

TEL-AVIV UNIVERSITY  
RAYMOND AND BEVERLY SACKLER FACULTY OF EXACT  
SCIENCES  
SCHOOL OF COMPUTER SCIENCE

**Designing Communication-Efficient  
Matrix Algorithms in  
Distributed-Memory Cilk**

Thesis submitted in partial fulfillment of the requirements for the M.Sc.  
degree of Tel-Aviv University by

Eyal Baruch

The research work for this thesis has been carried out at Tel-Aviv  
University under the direction of Dr. Sivan Toledo

November 2001



## Abstract

This thesis studies the relationship between parallelism, space and communication in dense matrix algorithms. We study existing matrix multiplication algorithms, specifically those that are designed for shared-memory multiprocessor machines (SMP's). These machines are rapidly becoming commodity in the computer industry, but exploiting their computing power remains difficult. We improve algorithms that originally were designed using an algorithmic multithreaded language called Cilk (pronounced silk), and we present new algorithms. We analyze the algorithms under Cilk's dag-consistent memory model. We show that by dividing the matrix-multiplication process into phases that are performed in a sequence, we can obtain lower communication bound without significantly limiting parallelism and without consuming significantly more space. Our new algorithms are inspired by distributed-memory matrix algorithms. In particular, we have developed algorithms that mimic the so-called two-dimensional and three-dimensional matrix multiplication algorithms, which are typically implemented using message-passing mechanisms, not using share-memory programming. We focus on three key matrix algorithms: matrix multiplication, solution of triangular linear systems of equations, and the factorization of matrices into triangular factors.

## Contents

Abstract	3
Chapter 1. Introduction	5
1.1. Two New Matrix Multiplication Algorithms	6
1.2. New Triangular Solver and LU Algorithms	7
1.3. Outline of The Thesis	7
Chapter 2. Background	8
2.1. Parallel Matrix Multiplication Algorithms	8
2.2. The Cilk Language	9
2.3. The Cilk Work Stealing Scheduler	12
2.4. Cilk's Memory Consistency Model	12
2.5. The BACKER Coherence Algorithm	14
2.6. A Model of Multithreaded Computation	15
Chapter 3. Communication-Efficient Dense Matrix Multiplication in Cilk	18
3.1. Space-Efficient Parallel Matrix Multiplication	18
3.2. Trading Space for Communication in Parallel Matrix Multiplication	23
3.3. A Comparison of Message-Passing and Cilk Matrix-Multiplication Algorithms	27
Chapter 4. A Communication-Efficient Triangular Solver in Cilk	29
4.1. Triangular Solvers in Cilk	29
4.2. Auxiliary Routines	31
4.3. Dynamic Cache-Size Control	34
4.4. Analysis of the New Solver with Dynamic Cache-Size Control	35
Chapter 5. LU Decomposition	39
Chapter 6. Conclusion and Open Problems	42
Bibliography	43

## CHAPTER 1

# Introduction

The purpose of parallel processing is to perform computations faster than can be done with a single processor by using a number of processors concurrently. The need for faster solutions and for solving large problems arises in wide variety of applications. These include fluid dynamics, weather prediction, image processing, artificial intelligence and automated manufacturing.

Parallel computers can be classified according to variety of architectural features and modes of operations. In particular, most of the existing machines may be broadly grouped into two classes: machines with shared-memory architectures (examples include most of the small multiprocessors in the market, such as Pentium-based servers, and several large multiprocessors, such as the SGI Origin 2000) and machines with distributed-memory architecture (examples include the IBM SP systems and clusters of workstations and servers). In a shared-memory architecture, processors communicate by reading from and writing into the shared memory. In distributed-memory architectures, processors communicate by sending messages to each other.

This thesis focuses on the efficiency of parallel programs that run under the Cilk programming environment. Cilk is a parallel programming system that offers the programmer a shared-memory abstraction on top a distributed memory hardware. Cilk includes a compiler for its programming language, which is also referred to as Cilk, and a run-time system consisting of a scheduler and a memory consistency protocol. (The memory consistency protocol, which this thesis focuses on, is only part of one version of Cilk; the other versions assume a shared-memory hardware.)

The Cilk parallel multithreaded language has been developed in order to make high-performance parallel shared-memory programming easier. Cilk is built around a provably efficient algorithm for scheduling the execution of fully strict multithreaded computations, based on the technique of work stealing [21][4][26][22][5]. In his PhD thesis [21], Randall developed a memory-consistency for running Cilk programs on distributed-memory parallel computers and clusters. His protocol allows the algorithm designer to analyze the amount of communication in a Cilk program and the impact of this communication on the total running time of the program. The analytical tools that he developed, along with earlier tools, also allows the designer to estimate the space requirements of a program. Randall demonstrated the power of these results by implementing and analyzing several algorithms, including matrix multiplication and LU factorization algorithms.

However, the communication bounds of Randall's algorithms are quite loose compared to known distributed-memory message-passing algorithms.

This is alarming, since extensive communication between processors may significantly slow down parallel computations even if the work and communication is equally distributed between processors.

In this thesis we show that it is possible to tighten the communication bound with respect to the cache size using new Cilk algorithms that we have designed. We demonstrate new algorithms for matrix multiplication, for solution of triangular linear systems of equations, and for the factorization of matrices into triangular factors.

By the term *Cilk algorithms* we essentially mean Cilk implementation of conventional matrix algorithms. Programming languages allow the programmer to specify a computation (how to compute intermediate and final results from previously-computed results). But most programming languages also force the designer to constrain the schedule of the computation. For example, a C program essentially specifies a complete ordering of the primitive operations. The compiler may change the order of computations only if it can prove that the new ordering produces equivalent results. Parallel message-passing programs fully specify the schedule of the parallel computation. Cilk programs, in contrast, declare that some computations may be performed in parallel but let a run-time scheduler decide on the exact schedule. Our analysis, as well as previous analyses of Cilk programs, essentially show that a given program admits an efficient schedule and that Cilk's run-time scheduler is indeed likely to choose such a schedule.

### 1.1. Two New Matrix Multiplication Algorithms

The main contribution of this thesis is in presenting a new approach for designing algorithms implemented in Cilk for achieving lower communication bound. In the distributed-memory application world there exists a traditional classification of matrix multiplication algorithms. So-called two-dimensional (2D) algorithms, such as those of Cannon [7], or of Ho, Johnson and Edelman [18], use only a little amount of extra memory. Three-dimensional (3D) algorithms use more memory but perform asymptotically less communication; examples include the algorithms of Gupta and Sadayappan [14], of Berntsen [3], of Dekel, Nassimi and Sahni [8] and of Fox, Otto and Hey [10].

Cilk's shared-memory abstraction, in comparison to message-passing mechanisms, simplifies programming by allowing of each procedure, no matter which processor runs it, to access the entire memory space of the program. The Cilk runtime system provides support for scheduling decisions and the programmer needs not specify which processor executes which procedure, nor exactly when each procedure should be executed. These factors make it substantially easier to develop parallel programs using Cilk than with other parallel-programming environments. One may suspect that the ease-of-programming comes at a cost: reduced performance. We show in this thesis that this is not necessarily the case, at least theoretically (up to logarithmic factors), but that careful programming is required in order to match existing bounds. More naive implementations of algorithms, including those proposed by Randall, do indeed suffer from relatively poor theoretical performance bounds.

We give tighter communication bounds for new Cilk matrix multiplication algorithms that can be classified as 2D and 3D algorithms and prove that it is possible to design such algorithms with the simple programming environment of Cilk almost without compromising on performance. In the 3D case we have even slightly improved parallelism. The analysis shows we can implement a 2D-like algorithm for multiplying  $n \times n$  matrices on a machine with  $P$  processors, each with  $\frac{n^2}{P}$  memory, with communication bound  $O(\sqrt{P}n^2 \log n)$ . In comparison, Randall's `notempul` algorithm, which is equivalent in the sense that it uses little space beyond the space required for the input and output, performs  $O(n^3)$  communication. We also present a 3D-like algorithm with communication bound  $O(\frac{n^3}{\sqrt{C}} + CP \log \frac{n}{\sqrt{C}} \log n)$ , where  $C$  is the memory size of each processor, which is lower than existing Cilk implementations for any amount of memory per processor.

### 1.2. New Triangular Solver and LU Algorithms

Solving a linear system of equations is one of the most fundamental problems in numerical linear algebra. The classic Gaussian elimination scheme to solve an arbitrary linear system of equations reduces the given system to a triangular form and then generates the solution by using the standard forward and backward substitution algorithm. This essentially factors the coefficient matrix into two triangular factors, one lower triangular and the other upper triangular. The contribution of this thesis is showing that if we can dynamically control the amount of memory that processors use to cache data locally, then we design communication efficient algorithms for solving dense linear systems. In other words, to achieve low communication bounds, we limit the amount of data that a processor may cache during certain phases of the algorithm. Our algorithms perform asymptotically a factor of  $\frac{\sqrt{C}}{\log(n\sqrt{C})}$  less communication than Randall's (where  $\sqrt{C} > \log(n\sqrt{C})$ ), but our algorithms have somewhat less parallelism.

### 1.3. Outline of The Thesis

The rest of the thesis is organized as follows. In Chapter 2 we present an overview of existing parallel linear-algebra algorithms together, and we present Cilk, an algorithmic multithreaded language. Chapter 2 also introduces the tools that Randall and others have developed for analysing the performance of Cilk programs. In Chapter 3 we present new Cilk algorithms for parallel matrix multiplication and analyze our algorithms. In Chapter 4 we present a new triangular solver and demonstrate how controlling the size of the cache can reduce communication. In Chapter 5 we use the results concerning the triangular solver to design communication-efficient LU decomposition algorithm. We present our conclusions and discuss open problems in Chapter 6.

## CHAPTER 2

# Background

This chapter provides background material required in the rest of the thesis. The first section describes parallel distributed-memory matrix multiplication algorithms. Our new Cilk algorithms are inspired by these algorithms. The other sections describe Cilk and the analytical tools that allow us to analyze the performance of Cilk programs. Some of the material on Cilk follows quite closely the Cilk documentation and papers.

### 2.1. Parallel Matrix Multiplication Algorithms

The product  $R = AB$  is defined as  $r_{ij} = \sum_{k=1}^n a_{ik}b_{kj}$ , where  $n$  is the number of columns of  $A$  and rows in  $B$ . Implementing matrix multiplication according to the definition requires  $n^3$  multiplications and  $n^2(n-1)$  additions when the matrices are  $n$ -by- $n$ . In this thesis we ignore  $o(n^3)$  algorithms, such as Strassen's, which are not widely-used in practice. Matrix multiplication is a regular computation that parallelizes well.

The first issue when implementing such algorithms on parallel machines is how to assign tasks to processors. We can compute all the elements of the product in parallel, so we can clearly employ  $n^2$  processors for  $n$  time steps. We can actually compute all the products in a matrix multiplication computation in one step if we can use  $n^3$  processors. But to compute the  $n^2$  sums of product, we need additional  $\log n$  steps for the summations. Note that with  $n^3$  processors, most of them would remain idle during most of the time, since there are only  $2n^3 - 1$  arithmetic operations to perform during  $\log n + 1$  time steps.

Another issue when implementing parallel algorithms is the mechanisms used to support communication among different processors. In a distributed-memory architectures each processor has its own local memory which it can address directly and quickly. A processor may or may not be able to address the memory of other processors directly and in any case, accessing remote memories is slower than accessing its own local memory.

Programming a distributed-memory machine with message passing poses two challenges. The first challenge is a software engineering one, since the memory of the computer is distributed and since the running program is composed of multiple processes, each with its own variables, we must distribute data structures among the processors. The second and more fundamental challenge is to choose the assignment of data-structure elements and computational tasks to process in a way that minimizes communication, since transferring data between memories of different processors is much slower than accessing data in a processor own local memory, reducing data transfers usually reduces the running time of a program. Therefore, we

must analyze the amount of communication in matrix algorithms when we attempt to design efficient parallel algorithms and predict their performance.

There are two well known implementation concepts for parallel matrix multiplication on distributed-memory machines. The first and more natural implementation concept is to lay out the matrix in blocks. The  $P$  processors are arranged in a 2-dimensional  $\sqrt{P}$ -by- $\sqrt{P}$  grid (we assume for simplicity here that  $\sqrt{P}$  is an integer), split the three matrices into  $\sqrt{P}$ -by- $\sqrt{P}$  block matrices and store each block on the corresponding processor. The grid of processors is simply a map from 2 dimensional processor indices to the usual 1-dimensional rank (processor indexing). This form of distributing a matrix is called a 2-dimensional (2D) block distribution, because we distribute both the rows and the columns of the matrix among processors. The basic idea of the algorithm is to assign processor  $(i, j)$  the computation of  $R_{ij} = \sum_{k=1}^{\sqrt{P}} A_{ik}B_{kj}$  (here  $R_{ij}$  is a block of the matrix  $R$ , and similarly for  $A$  and  $B$ ). The algorithm consists of  $\sqrt{P}$  main phases. In each phase, every processor sends two messages of size  $\frac{n^2}{P}$  words (and receives two such messages as well), and performs  $2(\frac{n}{\sqrt{P}})^3$  floating point operations, resulting in  $\sqrt{P}\frac{n^2}{P} = \frac{n^2}{\sqrt{P}}$  communication cost and  $\frac{n^2}{P}$  memory space per processor.

A second kind of distributed-memory matrix multiplication algorithm uses less communication but more space than the 2D algorithm. The basic idea is to arrange the processors in a  $p$ -by- $p$ -by- $p$  3D grid, where  $p = P^{1/3}$ , and to split the matrices into  $p$ -by- $p$  block matrices. The first phase of the algorithm distributes the matrix so that processor  $(i, j, k)$  stores  $A_{ik}$  and  $B_{kj}$ . The next phase computes on processor  $(i, j, k)$  the product  $A_{ik}B_{kj}$ . In the third and last phase of the algorithm the processors sum up the products  $A_{ik}B_{kj}$  to produce  $R_{ij}$ . More specifically, the group of processors with indices  $(i, j, k)$  with  $k = 1..p$  sum up  $R_{ij}$ . The computational load in the 3D algorithm is nearly perfectly balanced. Each processor multiplies two blocks and adds at most two. Some processors add none.

The 2D algorithm requires each processor to store exactly 3 submatrices of order  $\frac{n}{\sqrt{P}}$  during the algorithm and performs total of  $P(\frac{n^2}{\sqrt{P}}) = n^2\sqrt{P}$  communication. The 3D algorithm stores at each processor 3 submatrices of order  $\frac{n}{P^{1/3}}$  and performs a total of  $P(\frac{n^2}{P^{2/3}}) = n^2P^{1/3}$  communication.

## 2.2. The Cilk 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 the 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.

Cilk's algorithmic multithreaded language for parallel programming generalizes the semantics of C by introducing linguistic constructs for parallel control. The basic Cilk language is simple. It consists of C with the addition of three keywords: *cilk*, *spawn* and *sync* to indicate parallelism and synchronization. A Cilk program, when run on one processor, has the same

semantics as the C program that results when the Cilk keywords are deleted. Cilk extends the semantics of C in a natural way for parallel execution so procedure may spawn subprocedures in parallel and synchronize upon their completion.

A Cilk procedure definition is identified by the keyword *cilk* and has an argument list and body just like a C function and its declaration can be used anywhere an ordinary C function declarations can be used. The main procedure must be named *main*, as in C, but unlike C, however, Cilk insists that the return type of *main* be *int*. Since the *main* procedure must also be Cilk procedure, it must be defined with the *cilk* keyword.

Most of the work in a Cilk procedure is executed serially, just like C, but parallelism is created when the invocation of a Cilk procedure is immediately preceded by the keyword *spawn*. A spawn is the parallel analog of a C function call, and like a C function call, when a Cilk procedure is spawned, execution proceeds to the child. Unlike a C function call, however, where the parent is not resumed until after its child returns, in the case of a Cilk spawn, the parent can continue to execute in parallel with the child. Indeed, the parent can continue to spawn off children, producing a high degree of parallelism. Cilk's scheduler takes the responsibility of scheduling the spawned procedures on the processors of the parallel computer.

A Cilk procedure can not safely use the return values (or data written to shared data structures) of the children it has spawned until it executes a *sync* statement. If all of its children have not completed when it executes a *sync*, the procedure suspends and does not resume until all of its children have completed. In Cilk, a *sync* waits only for the spawned children of the procedure to complete and not for all procedures currently executing. When all its children return, execution of the procedure resumes at the point immediately following the *sync* statement. As an aid to programmers, Cilk inserts an implicit *sync* before every return, if it is not present already. As a consequence, a procedure never terminates while it has outstanding children.

The program in Figure 2.2.1 demonstrates how Cilk works. The figure shows a Cilk procedure that computes the  $n$ -th Fibonacci number.

In Cilk's terminology, a **thread** is a maximal sequence of instructions that ends with a *spawn*, *sync* or *return* (either explicit or implicit) statement (the evaluation of arguments to these statements is considered part of the thread preceding the statement). Therefore, We can visualize a Cilk program execution as a directed acyclic graph, or **dag**, in which vertices are threads (instructions) and edges denote ordering constraints imposed by control statements. A Cilk program execution consists of a collection of procedures, each of which is broken into a sequence of non blocking threads. The first thread that executes when a procedure is called is the procedure **initial thread**, and the subsequent threads are **successor threads**. At runtime, the binary *spawn* relation causes procedure instances to be structured as a rooted tree, and the dependencies among their threads form a dag embedded in this *spawn*-tree. For example, the computation generated by the execution of `fib(4)` from the program in Figure 2.2.1 generates the dag shown in Figure 2.2.2. A correct execution of a Cilk program must obey all the dependencies in the dag, since a thread can not be executed until

```

cilk int fib(int n)
{
  if (n<2) return n;
  else {
    int x,y;
    x = spawn fib(n-1);
    y = spawn fib(n-2);
    sync;
    return (x+y);
  }
}

```

FIGURE 2.2.1. A simple Cilk procedure to compute the  $n$ th Fibonacci number in parallel (using an exponential work method while logarithmic-time methods are known). Deleting the *cilk*, *spawn*, and *sync* keywords would reduce the procedure to a valid and correct C procedure.

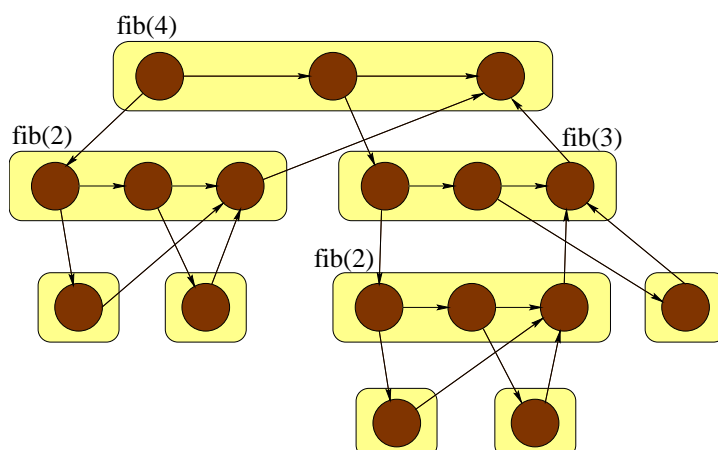


FIGURE 2.2.2. A dag of threads representing the multi-threaded computation of  $\text{fib}(4)$  from Figure 2.2.1. Each procedure, shown as a rounded rectangle, is broken into sequences of threads, shown as circles. A downward edge indicates the spawning of a subprocedure. A horizontal edge indicates the continuation to a successor thread. An upward edge indicates the returning of a value to a parent procedure. All three types of edges are dependencies which constrain the orders in which thread may be scheduled. The figure is from [22].

all the threads on which it depends have completed. Note that the use of the term **thread** here is different from the common use in programming environments such as Win32 or POSIX threads, where the same term refers to a process-like object that shares an address space with other threads and which competes with other threads and processes for CPU time.

### 2.3. The Cilk Work Stealing Scheduler

The *spawn* and *sync* keywords specify logical parallelism, as opposed to actual parallelism. That is, these keywords indicate which code may possibly execute in parallel but what actually runs in parallel is determined by the scheduler, which maps dynamically unfolding computations onto the available processors. To execute a Cilk program correctly, Cilk's underlying scheduler must obey all the dependencies in the dag, since a thread can not be executed until all the threads on which it depends have completed. These dependencies form a partial order, permitting many ways of scheduling the threads in the dag. A scheduling algorithm must ensure that enough threads remain concurrently active to keep the processors busy. Simultaneously, it should ensure that the number of concurrently active threads remains within reasonable limits so that memory requirements can be bounded. Moreover, the scheduler should also try to maintain related threads on the same processor, if possible, so that communication between them can be minimized. Needless to say, achieving all these goals simultaneously can be difficult.

Two scheduling paradigms address the problem of scheduling multi-threaded computations: work sharing and work stealing. In work sharing, whenever a processor generates new threads, the scheduler attempts to migrate some of them to other processors in hopes of distributing the work to under utilized processors. In work stealing, however, under utilized processors take the initiative: they attempt to steal threads from other processors. Intuitively, the migration of threads occurs less frequently with work stealing than with work sharing, since if all processors have plenty of work to do, no threads are migrated by a work-stealing scheduler, but threads are always migrated by a work-sharing scheduler.

Cilk's work stealing scheduler executes any Cilk computation in nearly optimal time [21][4][5]. Along the execution of a Cilk program, when a processor runs out of work, it asks another processor, chosen at random, for work to do. Locally, a processor executes procedures in ordinary serial order (just like C), exploring the spawn tree in a depth-first manner. When a child procedure is spawned, the processor saves local variables of the parent (activation frame) on the bottom of a stack, which is a ready deque (doubly ended queue from which procedures can be added or deleted) and commences work on the child (the convention is that the stack grows downward, and the items are pushed and popped from the bottom of the stack). When the child returns, the bottom of the stack is popped (just like C) and the parent resumes. When another processor requests work, however, work is stolen from the top of the stack, that is, from the end opposite to the one normally used by the worker.

### 2.4. Cilk's Memory Consistency Model

Cilk's shared memory abstraction greatly enhances the programmability of a multiprocessor. In comparison to a message-passing architecture, the ability of each processor to access the entire memory simplifies programming by reducing the need for explicit data partitioning and data movement. The single address space also provides better support for parallelizing compilers

and standard operating systems. Since shared-memory systems allow multiple processors to simultaneously read and write the same memory locations, programmers require a conceptual model for the semantics of memory operations to allow them to correctly use the shared memory. This model is typically referred to as a **memory consistency model** or **memory model**. To maintain the programmability of shared-memory systems, such a model should be intuitive and simple to use.

The intuitive memory model assumed by most programmers requires the execution of a parallel program on a multiprocessor to appear as some interleaving of the execution of the parallel processes on a uniprocessor. This intuitive model was formally defined by Lamport as *sequential consistency* [19]:

DEFINITION 2.4.1. A multiprocessor 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.

Sequential consistency maintains the memory behavior that is intuitively expected by most programmers. Each processor is required to issue its memory operations in program order. Operations are serviced by memory one-at-a-time, and thus, they appear to execute atomically with respect to other memory operations. The memory services operations from different processors based on an arbitrary but fair global schedule. This leads to an arbitrary interleaving of operations from different processors into a single sequential order. The fairness criteria guarantees eventual completion of all processor requests. The above requirements lead to a total order on all memory operations that is consistent with the program order dictated by each processors program.

Unfortunately, architects of shared memory systems for parallel computers who have attempted to support Lamport's strong model of sequential consistency have generally found that Lamport's model is difficult to implement efficiently and hence relaxed models of shared-memory consistency have been developed [9][11][12]. These models adopt weaker semantics to allow a faster implementation. By and large, all of these consistency models have had one thing in common: they are processor centric in the sense that they define consistency in terms of actions by physical processors. In contrast, Cilk's **dag consistency** is defined on the abstract computation dag of a Cilk program and hence is computational centric. To define a computation-centric memory model like dag consistency it suffices to define what values are allowed to be returned by a read.

We now define dag consistency in terms of the computation. A computation is represented by its graph  $G = (V, E)$ , where  $V$  is a set of vertices representing threads of the computation, and  $E$  is a set of edges representing ordering constraints on the threads. For two threads  $u$  and  $v$ , we say that  $u$  (strictly) precedes  $v$  which we write  $u \prec v$  if  $u \neq v$  and there is a directed path in  $G$  from  $u$  to  $v$ .

DEFINITION 2.4.2. The shared memory  $M$  of a computation  $G = (V, E)$  is a **dag consistent** if for every object  $x$  in the shared memory, there exists an **observer function**  $f_x : V \rightarrow V$  such that the following conditions hold:

1. For all instructions  $u \in V$ , the instruction  $f_x(u)$  writes to  $x$ .
2. If an instructions  $u$  writes to  $x$ , then we have  $f_x(u) = u$ .
3. If an instructions  $u$  reads to  $x$ , it receives a value of  $f_x(u)$ .
4. For all instructions  $u \in V$ , we have  $u \not\prec f_x(u)$ .
5. For each triple  $u, v, w$  of instructions such that  $u \prec v \prec w$ , if  $f_x(u) \neq f_x(v)$  holds, then we have  $f_x(w) \neq f_x(u)$ .

Informally, the observer function  $f_x(u)$  represents the viewpoint of instruction  $u$  on the content of object  $x$ . For deterministic programs, this definition implies the intuitive notion that a read can see a write in the dag consistency model only if there is some serial execution order consistent with the dag in which the read sees the write. Unlike sequential consistency, but similar to certain processor-centric models [11][12] dag consistency allows different reads to return values that are based on different serial orders, but the values returned must respect the dependency in the dag. Thus, the writes performed by a thread are seen by its successors, but threads that are incomparable in the dag may or may not see each other's writes.

The Primary motivation for any weak consistency model, including dag consistency, is performance. In addition, however, a memory model must be understandable by a programmer. In the dag consistency model if the programmer wishes to ensure that a read sees the write, he must ensure that there is a path in the computation dag from the write to the read. Using Cilk can ensure that such a path exists by placing a *sync* statement between the write and the read in his program.

## 2.5. The BACKER Coherence Algorithm

Cilk's maintains dag consistency using a coherence protocol called BACKER<sup>1</sup> [21]. In this protocol, versions of shared-memory objects can reside simultaneously in any of the processors' local caches or in the global backing store. Each procedure's cache contains objects recently used by the threads that have executed on that processor and the backing store provides a global storage location for each object. In order for a thread executing on the processor to read or write an object, the object must be in the processor's cache. Each object in the cache has a dirty bit to record whether the object has been modified since it was brought into the cache.

Three basic actions are used by the BACKER to manipulate shared-memory objects: fetch, reconcile and flush. A **fetch** copies an object from the backing store to a processor cache and marks the cached object as clean. A **reconcile** copies a dirty object from a processor cache to the backing store and marks the cached as clean. Finally, a **flush** removes a clean object from a processor cache. Unlike implementations of other models of consistency, all three actions are bilateral between a processors cache and the backing store and other processors' caches are never involved.

---

<sup>1</sup>The BACKER coherence algorithm was designed and implemented as part of Keith Randall's PhD thesis, but it is not included in the Cilk versions that are actively maintained.

The BACKER coherence algorithm operates as follows. When the program performs a read or write action on an object, the action is performed directly on a cached copy of the object. If the object is not in the cache, it is fetched from the backing store before the action is performed. If the action is a write, the dirty bit of the object is set. To make space in the cache for a new object, a clean object can be removed by flushing it from the cache. To remove a dirty object, it is reconciled and then flushed. Besides performing these basic operations in response to user reads and writes, the BACKER performs additional reconciles and flushes to enforce dag consistency. For each edge  $i \rightarrow j$  in the computation dag, if threads  $i$  and  $j$  are scheduled on different processors, say  $P$  and  $Q$ , then BACKER reconciles all of  $P$ 's cached objects after  $P$  executes  $i$ , but before  $P$  enables  $j$ , and it reconciles and flushes all of  $Q$ 's cached object before  $Q$  executes  $j$ .

The key reason BACKER works is that it is always safe, at any point during the execution, for a processor  $P$  to reconcile an object or to flush a clean object. The BACKER algorithm uses this safety property to guarantee dag consistency even when there is communication. BACKER causes  $P$  to reconcile all its cached object after executing  $i$  but before enabling  $j$  and it causes  $Q$  to reconcile and flush its entire cache before executing  $j$ . At this point, the state of  $Q$ 's cache (empty) is the same as  $P$ 's if  $j$  had executed with  $i$  on processor  $P$ , but a reconcile and flush had occurred between them. Consequently, BACKER, ensures dag consistency.

## 2.6. A Model of Multithreaded Computation

Cilk supports an algorithmic model of multithreaded computation which equips us with an algorithmic foundation for predicting the performance of Cilk programs. A multithreaded computation is composed of a set of threads, each of which is a sequential ordering of unit-size instructions. A processor takes one unit of time to execute one instruction. The instructions of a threads must execute in sequential order from the first instruction to the last instruction.

From an abstract theoretical perspective, there are two fundamental limits to how fast a Cilk program could run [21][4][5]. Let us denote by  $T_p$  the execution time of a given computation on  $P$  processors. The **work** of the computation, denoted  $T_1$ , is the total number of instructions in the dag, which corresponds to the amount of time required by a one-processor execution (ignoring cache misses and other complications). Notice that with  $T_1$  work and  $P$  processors, the lower bound  $T_p \geq \frac{T_1}{P}$  must hold, since in one step, a P-processor computer can do at most  $P$  work (this, again, ignores cache misses). The second limit is based on the program's **critical path length**, denoted by  $T_\infty$ , which is the maximum number of instructions on any directed path in the dag, which corresponds to the amount of time required by an infinite-processor execution, or equivalently, the time needed to execute threads along the longest path of dependency. The corresponding lower bound is simply  $T_p \geq T_\infty$ , since a P-processor computer can do no more work in one step than an infinite-processor computer.

The work  $T_1$  and the critical path length  $T_\infty$  are not intended to denote the execution time on any real single-processor or infinite-processor machine.

These quantities are abstractions of a computation and are independent of any real machine characteristics such as communication latency. We can think of  $T_1$  and  $T_\infty$  as execution times on an ideal machine with no scheduling overhead and with a unit-access-time memory system. Nevertheless, Cilk work-stealing scheduler executes a Cilk computation that does not use locks on  $P$  processors in expected time [21]

$$T_P = \frac{T_1}{P} + O(T_\infty),$$

which is asymptotically optimal, since  $\frac{T_1}{P}$  and  $T_\infty$  are both lower bounds. Empirically, the constant factor hidden by the big  $O$  is often close to 1 or 2 [22] and the formula  $T_p = \frac{T_1}{P} + T_\infty$  provides a good approximation of the running time on shared-memory multiprocessors. This performance model holds for Cilk programs that do not use locks. If locks are used, Cilk cannot not guarantee anything [22]. This simple performance model allows the programmer to reason about the performance of his Cilk program by examining the two simple metrics: work and critical path.

The **speedup** computation on  $P$  processors is the ratio  $\frac{T_1}{T_p}$  which indicates how many times faster the  $P$ -processor execution is than a one-processor execution. if  $\frac{T_1}{T_p} = \Theta(P)$ , then we say that the  $P$ -processor execution exhibits linear speedup. The maximum possible speedup is  $\frac{T_1}{T_\infty}$  which is also called the **parallelism of the computation**, because it represents the average amount of work that can be done in parallel in each time step along the critical path. We denote the parallelism of a computation by  $\bar{P}$ .

In order to model performance for Cilk programs that use dag-consistent shared memory, we observe that running times will vary as a function of the size  $C$  of the cache that each processor uses. Therefore, we must introduce metrics that account for this dependence. We define a new work measure, the total work, that accounts for the cost of cache misses in the serial execution. Let  $\Gamma$  be the time to service a cache miss in the serial execution. We assign weight to the instructions of the dag. Each instruction that generates a cache miss in the one-processor execution with the standard, depth-first serial execution order and with a cache of size  $C$  has weight  $\Gamma + 1$ , and all other instructions have weight 1. The **total work**, denoted  $T_1(C)$ , is the total weight of all instructions in the dag, which corresponds to the serial execution time if cache misses take  $\Gamma$  units of time to be serviced. The work term  $T_1$ , which was defined before, corresponds to the serial execution time if all cache misses take zero time to be serviced. Unlike  $T_1$ ,  $T_1(C)$  depends on the serial execution order of the computation. It further differs from  $T_1$  in that  $\frac{T_1(C)}{P}$  is not a lower bound on the execution time for  $P$  processors. Consequently, the ratio  $\frac{T_1(C)}{T_\infty}$  is defined to be the **average parallelism** of the computation.

We can bound the amount of space used by parallel Cilk execution in terms of its serial space. Denote by  $S_p$  the space required for a  $P$ -processor execution. Then  $S_1$  is the space required for an execution on one processor. Cilk's guarantees [21] that for a  $P$  processor execution we have  $S_P \leq S_1 P$ . This bound implies that if a computation uses a certain amount of memory

on one processor, it will use no more space per processor on average when it runs in parallel.

The amount of interprocessors communication can be related to the number of cache misses that a Cilk computation incurs when it runs on  $P$  processors using the implementation of the BACKER coherence algorithm with cache size  $C$ . Let us denote by  $F_p(C)$  the amount of cache misses performed by a  $P$ -processor Cilk computation. Randall [21] shows that  $F_p(C) \leq F_1(C) + 2Cs$ , where  $s$  is the total number of steals executed by the scheduler. The  $2Cs$  term represents cache misses due to warming up the processors' caches. Randall has performed empirical measurements that indicated that the warm-up events are much smaller in practice than the theoretical bound. Randall shows that this bound can be further tightened, if we assume that the accesses to the backing store behave as if they were random and independent. Under this assumption, the following theorem predicts the performance of a distributed-memory Cilk program [21]:

**THEOREM 2.6.1.** *Consider any Cilk program executed on  $P$  processors, each with an LRU cache of  $C$  elements, using Cilk's work stealing scheduler in conjunction with the BACKER coherence algorithm. Assume that accesses to the backing store are random and independent. Suppose the computation has  $F_1(C)$  serial cache misses and  $T_\infty$  critical path length. Then, for any  $\epsilon > 0$ , the number of cache misses is at most  $F_1(C) + O(CPT_\infty + CP \log(\frac{1}{\epsilon}))$  with probability at least  $1 - \epsilon$ . Moreover, the expected number of cache misses is at most  $F_1(C) + O(CPT_\infty)$ .*

The standard assumption in [21] is that the backing store consists half the physical memory of each processor, and that the other half is used as a cache. In other words,  $C$  is roughly a  $1/2P$  fraction of the total memory of the machine. It is, therefore, convenient to assess the communication requirements of algorithms under this assumption, although  $C$  can, of course, be smaller.

Finally, from here on we focus on the expected performance measures (communication, time, cache misses, and space).

## Communication-Efficient Dense Matrix Multiplication in Cilk

Dense matrix multiplication is used in a variety of applications and is one of the core component in many scientific computations. The standard way of multiplying two matrices of size  $n \times n$  requires  $O(n^3)$  floating-point operations on a sequential machine. Since dense matrix multiplication is computationally expensive, the development of efficient algorithms is of great interest.

This chapter discusses two types of parallel algorithms for multiplying  $n \times n$  dense matrices  $A$  and  $B$  to yield the product matrix  $R = A \times B$  using Cilk programs. We analyze the communication cost and space requirements of specific Cilk algorithms and show new algorithms that are efficient with respect to the measures of communication and space. Specifically, we prove upper bounds on the amount of communication on SMP machines with  $P$  processors and shared-memory cache of size  $C$  when dag consistency is maintained by the BACKER coherence algorithm and under the assumption that accesses to the backing store are random and independent.

### 3.1. Space-Efficient Parallel Matrix Multiplication

Previous papers on Cilk [21] [20][6] presented two divide-and-conquer algorithms for multiplying  $n$ -by- $n$  matrices. The first algorithm uses  $\Theta(n^2)$  memory and  $\Theta(n)$  critical-path length (as stated above, we only focus on conventional  $\Theta(n^3)$ -work algorithms). In [21, page 56], this algorithm is called `notempmul`, which is the name we will use to refer to it. This algorithm divides the two input matrices into four  $\frac{n}{2}$ -by- $\frac{n}{2}$  blocks or submatrices, computes recursively the first four products and store the result in the output matrix, then computes recursively, in parallel, the last four products and then concurrently adds the new results to the output matrix. The `notempmul` algorithm is shown in Figure 3.1.1. In essence, the algorithm uses the following formulation:

$$\begin{bmatrix} R_{11} & R_{12} \\ R_{21} & R_{22} \end{bmatrix} = \begin{bmatrix} A_{11}B_{11} & A_{11}B_{12} \\ A_{21}B_{11} & A_{21}B_{12} \end{bmatrix} \\ \begin{bmatrix} R_{11} & R_{12} \\ R_{21} & R_{22} \end{bmatrix} + = \begin{bmatrix} A_{12}B_{21} & A_{12}B_{22} \\ A_{22}B_{21} & A_{22}B_{22} \end{bmatrix} .$$

Under the assumption that  $C$  is  $1/2P$  of the total memory of the machine, and that the backing store's size is  $\Theta(n^2)$  (so  $C = n^2/P$ ), the communication upper bound for `notempmul` that Theorem 2.6.1 implies is  $O(n^3)$ , which is a lot more than the  $\Theta(n^2\sqrt{P})$  bound for 2D message-passing algorithms.

```

cilk void notempmul (long nb, block *A, block *B, block *R)
{
    if (nb == 1)
        multiplyadd_block(A,B,R);
    else{
        block *C, *D, *E, *F, *G, *H, *I, *J;
        block *CGDI, *CHDJ, *EGFI, *EHFJ;

        /* get pointers to input submatrices */
        partition (nb, A, &C, &D, &E, &F);
        partition (nb, B, &G, &H, &I, &J);

        /* get pointers to result submatrices */

        partition (nb, R, &CGDI, &CHDJ, &EGFI, &EHFJ);
        /* solve subproblem recursively */
        spawn notempmul(nb/2, C, G, CGDI);
        spawn notempmul(nb/2, C, H, CHDJ);
        spawn notempmul(nb/2, E, H, EHFJ);
        spawn notempmul(nb/2, E, G, EGFI);
        sync;
        spawn notempmul(nb/2, D, I, CGDI);
        spawn notempmul(nb/2, D, J, CHDJ);
        spawn notempmul(nb/2, F, J, EHFJ);
        spawn notempmul(nb/2, F, I, EGFI);
        sync;
    }
    return;
}

```

FIGURE 3.1.1. A Cilk code for notempmul algorithm. It is a no-temporary version of recursive blocked matrix multiplication.

We suggest another way to perform  $n \times n$  matrix multiplication. Our new algorithm, called `CilkSUMMA`, is inspired by the SUMMA matrix multiplication algorithm [23]. The algorithm divides the multiplication process into phases that are performed one after the other, not concurrently; all the parallelism is within phases.

Figure 3.1.2 illustrates the algorithm. For some constant  $r$ ,  $0 < r \leq n$ , we divide  $A$  into  $\frac{n}{r}$  blocks  $A_1, A_2, \dots, A_{n/r}$  of size  $n \times r$ , we divide matrix  $B$  into  $\frac{n}{r}$  blocks  $B_1, \dots, B_{n/r}$  of size  $r \times n$ , and we then compute the  $\frac{n}{r}$  multiplications in  $\frac{n}{r}$  phases. At each phase the appropriate blocks of the two matrices are multiplied in parallel and added to the  $n \times n$  result matrix. The multiplication of  $A_k$  by  $B_k$  is performed by an auxiliary procedure, `RankRUpdate`. The algorithm's code is as follows (we ignore the details of generating references to submatrices):

```

cilk CilkSUMMA(A, B, R, n)

```

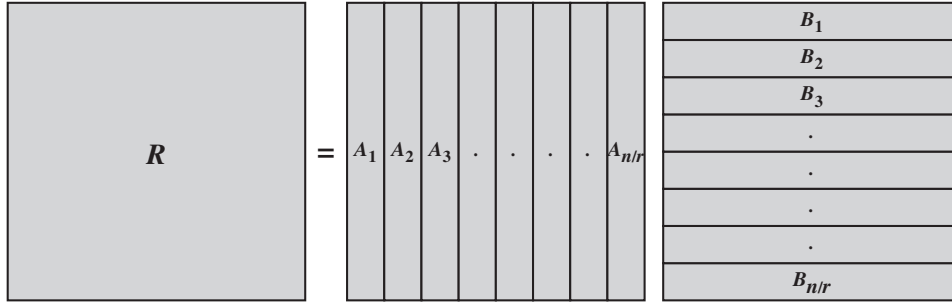


FIGURE 3.1.2. The `CilkSUMMA` algorithm for parallel matrix multiplication. `CilkSUMMA` divides  $A$  into  $\frac{n}{r}$  vertical blocks of size  $n \times r$  and  $B$  into  $\frac{n}{r}$  horizontal blocks of size  $r \times n$ . The corresponding blocks of  $A$  and  $B$  are iteratively multiplied to produce  $n \times n$  product matrix. Each such matrix is stored in matrix  $R$  with the previous iteration result values. Finally  $R$  will hold the product of  $A$  and  $B$ .

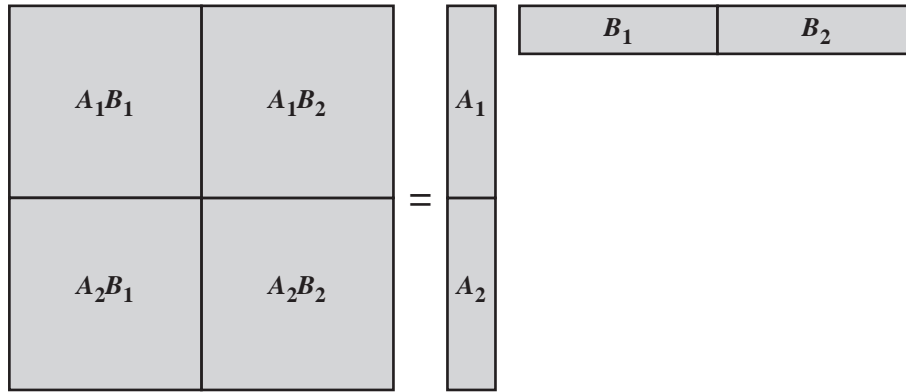


FIGURE 3.1.3. Recursive rank- $r$  update to an  $n$ -by- $n$  matrix  $R$ .

```

{
  R = 0
  for k = 1..n/r {
    spawn RankRUpdate(Ak, Bk, R, n, r)
    sync
  }
}

```

The `RankRUpdate` procedure recursively divides an  $n \times r$  block of  $A$  into two blocks of size  $\frac{n}{2} \times r$  and an  $r \times n$  block of  $B$  into two blocks of size  $r \times \frac{n}{2}$  and multiplies the corresponding blocks in parallel, recursively. If  $n = 1$ , then the multiplication reduces to a dot product, for which we give the code later. The algorithm is shown in Figure 3.1.3, and its code is given below:

```
cilk RankRUpdate(A, B, R, n, r)
```

```

{
  if n=1
    R += DotProd(A, B, r)
  else {
    spawn RankRUpdate(A1, B1, R11,  $\frac{n}{2}$ , r)
    spawn RankRUpdate(A1, B2, R12,  $\frac{n}{2}$ , r)
    spawn RankRUpdate(A2, B1, R21,  $\frac{n}{2}$ , r)
    spawn RankRUpdate(A2, B2, R22,  $\frac{n}{2}$ , r)
  }
}

```

The recursive Cilk procedure `DotProd`, shown below, is executed at the bottom of the rank- $r$  recursion. If  $r = 1$ , the code returns the scalar multiplication of the inputs. Otherwise, the code splits each of the the  $r$ -length input vectors  $a$  and  $b$  into two subvectors of  $r/2$  elements, and multiplies the two halves recursively, and returns the sum of the two dot products. Clearly, the code performs  $\Theta(r)$  work and has critical path  $\Theta(\log r)$ . The details are as follows:

```

cilk DotProd(a, b, r)
{
  if (r = 1) return a1 · b1
  else {
    x = spawn DotProd(a[1, ..., [r/2]], b[1, ..., [r/2]],  $\frac{r}{2}$ )

    y = spawn DotProd(a[[r/2]+1, ..., r], b[[r/2]+1, ..., r],  $\frac{r}{2}$ )
    sync
    return(x + y)
  }
}

```

The analysis of communication cost is organized as follows. First, we prove a lemma describing the amount of communication performed by `RankRUpdate`. Next, we obtain a bound on the amount of communication in `CilkSUMMA`.

**LEMMA 3.1.1.** *The amount of communication in `RankRUpdate`,  $F_P^{RRU}(C, n)$ , incurred by `BACKER` running on  $P$  processors, each with a shared-memory cache of  $C$  elements, and with block size  $r = \sqrt{C}/3$ , when solving a problem of size  $n$ , is  $O(n^2 + CP \log(n\sqrt{C}))$ .*

**PROOF.** To find the number of `RankRUpdate` cache misses, we use Theorem 2.6.1.

The work and critical path for `RankRUpdate` can be computed using recurrences. We find the number of cache misses incurred when `RankRUpdate` algorithm is executed on a single processor and then assign them in Theorem 2.6.1.

The work bound  $T_1^{RRU}(n, r)$  satisfies  $T_1^{RRU}(1, r) = T_1^{DOTPROD}(r) = \Theta(r)$  and  $T_1^{RRU}(n, r) = 4T_1^{RRU}(n/2)$  for  $n > 1$ . Therefore,  $T_1^{RRU}(n, r) = \Theta(n^2 r)$ .

To derive recurrence for the critical path length  $T_\infty^{RRU}(n, r)$ , we observe that with an infinite number of processors the 4 block multiplications can execute in parallel, therefore  $T_\infty^{RRU}(n, r) = T_\infty^{RRU}(n/2, r) + \Theta(1)$  for  $n > 1$ .

For  $n = 1$   $T_\infty^{RRU}(1, r) = T_\infty^{DOTPROD}(r) + \Theta(1) = \Theta(\log r)$ . Consequently, the critical path satisfies  $T_\infty^{RRU}(n, r) = \Theta(\log n + \log r)$ .

Next, we bound  $F_1^{RRU}(C, n)$ , the number of cache misses occur when the **RankRUUpdate** algorithm is used to solve a problem of size  $n$  with the standard, depth-first serial execution order on a single processor with an LRU cache of size  $C$ . At each node of the computational tree of **RankRUUpdate**,  $k^2$  elements of  $R$  in a  $k \times k$  block are updated using the results of  $k^2$  dot products of size  $r$ . To perform such an operation entirely in the cache, the cache must store  $k^2$  elements of  $R$ ,  $kr$  elements of  $A$ , and  $kr$  elements of  $B$ . When  $k \leq \sqrt{C}/3$ , the three submatrices fit into the cache. Let  $k' = r = \sqrt{C}/3$ . Clearly, the  $(n/k')^2$  updates to  $k'$ -by- $k'$  blocks can be performed entirely in the cache, so the total number of cache misses is at most  $F_1^{RRU}(C, n) \leq (n/k')^2 \times [(k')^2 + 2k'r] \leq \Theta(n^2)$ .

By Theorem 2.6.1 the amount of communication that **RankRUUpdate** performs, when run on  $P$  processors using Cilk's scheduler and the **BACKER** coherence algorithm, is

$$F_p^{RRU}(C, n) = F_1^{RRU}(C, n) + O(CPT_\infty^{RRU}).$$

Since the critical path length of **RankRUUpdate** is  $\Theta(\log nr)$ , the total number of cache misses is  $O(n^2 + CP \log(n\sqrt{C}))$ .  $\square$

Next, we analyze the amount of communication in **CilkSUMMA**.

**THEOREM 3.1.2.** *The number of **CilkSUMMA** cache misses  $F_P^{CS}(C, n)$ , incurred by **BACKER** running on  $P$  processors, each with a shared-memory cache of  $C$  elements and block size  $r = \sqrt{C}/3$  when solving a problem of size  $n$  is  $O(\frac{n}{\sqrt{C}}(n^2 + CP \log(n\sqrt{C})))$ . In addition the total amount  $S_P^{CS}(n)$  of space taken by the algorithm is  $O(n^2 + P \log(n\sqrt{C}))$ .*

**PROOF.** Notice that the **CilkSUMMA** algorithm only performs sequential calls to the parallel algorithm **RankRUUpdate**. The *sync* statement at the end of each iteration guarantees that the procedure suspends and does not resume until all the **RankRUUpdate** children have completed. Each such iteration is a phase in which only one call to **RankRUUpdate** is invoked so the only parallel execution is of the parent procedure and its own children. Thus, we can bound the total number of cache misses to be at most the sum of all the cache misses incurred during each phase

$$F_P^{CS}(C, n) \leq \sum_1^{n/r} F_P^{RRU}(C, n).$$

By Lemma 4.2.1, we have  $F_P^{RRU}(C, n) = O(n^2 + CP \log(n\sqrt{C}))$ , yielding

$$\begin{aligned} F_P^{CS}(C, n) &= O\left(\frac{n}{r}(n^2 + CP \log(nr))\right) \\ &= O\left(\frac{n}{\sqrt{C}}\left(n^2 + CP \log(n\sqrt{C})\right)\right). \end{aligned}$$

The total amount of space used by the algorithm, is the space allocated for the product matrix, and the space for the activation frames allocated by the run time system. The Cilk run time system uses activation frames

to represent procedure instances. Each such representation is of constant size, including the program counter and all live, dirty variables. The frame is pushed into a deque on the heap and it is deallocated at the end of the procedure call. Therefore, in the worst case the total space saved for the activation frames is the longest possible chain of procedure instances, for each processor, which is the critical path length, resulting in  $O(P \log nr)$  total space allocated at any time, thus  $S_P^{CS}(n) = \Theta(n^2) + O(P \log nr) = O(n^2 + P \log(n\sqrt{C}))$ .  $\square$

We have bounded the work and critical path of `CilkSUMMA`. Using these values we can compute the total work and estimate the total running time  $T_P^{CS}(C, n)$ . The computational work of `CilkSUMMA` is  $T_1^{CS}(n) = \Theta(n^3)$ , so the total work is  $T_1^{CS}(C, n) = T_1^{CS}(n) + \Gamma F_1^{CS}(C, n) = \Theta(n^3)$ , assuming  $\Gamma$  is a constant. The critical path length is  $T_\infty^{CS}(C, n) = \frac{n}{\sqrt{C}} \log(n\sqrt{C})$ , so using the performance model in [21], the total expected time for `CilkSUMMA` on  $P$  processors is

$$T_P(C, n) = O\left(\frac{T_1(C, n)}{P} + \Gamma C T_\infty(n)\right) = O\left(\frac{n^3}{P} + \Gamma n\sqrt{C} \log(n\sqrt{C})\right).$$

Consequently, if  $P = O\left(\frac{n^2}{\Gamma\sqrt{C} \log(n\sqrt{C})}\right)$ , the algorithm runs in  $O\left(\frac{n^3}{P}\right)$  time, obtaining linear speedup.

`CilkSUMMA` uses the processor cache more effectively than `notempmul` whenever  $\sqrt{C} > \log(n\sqrt{C})$ , which holds asymptotically for all  $C = \Omega(n)$ . If we consider the size of the cache to be  $C = \frac{n^2}{P}$ , which is the memory size of each one of the  $P$  processors in a distributed-memory machine when solving 2D,  $n \times n$  matrix multiplication algorithms, then  $F_p^{CS}(C, n) = \Theta(\sqrt{P}n^2 \log n)$  and  $S_P^{CS}(n) = O(n^2)$ . These results are comparable to the communication and space requirements of 2D distributed-memory matrix multiplication algorithms, and they are significantly better than the  $\Theta(n^3)$  communication bound.

We also improved the average parallelism of the algorithm over `notempmul` for  $r = \Omega(n)$ , since  $\frac{T_1(C, n)}{T_\infty(n)} = \frac{n^3}{\frac{n}{r} \log(nr)} > n^2$ .

### 3.2. Trading Space for Communication in Parallel Matrix Multiplication

The matrix-multiplication algorithm shown in Figure 3.2.1 is perhaps the most natural matrix-multiplication algorithm in Cilk. The code is from [21, page 55], but the same algorithm also appears in [20]. The motivation for this algorithm, called `blockedmul`, is to increase parallelism, at the expense of using more memory. Its critical path is only  $\Theta(\log^2 n)$ , as opposed to  $\Theta(n)$  in `notempmul`, but it uses  $\Theta(n^2 P^{1/3})$  space [21, page 148], as opposed to only  $\Theta(n^2)$  in `notempmul`.

In the message-passing literature on matrix algorithms, space is traded for a reduction in communication, not for parallelism. So-called 3D matrix multiplication algorithm [1, 2, 3, 13, 14, 17] replicate the input matrices  $P^{1/3}$  times in order to reduce the total amount of communication from  $\Theta(n^2 P^{1/2})$  in 2D algorithms down to  $\Theta(n^2 P^{1/3})$ . Irony and Toledo have

### 3.2. TRADING SPACE FOR COMMUNICATION IN PARALLEL MATRIX MULTIPLICATION

```

cilk void blockedmul (long nb, block *A, block *B, block *R)
{
    if (nb == 1)
        multiply_block(A,B,R);
    else{
        block *C, *D, *E, *F, *G, *H, *I, *J;

        block *CG, *CH, *EG, *EH, *DI, *DJ, *FI, *FJ;
        block tmp[nb*nb];

        /* get pointers to input submatrices */
        partition (nb, A, &C, &D, &E, &F);
        partition (nb, B, &G, &H, &I, &J);

        /* get pointers to result submatrices */
        partition (nb, R, &CG, &CH, &EG, &EH);

        partition (nb, tmp, &DI, &DJ, &FI, &FJ);
        /* solve subproblem recursively */
        spawn blockedmul(nb/2, C, G, CG);
        spawn blockedmul(nb/2, C, H, CH);
        spawn blockedmul(nb/2, E, H, EH);
        spawn blockedmul(nb/2, E, G, EG);
        spawn blockedmul(nb/2, D, I, DI);
        spawn blockedmul(nb/2, D, J, DJ);
        spawn blockedmul(nb/2, F, J, FJ);
        spawn blockedmul(nb/2, F, I, FI);
        sync;

        /* add results together into R*/
        spawn matrixadd(nb,tmp,R);
        sync;
    }
    return;
}

```

FIGURE 3.2.1. A Cilk code for recursive blocked matrix multiplication. It uses divide-and-conquer to solve one  $n \times n$  multiplication problem by splitting it into  $8 \frac{n}{2} \times \frac{n}{2}$  multiplication subproblems and combining the results with one  $n \times n$  addition. A temporary matrix of size  $n \times n$  is allocated at each divide step. A serial matrix multiplication routine is called to do the base case.

shown that the additional memory is necessary for reducing communication, and that the tradeoff is asymptotically tight [16].

Substituting  $C = n^2/P^{2/3}$  in Randall's communication analysis for `blockedmul`, we find out that with that much memory, the algorithm performs  $O(n^2 P^{1/3} \log^2 n)$

communication. That is, if we provide the program with caches large enough to replicate the input  $\Theta(P^{1/3})$  times, as in 3D message-passing algorithms, the amount of communication that it performs is at most a factor of  $\Theta(\log^2 n)$  more than message-passing 3D algorithms. In other words, `blockedmul` is a Cilk analog of 3D algorithms.

We propose a slightly more communication efficient algorithm than `blockedmul`. Like our previous algorithm, `CilkSUMMA`, obtaining optimal performance from this algorithm requires explicit knowledge and use of the cache size parameter  $C$ . This makes the algorithm more efficient but less elegant than `blockedmul`, which exploits a large cache automatically without explicit use of the cache-size parameter. On the other hand, `blockedmul` may simply fail if it cannot allocate temporary storage (a real-world implementation should probably synchronize after 4 recursive calls, as `notempmul` does, if it cannot allocate a temporary matrix).

The code for `SpaceMul` is given below. It uses an auxiliary procedure, `MatrixAdd`, which is not shown here, to sum an array of  $n \times n$  matrices. We assume that `MatrixAdd` sums  $k$  matrices of dimension  $n$  using  $\Theta(kn^2)$  work and critical path  $\Theta(\log k \log n)$ ; such an algorithm is trivial to implement in Cilk. For simplicity, we assume that  $n$  is a power of 2.

```

cilk spawnhelper(cilk procedure f, array [Y1, Y2, ... Yk])
{
  if (k = 1) spawn f(Y1)
  else {
    spawn spawnhelper(f, [Y1, Y2, ... Y[k/2]])
    spawn spawnhelper(f, [Y[k/2]+1, ... Yk])
  }
}
cilk SpaceMul(A, B, R)
{
  /* comment: A, B, R are n-by-n */

  Allocate  $\frac{n}{r}$  matrices, each n-by-n, denoted R1..R $\frac{n}{r}$ 
  Partition A into  $\frac{n}{r}$  block columns A1..A $\frac{n}{r}$ 
  Partition B into  $\frac{n}{r}$  block rows B1..B $\frac{n}{r}$ 
  spawn spawnhelper(RankRUpdate,

[(A1, B1, R1), ..., (A $\frac{n}{r}$ , B $\frac{n}{r}$ , R $\frac{n}{r}$ )
)

  sync
  spawn MatrixAdd(R, R1, R2, ..., R $\frac{n}{r}$ )
  return R
}

```

**THEOREM 3.2.1.** *The number of `SpaceMul` cache misses  $F_P^{SM}(C, n)$ , incurred by `BACKER` running on  $P$  processors, each with a shared-memory cache of  $C$  elements and block size  $r = \sqrt{C}/3$  when solving a problem of size*

$n$  is  $O(\frac{n^3}{\sqrt{C}} + CP \log \frac{n}{\sqrt{C}} \log n)$ . The total amount  $S_P^{SM}(n)$  of space used by the algorithm is  $O(\frac{n^3}{\sqrt{C}})$ .

PROOF. The amount of communication in **SpaceMul** is bounded by the sum of the communication incurred until the **sync** and the communication incurred after the **sync**. Thus,  $F_P^{SM}(C, n) = F_P^1(C, n) + F_P^2(C, n)$  where  $F_P^1$ ,  $F_P^2$  represent the communication in the two phases of the algorithm.

First, we compute the work and critical path of **SpaceMul**. Let  $T_P^{ADD}(n, k)$  be the  $P$ -processor running time of **MatrixAdd** for summing  $k$  matrices of dimension  $n$ , let  $T_P^{RRU}(n, r)$  be the  $P$ -processor running time of **RankRUUpdate** to perform a rank- $r$  update on an  $n \times n$  matrix, and let  $T_P^{SM}(n)$  be the total running time of **SpaceMul**.

Recall from the previous section that is  $T_1^{RRU}(n, r) = \Theta(n^2 r)$  and that  $T_\infty^{RRU}(n, r) = \Theta(\log nr)$ . As discussed above, it is trivial to implement **MatrixAdd** so that  $T_1^{ADD}(n, n/r) = \Theta(n^3/r)$  and  $T_\infty^{ADD}(n, n/r) = \Theta(\log \frac{n}{r} \log n)$ .

We now bound the work and critical path of **SpaceMul**. The work for **SpaceMul** is

$$\begin{aligned} T_1^{SM}(n, r) &= \frac{n}{r} + \frac{n}{r} T_1^{RRU}(n, r) + T_1^{ADD}(n, \frac{n}{r}) \\ &= \frac{n}{r} + \frac{n}{r} n^2 r + \frac{n}{r} n^2 = \Theta(n^3) \end{aligned}$$

(There is nothing surprising about this: this is essentially a schedule for the conventional algorithm). The critical path for **SpaceMul** is  $T_\infty^{SM}(n) = \log \frac{n}{r} + T_\infty^{RRU}(n, r) + T_\infty^{ADD}(n, \frac{n}{r})$ . The first term accounts for spawning the  $\frac{n}{r}$  parallel rank- $r$  updates. Therefore,  $T_\infty^{SM}(n) = \Theta(\log \frac{n}{r} + \log nr + \log \frac{n}{r} \log n) = \Theta(\log \frac{n}{r} \log n)$ .

Next, we compute the amount of communication in **SpaceMul**. From the proof of Lemma 3.1.1 we know that  $F_1^{RRU}(C, n, r) = n^2$  (recall that  $r = \sqrt{C}/3$ ). A sequential execution of **MatrixAdd** performs  $O(\frac{n}{r} n^2)$  cache misses, at most  $3n^2$  during the addition of every pair of  $n$ -by- $n$  matrices.

Using Theorem 2.6.1, we can bound the total communication  $F_P^{SM}(C, n)$  in **SpaceMul**,

$$\begin{aligned} F_P^{SM}(C, n) &= F_P^1(C, n) + F_P^2(C, n) \\ &= O\left(\frac{n^3}{r} + CP \log n\right) + O\left(\frac{n^3}{r} + CP \log \frac{n}{r} \log n\right) \\ &= O\left(\frac{n^3}{\sqrt{C}} + CP \log \frac{n}{\sqrt{C}} \log n\right). \end{aligned}$$

The space used by the algorithm consists of the space for the  $\frac{n}{r}$  product matrices and the space of the activation frames which are bounded by  $PT_\infty$ . Therefore the total space cost is  $O(\frac{n}{r} n^2 + P \log \frac{n}{r} \log n) = O(\frac{n^3}{\sqrt{C}})$ .  $\square$

CONCLUSION 3.2.2. The communication upper bound of **SpaceMul** is smaller a factor of  $\Omega\left(\frac{\log n}{\log \frac{n}{\sqrt{C}}}\right)$  than the bound of **blockedmul** for any cache size.

Algorithm	$S_P$	$F_P$	$T_1$	$T_\infty$
notempmul	$n^2$	$\frac{n^3}{\sqrt{C}} + CPn$	$n^3$	$n$
blockedmul	$n^2 P^{1/3}$	$\frac{n^3}{\sqrt{C}} + CP \log^2 n$	$n^3$	$\log^2 n$
CilkSUMMA	$n^2$	$\frac{n^3}{\sqrt{C}} + \sqrt{C} P n \log(n\sqrt{C})$	$n^3$	$\frac{n}{\sqrt{C}} \log(n\sqrt{C})$
SpaceMul	$\frac{n^3}{\sqrt{C}}$	$\frac{n^3}{\sqrt{C}} + CP \log n \log(\frac{n}{\sqrt{C}})$	$n^3$	$\log \frac{n}{\sqrt{C}} \log n$

TABLE 1. Asymptotic upper bounds on the performance metrics of the four Cilk matrix multiplication algorithm, when applied to  $n$ -by- $n$  matrices on a computer with  $P$  processors and cache size  $C$ .

PROOF. The amount of communication in `blockedmul` is bounded by  $\Theta\left(\frac{n^3}{\sqrt{C}} + CP \log^2 n\right)$ . The result follows from Theorem 3.2.1.  $\square$

in particular, for  $C = \frac{n^2}{P^{2/3}}$  and  $r = \sqrt{C}/3$ ,

$$\begin{aligned} F_P^{SM}(n) &= O\left(n^2 P^{1/3} + \frac{1}{3} n^2 P^{1/3} \log P \log n\right) \\ &= O\left(n^2 P^{1/3} \log P \log n\right). \end{aligned}$$

This bound is a factor of  $\frac{P^{1/6}}{\log P}$  smaller than the corresponding bound for `CilkSUMMA`; The average parallelism is  $\frac{T_1^{SM}(C,n)}{T_\infty^{SM}(n)} = \Omega\left(\frac{n^3}{\log^2 n}\right)$ , which is the same as in `blockedmul`.

### 3.3. A Comparison of Message-Passing and Cilk Matrix-Multiplication Algorithms

Table 1 summarized the performance bounds of the four Cilk algorithms that we have discussed in this chapter. The table compares the space, communication, work, and critical path in the four algorithms as a function of the input size  $n$  and the cache size  $C$ .

Table 2 compares message-passing algorithms to their Cilk analogs. Message-passing algorithms have fixed space requirements, and the table shows the required amount of space and the corresponding bound on communication. In Cilk algorithms the amount of communication depends on the size of local memories (the cache size), and the table fixes these sizes to match the space requirements of message-passing algorithms. The communication bounds or the Cilk algorithms were derived by substituting the cache size  $C$  in the general bounds shown in Table 1. The main conclusions that we draw from the table are

- The `notempmul` algorithm is communication inefficient. `CilkSUMMA` is a much better alternative.
- The communication upper bounds for even the best Cilk algorithms are worse by factor of between  $\log n$  and  $\log^2 n$  than the communication in message-passing algorithms.

Can the performance of `notempmul` and `blockedmul` be improved by tuning the cache size to the problem size and machine size at hand? Table 3

Algorithm	Memory per Processor	Total Communication
Distributed 2D	$\Theta\left(\frac{n^2}{P}\right)$	$O\left(n^2\sqrt{P}\right)$
Distributed 3D	$\Theta\left(\frac{n^2}{P^{2/3}}\right)$	$O\left(n^2P^{1/3}\right)$
<code>notempmul</code>	$\Theta\left(\frac{n^2}{P}\right)$	$O\left(n^3\right)$
<code>blockedmul</code>	$\Theta\left(\frac{n^2}{P^{2/3}}\right)$	$O\left(n^2P^{1/3}\log^2 n\right)$
<code>CilkSUMMA</code>	$\Theta\left(\frac{n^2}{P}\right)$	$O\left(n^2\sqrt{P}\log n\right)$
<code>SpaceMul</code>	$\Theta\left(\frac{n^2}{P^{2/3}}\right)$	$O\left(n^2P^{1/3}\log P\log n\right)$

TABLE 2. Communication overhead for Cilk shared-memory algorithms when the processor cache size is the same as the processor memory size when performing distributed-memory algorithms.

Algorithm	Optimal Cache Size	Overall Communication
<code>notempmul</code>	$C = \Theta\left(\frac{n^{4/3}}{P^{2/3}}\right)$	$O\left(n^{7/3}P^{1/3}\right)$
<code>blockedmul</code>	$C = \Theta\left(\frac{n^2}{P^{2/3}\log^{4/3} n}\right)$	$O\left(n^2P^{1/3}\log^{2/3} n\right)$

TABLE 3. Optimized communication overhead of Cilk  $n \times n$  matrix multiplication algorithms and their appropriate cache size values.

shows that the performance of `notempmul` can indeed be improved by reducing the caches slightly. Obviously, the size of the backing store cannot be shrunk if it to hold the input and output, so the implication is that for this algorithm, the size of caches should be smaller than half the local memory of the processor. (This setting reduces the provable upper bound; whether it reduces communication in practice is another matter, and we conjecture that it does not.) However, even after the reduction the communication upper bound is significantly worse than that of all the other algorithms. The table also shows that the performance of `blockedmul` can also be improved slightly by reducing the size of the cache, but not by much. Since `SpaceMul` always performs less communication, the same applied to it.

## A Communication-Efficient Triangular Solver in Cilk

This chapter focuses on the solution of linear systems of equations with triangular coefficient matrices. Such systems are solved nearly always by substitution, which creates a relatively long critical path in the computation. In Randall's analyses of communication, long critical paths weaken the upper bounds because of the  $CPT_\infty$  term. In this chapter we show that by allowing the programmer to dynamically control the size of local caches, we can derive tighter communication upper bounds. More specifically, the combination of dynamic cache-size control and a new Cilk algorithm allows us to achieve performance bounds similar to those of state-of-the-art message-passing algorithms.

### 4.1. Triangular Solvers in Cilk

A lower triangular linear system of equations

$$\begin{aligned} l_{11}x_1 &= b_1 \\ l_{21}x_1 + l_{22}x_2 &= b_2 \\ &\vdots \\ l_{n1}x_1 + l_{n2}x_2 + \cdots + l_{nn}x_n &= b_n \end{aligned}$$

which we can also write as a matrix equation  $Lx = b$ , is solved by substitution. Here  $L$  is a known coefficient matrix,  $b$  is a known vector, and  $x$  is a vector of unknown to be solved for. This chapter actually focuses on solution of multiple linear systems with the same coefficient matrix  $L$  but with different right hand sides  $b$ , which we write  $LX = B$ . More specifically, we focus on the case in which  $B$  has exactly  $n$  columns, which is the case that comes up in the factorization of general matrices, the subject of the next chapter. We assume that  $L$  is nonsingular, which for a triangular matrix is equivalent to saying that it has no zeros on the diagonal. Although we focus on lower triangular systems, upper triangular systems are handled in exactly the same way.

We can solve such systems by substitution. In a lower triangular system the first equation involves only one variable,  $x_1$ . We can, therefore, solve it directly,  $x_1 = \frac{b_1}{l_{11}}$ . Now that we know the value of  $x_1$ , we can substitute its value in all the other equations. Now the second equation involves only one unknown, which we solve for, and so on.

Randall presents and analyzes a recursive formulation of the substitution algorithm in Cilk [21, pg. 58]. This recursive solver partitions the matrix as shown in Figure 4.1.1. The code is as follows:

```
cilk RecursiveTriSolver(L, X, B)
```

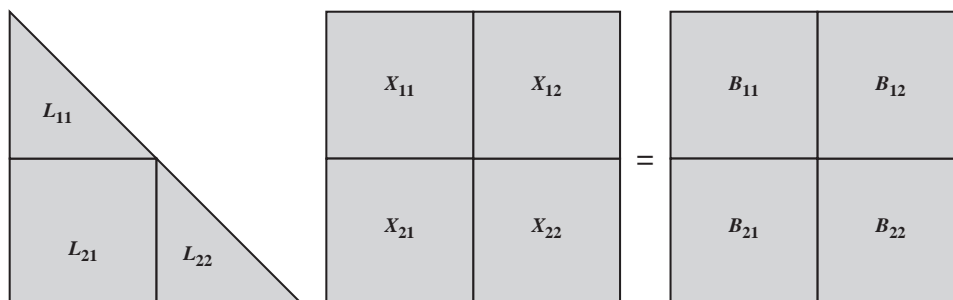


FIGURE 4.1.1. Recursive decomposition of a traditional triangular solver algorithm. Subdividing the three matrices into  $\frac{n}{2} \times \frac{n}{2}$  matrices. First solving the recursive matrix equations  $L_{11}X_{11} = B_{11}$  for  $X_{11}$  and in parallel  $L_{11}X_{12} = B_{12}$  for  $X_{12}$ . Then, compute  $B'_{21} = B_{21} - L_{21}X_{11}$  and in parallel  $B'_{22} = B_{22} - L_{21}X_{12}$ . Finally solve recursively  $B'_{21} = L_{22}X_{21}$  and in parallel  $B'_{22} = L_{22}X_{22}$ .

```

{
  if (L is 1-by-1)  $x_{11} = b_{11}/l_{11}$ 
  else {

    partition L, B, and X as in Figure 4.1.1
      spawn RecursiveTriSolver( $L_{11}, X_{11}, B_{11}$ )
      spawn RecursiveTriSolver( $L_{11}, X_{12}, B_{12}$ )
      sync

    spawn multiply-and-update( $B_{21} = B_{21} - L_{21}X_{11}$ )

    spawn multiply-and-update( $B_{22} = B_{22} - L_{21}X_{12}$ )
    sync
    spawn RecursiveTriSolver( $L_{22}, X_{21}, B_{21}$ )
    spawn RecursiveTriSolver( $L_{22}, X_{22}, B_{22}$ )
  }
}

```

This algorithm performs  $\Theta(n^3)$  work and the length of its critical path is  $\Theta(n \log n)$  when the `multiply-and-updates` are performed using `notempmul`. Here too, the advantage of `notempmul` is that it uses no auxiliary space. But because the substitution algorithm solves for one unknown after the other, the critical path cannot be shorter than  $n$  even if the `multiply-and-updates` are performed using a short-critical-path multiplier, so `notempmul` does not worsen the parallelism significantly. For  $C = \frac{n^2}{P}$ , Randall's result implies that the amount of communication is bounded by  $F_P(C) = O(n^3 \log n)$ . This is a meaningless bound, since even without local caching at all, a  $\Theta(n^3)$  algorithm does not perform more than  $\Theta(n^3)$  communication.

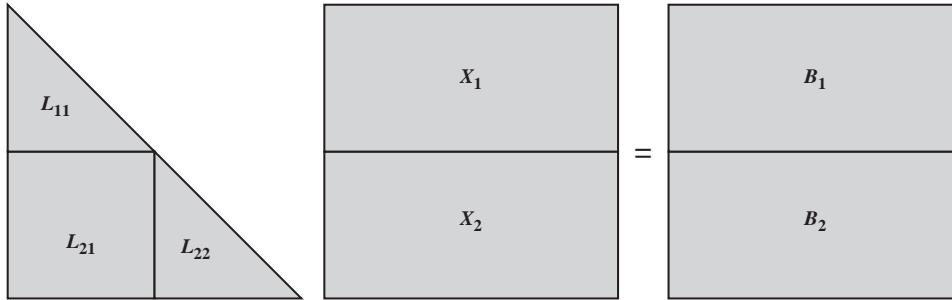


FIGURE 4.1.2. The recursive partitioning in `NewTriSolver`. The algorithm first recursively solves  $L_{11}X_1 = B_1$  for  $X_{11}$ , then updates  $B'_2 = B_2 - L_{21}X_1$ , and then recursively solves  $L_{22}X_2 = B'_2$ .

We use a slightly different recursive formulation of the substitution algorithm, which will later allow us to develop a communication-efficient algorithm. Our new algorithm, `NewTriSolver`, partitions the matrices as shown in Figure 4.1.2. Its code is actually simpler than that of `RecursiveTriSolver`:

```

cilk NewTriSolver(L, X, B)
{
    if (L is 1-by-1)

        call VectorScale(1/l11, X, B) /* X = B/l11 */
        else {

            partition L, B, and X as in Figure 4.1.2
            call NewTriSolver(L11, X1, B1)
            call RectangularMatMult(B2 = B2 - L21X1)
            call NewTriSolver(L22, X2, B2)
        }
}

```

This algorithm exposes no parallelism—all the parallelism is exposed by the two auxiliary routines that it calls. We use the keyword `call` to emphasize that the caller is suspended until the callee returns, unlike a `spawn`, which allows the caller to continue to execute concurrently with the callee. The `call` keyword uses the normal C function call mechanism. We could achieve the same scheduling result by each of the called routines in `NewTriSolver` and immediately following each `spawn` with a `sync`.

## 4.2. Auxiliary Routines

We now turn to the description of the auxiliary routines that `NewTriSolver` uses. The `VectorScale` routine scales a vector  $Y = (y_1, \dots, y_n)$  by a scalar  $\alpha$  and return the result in another vector  $X = (x_1, \dots, x_n)$ .

```

cilk VectorScale( $\alpha$ , [x1, ..., xn], [y1, ..., yn])

```

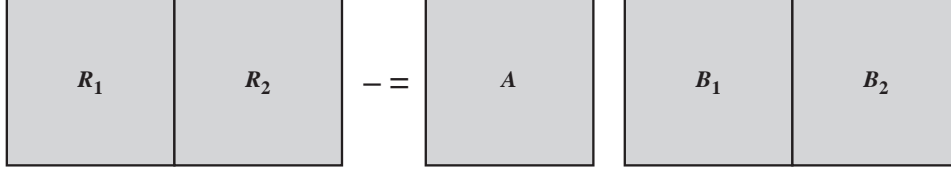


FIGURE 4.2.1. The recursive partitioning in `RectangularMatMult`. The algorithm recursively partitions the long matrices until they are square and then calls `CilkSUMMA`.

```

{
  if (n = 1) x1 = αy1
  else
  {
    spawn VectorScale(α, [x1, ..., x⌊n/2⌋}], [y1, ..., y⌊n/2⌋}]);
    spawn VectorScale(α, [x⌊n/2⌋+1}, ..., xn], [y⌊n/2⌋+1}, ..., yn]);
  }
}

```

Analyzing the performance of this simple algorithm is easy. Clearly, it performs  $\Theta(n)$  work with critical path  $\Theta(\log n)$ . The next lemma analyzes the amount of communication in the code. Intuitively, we expect the amount of communication to be  $\Theta(n)$ , since there is no data reuse in the algorithm.

LEMMA 4.2.1. *The amount of communication in `VectorScale` is bounded by  $F_P^{VS}(C = 3, n) = O(n + P \log n)$ .*

PROOF. We use Theorem 2.6.1 to show that  $F_P^{VS}(C, n) = O(n + CP \log n)$ . The work in `VectorScale` is  $T_1^{VS}(n) = 2T_1(n/2) = \Theta(n)$  and the critical path length is  $T_\infty^{VS}(n) = \Theta(\log n)$ . The amount of serial cache misses is  $O(n)$  independently in the cache size because the work bounds the number of cache misses. Applying Theorem 2.6.1 yields  $F_P^{VS}(C, n) = O(n + CP \log n)$ . The result follows by substituting  $C = 3$ .  $\square$

The second algorithm that `NewTriSolver` uses, `RectangularMatMult`, is a rectangular matrix multiplier that calls `CilkSUMMA`. This algorithm is always called to multiply a square  $m$ -by- $m$  matrix by an  $m$ -by- $n$  matrix, where  $m < n$ . We assume for simplicity that  $m$  divides  $n$ . It works by recursively dividing the long matrix into block columns until it is square (or roughly square in practice), and then calls `CilkSUMMA`.

```

cilk RectangularMatMult(A, B, R) /* R =
R - AB */

```

```

{
  partition  $A$  and  $B$  into as shown in Figure 4.2.1
  if ( $R, B$  have more columns than rows ) {
    spawn RectangularMatMult( $A, B_1, R_1$ )
    spawn RectangularMatMult( $A, B_2, R_2$ )
  } else
    call CilkSUMMA( $A, B, R, m$ )
}

```

Let us now analyze the performance of this algorithm. The next lemma analyzes the amount of work and the length of the critical path, and lemma 4.2.3 that follows analyzes communication. To keep the analysis simple, we assume that  $n$  is a power of 2 and that  $m$  divides  $n$ .

LEMMA 4.2.2. *Let  $A$  be  $m$ -by- $m$  and let  $B$  and  $R$  be  $m$ -by- $n$ . The amount of work in `RectangularMatMult`( $A, B, R$ ) is  $\Theta(m^2n)$  and the length of its critical path is  $O\left(\log(n/m) + m \log(m\sqrt{C})/\sqrt{C}\right)$ .*

PROOF. The work in `RectangularMatMult` is equal to the work in `CilkSUMMA` if  $n = m$  and is  $T_1^{RMM}(m, n) = 2T_1^{RMM}(m, \frac{n}{2})$  otherwise. Therefore,  $T_1^{RMM}(m, n) = 2T_1^{RMM}(m, \frac{n}{2}) = 2^{\log \frac{n}{m}} T_1^{CS}(m) = \frac{n}{m} \Theta(m^3) = \Theta(m^2n)$ .

The critical path of `RectangularMatMult` is bounded by  $\Theta(1) + T_\infty^{CS}(C, m)$  if  $n = m$  and by  $\Theta(1) + T_\infty^{RMM}(C, m, \frac{n}{2})$  otherwise. Therefore, the critical path is

$$\begin{aligned}
T_\infty^{RMM}(C, m, n) &= \Theta(1) + T_\infty^{RMM}(C, m, \frac{n}{2}) \\
&= \Theta(\log \frac{n}{m}) + T_\infty^{CS}(C, m) \\
&= \Theta(\log \frac{n}{m}) + \frac{m}{\sqrt{C}} \log(m\sqrt{C}).
\end{aligned}$$

□

We now bound the amount of communication in `RectangularMatMult`. Although the bound seems complex, we actually need to use this result in only one very special case, which will allow us to simplify the expression.

LEMMA 4.2.3. *Let  $A$  be  $m$ -by- $m$  and let  $B$  and  $R$  be  $m$ -by- $n$ . The amount of communication in `RectangularMatMult`( $A, B, R$ ) is bounded by*

$$n \max(m, m^2/\sqrt{C}) + O\left(CP\left(\log \frac{n}{m} + \frac{m}{\sqrt{C}} \log(m\sqrt{C})\right)\right).$$

PROOF. The number of cache misses in a sequential execution is bounded by  $n/m$  times the number of cache misses in each call to `CilkSUMMA`. The number of cache misses in `CilkSUMMA` on matrices of size  $m$  is at most  $\max(m^2, m^3/\sqrt{C})$ , since even if  $C$  is large, we still have to read the arguments into the cache.

Therefore, the amount of communication in a parallel execution is bounded by

$$\begin{aligned} F_P^{RMM}(C, m, n) &= \frac{n}{m} \max(m^2, m^3/\sqrt{C}) + O(CPT_\infty^{RMM}(C, m, n)) \\ &= \frac{n}{m} \max(m^2, m^3/\sqrt{C}) + O\left(CP\left(\log \frac{n}{m} + \frac{m}{\sqrt{C}} \log(m\sqrt{C})\right)\right). \end{aligned}$$

□

### 4.3. Dynamic Cache-Size Control

The bound in Theorem 2.6.1 has two terms. In the first term,  $F_1(C)$ , larger caches normally lead to a tighter communication bound, which is intuitive. The second term,  $CPT_\infty$  causes larger caches to weaken the communication bound. This happens because the cost of flushing the caches rises. Randall [21] addresses this issue in two ways. First, he suggests that good parallel algorithms have short critical paths, so the  $CPT_\infty$  term should usually be small. This argument fails in many numerical-linear-algebra algorithms, which have long critical paths but which parallelize well thanks to the amount of work they must perform. In particular, triangular solvers and triangular factorization algorithms, which have  $\Omega(n)$  critical paths, parallelize well, but Randall's communication bounds for them are too loose. The second argument that Randall suggests is empirical: in his experiments, the actual amount of communication that can be attributed to the  $CPT_\infty$  is insignificant. While this empirical evidence is encouraging, we would like to have tighter provable bounds.

Our main observation is that most of the tasks along the (rather long) critical path in triangular solvers do not benefit from large caches. That is, the critical path is long, but most of the tasks along it perform little work on small amounts of data, and such tasks do not benefit from large caches. Consider square matrix multiplication: a data item is used at most  $n$  times, the dimension of the matrices, and caches of size  $n^2$  minimize the amount of communication. Larger caches do not reduce the  $F_1(C)$  term, but they do increase the  $CPT_\infty$  term.

This observation leads us to suggest a new feature in the Cilk run-time system. This feature allows us to temporarily instruct processors to use only part of their local caches to cache data.

**DEFINITION 4.3.1.** The programmer can set the *effective cache size* when calling a Cilk procedure. This effective size is registered in the activation frame of the newly-created procedure instance. When an effective cache size is specified in a procedure instance, it is inherited by all the procedures that it calls or spawns, unless a different cache size is explicitly set in the invocation of one of these descendant procedures. When a processor starts to execute a thread from an activation frame with new specified effective cache size, but its current effective cache size is larger, it flushes its local cache and limits its effective size to the specified size. When a child procedure returns the cache size returns to its parent effective cache size as is stored in the parent's activation frame.

Although the ability to limit the effective cache size allows us to reduce the effect of the  $CPT_\infty$  term, we need to account for the cost of the extra cache flush.

#### 4.4. Analysis of the New Solver with Dynamic Cache-Size Control

We are now ready to add cache-size control to `NewTriSolver`. Our aim is simple: to set the cache size to zero during calls to `VectorScale`, which does not benefit from the cache, and to set the cache size in `RectangularMatMult` to the minimum size that ensures optimal data reuse. In particular, when multiplying an  $m$ -by- $m$  matrix by an  $m$ -by- $n$  matrix, data is used at most  $m$  times, so a cache of size  $C = m^2$  ensures optimal data reuse. The complete algorithm is shown below.

```

cilk NewTriSolver(L, X, B)
{
  if (L is 1-by-1)
    SetCacheSize(0)
    call VectorScale(1/l11, X, B)
  else {

    partition L, B, and X as in Figure 4.1.2
    SetCacheSize(dim(L21)2)
    call NewTriSolver(L11, X1, B1)
    call RectangularMatMult(B2 = B2 - L21X1)
    call NewTriSolver(L22, X2, B2)
  }
}

```

The next lemma bounds the amount of communication that `RectangularMatMult` performs in the context of `NewTriSolver`, when it uses a cache of size at most  $m^2$ .

LEMMA 4.4.1. *The amount of communication that `RectangularMatMult`( $A, B, R$ ) performs with cache size at most  $m^2$  (the dimension of  $A$ ) is bounded by*

$$F_P^{RMM}(\min(C, m^2), m, n) \leq O\left(\frac{nm^2}{\sqrt{C}} + nm + m\sqrt{C}P \log(n\sqrt{C})\right).$$

PROOF. In general, the amount of communication is bounded by

$$n \max(m, m^2/\sqrt{C}) + O\left(CP \left(\log \frac{n}{m} + \frac{m}{\sqrt{C}} \log(m\sqrt{C})\right)\right).$$

Since  $\sqrt{C} \leq m$ ,  $m/\sqrt{C} \geq 1$ . We also have  $m \leq n$  so  $\log(m\sqrt{C}) \leq \log(n\sqrt{C})$ . Therefore,  $\log \frac{n}{m} + \frac{m}{\sqrt{C}} \log(m\sqrt{C}) = O\left(\frac{m}{\sqrt{C}} \log(n\sqrt{C})\right)$ . The result follows from this bound and from replacing the max by the sum of its arguments.  $\square$

We can use now Lemma 4.4.1 to bound the amount of communication in `NewTriSolver`. Since the algorithm is essentially sequential, we do not use Theorem 2.6.1 directly, so we do not need to know the length of the critical path. (We do analyze the critical path later in this section, but the critical-path length bound is not used to bound communication.)

The following theorem bounds the amount of communication with the additional communication cost for the flush performed when decreasing the cache size:

**THEOREM 4.4.2.** *The amount of communication  $F_P^{TS}(C, n)$  in *NewTriSolver*, using cache-size control, is bounded by  $O\left(\frac{n^3}{\sqrt{C}} + n\sqrt{C}P \log(n\sqrt{C}) \log n\right)$ .*

**PROOF.** *NewTriSolver* is essentially a sequential algorithm that calls parallel Cilk subroutines. Before calling a parallel subroutine, it performs a `SetCacheSize`, which sets the maximum cache size of the processor executing the algorithm. The other processors participating in the parallel sub-computation inherit this cache size when they steal work for executing the parallel sub-computation. This behavior eliminates interaction between parallel subroutines, in that it ensures that each parallel computation uses all  $P$  processors and that each parallel computation starts with specified cache size and in a specific order. This allows us to simply sum the communication upper bounds for the parallel subcomputations in order to derive an upper bound for *NewTriSolver* as a whole.

Let  $\varphi$  be a cost function that accounts for the communication cost incurred by cleaning the cache when changing the cache size from size  $m_1$  to size  $m_2$ , which is implemented by adding the command `SetCacheSize( $m_2$ )` to *NewTriSolver* algorithm, when the actual cache size is  $m_1$ .

$$\varphi(C, m_1, m_2) = \begin{cases} 0 & m_2 \geq C \text{ or } m_2 \geq m_1 \\ m_1 & m_2 < C \end{cases}$$

Thus,

$$F_p^{TS}(C, n) \leq \sum_{k=1}^{\phi_1} (\varphi(C, m_{k-1}, m_k) + F_p^{RRM}(C, m_k, n) + 3\varphi(C, m_k, m_{k-1})) + \phi_2 \cdot F_p^{VS}(C, n),$$

where  $\phi_1$  is the number of `SetCacheSize` phases and  $m_k$  is the problem size at each such phase, and  $\phi_2$  is the number of `VectorScale` phases. At each phase all the  $P$  processors have the same cache size. We count the communication incurred by reducing the cache, for each phase, only for one processor, since all the other processors before they steal work their cache is clean and therefore changing (reducing) their cache size does not include extra cache flush. Recall that the processor cache is not changed when it finishes the execution of a computation. The sequence of calls to `RectangularMatMult` creates a binary tree in which the current phase is partial problem size of the previous one, but it is done twice as many times. Notice that before `VectorScale` is called, the cache size is always changed to constant size (0) and that it is called exactly  $n$  times. Also notice that the factor  $\varphi(C, m_0, m_1) = 0$  since  $m_0$  is the initial phase, before program execution, where each processor cache is clean and the factor  $\varphi(C, m_k, m_{k-1}) = 0$ , since no communication is performed for not changing or increasing the cache size if  $m_k \leq m_{k-1}$ . Therefore,

$$\begin{aligned}
F_P^{TS}(C, n) &\leq \sum_{k=1}^{\log n} 2^{k-1} \left[ \varphi \left( C, \left( \frac{n}{2^{k-1}} \right)^2, \left( \frac{n}{2^k} \right)^2 \right) + F_P^{RMM} \left( C, \frac{n}{2^k}, n \right) + 0 \right] \\
&\quad + n \cdot F_P^{VS}(C, n) \\
&= O \left( \sum_{k=1}^{\log n} \left[ \frac{n^2}{2^{k-1}} + 2^{k-1} \left( \frac{n^3}{2^{2k}\sqrt{C}} + \frac{n^2}{2^k} + \frac{n}{2^k} \sqrt{C} P \log(n\sqrt{C}) \right) \right] + n^2 \right) \\
&= O \left( n^2 \left( \sum_{k=1}^{\log n} \frac{1}{2^{k-1}} \right) + \frac{n^3}{2\sqrt{C}} \left( \sum_{k=1}^{\log n} 2^{-k} \right) + \frac{n^2}{2} \left( \sum_{k=1}^{\log n} 1 \right) \right) \\
&\quad + O \left( \frac{n}{2} \sqrt{C} P \log(n\sqrt{C}) \left( \sum_{k=1}^{\log n} 1 \right) + n^2 \right) \\
&= O \left( n^2 + \frac{n^3}{\sqrt{C}} + n^2 \log n + n\sqrt{C} P \log(n\sqrt{C}) \log n + n^2 \right) \\
&= O \left( \frac{n^3}{\sqrt{C}} + n^2 \log n + n\sqrt{C} P \log(n\sqrt{C}) \log n \right).
\end{aligned}$$

if  $C < (\frac{n}{\log n})^2$  then  $n^2 \log n = O(\frac{n^3}{\sqrt{C}})$ , else  $n^2 \log n = O(n\sqrt{C} P \log(n\sqrt{C}) \log n)$ , therefore  $F_P^{TS}(C, n) = O \left( \frac{n^3}{\sqrt{C}} + n\sqrt{C} P \log(n\sqrt{C}) \log n \right)$ .  $\square$

We now bound the amount of work and parallelism in **NewTriSolver**.

**THEOREM 4.4.3.** *NewTriSolver performs  $\Theta(n^3)$  work when its arguments are all  $n$ -by- $n$ , and the length of its critical path when the cache-size-control feature is used is  $O(n \log(n\sqrt{C}) \log n)$ .*

**PROOF.** The work for **NewTriSolver** is expressed by the recurrence

$$\begin{aligned}
T_1^{TS}(n, n) &= 2T_1^{TS}(\frac{n}{2}, n) + T_1^{RMM}(\frac{n}{2}, n) \\
&= 2^{\log n} n + \Theta(n^3) = \Theta(n^3),
\end{aligned}$$

since **VectorScale** performs  $\Theta(n)$  work. The length of the critical path satisfies

$$T_\infty^{TS}(C, m, n) = \begin{cases} \log n & m = 1 \\ 2T_\infty^{TS}(C, \frac{m}{2}, n) + T_\infty^{RMM}(\frac{m^2}{4}, \frac{m}{2}, n) & 1 \leq \frac{m}{2} \leq \sqrt{C} \\ 2T_\infty^{TS}(C, \frac{m}{2}, n) + T_\infty^{RMM}(C, \frac{m}{2}, n) & \sqrt{C} < \frac{m}{2}. \end{cases}$$

Recall that

$$T_\infty^{RMM}(\min(C, m^2), m, n) = O \left( \log \frac{n}{m} + \frac{m}{\min(\sqrt{C}, m)} \log \left( m \min(\sqrt{C}, m) \right) \right).$$

Therefore,

$$\begin{aligned}
T_{\infty}^{TS}(C, n, n) &= 2T_{\infty}^{TS}(C, \frac{n}{2}, n) + T_{\infty}^{RMM}(\min(C, \frac{n^2}{4}), \frac{n}{2}, n) \\
&= O\left(n \log n + \sum_{k=1}^{\log n} 2^{k-1} \left(\log \frac{n}{n/2^k} + \frac{n/2^k}{\min(\sqrt{C}, n/2^k)} \log(n\sqrt{C})\right)\right) \\
&= O\left(n \log n + \frac{n}{2} \log(n\sqrt{C}) \sum_{k=1}^{\log n} \frac{1}{\min(\sqrt{C}, n/2^k)}\right) \\
&= O\left(n \log n + n \log(n\sqrt{C}) \log n\right). \\
&= O\left(n \log(n\sqrt{C}) \log n\right).
\end{aligned}$$

□

`NewTriSolver` performs less communication than Randall's `RecursiveTriSolver` [21] when  $\sqrt{C} > \log(n\sqrt{C})$ . In particular, for  $C = O(\frac{n^2}{P})$  the new algorithm performs  $O(n^2 \sqrt{P} \log^2 n)$  communication, a factor of  $\frac{n}{\sqrt{P} \log n}$  less than `RecursiveTriSolver`. The critical path of the new algorithm, however, is slightly longer than that of `RecursiveTriSolver`. The new algorithm uses  $O(n^2)$  space, since no external space is allocated by the algorithm and the additional space taken by the run time system activation frames is  $O(Pn \log(n\sqrt{C}) \log n)$ .

## LU Decomposition

In this chapter we describe a communication-efficient  $LU$  decomposition algorithm. A general linear system  $Ax = b$  can often be solved by factoring  $A$  into a product of triangular factors  $A = LU$ , where  $L$  is lower triangular and  $U$  is upper triangular. Not all matrices have such a decomposition, but many classes of matrices do, such as diagonally-dominant matrices and symmetric positive-definite matrices. In this chapter we assume that  $A$  has an  $LU$  decomposition. Once the matrix has been factored, we can solve the linear system using forward and backward substitution, that is, by solving  $Ly = b$  for  $y$  and then  $Ux = y$  for  $x$ .

To achieve a high level of data reuse in a sequential factorization, the matrix must be factored in blocks or recursively (see [24, 25] and the references therein). A more conventional factorization by row or by column performs  $\Theta(n^3)$  cache misses when the dimension  $n$  of the matrix is twice  $\sqrt{C}$  or larger. Since factorization algorithms perform  $\Theta(n^3)$  work, it follows that data reuse in the cache in factorizations by row or by column is limited to a constant. Recursive factorizations, on the other hand, perform only  $\Theta(n^3/\sqrt{C})$  cache misses.

Like triangular linear solvers,  $LU$  factorizations have long critical paths,  $\Theta(n)$  or longer. Randall analyzed a straightforward recursive factorization in Cilk [21, section 4.1.2]. He used the formulation shown in Figure 5.0.1, which partitions  $A$ ,  $L$ , and  $U$  into 2-by-2 block matrices, all square or nearly square. The algorithm begins by factoring  $A_{11} = L_{11}U_{11}$ , then solves in parallel for the off-diagonal blocks of  $L$  and  $U$  using the equations  $L_{21}U_{11} = A_{21}$  and  $L_{11}U_{12} = A_{12}$ , updates the remaining equations,  $\hat{A}_{22} = A_{22} - L_{21}U_{12}$ , and factors  $\hat{A}_{22} = L_{22}U_{22}$ . The performance characteristics of this algorithm depend, of course, on the specific triangular solver and matrix multiplication subroutines. Randall's choices for these subroutines lead to an algorithm with overall critical path  $T_\infty(n) = \Theta(n \log^2 n)$ . When  $C = \frac{n^2}{P}$ , the  $CPT_\infty$  term in Theorem 2.6.1 causes the communication bound to reach  $\Theta(n^3 \log^2 n)$ . This is a meaningless bound, since a Cilk algorithm never performs more communication than work. (Randall does show that this algorithm, like most matrix algorithm, can be arranged to achieve good spatial locality, but he does not show a meaningful temporal locality bound.)

We propose a better Cilk  $LU$ -decomposition algorithm that performs only  $O(\sqrt{P}n^2 \log^3 n)$  communication. Our algorithm differs from Randall's in several ways. The most important difference is that we rely on our

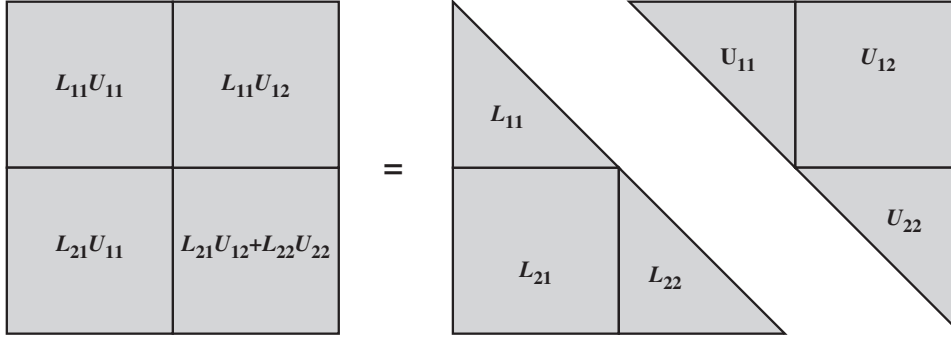


FIGURE 5.0.1. A divide-and-conquer algorithm for LU decomposition.

communication-efficient triangular solver, which was described in the previous chapter. To allow the triangular solver to control cache sizes without interference, we perform the two triangular solves in the algorithm sequentially, not in parallel (the work of the algorithm is  $\Theta(n^3)$ ). It turns out that this lengthens the critical path, but not by much. We also use our communication- and space-efficient matrix multiplier, `CilkSUMMA`. The pseudo code of the algorithm follows.

```

cilk LUD(A,L,U,n)
{
  call LUD(A11,L11,U11)
  call NewTriSolver(L11,U12,A12)
  call NewTriSolver(U11,L21,A21)
  call CilkSUMMA(L21,U12, $\hat{A}_{22}$ , $\frac{n}{2}$ )
  call LUD( $\hat{A}_{22}$ ,L22,U22)
}

```

To bound the amount of communication, we do not use Theorem 2.6.1 directly (we do not use the critical-path length to bound communication but, we do analyze the critical path later in this section). Since there is no parallel execution of different procedures on the same time, it is possible to count the number of cache misses by summing the cache misses incurred along any phase while executing `NewTriSolver` or `CilkSUMMA`.

**THEOREM 5.0.4.** *The amount of communication  $F_P^{LUD}(C,n)$  in LUD when solving a problem of size  $n$  is  $O(\frac{n^3}{\sqrt{C}} + n\sqrt{C}P \log^2 n \log(n\sqrt{C}))$ .*

**PROOF.** We count the amount of communication by phases. There are two `NewTriSolver` phases and one `CilkSUMMA` phase, which lead to the

following recurrence:

$$\begin{aligned}
F_P^{LUD}(C, n) &= 2F_P^{LUD}(C, n/2) + 2F_P^{TS}(C, n/2) + F_P^{CS}(C, n/2) + \Theta(1) \\
&= 2F_P^{LUD}(C, n/2) + O\left(\frac{n^3}{\sqrt{C}} + n\sqrt{C}P \log(n\sqrt{C}) \log n\right) \\
&\quad + O\left(\frac{n^3}{\sqrt{C}} + n\sqrt{C}P \log(n\sqrt{C})\right) + \Theta(1) \\
&= 2F_P^{LUD}(C, n/2) + O\left(\frac{n^3}{\sqrt{C}} + n\sqrt{C}P \log(n\sqrt{C}) \log n\right) + \Theta(1) \\
&= \sum_{k=1}^{\log n} 2^k \cdot \left[ \frac{\left(\frac{n}{2^k}\right)^3}{\sqrt{C}} + \frac{n}{2^k} \sqrt{C} P \log\left(\frac{n\sqrt{C}}{2^k}\right) \log \frac{n}{2^k} \right] \\
&= \frac{n^3}{\sqrt{C}} \sum_{k=1}^{\log n} 4^{-k} + n\sqrt{C}P \sum_{k=1}^{\log n} \log\left(\frac{n\sqrt{C}}{2^k}\right) \log \frac{n}{2^k} \\
&= O\left(\frac{n^3}{\sqrt{C}} + n\sqrt{C}P \log^2 n \log(n\sqrt{C})\right).
\end{aligned}$$

□

**THEOREM 5.0.5.** *The critical path length of LUD when its arguments are all  $n$ -by- $n$  is  $O(n \log^2(n\sqrt{C}) \log n)$ .*

**PROOF.** The critical path of LUD depends on the critical path of the `NewTriSolver` algorithm to solve triangular systems, and of `CilkSUMMA` for matrix multiplication, so the critical path is of length

$$T_\infty^{LUD}(C, n) = 2T_\infty^{LUD}(n/2) + 2T_\infty^{TS}(C, n/2, n/2) + T_\infty^{CS}(C, n/2) + \Theta(1)$$

Recall that  $T_\infty^{CS}(C, n) = \frac{n}{\sqrt{C}} \log(n\sqrt{C})$  and from Theorem 4.4.3 that  $T_\infty^{TS}(C, n, n) = O(n \log(n\sqrt{C}) \log n)$ . Therefore,

$$\begin{aligned}
T_\infty^{LUD}(C, n) &= 2T_\infty^{LUD}(n/2) + 2T_\infty^{TS}(C, n/2, n/2) + T_\infty^{CS}(C, n/2) + \Theta(1) \\
&= O\left(2^{\log n} + \sum_{k=1}^{\log n} 2^k \cdot \left(\frac{n}{2^k} \log^2\left(\frac{n}{2^k} \sqrt{C}\right)\right) + \sum_{k=1}^{\log n} 2^{k-1} \left(\frac{n}{2^k \sqrt{C}} \log\left(\frac{n}{2^k} \sqrt{C}\right)\right)\right) \\
&= O\left(n + n \log^2(n\sqrt{C}) \log n + \frac{n}{\sqrt{C}} \log(n\sqrt{C}) \log n\right) \\
&= O\left(n \log^2(n\sqrt{C}) \log n\right).
\end{aligned}$$

□

The amount of communication incurred in LUD for  $C = \frac{n^2}{P}$  is  $O\left(\sqrt{P} n^2 \log^3 n\right)$  which for large  $n$  is much tighter than the  $O(n^3 \log^2 n)$  of the LU decomposition algorithm shown in [21]. The parallelism of LUD algorithm is  $\frac{T_\infty^{LUD}}{T_\infty^{LUD}} = \Omega\left(\frac{n^3}{n \log^3 n}\right) = \Omega\left(\frac{n^2}{\log^3 n}\right)$  which is factor of  $\log n$  smaller than the implementation presented in [21].

## CHAPTER 6

### Conclusion and Open Problems

Cilk's dag consistency employs relaxed consistency model in order to realize performance gains, but unlike dag consistency most distributed shared memories take a low level view of parallel programs and can not give analytical performance bound. In this thesis we used the analytical tools of Cilk to design algorithms with tighter communication bounds for existing dense matrix multiplication, triangular solver and LU factorization algorithms, than the bounds obtained by [21].

Several experimental versions, such as Cilk-3 with the implementation of BACKER coherence algorithm were developed for the runtime system of the Connection Machine Model CM5 parallel super computer. However, official distribution of Cilk version with dag-consistent shared memory was never released and therefore it was not feasible to implement the above algorithms for distributed-shared memory environment.

We leave it as an open question whether it is possible to tighten the bounds on the number of communication and memory requirements for a factor of  $\log n$  than the existing bounds, without compromising the other performance parameters and whether the dynamic cache size control property is required for obtaining such low communication bounds.

## Bibliography

- [1] R. C. Agarwal, S. M. Balle, F. G. Gustavson, M. Joshi and P. Palkar. A Three-Dimensional Approach to Parallel Matrix Multiplication. *IBM Journal of Research and Development*, 39:575-582, 1995.
- [2] Alok Aggarwal, Ashok K. Chandra and Marc Snir. Communication Complexity of PRAMs. *Theoretical Computer Science*, 71:3-28, 1990.
- [3] J. Berntsen. Communication efficient matrix multiplication on hypercubes. *Parallel Computing*, 12:335-342, 1989.
- [4] Robert D. Blumof. Executing Multithreaded Programs Efficiently. Phd thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology. September 1995.
- [5] Robert D. Blumof and Charles 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, November 1994.
- [6] Robert D. Blumof, Matteo Frigo, Christopher F. Joerg, Charles E. Leiserson, and Keith H. Randall. Dag-consistent distributed shared memory. In *Proceedings of the 10th International Parallel Processing Symposium (IPPS)*, pages 132-141, Honolulu, Hawaii, April 1996.
- [7] L.E. Cannon. A cellular computer to implement the Kalman Filter Algorithm. Technical report, Ph.D. Thesis, Montana State University, 1969.
- [8] E. Dekel, D. Nassimi, and S. Sahni. Parallel matrix and graph algorithms. *SIAM Journal of Computing*, 10:657-673, 1981.
- [9] Michel Dubois, Christoph Schurich, and Faye Briggs. Memory access buffering in multiprocessors. In *Proceedings of the 13th Annual International Symposium on Computer Architecture*, pages 434-442, June 1986.
- [10] G.C. Fox, S.W. Otto, and A.J.G. Hey, Matrix algorithms on a hypercube I: *Matrix multiplication* *Parallel Computing*, 4:17-31, 1987.
- [11] Guang R. Gao and Vivek Sarkar. Location consistency: Stepping beyond the memory coherence barrier. In *proceedings of the 24th International Conference on Parallel Processing*, Aconomowoc, Wisconsin, August 1995.
- [12] Kourosh Gharachorloo, Daniel Lenoski, James Laudon, Phillip Gibbons, Anoop Gupta, and John Hennessy. Memory consistency and event ordering in scalable shared-memory multiprocessors. In *Proceedings of the 17th Annual International Symposium on Computer Architecture (ISCA)*, pages 15-26, Seattle, Washington, June 1990.
- [13] Anshul Gupta and Vipin Kumar. The Scalability of Matrix Multiplication Algorithms on Parallel Computers. Department of Computer Science, University of Minnesota, 1991. Available on the Internet from <ftp://ftp.cs.umn.edu/users/kumar/matrix.ps>.
- [14] H. Gupta and P. Sadayappan. Communication efficient matrix multiplication on hypercubes. In *Proceedings of the 6th Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA94)*, pages 320-329, June 1994.
- [15] Dror Irony and Sivan Toledo. Trading replication for communication in parallel distributed-memory dense solvers. Submitted to *Parallel Processing Letters*, July 2001.
- [16] Dror Irony and Sivan Toledo. Communication lower bounds for distributed-memory matrix multiplication. Submitted to *Journal of Parallel and Distributed Computing*, April 2001.

- [17] S. Lennart Johnsson. Minimizing the communication time for matrix multiplication on multiprocessors. *Parallel Computing*, 19:1235–1257, 1993.
- [18] C.T. Ho, S.L. Johnsson and A. Edelman. Matrix multiplication on hypercubes using full bandwidth and constant storage. In *Proceeding of the Sixth Distributed Memory Computing Conference*, 447-451,1991.
- [19] Leslie Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Transactions on Computers*, C-28(9):690-691, September 1979.
- [20] Charles E. Leiserson and Harald Prokop. A Minicourse on Multithreaded Programming. MIT Laboratory for Computer Science, Cambridge, Massachusetts, July 1998.
- [21] Keith H. Randall. Cilk:Efficient Multithreaded Computing. Phd thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology. May 1998.
- [22] Cilk-5.3.1 Reference Manual. Supercomputing Technologies Group. MIT Laboratory for Computer Science. June 2000. Available on the Internet from <http://supertech.lcs.mit.edu/Cilk>
- [23] Robert van de Geijn and Jerrell Watts. SUMMA: scalable universal matrix multiplication algorithm. *Concurrency: Practice and Experience*, 9(1997):255–274.
- [24] Sivan Toledo. Locality of reference in LU decomposition with partial pivoting, *SIAM Journal on Matrix Analysis and Applications*, 18(1997):1065–1081.
- [25] Sivan Toledo. A survey of out-of-core algorithms in numerical linear algebra. in *External Memory Algorithms*, James M. Abello and Jeffrey Scott Vitter, eds., DIMACS Series in Discrete Mathematics and Theoretical Computer Science, American Mathematical Society, 1999, pages 161–179.
- [26] I-Chen Wu and H. T. Kong. Communication Complexity for Parallel Divide-and-Conquer. School of Computer Science. Camegie Mellon University, Pittsburgh.