarks version of those programs also behave similarly (i.e. performance decrease with sixteen nodes) in our experiments, which implies not only the runtime system matters, but also the characteristics of programs can affect the performance in large clusters. In summary, the experiments show that SilkRoad is able to allow new types of applications to be programmed, which is our main target. The inefficiency of some newly supported programs with large clus-

ter size implies that the reduction in messages may generate new overheads and some of the implementations need improvement, so the overall efficiency is not necessarily improved (i.e. it depends on the programming paradigm used by the applications).

Last, the experiments also show that the extension of memory model in SilkRoad does not hurt the load balancing inherited from Cilk. With the inherited multithreading and dynamic parallelism, SilkRoad can achieve good load balancing with Divide-and-Conquer paradigm. Table 6.8 shows some statistical data in one typical execution of the *Matmul* example and Table 6.9 shows the results of *Matmul* program implemented in TreadMarks with SPMD paradigm. Even though not all data is directly comparable, the result show SilkRoad got good load balance among processors.

Processor No.	Total work	Received Messages
0	41.04	29851
1	53.35	20824
2	53.94	20616
3	40.95	23079
TOTAL	189.28	94370

Table 6.8: Output of processor load (in seconds) and messages in one execution of *Matmul* (1024×1024) on 4 processors in SilkRoad.

Processor No.	messages	barrier waiting time (seconds)
0	7274	1.3
1	3593	1.61
2	3530	0.49
3	5838	0.49
TOTAL	20235	3.89

Table 6.9: Some statistic data in one execution of *matmul* (1024×1024) on 4 processors in TreadMarks.

6.4 Conclusion

In this chapter, the performance of SilkRoad is evaluated and analyzed to show its efficiency. In addition to Cilk's test programs with Divide and Conquer, to test the user level shared virtual memory, some applications with other paradigms are also selected for the evaluation. The overall performance of SilkRoad and its scalability with problem size are examined. A comparison between SilkRoad and Cilk is presented in order to examine the side-effects of extending Cilk's Location Consistency and to show the efficiency of RC_dag consistency model in reducing the network traffic when solving problems with Divide-and-Conquer paradigm. For those applications with other paradigms, we compared them with TreadMarks applications.

In Chapter 4 and Chapter 5 the idea of removing Cilk's backing store is introduced and in this chapter the experimental programming work shows that the idea can be used empirically. The programs in this chapter are mainly to show the performance (which is more relevant to our topic), but they also show how the additional facilities of SilkRoad (i.e. global lock and barrier mechanisms) can be used in programming.

In summary, our experimental results show that with the extended memory consistency model (i.e. RC_dag consistency), SilkRoad has the performance comparable to Cilk while reducing the network traffic and supporting more paradigms (using user-level shared memory) with fairly good efficiency.

Chapter 7

Conclusions

This chapter summarizes the thesis and outlines areas which merit further investigation.

7.1 Conclusions

In this dissertation, we explored the techniques to support multiple parallel programming paradigms theoretically and empirically.

Theoretically, a graph-theoretical analysis approach is presented in order to analyze parallel programming paradigms more generically. Under this framework, several paradigms are defined uniformly based on the concept of execution instance dag. Moreover, it is shown that the underlying memory model of a parallel computing system plays an important role in supporting multiple paradigms.

In order to achieve our goal of supporting multiple paradigms, we find the cross point of the "orthogonal" Cilk and DSM, which is the underlying memory consistency model. Empirically, it is shown that extending the memory model of one existing parallel system is a feasible way to support more paradigms. By extending LC to RC_dag,

we developed a variant system of Cilk, i.e. SilkRoad, to provide user level shared memory (with linguistic support for mutual exclusion and global synchronization) and hence support more programming paradigms. The performance evaluation showed the efficiency of RC_dag consistency and SilkRoad's ability to support multiple parallel programming paradigms with the utilization of user level shared memory. The comparison between SilkRoad and Cilk showed that SilkRoad achieves good overall performance while extending the memory model of Cilk. With the comparison to Cilk and TreadMarks system, we showed that SilkRoad supports wider paradigms based on Cilk, while at the same time achieves rather good performance. Moreover, the programmability of Cilk/SilkRoad as a parallel programming language is also examined. Cilk/SilkRoad's solutions to various examples show its effectiveness in parallel programming. With SilkRoad's extension, Cilk/SilkRoad's programmability is also enhanced.

In summary, our work explored an approach to support more paradigms by strengthening the underlying memory model of an existing parallel system, and the performance of SilkRoad system showed the feasibility of this approach.

7.2 Future work

The memory consistency model in SilkRoad was originally inspired by the need of extensions to support other kinds of synchronization in Cilk. RC_dag consistency provides the additional operations like global mutual exclusion and synchronization (which is actually the way it extends the *LC* of Cilk), and the experimental programs demonstrate their use in programming with different paradigms. In our work we explored a way of extending the memory consistency model, but it is too early to say that the

synchronization in SilkRoad is very easy-to-use and efficient. There may be better ways for synchronization. The efficiency of locks in SilkRoad still needs improvement. Moreover, the efficient support and management for producer/consumer-like synchronizations still remain to be explored.

In the future the following improvements are to be achieved:

Other approaches to extend *LC*. In SilkRoad, we introduce the semantics of lazy release consistency. It is also possible to use some other ways to extend *LC*. For example, introducing the semantics of the scope consistency, etc. A "lazy" style policy without "home" is used to achieve less network traffic. However, other approaches are also worth exploring.

In SMP systems or other centralized environments, Cilk mainly supports Divide-and-Conquer paradigm. Since shared variables are already used in Cilk (the SMP version), it should be easier to write Cilk programs with other paradigms. It can be examined to find out what are the concrete requirements (at both runtime level and user level) for supporting multiple paradigms based on Divide-and-Conquer.

Besides providing user level shared memory, are there other ways to enlarge the supported paradigms? This depends on the particular paradigms and more further study work is needed. Even for the user level shared memory, besides lock and barrier mechanisms, how to implement other mechanisms so that a more powerful parallel programming system can be provided also needs to be figured out.

SilkRoad is still in experimental stage and more applications are to be developed with various paradigms. Besides the benchmarking programs used in this thesis, the applications solving real problems are still needed to test the system.

Supporting multiple parallel programming paradigms is in demand but the theory

and technology are not ready to be widely accepted and used. SilkRoad shows a practical approach in this field, but it is far from complete and mature. Meanwhile, the current implementation can be further optimized to achieve better performance.

Bibliography

[1] G. A. Abandah and E. S. Davidson. Characterizing Distributed Shared Memroy Performance: A Case Study of the Convex SPP1000. *IEEE Transactions on Parallel and Distributed Systems*, 9(2):206–216, Feb. 1998.

[2] Ada9X Project. Ada9x Requirements, 1990. Office of the Under Secretary of Defense for Acquisition, Washington, D.C.

[3] S. V. Adve and K. Gharachorloo. Shared memory consistency models: A tutorial. In *Rice University ECE Technical Report 9512*, 1995.

[4] S. V. Adve and M. D. Hill. Weak ordering- new definition. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, pages 2–14, May 1990.

[5] A. Agarwal, G. D'Souza, K. Johnson, D. Kranz, J. Kubiatowicz, K. oshi Kurihara, B.-H. Lim, G. Maa, D. Nussbaum, M. Parkin, and D. Yeung. The MIT Alewife machine : A large-scale distributed-memory multiprocessor. In *Proceedings of Workshop on Scalable Shared Memory Multiprocessors*. Kluwer Academic, 1991.

[6] S. Ahuja, N. Carriero, and D. Gelernter. Linda and friends. *IEEE Computer*, 19(8):26–34, Aug. 1986.

[7] A. Aiken and D. Gay. Barrier inference. In *Symposium on Principles of Programming Languages*, pages 342–354, 1998.

[8] C. Amza, A. Cox, K. Rajamani, and W. Zwaenepoel. Tradeoffs between false sharing and aggregation in software distributed shared memory. In *ACM Symposium on Princple and Practics of Parallel Programming (PPoPP)*, pages 90–99, 1997.

[9] C. Amza, A. L. Cox, S. Dwarkadas, K. Rajamani, and W. Zwaenepoel. Adaptive protocols for software distributed shared memory. *Proceedings of the IEEE*, 87(3):467–475, Mar. 1999.

[10] T. E. Anderson, D. E. Culler, and D. A. Patterson. A Case for Networks of Workstations: NOW. *IEEE Micro*, feb 1995.

[11] D. Bailey, J. Barton, T. Lasinski, and H. Simon. The NAS parallel benchmarks. In *Report RNR-91002 Revision 2. Moffett Field, Calif.: Numerical Aerodynamic Simulation (NAS) systems Division, NASA Ames Research Center*, 1991.

[12] H. Bal, M. Kaashoek, and A. Tanenbaum. Experience with Distributed Programming in Orca. In *IEEE CS Int. Conf. on Computer Languages*, pages 79–89, Mar. 1990.

[13] T. Ball and S. Horwitz. Constructing control flow from data dependence. In *Technical report, University of Wisconsin-Madison, TR No. 1091*, 1992.

[14] S. Balsamo, L. Donatiello, and N. M. V. Dijk. Bound performance models of heterogeneous parallel processing systems. *IEEE Transactions on Parallel and Distributed Systems*, 9(10):1041–1056, Oct. 1998.

[15] A. Barak and O. La'adan. The MOSIX Multicomputer Operating System for High Performance Cluster Computing, 1998.

[16] M. Bari, L. Jaffe, S. Zur, A. Itzkovich, and A. Schuster. Cparpar-a natural parallel extention of C++. *Technion's laboratory for distributed-parallel computing internal document*, 1996.

[17] Y. Ben-Asher, D. G. Feitelson, and L. Rudolph. ParC: An Extension of C for Shared Memory Parallel Processing. *Software Parctice and Experience*, 26(5):581–612, May 1996.

[18] P. Berenbrink, T. Friedetzky, and A. Steger. Randomized and adversarial load balancing. In *Proceedings of the 11th Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA)*, pages 175–184, 1999.

[19] B. Bershad. Shared memory parallel programming with entry consistency for distributed memory multiprocessors. In *CMU Technical Report CMU-CS-91-170*, 1991.

[20] B. N. Bershad, M. J. Zekauskas, and W. A. Sawdon. The midway distributed shared memory system. In *Digest of Papers from the 38th IEEE Computer Society International Conference (Spring COMPCON)*, pages 528–537, Feb. 1993.

[21] B. N. Bershard and M. J. Zekauskas. Midway: Shared Memory Parallel Programming with Entry Consistency for Distributed Memory Multiprocessors. In *Tech. Report CMU-CS-91170*, 1991.

[22] A. Bilas. *Improving the Performance of Shared Virtual Memory on System Area Networks*. PhD thesis, Department of Computer Science, Princeton University, Nov. 1998.

[23] A. Bilas and E. W. Felten. Fast RPC on the SHRIMP Virtual Memory Mapped Network Interface. *Journal of Parallel and Distributed Computing. Special Issue on Workstation Cluster and Network-bas ed Computing*, Feb. 1997.

[24] A. Bilas, D. Jiang, Y. Zhou, and J. P. Singh. Limits to the performance of software shared memory: A layered approach. In *Proceedings of fifth International Symposium on High Performance Computer Architecture*, pages 193–202, 1999.

[25] G. E. Blelloch. Programming parallel algorithms. In D. B. Johnson, F. Makedon, and P. Metaxas, editors, *Proceedings of the Dartmouth Institute for Advanced Graduate Study in Parallel Computation Symposium*, pages 11–18, 1992.

[26] G. E. Blelloch, P. B. Gibbons, and Y. Matias. Provably efficient scheduling for languages with fine-grained parallelism. *Journal of the ACM*, 46(2):281–321, 1999.

[27] G. E. Blelloch, P. B. Gibbons, Y. Matias, and G. J. Narlikar. Space-efficient scheduling of parallism with synchronization variables. In *Proceedings of the 9th Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA)*, pages 12–23, June 1997.

[28] G. E. Blelloch and J. Greiner. A provable time and space efficient implementation of NESL. *ACM SIGPLAN Notices*, 31(6):213–225, 1996.

[29] J. Blieberger, B. Burgstaller, and B. Scholz. Symbolic Data Flow Analysis for Detecting Deadlocks in Ada Tasking Programs. In *Ada-Europe*, pages 225–237, 2000.

[30] W. Blume and R. Eigenmann. Performance Analysis of Parallelizing Compilers on the Perfect Benchmarks Programs. *IEEE Trans. on Parallel and Distributed Systems*, 3(6):643–656, 1992.

[31] R. D. Blumofe. *Executing Multithreaded Programs Efficiently*. PhD thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, Sept. 1995.

[32] R. D. Blumofe, M. Frigo, C. F. Joerg, C. E. Leiserson, and K. H. Randall. An Analysis of Dag-Consistent Distributed Shared Memory Algorithms. In *Proceedings of the 8th Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA)*, Padua, Italy, June 1996.

[33] R. D. Blumofe, M. Frigo, C. F. Joerg, C. E. Leiserson, and K. H. Randall. Dag-consistent distributed shared memory. In *Proceedings of the 10th International Parallel Processing Symposium (IPPS)*, pages 132–141, Honolulu, Hawaii, Apr. 1996.

[34] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou. Cilk: An efficient multithreaded runtime system. In *Proceedings of the ACM SIGPLAN'95 Symposium on Principles and Practice of Parallel Programming (PPoPP)*, Santa Barbara, CA, July 1995.

[35] R. D. Blumofe and C. E. Leiserson. Space-efficient scheduling of multithreaded computations. In *Proceedings of the Trenty-Fifth Annual ACM Symposium on the Theory of Computing (STOC'93)*, May 1993.

[36] R. D. Blumofe and C. E. Leiserson. Scheduling multithreaded computations by work stealing. In *Proceedings of the 35th Annual Symposium on Foundations of Computer Science (FOCS)*, pages 356–368, Santa Fe, New Mexico, Nov. 1994.

[37] R. D. Blumofe and C. E. Leiserson. Scheduling multithreaded computations by work stealing. *Journal of the ACM*, 46(5):720–748, Sept. 1999.

[38] L. Brunie and L. Lefevre. DOSMOS : A Distributed Shared Memory based on PVM. In *First european PVM users group meeting*, Oct. 1994.

[39] R. Buyya. *High Performance Cluster Computing:programming and applications. Volume 2*. Prentice Hall, 1999.

[40] N. Carriero and D. Gelernter. Linda in context. *Communication of the ACM*, 32(4):444–458, Apr. 1989.

[41] J. B. Carter, J. K. Bennet, and W.Zwaenepoel. Implementation and Performance of Munin. In *Proc. of the 13th ACM Symposium on Operating System Principles*, pages 152–164, Oct. 1991.

[42] M. Chandy, I. Foster, K. Kennedy, C. Koelbel, and C.-W. Tseng. Integrated support for task and data parallelism. *The International Journal of Supercomputer Applications*, 8(2):80–98, Summer 1994.

[43] Y.-K. Chong and K. Hwang. Evaluation of relaxed memory consistency models for multithreaded multiprocessors. 1994.

[44] Cilk-5.2 Reference Manual. Available on the website `http://supertech.lcs.mit.edu/cilk`.

[45] A. L. Cox, E. de Lara, Y. C. Hu, and W. Zwaenepoel. A performance comparison of homeless and home-based lazy release consistency protocols in software shared memory.

In *Proceedings of the fifth High Performance Computer Architecture Conference*, Jan. 1999.

[46] R. Cytron, M. Hind, and W. Hsieh. Automatic generation of DAG parallelism. In *Proceedings of the ACM SIGPLAN '89 Conference on Programming Language Design and Implemen tation*, volume 24, pages 54–68, Portland, OR, June 1989.

[47] S. Darbha and D. P. Agrawal. Optimal scheduling algorithm for distributed-memory machines. *IEEE Transactions on Parallel and Distributed Systems*, 9(1):87–95, Jan. 1998.

[48] A. Darte, Y. Robert, and F. Vivien. *Scheduling and Automatic Parallelization*. Birkhauser, 2000.

[49] E. de. Lara, Y. C. Hu, H. Lu, A. L. Cox, and W. Zwaenepoel. The effect of contention on the scalability of page-based software shared memroy systems. In *Proceedings of Languages, Compilers, and Runtimes for Scalable Computing*, May 2000.

[50] Distributed Cilk - Release 5.1 alpha 1. Available on the website `http://supertech.lcs.mit.edu/cilk/release/distcilk5.1.html`.

[51] M. Dubois, C. Scheurich, and F. Briggs. Memory access buffering in multiprocessors. In *Proceedings of the 13th Annual International Symposium on Computer Architecture (ISCA)*, pages 434–442, June 1986.

[52] S. Eggers and R. Katz. Evaluation of the performance of four snooping cache coherency protocols. In *Proceedings of the 16th International Symposium on Computer Architecture*, pages 2–15, 1989.

[53] T. V. Eicken, D. E. Culler, S. C. Goldstein, and K. E. Schauser. Active messages: a mechanism for integrated communication and computation. In *Proceedings of the 19th Annual International Symposium on Computer Architecture (ISCA)*, pages 256–266, Gold Coast, Australia, May 1992.

[54] A. Erlichson, N. Nuckolls, G. Chesson, and J. Hennessy. Softflash: Analyzing the performance of clustered distributed virtual shared memory. In *ACM Symposium on Architectural Support for Programming Languages and Operating Systems*, pages 210–220, 1996.

[55] J. T. Feo. *A Comparative Study of Parallel Programming Languages: The Salishan Problems*. Elsevier Science Publishers, Holland, 1992.

[56] J. Ferrante, K. J. Ottenstein, and J. D. Warren. The program dependence graph and its use in optimization. *ACM Transactions on Programming Languages and Systems*, (9(3)):319–349, 1987.

[57] A. L. O. Filho and V. C. Barbosa. A Graph-Theoretic Model of Shared-Memory Legality. Technical Report ES-531/00, Apr. 2000. UFRJ technical report, Rio de Janeiro, Brazil.

[58] B. D. Fleisch, N. C. Juul, and R. L. Hyde. Mirage+: A kernel implementation of distributed shared memory for a network of personal computers. *Software Practice and Experience*, 24(10), Oct. 1994.

[59] I. Foster. *Designing and Building Parallel Programs*. Addison Wesley, 1996. available at http://www.mcs.anl.gov/dbpp.

[60] M. Frigo. The weakest resonable memory model. Master's thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, 1998.

[61] M. Frigo. *Portable High-Performance Programs*. PhD thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, June 1999.

[62] M. Frigo and V. Luchangco. Computation-centric memory models. In *Proceedings of the 10th Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA)*, June 1998.

[63] M. Frigo, K. H. Randall, and C. E. Leiserson. The implementation of the Cilk-5 multithreaded language. In *Proceedings of the ACM SIGPLAN '98 Conference on Programming Language Design and Implementation (PLDI)*, Montreal, Canada, June 1998.

[64] G. R. Gao and V. Sarkar. Stepping beyong memory coherence barrier. In *Proceedings of the 1995 International Conference on Parallel Processing (ICPP'95)*, pages 73–76, Aug. 1995.

[65] A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Manchek, and V. Sunderam. *PVM: Parallel Virtual Machine*. MIT Press, 1994. PVM homepage `http://www.epm.ornl.gov/pvm/pvm_home.html`.

[66] K. Gharachorloo, D.E.Lenoski, J.Laudon, P.Gibbons, A.Gupta, and J.L.Hennessy. Memory consistency an event ordering in scalable shared-memory multiprocessors. In *Proc. of the 17th Annual Int'l Symp. on Computer Architecture (ISCA'90)*, pages 15–26, May 1990.

[67] P. B. Gibbons. What good are shared-memory models? In *International Conference on Parallel Processing Workshop*, pages 103–114, 1996.

[68] J. R. Goodman. Cache consistency and sequential consistency. In *Technical Report No. 61, SCI Committee*, 1989.

[69] J. Gosling, B. Joy, and G. Steele. *The Java Language Specification*. Addison-Wesley, 1996.

[70] W. Gropp, S. Huss-Lederman, A. Lumsdaine, E. Lusk, B. Nitzberg, W. Saphir, and M. Snir. *MPI: The Complete Reference. Volume 2 – The MPI-2 Extensions*. MIT Press, 1998. MPI forum homepage `http://www.mpi-forum.org`.

[71] F. Hamelin, J. Jezequel, and T. Priol. A multi-paradigm object oriented parallel evironment. In *Proceedings of the 8th International Parallel Processing Symposium*, pages 182–186, 1994.

[72] P. B. Hansen. Model programs for computational science: A programming methodology for multicomputers. *Currency: practice and experience*, (5(5)):407–427, 1993.

[73] A. Hey, D. Pritchard, and C. Whitby-Strevens. Multi-paradigm parallel programming. In *Proceedings of the Twenty-Second Annual Hawaii International Conference on System Sciences*, pages 716–725, 1989.

[74] W. Hu, W. Shi, and Z. Tang. JIAJIA: An SVM System Based on A New Cache Coherence Protocol. In *In Proceedings of the High Performance Computing and Networking (HPCN'99), LNCS 1593*, pages 463–472, Apr. 1999.

[75] K. Hwang and Z. Xu. *Scalable Parallel Computing*. McGraw-Hill, 1998.

[76] IEEE. Information technology–Portable Operating System Interface (POSIX)-Part1: System Application: Program Interface (API) [C Language], 1996. ANSI/IEEE Std 1003.1, 1996 Edition.

[77] L. Iftode and J. P. Singh. Shared virtual memory: Progress and challenges. In *Proceedings of the IEEE*, pages 498–507, Mar. 1999.

[78] L. Iftode, J. P. Singh, and K. Li. Scope consistency: A bridge between release consistency and entry consistency. In *Proceedings of the 8th Annual ACM Symposium on Parallel Algorithms and Architecture (SPAA)*, pages 277–287, June 1996.

[79] Inmos. Programming in Occam 2, 1988. Prentice Hall.

[80] A. Itzkovitz and A. Schuster. MultiView and Millipage – Fine-Grain Sharing in Page Based DSMs. In *3rd Symposium on Operating Systems Design and Implementation (OSDI '99)*, Feb. 1999.

[81] A. Itzkovitz, A. Schuster, and L. Shalev. Supporting Multiple Parallel Programming Paradigms on Top of the Millipede Virtual Parallel Machine. In *Proceedings of the Second International Workshop on High Level Programming Models and Supportive Environments (HIPS '97)*, pages 25–34, Apr. 1997.

[82] A. Itzkovitz, A. Schuster, and L. Shalev. Thread migration and its application in distributed shared memory systems. *The Journal of Systems and Software*, 42:71–87, 1998.

[83] K. L. Johnson, M. F. Kaashoek, and D. A. Wallach. CRL: High-Performance All-Software Distributed Shared Memory. In *the Fifteenth Symposium on Operating Systems Principles*, Dec. 1995.

[84] P. Keleher. *Lazy Release Consistency for Distributed Shared Memory*. PhD thesis, Department of Electrical Engineering and Computer Science, Rice University, Jan. 1995.

[85] P. Keleher, A. L. Cox, S. Dwarkadas, and W. Zwaenepoel. TreadMarks: Distributed shared memory on standard workstations and operating systems. In *USENIX Winter 1994 Conference Proceedings*, pages 115–132, San Francisco, Ca, Jan. 1994.

[86] P. Keleher, A. L. Cox, and W. Zwaenepoel. Lazy release consistency for software distributed shared memory. In *Proc. of the 19th Anaual International Symposium on Computer Architecture (ISCA'92)*, pages 13–21, May 1992.

[87] J.-H. Kim and N. H. Vaidya. A cost model for distributed shared memory using competitive update. 1997.

[88] C. H. Koelbel, D. B. Loveman, R. S. Schreiber, J. Guy L. Steele, and M. E. Zosel. *The High Performance Fortran Handbook*. MIT Press, 1994.

[89] A. Kumar and R. Shorey. Performance analysis and scheduling of stochastic fork-join jobs in a multicomputer system. *IEEE Transactions on Parallel and Distributed Systems*, 4(10):1147–1164, Oct. 1993.

[90] L. Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs? In *IEEE Transactions on Computers*, pages 690–691, Sept. 1979.

[91] D. Lea. *Concurrent Programming in Java*. Addison-Wesley, 1996.

[92] J. Leichtl, P. E. Crandall, and M. J. Clement. Parallel programming in multi-paradigm clusters. In *Sixth IEEE International Symposium on High Performance Distributed Computing*, pages 326–335, 1997.

[93] K. Li. IVY: A shared virtual memory system for parallel computing. In *Proceedings of the International Conference on Parallel Computing (ICPP)*, pages 94–101, Aug. 1988.

[94] B.-H. Lim, C.-C. Chang, G. Czajkowski, and T. von Eicken. Performance implications of communication mechanisms in all-software global address space systems. In *ACM Symposium on Princple and Practics of Parallel Programming (PPoPP)*, pages 230–239, 1997.

[95] C. Lindemann and F. Schon. Performance evaluation of consistency models for multi-computers with virtually shared memory. 1993.

[96] P. Lu. Implementing Scoped Behaviour for Flexible Distributed Data Sharing. *IEEE Concurrency*, 8(3):63–73, July–September 2000. Available at http://www.cs.ualberta.ca/˜paullu/.

[97] J. C. Lui, R. R. Muntz, and D. Towsley. Computing performance bounds of fork-join parallel programs under a multiprocessing environment. *IEEE Transactions on Parallel and Distributed Systems*, 9(3):295–311, Mar. 1998.

[98] D. Marinov, D. Magdic, A. Milenkovic, J. Protic, I. Tartalja, and V. Milutinovic. An Approach to Characterization of Parallel Applications for DSM Systems. In *Proceedings of the 31st Annual Hawaii International Conference on System Sciences (HICSS)*, pages 782–783, 1998.

[99] G. J. Narlikar. Scheduling threads for low space requirement and good locality. In *Proceedings of the 11th Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA)*, pages 83–95, 1999.

[100] R. Nelson and A. N. Tantawi. Approximate analysis of fork/join synchronization in parallel queues. *IEEE Transactions on Computers*, 37(6):739–743, June 1988.

[101] M. C. Ng and W. F. Wong. ORION: An Adaptive Home-Based Software Distributed Shared Memory System. In *Proc. of 2000 International Conference on Parallel and Distributed Systems (ICPADS 2000)*, pages 187–194, 2000.

[102] R. V. V. Nieuwpoort, T. Kielmann, and H. E. Bal. Efficient load balancing for wide-area divide-and-conquer applications. In *ACM Symposium on Principles and Practice of Parallel Programming (PPoPP)*, pages 34–43, June 2001.

[103] B. Nitzberg and V. Lo. Distributed shared memory: A survey of issues and algorithms. *IEEE Computer*, pages 52–60, Aug. 1991.

[104] OpenMP Architecture Review Board. OpenMP C and C++ application program interface, 1998. Available on the Internet from `http://www.openmp.org/`.

[105] L. Peng, W. F. Wong, M. D. Feng, and C. K. Yuen. SilkRoad: A Multithreaded Runtime System with Software Distributed Shared Memory for SMP Clusters. In *Proc. of the 2nd IEEE International Conference on Cluster Computing (CLUSTER2000)*, Nov. 2000.

[106] L. Peng, W. F. Wong, and C. K. Yuen. SilkRoad II: Mixed Paradigm Cluster Computing with RC_dag Consistency. *Journal of Parallel Computing*. Accepted.

[107] L. Peng, W. F. Wong, and C. K. Yuen. SilkRoad II: A Multiple-Paradigm Runtime System for Cluster Computing. In *Proc. of the 4th IEEE International Conference on Cluster Computing (CLUSTER2002)*, Chicago, U.S.A, Sept. 2002.

[108] G. F. Pfister. *In search of clusters, Edition 2nd ed.* Imprint Upper Saddle River, NJ : Prentice Hall PTR, 1998.

[109] J. Protic, M. Tomasevic, and V. Milutinovic. A survey of distributed shared memory systems. In *Proceedings of the 28th IEEE/ACM Hawaii International Conference on System Sciences (HICSS)*, pages 74–84, Jan. 1995.

[110] J. Protic, M. Tomasevic, and V. Milutinovic. *Distributed Shared Memory: concepts and systems*. IEEE Computer Society, 1997.

[111] F. Rabhi. A parallel programming methodology based on paradigms. In *Proceedings of WoTUG-18: Transputer and occam Developments*, volume 44, pages 239–251, Amsterdam, 1995. IOS Press.

[112] K. H. Randall. *Cilk: Efficient Multithreaded Computing*. PhD thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, May 1998. Available as MIT Technical Report MIT/LCS/TR-749.

[113] J. Rehg, K. Knobe, U. Ramachandran, R. S. Nikhil, and A. Chauhan. Integrated task and data parallel support for dynamic applications. In *Languages, Compilers, and Run-Time Systems for Scalable Computers*, pages 167–180, 1998.

[114] M. C. Rinard, D. J. Scales, and M. S. Lam. Jade: A High-Level, Machine-Independent Language for Parallel Programming. *IEEE Computer*, 26:28–38, June 1993.

[115] R. Rugina and M. Rinard. Automatic parallelization of divide and conquer algorithms. In *ACM Symposium on Princple and Practics of Parallel Programming (PPoPP)*, pages 72–83, 1999.

[116] V. Sarkar. Analysis and optimization of explicitly parallel programs using the parallel program graph repres entation. In *Languages and Compilers for Parallel Computing*, pages 94–113, 1997.

[117] V. Sarkar and B. Simons. Parallel program graph and their classification. In *LNCS 768 Languages and Compilers for Parallel Computing*, pages 633–655, 1993.

[118] M. L. Scott, T. J. LeBlanc, and B. D. Marsh. Multi-model parallel programming in Psyche. In *Proc. 2nd Annual ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming*, pages 70–78, Seattle, WA (USA), 1990.

[119] Y. J. Soo, M. D. Feng, and C. K. Yuen. Parallel C Programming System on Cluster of Workstations with Process Migration. In *Proceedings of the Eleventh IASTED International Conference on Parallel and Distributed Computing and Systems (PDCS'99)*, Boston, USA, Nov. 1999.

[120] S. Srbljic and L. Budin. Analytical performance evaluation of data replication based shared memory model. In *Proceedings of 2nd International Symposium on High Performance Distributed Commputing*, pages 326–335, 1999.

[121] S. Srbljic, Z. G. Vranesic, and L. Budin. Performance prediction for different consistency schemes in distributed shared memory systems. 1994.

[122] T. Sterling, D. Savarese, D. J. Becker, J. E. Dorband, U. A. Ranawake, and C. V. P. er. BEOWULF: A parallel workstation for scientific computation. In *Proceedings of the 24th International Conference on Parallel Processing*, pages I:11–14, Oconomowoc, WI, 1995.

[123] E. Stoltz, H. Srinivasan, J. Hook, and M. Wolfe. Static single assignment form for explicitly parallel programs: Theory and practice. In *Tech. report, Dept. of Computer Science and Engineering, Oregon Graduate Institute of Science a nd Technology, Portland, Oregon*, 1994.

[124] E. Stoltz and M. Wolfe. Sparse data-flow analysis for dag parallel programs. 1994. http://citeseer.nj.nec.com/stoltz94sparse.html.

[125] M. Stumm and S. Zhou. Algorithms implementing distributed shared memory. *IEEE Computer*, pages 54–64, May 1990.

[126] The MPI Forum. MPI: A message passing interface. In *Supercomputing '93*, pages 878–883, Portland, Oregon, Nov. 1993. MPI homepage `http://www.mpi-forum.org`.

[127] Thinking Machines Co. C^* Reference Manual. Version 4.3., 1988. Thinking Machines Corporation, Cambridge, MA.

[128] K. Thitikamol and P. Keleher. Multi-threading and remote latency in software DSMs (award paper). In *The 17th International Conference on Distributed Computing Systems(ICDCS)*, May 1997.

[129] K. Thitikamol and P. Keleher. Thread Migration and Communication Minimization in DSM Systems. *Proceedings of the IEEE*, 87(3):487–497, Mar. 1999.

[130] J. Thornley. Integrating functional and imperative parallel programming: CC++ solutions to the salishan problems. In *Proceedings of the 1994 International Parallel Processing Symposium (IPPS'94)*, 1994.

[131] M. Thornton and D. Andrews. Graph analysis and transformation techniques for runtime minimization in multithreaded architectures. In *Proc. of the 30th Hawaii International Conference on System Sciences*, 1996.

[132] TreadMarks User Manual. Available in TreadMarks' distribution.

[133] A. Waheed and J. Yan. Performance modeling and measurement of parallelized code for distributed shared memory multiprocessors. In *Proceedings of Sixth International Symposium on Modeling, Analysis and Simulation of computer and Telecommunication Systems*, pages 161–166, 1998.

[134] Z. Xu and K. Hwang. Coherent Parallel Programming in C//. In *Proceedings of International Conference on Advances in Parallel and Distributed Computing*, pages 116–122, Mar. 1997.

[135] Z. Xu, J. R. Larus, and B. P. Miller. Shared-memory performance profiling. In *ACM Symposium on Princple and Practics of Parallel Programming (PPoPP)*, pages 240–251, 1997.

[136] T. Yang and C. Fu. Space/time-efficient scheduling and execution of parallel irregular computations. *ACM Transactions on Programming Languages and Systems*, 20(6):1195–1222, Nov. 1998.

[137] D. Yeung. The scalability of multigrain systems. In *13th Annual International Conference on Supercomputing*, June 1999.

[138] D. Yeung, J.Kubiatowicz, and A.Agarwal. Mgs: A multigrain shared memory system. In *23th Annual Symposium on Computer Architecture*, pages 44–55, May 1996.

[139] C. K. Yuen. BaLinda Lisp: Realization of a pragmatic parallel programming model. In *Proceedings of ACM Japan Chapter Inaugural Conference*, pages 253–260, Mar. 1994.

[140] C. K. Yuen. Function families and reflective active objects in BaLinda K. *Journal of High Performance Computing*, 3(1):3–6, 1996.

[141] M. J. Zekauskas, W. A. Sawdon, and B. N. Bershad. Software write detection for a distributed shared memory. In *Proceedings of the First Symposium on Operating Systems Design and Implementation (OSDI)*, 1994.