TEL-AVIV UNIVERSITY

RAYMOND AND BEVERLY SACKLER FACULTY OF
EXACT SCIENCES

SCHOOL OF COMPUTER SCIENCE

Cilk on CC-NUMA Machines

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

Eitan Ben Amos

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

July 2006

Abstract

This thesis shows how to improve the performance of Cilk Programs when executed on CC-NUMA Distributed Shared Memory Machines. The work was carried out on a SGI Origin 2000 Machine (highly scalable CC-NUMA DSM machine). The thesis also presents improvements to the `cilk2c` translator to generate code that is compatible with ANSI C. this makes it possible to use Cilk with compilers other than `gcc` which promise better binary code generation, compatibility with existing vendor libraries and the use of vendor specific features by the Cilk program. Completion of this phase enabled to compile the Cilk example programs which are available in the Cilk distribution package with both the MIPS-Pro compiler and the Intel C Compiler. The Mips-Pro compiler shortened the running time of the example programs by 10-30 percent. The Intel compiler shortened the running times by 20% on average with a large range of changes.

A later phase introduced a new scheduler to the Cilk runtime. This scheduler is aware of the NUMA architecture and enables optimization not available in the original work-stealing scheduler such as preferring to steal work from physically nearby CPUs over distant CPUs and enabling placement of memory buffers on specific physical nodes. The later one creates the potential for new optimization and research in the field of data placement of parallel algorithms that was not addressed in this thesis.

TABLE OF CONTENTS

Chapter 1

INTRODUCTION

Traditionally, software has been written for ***serial*** computation. A program was executed by a single computer having a single CPU – only one instruction may be executed at any moment in time.

Parallel computing is the simultaneous use of multiple compute resources to solve a computational problem. Compute resources can include a single computer with multiple CPUs, multiple computers connected over a network or any combination of these forms.

The purpose of parallel processing is to perform the computation 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 traditional applications such as weather and climate modeling, chemical physical and biological modeling, as well as commercial applications such as parallel databases, data mining, oil exploration, web search engines and computer-aided design of drugs.

Parallel computers can be classified according to variety of architectural characteristics. However, most of the existing machines can be broadly grouped into two classes: machines with shared memory architectures (such as the SGI Origin and commodity Intel Pentium based machines) and machines with distributed-memory architecture (such as the IBM SP systems and clusters of workstations and servers).

In shared memory machines, processors communicate with one another by writing and reading regions of memory that are accessible by all the processors. In distributed memory architectures, processors communicate with one another by sending and receiving messages which are sent over a communication channel.

The Cilk language (version 5.3 was used for this thesis) was designed for the shared memory architecture [1]. Cilk is a language for multithreaded parallel programming that is based on ANSI C. Cilk is designed for general-purpose parallel programming. It is especially effective for exploiting dynamic, highly asynchronous parallelism, which can be difficult to exploit in data-parallel or message-passing style. Unlike many other multithreaded programming systems, Cilk is algorithmic in that the runtime system employs a scheduler that allows the performance of programs to be estimated accurately [2] based on abstract complexity measures. Chapter two further describes the details of the Cilk language and its work stealing scheduler.

1.1 CONTRIBUTION OF THE THESIS

The first contribution of this thesis is in making the Cilk distribution (*`cilk2c`* translator, runtime library and compiler driver) compliant with the ANSI C standard. The code was enhanced in areas that were not portable from the `gcc` compiler to the native compiler of the platform, the SGI MIPS Pro compiler suit (version 7.4) - this work also makes it easier to port the code to other

ANSI C compliant compilers (as attempted later on with the Intel C Compiler).

Having the runtime compile with the native compiler showed improved performance even before optimizing the code. Analysis of the performance gain of using the native compiler of the platform and using the vendor supplied BLAS library can be seen in [3]. In that article, the authors have criticized the intimate connections between Cilk (version 5.2.1) and gcc, and specifically the limitation on linking non-Cilk libraries. Being unable to link Cilk programs with vendor supplied libraries forced them to code the low level BLAS routines that were used as the building blocks of the implemented algorithms. They made some tests to measure the performance impact of using gcc instead of the vendor compiler and also of coding the low-level BLAS routines instead of using the vendor library (Sun's perflib library). They measured that the lack of native BLAS costs a factor of 1.2–1.4, while the switch to gcc increases the costs to a factor of 1.5–1.9 (accumulated). This is an example of the value of native BLAS library and compiler. On newly introduced processors (e.g., Intel IA-64) the gain might be considerably higher since the vendor compiler is highly optimized while gcc needs time to digest the new processor architecture and features.

The second contribution is a new scheduler that is aware of the additional hierarchies which are introduced by the NUMA architecture. It attempts to take these into consideration when scheduling the tasks of a Cilk program. The scheduler was implemented for the SGI Irix operating system using Origin 2000 hardware. It uses advanced operating system features that are specific for this operating system and hence requires the ability to compile and link the Cilk programs with the vendors' native compiler and libraries. These libraries offer sophisticated features such as binding threads to processors, management of memory placement over the distributed shared physical memory and setting memory allocation policies.

The scheduler that was developed in order to better support NUMA architecture maintains this algorithmic feature which is important when it comes to analyzing the work, critical path and parallelism.

1.2 OUTLINE OF THIS THESIS

The rest of the thesis is organized as follows. In Chapter 2 I present Cilk. This chapter explains the language syntax, the transformation into ANSI C code and the tools used to compile and link Cilk programs. In chapter 3 I present the work done to make Cilk compile using the MIPS Pro compiler suit (the compiler from SGI for its Origin distributed shared memory machines). Compiling the examples that come with Cilk with this compiler shows improved performance for almost all of them. It shows that support for the vendor compiler can improve the performance of the final code considerably and has value on its own even before turning to sophisticated machine specific techniques such as the NUMA architecture. To further validate the portability of Cilk I have also tested with the Intel C Compiler for Linux. These tests also showed considerable performance improvement which further justifies the need to support non `gcc` compiler. In chapter 4 I present the NUMA aware scheduler that was developed as part of this thesis. The chapter first presents the idea behind the scheduler and then continues to present the techniques that were used in order to implement the scheduler and verify that it is working as expected. In chapter 5 I present the problem of memory placement and how it can be addressed with explicit memory placement - this ends with scalability tests of matrix multiplication and cholesky factorization codes. In chapter 6 I present some test programs that were used to understand the behavior of the Irix operating system and its accompanying libraries. Chapter 7 wraps up with conclusions of the achievements of this work.

Chapter 2

BACKGROUND AND RELATED WORK

The following chapter presents an overview of the Cilk language (version 5.3) together with its tools [1]. This is not a complete reference of the language as such a reference is available through [1].

2.1 CILK OVERVIEW AND BACKGROUND

Cilk 5.3 is a language for multithreaded parallel programming based on ANSI C. Cilk is designed for general-purpose parallel programming, but is especially effective for exploiting dynamic, asynchronous parallelism. Unlike many other multithreaded programming systems, Cilk is algorithmic in that the runtime system employs a scheduler that allows the performance of programs to be estimated accurately [2] based on abstract complexity measures.

Cilk is made up of the Cilk runtime system and the Cilk compiler. Prior to the Cilk development made during this thesis, Cilk was intended to run on Unix-like systems, provided that `gcc`, POSIX threads, and GNU make are available. The philosophy behind Cilk is that a programmer should concentrate on structuring the program to expose parallelism and exploit locality, leaving the runtime system with the responsibility of scheduling the computation to run efficiently on a given platform. Cilk has a runtime system that takes care of details like load balancing, and communication protocols. An exceptional advantage of Cilk over other multithreaded languages is that Cilk is algorithmic in that the runtime system guarantees efficient and predictable performance.

The theory behind Cilk is based on research of scheduling multithreaded computations, and especially of the performance of work-stealing. The work stealing model has been a focal point in the development of Cilk. The results from this research led to the development of a performance model that predicts the efficiency of a Cilk program using two simple parameters: work and critical-path length [2, 4, 5]. Beginning with version 3, Cilk features an implementation of dag consistent distributed shared memory [6, 7].

Informally, a dag consistent memory model for a deterministic programs means that thread A reading from memory can observe the value written by thread B only if there is a serial execution order that is consistent with the DAG and in which the read made by A can observe the write made by B.

2.2 PROGRAMMING WITH CILK

The basic Cilk language consists of C with additional three keywords to indicate parallelism and synchronization. When a Cilk program is executed on a single processor it has the same semantics as the C program that results when the Cilk keywords are deleted. This C program is named the `serial elision` or `C elision` of the Cilk program. A very simple example of a Cilk program is a recursive program to compute the *n*th Fibonacci number. A C program to compute the *n*th Fibonacci number is shown in Figure 2.1(a). The Cilk program to compute the *n*th Fibonacci number in parallel is seen in Figure 2.1(b). Pay attention to the great similarity between the 2 programs. In fact, the only differences between them are the inclusion of the library header file "cilk.h" and the 3 Cilk keywords (`cilk, spawn`, and `sync`) that expose the parallelism of the code.

```
#include <stdlib.h>
#include <stdio.h>

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


int main (int argc, char
*argv[])
{
  int n, result;
  n = atoi(argv[1]);
  result = fib (n);
  printf ("Result: %d\n",
result);
  return 0;
}
```

Figure 2.1(a): A serial C program to compute the *n*th Fibonacci number.

```
#include <stdlib.h>
#include <stdio.h>
#include <cilk.h>

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);
  }
}

cilk int main (int argc,
char *argv[])
{
  int n, result;
  n = atoi(argv[1]);
  result = spawn fib(n);
  sync;
  printf ("Result: %d\n",
result);
  return 0;
}
```

Figure 2.1(b): A parallel Cilk program to compute the *n*th Fibonacci number.

Figure 2.1: A Comparison of Cilk vs. C code to compute the requested Fibonacci number as described in [1]. Notice the similarity between the two programs.

The keyword `cilk` identifies a Cilk procedure, which is the parallel equivalent of a C function. A Cilk procedure may spawn sub-procedures in parallel and synchronize (wait) for their completion. A Cilk procedure definition has an argument list and body, both identical to that of a C function.

Most of the work in a Cilk procedure is executed serially, just like in C, but parallelism is created when the invocation of a Cilk procedure is preceded by the `spawn` keyword. A spawn is the parallel equivalent of a C function call and just like a C function call, when a Cilk procedure is spawned, execution proceeds to the child. However, unlike a C function call in which the parent is not resumed until its child returns, in Cilk, the parent can continue to execute in parallel with the child. In fact, the parent can continue to spawn off children, producing a high degree of parallelism. Scheduling the spawned procedures on the processors of the parallel computer is the responsibility of the Cilk scheduler.

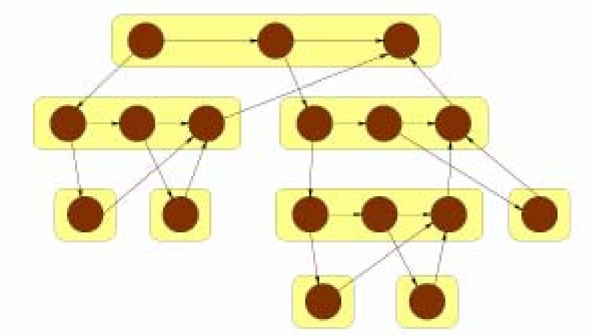

Figure 2.2: The Cilk model of multithreaded computation as described in [1]. Each procedure, shown as a rounded rectangle, is broken into sequences of threads, shown as circles. A downward edge indicates the spawning of a sub-procedure. A horizontal edge indicates the continuation to a successor thread. An upward edge indicates the returning of a value to a parent procedure. All three types of edges are dependencies which constrain the order in which threads may be scheduled.

A Cilk procedure cannot safely use the return values of the children it has spawned until it executes a `sync` statement. A `sync` statement will cause the procedure to suspend until all of its children have completed. In case all of the children of a procedure have already completed then it can continue execution immediately with very little overhead. The procedure resumes/continues its execution at the point following the `sync` statement. The `sync` statement is thus a local barrier in the sense that it waits only for the spawned children of the procedure to complete rather than waiting for all procedures currently executing. In the Fibonacci example, a `sync` statement is required before the statement "`return (x+y)`", to prevent the summation of `x` and `y` before they are both computed. A Cilk programmer uses `spawn` and `sync` keywords to expose the parallelism in a program, and the Cilk runtime system takes the responsibility of scheduling the procedures efficiently.

A Cilk program execution can be visualized as a directed acyclic graph as is illustrated in Figure 2.2. A Cilk program execution consists of a collection of procedure instances, each is made of a sequence of non-blocking threads. In Cilk, a thread is the longest sequence of instructions that ends with a spawn, sync, or return (either explicit or implicit) statement. The first thread that is executed upon procedure activation is the procedure's initial thread. The rest of the threads that make up the procedure are successor threads. At runtime, the binary `spawn` relation causes procedure instances to be structured as a rooted tree.

A correct execution of a Cilk program must obey all the dependencies that are visualized by the DAG. Some scheduling restrictions are implemented by the Cilk scheduler while others must be handled by the Cilk programmer.

2.3 COMPILING AND RUNNING CILK PROGRAMS

The Cilk 5.3 distribution installs the cilk command, which is a special version of the gcc compiler. More accurately the command changes the gcc compiler driver to accept new command line switches, activate the cilk2c translator and chain the various tools to compile a Cilk file (preprocessing, cilk2c translation, compiling and linking). cilk accepts the same arguments as the gcc compiler and in addition some new command line switches to control Cilk specific issue. Compiling the source code for the Fibonacci example from figure 2.1(b) requires the following command line (all other gcc switches such as –g –W, etc. can be used as well):

```
> cilk –O3 fib.cilk -o fib
```

In order to execute the program do the following:

```
> ./fib --nproc 8 100
```

This executes the program using 8 worker threads and output the value of the 100th Fibonacci number.

During development, this command was replaced with a driver program named cilkc (or cilkclocal). This driver parses the command line switches and decides whether the file requires standard compilation (files with extensions of ".c", ".o", ".s", etc) or that it is a file with ".cilk" extension in which case it requires the cilk2c translator in order to transform it to C code that will be compiled in later phases to produce an object file.

Cilk related compilation switches compile the code so that it collects statistics about the execution of the program (such as the maximum depth of the DAG and the amount of work in the DAG). These statistics can then be used by the developer to understand the bottlenecks in the program and enhance the program in order to achieve better performance and scalability. Various levels of statistics exist and the more statistics being collected the more it affects the program behavior.

2.4 SHARED MEMORY MODEL

Cilk supports the shared memory model. Sharing occurs when 2 different procedure instances that execute in parallel (be it instances of the same procedure or not) access a global variable or indirectly by passing a pointer to spawned procedures, allowing multiple procedures to access the (possibly stack based) object that is addressed by the pointer. Accessing shared objects in parallel can cause nondeterministic anomalies. Consequently, it is important to understand the basic semantics of shared memory model used by Cilk.

Figure 2.4 shows two Cilk procedures, *foo()* and *bar()*. The procedure *foo()* passes the variable *x* to the procedure *bar()* by reference, and then *foo()* proceeds to read *x* before the *sync*. The procedure *bar()* may be scheduled concurrently and in that case it reads *x* through the pointer *px*. Sharing the value of *x* in this way is safe because the shared variable *x* is assigned in *foo()* before *bar()* is spawned, and no write accesses happen on *x* thereafter.

Figure 2.5 shows a modified version of the Cilk procedures in that we saw in Figure 2.4. Here, the procedure *foo()* passes the variable *x* to the procedure *bar()* by reference, but now *foo()* proceeds to modify *x* before the *sync*. As a result, it is not clear what value will be seen when the procedure *bar()* reads *x* through pointer *px*: The value at the time the variable was passed, the value after *foo()* has modified *x*, or something else (in case the update of *x* is non atomic). In addition, it is not clear which value of *x* will the procedure *foo()* see because the procedure *bar()* might have already increased it. This situation (called a data race) causes a nondeterministic behavior of the program. In most cases, non-determinism of this sort represents a programming error.

```
cilk int foo (void)
{
  int x = 0, y;
  spawn bar(&x);
  y = x + 1;
  sync;
  return (y);
}
cilk void bar (int *px)
{
  printf("%d", *px + 1);
  return;
}
```

Figure 2.4: Passing the spawned procedure bar an argument consisting of a pointer to the variable x leads to the sharing of x. This code was taken from [1].

```
cilk int foo (void)
{
  int x = 0;
  spawn bar(&x);
  x = x + 1;
  sync;
  return (x);
}
cilk void bar (int *px)
{
  *px = *px + 1;
  return;
}
```

Figure 2.5: Nondeterministic behavior may result from shared access to the variable x when x is updated. This code was taken from [1].

2.5 INLETS

In most cases, when a spawned procedure returns a value, it is simply stored into a variable in the frame of its parent, as in the following example:

```
x = spawn foo(y);
```

But in some occasions, you need a more complex method of incorporating the returned value into the parent's frame. Cilk provides the `inlet` feature for this purpose.

An `inlet` is essentially a C function that is defined inside a Cilk procedure (similar to an inner function). Up till now we have see that Cilk requires a separate statement in order to spawn a Cilk procedure, as is seen above. But when the spawn is performed as an argument to an `inlet` call then an exception to this rule is allowed. In this case, the procedure is spawned, and upon its return, the `inlet` is invoked with the result that was returned from the child procedure. In the meantime, control of the parent procedure proceeds to the statement that follows the inlet.

```
cilk int fib (int n)
{
  int x = 0;

  inlet void sum (int result)
  {
    x += result;
    return;
  }

  if (n<2) {
    return n;
  } else {
    sum(spawn fib (n-1));
    sum(spawn fib (n-2));
    sync;
    return (x);
  }
}
```

Figure 2.6: Using an inlet to compute the n^{th} Fibonacci number as it appears in [1].

Figure 2.6 illustrates how to code the *fib()* function using inlets. The inlet *sum()* is defined to take the parameter *result* (returned by a returning Cilk procedure call) and add it to the variable *x* that is defined in the frame of the containing procedure, *fib()*. The variables of *fib()* are accessible within *sum()*, because *sum()* is an internal function of *fib()*.

Because an `inlet` is very similar to a C function it also has similar restrictions to that of C functions. These prevent it from containing `spawn` and `sync` statements making the inlet consist of a single Cilk thread. However, it is not entirely the same as a C function because Cilk provides special statements that can only be executed inside an inlet.

It may happen that an inlet is operating on the variables of a procedure frame concurrently with the procedure itself or other inlets of the procedure. Cilk guarantees that the threads of a procedure instance, including its inlets, operate atomically with respect to one another. This relives the programmer from the need to worry that variables in a frame are being updated by multiple threads (of the procedure) concurrently.

2.6 THE CILK MODEL OF MULTITHREADED COMPUTATION

This section briefly explores the major characteristics of Cilk's algorithmic model in order to understand the efficient scheduling that is guaranteed by the Cilk runtime system and scheduler.

Previously we have seen that a Cilk program can be visualized as a DAG. To execute a Cilk program correctly, the scheduler must enforce all of the dependencies in the DAG. These dependencies allow many ways of scheduling the threads of the DAG. The two most important decisions made by the scheduler are the order in which the DAG unfolds and the mapping of the threads to processors. Fortunately, every Cilk program generates a DAG that can be scheduled efficiently [5].

The Cilk runtime system implements a provably efficient scheduling policy that is based on randomized work-stealing. Locally, a processor executes procedures in ordinary serial order (just like C), thereby traversing the spawn tree in a depth-first order. When a child procedure is spawned, the processor saves local variables of the parent on the bottom of a stack and begins to work on the child procedure. When the child returns, the bottom of the stack is popped (just like C) and the execution resumes for the parent procedure. During the execution of a Cilk program, a processor will run out of work (in fact, upon startup only one thread has work to do and all others must steal in order to do useful work). When in that state, the processor asks another processor that is selected at random for work to do. When work is available on the selected processor, it is stolen from the top of the stack, that is, the end opposite to the one normally used by the worker when working locally. The following part was taken from section 2.8 of [1].

Cilk's work-stealing scheduler executes any Cilk computation in nearly optimal time. From an abstract theoretical perspective, there are two fundamental limits to how fast a Cilk program could run. Let us denote with T_p the execution time of a given computation on P processors. The work of the computation is the total time needed to execute all threads in the DAG. We can denote the work with T_1, since the work is essentially the execution time of the computation on one processor. Notice that with T_1 work and P processors, the lower bound $T_p \geq T_1/P$ must hold. The second limit is based on the program's critical-path length, denoted by T_∞, which is the execution time of the computation on an infinite number of processors, or equivalently, the time needed to execute threads along the longest path of dependency. The second lower bound is simply $T_p \geq T_\infty$.

Cilk's work-stealing scheduler executes a Cilk computation on P processors in time $T_p \leq T_1/P + O(T_\infty)$, which is asymptotically optimal. Empirically, the constant factor hidden by the big O is often close to 1 or 2 [5], and the formula

$$T_p \approx T_1/P + T_\infty \qquad (2.1)$$

is a good approximation of runtime. This performance model holds for Cilk programs that do not use locks.

We can explore this performance model using the notion of parallelism, which is defined as $P' = T_1/T_\infty$. The parallelism is the average amount of work for every step along the critical path. Whenever $P << P'$, that is, the actual number of processors is much smaller than the parallelism of the application, we have equivalently that $T_1/P >> T_\infty$. Thus, the model predicts that $T_p \approx T_1/P$, and therefore the Cilk program is predicted to run with almost perfect linear speedup. The measures of work and critical-path length provide an algorithmic basis for evaluating the performance of Cilk programs over the entire range of possible parallel machine sizes (Note that this theoretical model neglects important practical issues such as cache behavior, locality, etc but nevertheless, it has proven to be practical even so).

2.7 CILK PERFORMANCE ON NUMA MACHINES

The idea that Cilk suffers scalability problems on large NUMA machines is evident in [21] in which the authors have implemented a parallel Sparse Cholesky factorization algorithm using Cilk and attempted to optimize the algorithm and the code for scalability on the SGI Origin 3000 shared memory supercomputer. They show that the code is competitive with other codes for single processor tests. This implies that the serial code achieves high performance without regard to the parallel performance that is also related to many other factors such as scheduling, synchronization overhead, system architecture and more. They also show that the code scales well on SMPs with up to 16 processors but once more processors are added, the performance actually drops. They indicate that "when the memory system is too slow, as in an Origin 3000 with more than 16 processors, the overheads incurred by our code are intolerable" As one of the reasons for the performance drop. On this

Origin 3000 machine, up to 16 processors can communicate through a single router. When more than 16 processors participate in the computation, some of the memory accessed must go through a link between the 2 routers, which slow down the accesses. Figure 2.7 shows their graphs of parallel code performance and in these graphs we can see that the problem is also evident in the SCSL library which is supplied by the system vendor.

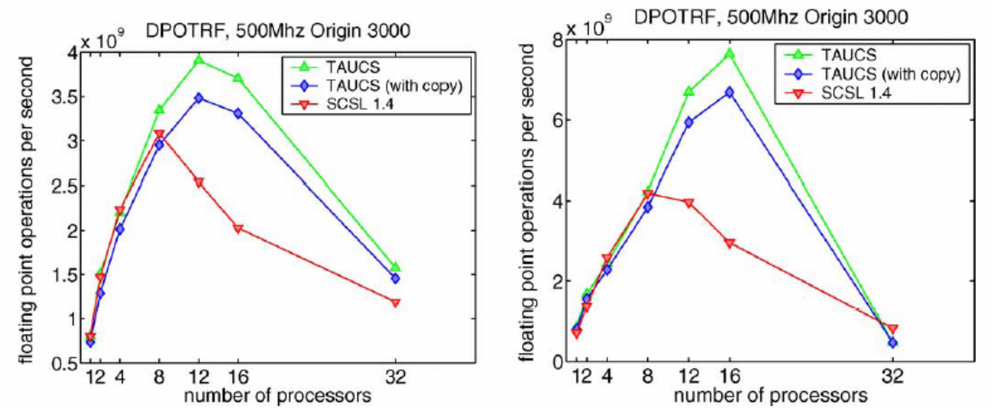

Figure 2.7: The parallel performance of the new dense Cholesky factorization on matrices of order 2000 (left) and 4000 (right), taken from [21].

The authors speculate that the second reason for the performance drop is the allocation of the entire matrix using a single memory allocation which will probably place the entire block on a single memory node. This might overload the memory subsystem of the node on which the memory is placed. It might also overload the memory links through which every remote memory access is performed from the node on which a processor is placed to the node on which the matrix is placed. They did not use memory placement or memory migration mechanisms to try to alleviate the problem.

The limitations seen so far in the Cholesky factorization code will be addressed in this thesis and an attempt will be made to improve the scalability of the Cilk code in order to use more processors efficiently.

2.8 EXPLICIT MEMORY PLACEMENT

Explicit Memory Placement was not on the checklist of this thesis. Attempting to use this technology was decided only after completing the implementation of the NUMA-aware scheduler and performing preliminary tests that showed no change compared to the original Cilk scheduler. Further tests pointed at the bad initial memory placement achieved with Irix and the MIPS-Pro runtime library as the cause for the problem. The work on Explicit Memory Placement was executed in the hope that it will eliminate the bad initial memory placement seen that far and enable the NUMA-aware scheduler to reveal its potential.

Searching the internet for information on explicit memory placement for pthreads based programs on Irix gave very little information apart from the SGI minimal documentation and none of them had information on tests/implementations that have tried to measure the performance gain from using this technology. At that point, I did not look for information on similar models (such as the widely used OpenMP standard) which later on proved to have the same problems due to poor performance of the Irix OS, the placement library, etc. Following is a survey of similar problems that were found in research on OpenMP which supports the final results shown in this thesis with Explicit Memory Placement. This survey was executed only after all attempts to improve the memory placement failed and other models were investigated to try to explain what might be wrong.

In [15] we can find the description of the various memory placement policies available for the Software developer and how they can be used with compiler directives of the MIPS-Pro compiler. The directives are available for Fortran programs and also for C/C++ programs which use the OpenMP standard but are not available for pthreads-based programs – these programs must either use the low-level MLD API or the `dplace` library which is built on top of the MLD API; the `dplace` library has a very small and simple API consisting of only two C functions to execute a single `dplace` command or all commands within a given file.

The document also enumerates the various memory placement policies which are supported by Irix:

First Touch - places memory pages in the node where they are first "touched", that is, referenced by a CPU.

Round Robin - distributes all data memory for the program across the nodes in which the program runs. Each new virtual page is allocated in a different node.

Dynamic Page Migration - When migration is enabled, Irix keeps track of the source of the references to each page of memory. When a page is being used predominately from a different node, Irix copies the page contents to the node that is using it, and resets the page tables to direct references to the new location. This effectively migrates the memory page to the node which uses it most.

Explicit Memory Placement - enables the program to request specific placement of the pages of a memory buffer on system nodes.

In the description of Explicit Memory Placement, the document indicates that "Dynamic migration is a relatively expensive operation: besides the overhead of a daemon that uses hardware counters to monitor page usage, a migration itself entails a memory copy of data and the forced invalidation of translate look aside registers in all affected nodes. For this reason, migration is not enabled by default. (The system administrator can turn it on for all programs, but this is not recommended)". These are quite unpromising words from the vendor about the technology but it still needs to be tested with real life code to see the performance and overheads.

In [16], the author attempts to evaluate the CC-NUMA DSM machines and one of the test machines was the SGI Origin 3000 machine (a second generation to the Origin 2000 model on which this thesis was executed). The Origin 3000 maintains the same architecture as the Origin 2000 with various improvements such as twice as fast interconnects (3.2 GB/sec vs. 1.6 GB/sec), lower latency over the NUMALink, double memory per node (8GB vs. 4 GB), four processors per node (instead of two) and more.

This work used several test programs to measure the performance and latency of various memory access patterns using multiple placements of thread and memory, showing the degradation of performance as the memory moves from local to remote node over several routers (Chapter 4). Section 4.4 specifically attempts to asses the performance gain that is possible when using data placement. From these tests, the author indicates that it did not manage to see the operating system relocating pages between different nodes even when the test program was using extreme access patterns (e.g.: allocating memory on the most distant node from where the executing thread is running and accessing only that memory for some time). This test was executed many times and yet the author has seen no data relocation in any run.

The next test attempted to measure the effect on performance of Explicit Memory Placement and Migration. Migration of memory pages will usually pay off when a thread is heavily accessing the memory pages such that the time spent on migrating the memory to the local node would be smaller than the latency paid on cache misses (which cause the remote memory to be accessed). It is hard to say what is the point at which the migration would pay off and this was tested by Nathan who designed a specific benchmark for this question. The results showed "an astonishingly low bandwidth" on page migration. He summarizes the test by postulating that the high kernel overhead of page migration (requires multiple node CPU synchronization, TLB flushes, etc) is the responsible for the low migration bandwidth and indicate in the results section that the usefulness of the page migration feature are nowhere near the theoretical peak which explain why Irix, by default, is reluctant to dynamically migrate pages without a programmer hint.

All in all, the results of [16] show that "the use of memory migration, either by explicit programming or by overriding the system default to become more aggressive in its implicit memory migration, is almost never useful for enhancing the performance of the code". He continues with "Even the value of programmer initiated memory migration is questionable, as the time penalty imposed by migrating memory would be hard to regain". These highly unpromising words conclude chapter 4.

Having the results from using the Explicit Memory Placement in Cilk, the notes from Irix in [15] indicating that "Dynamic migration is a relatively expensive operation" and also the conclusion of [16], it is quite understandable why SGI wouldn't recommend on this feature as it usually only degrades performance.

In [17] the authors have investigated the performance implications of data placement in OpenMP programs running on cc-NUMA machines. They show that "due to the low remote-to-local memory access latency ratio of state-of-the-art cc-NUMA architectures, reasonably balanced page placement schemes, such as round-robin or random distribution of pages incur modest performance losses". They also show that "performance leaks stemming from suboptimal page placement schemes can be remedied with a smart user-level page migration engine". The applicability of their scheme to the Cilk language is doubtful as it is based on "exploiting the iterative structure of most parallel codes". They also implemented experiments that support the effectiveness of these mechanisms. Luckily, they have implemented their tests on the SGI Origin 2000 machine which was used for this thesis as well and so some of their results and observations can more easily be used here although Cilk Algorithms are recursive in nature rather than iterative.

The first question posed by the authors was "up to what extent can data distribution affect the performance of OpenMP programs". Their tests covered 4 memory placement policies (First Touch, Round Robin, Random and Worst Case Page Placement). Their results showed that the "First Touch" policy is the optimal memory placement policy. They then compared the performance of all policies to the performance of the optimal policy and show that there is indeed a noticeable difference, meaning that the data placement has indeed a "significant impact on the performance of OpenMP Programs although this impact is not as pronounced as expected for reasonably balanced distributions of pages among processors, like round-robin and random distribution" in their words. They also feel that the main reason for the lower-than-anticipated memory placement effect is due to the very low remote-to-local memory access latency ratio.

In order to investigate the performance implications of data placement they implemented a user-level library called `UPMLib` that implemented page migration and used it in tests of OpenMP programs which have iterative behavior. These tests show that for the test programs they used, the "smart page migration engine can be as effective as a system that performs accurate initial data distribution, without losses in performance". The algorithms and heuristics that were used in the `UPMLib` user-level library might also be adaptable to other programming models but this was not checked in this thesis and it can be the subject of future work on Cilk. The library itself seems to have intricate relationships with the OpenMP compile directive and would not be easily portable to other languages/parallelism-models.

The user-level page migration library was tested with 5 OpenMP algorithms, each with all 4 placement policies and compared with an identical run aided with the migration library. The results show that for a worst-case placement, which had very low performance, the library improved performance considerably (90% on average) but for the other placement policies, the improvement was modest (10-20%) which means that with a reasonable placement policy supported by the system there is still a performance improvement but it is much less clear.

They also tried to measure the possible performance gain from activating the page migration feature of the Irix kernel and found it to be negligible. The tests also showed the First-Touch placement policy was competitive to the

Optimal Static Placement (less than 12% difference) which means that attempting to improve the First Touch Policy might not worth the trouble.

In [18] the authors have explored placement policies for DSM machines and report that “Even with the very simple policy of First-Use placement, we find significant improvements over Round-Robin placement for many applications on both hardware and software–coherent systems”. The performance improvement they saw was 20%-40%.

They further continue and indicate that “we have also investigated the performance impact of more sophisticated policies, including hardware support for page placement, dynamic page migration and page replication. We were surprised to find no performance advantage for the more sophisticated policies; in fact in most cases performance of our applications suffered”. This supports the conclusions made by [16] about migration and placement.

The final words of [17] together with the conclusions of [16] further amplify the conclusions for page migration and page placement as they were seen during the work carried out for this thesis.

In [19] the authors claim to have implemented a User-Level runtime library which implements the page migration policy such that it performs better than the operating system provided mechanism and also “demonstrate that OpenMP programs with user-level dynamic page migration are effectively immune to the page placement strategy of the operating system and have robust, non-degrading performance even with the worst possible initial page placement”. Although this sounds very promising it was tested on OpenMP programs which have iterative behavior in the sense that “they execute the same parallel computation for a number of iterations”. This behavior makes “well-defined execution points at which the program can obtain an accurate snapshot of its complete memory reference pattern”. The library also uses a modified compiler to “identify the hot memory areas of the application virtual address space which are likely to contain candidate pages for migration and instrument the programs to invoke the monitoring and page migration services of the runtime system”.

These assumptions are not necessarily true for fine-grain recursive programs written using Cilk and in addition, the requirement for a modified compiler to identify the hot spots introduces work which is out of the scope for this thesis. The requirement for a modified compiler hurts the portability of the system which is a major concern for a system that is meant for use outside the authors’ lab.

The authors state multiple motivations for their research which points to the ineffectiveness of the current page migration of existing commercial system. These motivations are the truly important part of [19] for this thesis as they bind the page migration and page placement policies to conclude that the combination of these two is the blame for the scalability problems on large DSM machines.

Some of the motivations listed in [19] are:

- “Although results from simulations in previous works have indicated that dynamic page migration is an effective technique to reduce memory latency on cache-coherent NUMA systems [24, 25], the real implementations of dynamic page migration on SGI Origin 2000 and Sun Wildfire have not demonstrated analogous results”.

- “Performance evaluations of the Origin 2000 from its vendor have not reported any results with parallel benchmarks using the Irix page migration engine” [23, 26]. This might suggest that the vendor cannot find good reasons to use the feature.

- “A preliminary evaluation of the Sun Wildfire prototype with OLTP workloads reported also no results with dynamic page migration enabled in the system [27]. A more recent evaluation of the Wildfire [20] has shown with a synthetic experiment that page migration can improve computational throughput in the long-term, but suffers from poor responsiveness and performs significantly worse compared to coherent memory replication at the granularity of a cache line”.

- “A relevant study of the complete SPLASH-2 benchmark suite on a large-scale Origin 2000 has shown that page migration was ineffective in dealing with the problems introduced by the operating system’s page placement strategy and some programs required hand-tuned page placement to scale reasonably [28]

All of these motivations and especially the last one (which is of special interest because it uses the same system as in this thesis) further emphasize the inapplicability of the page migration on the Origin 2000 and also point at the initial page placement strategy as the cause for scalability problem.

The outcome of this survey is that the Explicit Memory Placement that is built using memory migration primitives that have a low performance on their own, together with an ineffective initial placement of allocated memory causes the Explicit Memory Placement mechanism to perform poorly as well. This makes the two technologies ineffective to achieve greater scalability and performance.

Chapter 3

PORTING CILK TO A NON-GCC COMPILER

Since the early development of the Cilk language, Cilk had always maintained intimate relationship with the `gcc` compiler. This was apparent in the Cilk runtime library, in the code generated by *`cilk2c`* transformations and in the way `gcc` was extended to allow compilation of ".cilk" files. Making all these components work with native compiler and libraries of the vendor was an important step towards the development of a NUMA aware scheduler. NUMA is a relatively new architecture[1], so it was clear from the beginning of the design of the NUMA aware scheduler that the scheduler would not be portable across platforms. Having the code tied to a specific vendor meant that using its native compiler for improved performance would not hurt portability. Moreover, it was apparent from my early attempts on the SGI machine that most of the advanced features that the Irix OS offer the developer are not standard in any way and some important tools come with their libraries. These issues made it clear that we should have Cilk compile without relying on `gcc` in order to clear the way for later usage of the more advanced features of the vendor's compiler and libraries.

Making Cilk work with the vendor tools did not necessarily mean that the code had to be ANSI compliant. It was possible and simpler to fork the code and make the specific version of Cilk work with vendor tools, abandoning `gcc`

[1] Commercial CC-NUMA machines were introduced at the mid 1990’s (the Sequent STING and the SGI Origin 2000 were both introduced at 1996). These systems were influenced by the research done in the late 1980’s and early 1990’s on scalable shared-memory systems, including the Stanford DASH [12] and FLASH [13] projects, the MIT alewife [14] and more.

altogether. However, this would mean that the work will be lost afterward since the code will not contribute anything to the development of the Cilk open-source project. It was important for me to contribute the work back to the Cilk open source project and the best way to do so was to make the code more portable so that it can be used with **more** tools rather than with **other** tools.

This **portability** approach had several important advantages:

- The code could always be tested against the standard tools. This was very important for sanity testing during development in order to make sure there's no regression of other issues. It was also important for the final result in order to have a single version on which we can compare the performance of Cilk programs compiled with multiple compilers and measure the performance difference. A large difference would justify the portability on its own without the need for future benefits.

- The Cilk maintainer was interested in the portability of Cilk to new compilers. This was also important for me in order to ensure that my work will help others who will use Cilk in the future rather than simply be forgotten.

- It should be easier now to port Cilk to other tools. On some architectures there are very big differences between the vendor compilers (plus libraries) and `gcc`. One example is the Intel IA-64 architecture for which the Intel compiler generates code that is more than twice as fast as `gcc` floating point computation. On such a platform, being able to use the vendor compiler gives a major performance boost. The IMPACT project reports a speedup of up to

2.3 over gcc when testing the performance of various tools such as gzip, bzip2, vortex and more with their enhanced Itanium compiler.

3.1 PORTABILITY CODE CHANGES OF THE RUNTIME LIBRARY

Most of the runtime system was portable as is. Some work was required for cleanup of the code because the MIPS-Pro compiler is stricter than gcc and issues lots of warnings on code that is completely clean of warning when compiled with gcc. Other small issues were assembly language macros (read/write barriers, atomic register-memory exchange, etc) that had to be implemented using MIPS Pro language extensions (intrinsic compiler functions rather than assembly language), 64 bit ABI support, differences in how to specify function inlining, etc.

3.2 PORTABILITY CODE CHANGES OF THE COMPILE DRIVER

The compiler driver mimics the behavior of gcc as it parses the command line switches and decides what tools (preprocessor, compiler, linker, and *cilk2c*) need to be invoked and what part of the CLI relates to each tool. For instance, compiling a ".c" file requires activating a preprocessor, then compiler and then maybe a linker. Some command line switches relate to a single phase (such as specifying a macro definition) while others relate to multiple phases (such as ABI). The driver adds support for compilation of a ".cilk" file while all other standard file extensions were delegated to gcc for full treatment. A ".cilk" file requires 3 phases until it is made into a C file that can be compiled by gcc:

- A preprocessing phase to get a Cilk file with expansion of macros.

- A transformation of the Cilk constructs into equivalent C constructs and emitting proper calls to the Cilk runtime library. This phase requires `cilk2c` to process the code that was generated by the previous phase. `cilk2c` then generates code which uses many macros defined by the runtime system in order to make the generated code both shorter and simpler to understand by a human developer as it is quite simple to see most of the correlations between the original Cilk source code and the generated C source code.

- Activation of the preprocessor again in order to have all the macros used by the C code generator translated into C code.

When these 3 phases are completed we need to execute the same phases as any standard C file compilation requires. The driver program automates all of these phases into a single command line whose structure was the same as that of `gcc` only with the additional support for files with the ".cilk" extension.

When moving from `gcc` to MIPS-Pro, the compiler being used is not known to the driver program because of the decision to support them both: It might be either one of these compilers (support for Intel Compiler for Linux was added later on and other compilers might be supported in the future). So the compiler to be used needs to be indicated through a new command line switch targeted for the driver program. When this parameter is set, the parsing of the rest of the command line has to be dependent on the compiler being used. Although `gcc` and MIPS Pro have similar command line syntax, it is not entirely identical and so some of the existing parsing was made valid only when using `gcc` while new parsing was added for the case when using MIPS

Pro. Moreover, sometimes `cilk2c` is required to inject a command line switch to achieve a certain behavior yet the switch to be used must be different depending on the compiler being used. One example is the use of a switch to indicate for the C preprocessor that a file with ".cilk" extension should be parsed as a C file (`gcc` recognizes the `-x` switch while the Intel C Compiler treats this as instruction set generation directive that will cause the code to execute only on certain processors).

In addition there are differences not noticeable from the command line syntax such as:

- Multi-pass linker (`gcc`) vs. a single-pass linker (MIPS Pro): single-pass linker forces libraries to be specified on the command line in the order of their dependence. When processing modules on the command line, the IRIX loader (`ld`) requires that at least one reference to an unresolved symbol appear before inclusion of the library that resolves that reference [8]. Because the Cilk runtime libraries are added to the link phase without having them specified by the user, the driver must add them at the correct place in the list of arguments in order for the link to be successful. Simply inserting the Cilk runtime libraries usually causes a link failure because of missing symbols (which are in the badly placed library).

- The Cilk driver program used the "`-include`" switch of `gcc` in order to cause compiled ".cilk" files to include Cilk runtime header files without forcing the programmer to include them in the Cilk source code. MIPS-Pro does not support this switch or an equivalent. This problem was solved by using a temporary file into which the original Cilk file is copied with the proper inclusion of runtime header files generated in the beginning of that file. Using additional standard preprocessor directives cause the compiler to generate warning/error messages with proper files name and line numbers that the Cilk developer can understand.

The driver program maintained the use of `gcc` for the preprocessor phase because of requests from the MIT group (who developed some new features in parallel with my work and wanted to avoid future code merge problems). Later on this was handled by that team as well and the package could be compiled without `gcc` installation at all.

3.3 PORTABILITY CODE CHANGES OF THE COMPILE DRIVER

Last and most important was to make the *`cilk2c`* tool and the C code that it generates compile with an ANSI C compiler in general and with the MIPS-Pro compiler in specific. The code that was generated by *`cilk2c`* suffered 2 major portability problems:

- Cilk inlets were implemented as inner functions which are not supported by ANSI C.

- Initializing automatic variables at the point of definition requires some tricks to ensure that they are properly initialized in the generated code.

3.3.1 Supporting inlets

A special treatment was required to support Cilk inlets. Inlets are described in chapter 2 as special functions which allow the incorporation of the result of a spawn without assigning it first into a variable and it has a few more interesting features such as ensuring atomicity of access to parents frame variables between the inlet and the parent, etc. Because inlets are very similar in their syntax to inner functions, they were implemented as such by cilkc2c because gcc supports inner functions for the C language (as opposed to ANSI C compliant compilers) and such support is much easier to implement. For example, an inlet IN declared inside a Cilk procedure PROC was transformed by cilk2c into a C function IN_C that was an inner function in the output of the transformation of PROC into a C function called PROC_C. This was convenient because an inner function has access to local variables and formal arguments of the parent procedure in a natural way (as required for inlets) but it meant a lot of trouble for other compilers that do not support inner functions. Because cilk2c transforms the code using recursion to traverse the AST of the transformed module, it is easier to implement local transformations than non-local transformations. In "local transformations" I refer to a transformation made when the AST traversal points to a node N in which all of the information that is required for the transformation is available from N and its children. The side effect of the transformation (add/delete/update AST nodes) is also performed only on nodes that are accessible though N and its children. This kind of transformation is natural for Cilk (which is a small extension of ANSI C) because it transforms the new construct of the Cilk language into equivalent implementation using C constructs, one block at a time. Getting rid of these inner functions required the extraction of the inner function logic into a standard global scope C function but this does not allow the inner function to access variables from its enclosing function scope. The solution was to pack all the variables of the outer function into a structure before the call to the inlet generated function, pass a pointer to that structure into the inlet generated function and unpack the variables upon return (some variables might be updated by the inlet function and that update needs to be reflected into the outer function). The code of inlet itself is transformed to use the variables from the structure rather than expecting them on the stack. This solution is clean in terms of portability though it has a performance downside (packing and unpacking the automatic variables of the procedures which are used in the inlet). The performance downside was decided to be a non-issue because the inlets are not the construct with which the programmer expresses parallelism and from its atomicity behavior it is clear that it can degrade performance and parallelism because of the implicit synchronization. So a Cilk developer must ensure that inlets are not used too much in the code or otherwise the code will not be able to achieve linear speedup. The inlet code now has to be transformed so that every usage of a variable from the scope of the containing Cilk procedure, will access the variable through the new structure pointer introduced by the `cilk2c` translator into the function signature. This requires the code to know the scope of every variable yet it is already computed by the Data Flow module and the information is available at the tree node. An example of this change can be seen in Appendix A. Figure A.1 shows a Cilk procedure which uses an inlet constructs. Figure A.2 shows the C code that was originally generated by `cilk2c` – the code uses an inner function. Figure A.3 shows the C code that is generated by `cilk2c` after making inlets implementation using standard C function (global scope).

3.3.2 Supporting initialized automatic variables

Another problematic case was related to the transformation of Cilk procedures which define automatic variables and initialize them upon declaration. `gcc` supports an extension called “statement as expression” which means that a C expression can be made to include code that ANSI supports only as a statement - this includes the ability to declare new variables inside such an expression.

Every Cilk worker manages its part of the cactus stack. This management implies that prior to a spawn, the caller pushes a frame onto the stack, then it calls the function in a standard C convention and upon return from the called function it pops the frame from the stack. The frames maintain the state of the spawning function at the point that the spawn was made so that another worker that steals it can reconstruct the stack based variables. To accomplish this, *cilk2c* generates for a Cilk procedure named PROC a structure that is named PROC_frame. This structure contains all automatic variables of the procedure PROC. The variables of the frame are updated with the variables from the stack before the frame is pushed into the cactus stack.

```
typedef struct {
        int  i;
        char c;
        int* pi;
} a_type_t;

static int number = 123456789;
static const a_type_t    a_type_const = {
        1234,
        'z',
        &number
};

cilk void func (void)
{
  int       num = 1;
  a_type_t  a_type = a_type_const;
  long      time;


  < statements … >
} /* end of procedure */
```

Figure 3.1: A Cilk procedure declaring 3 stack-based variables. First 2 are transformed while the third is not.

The problem begins when in some cases there is no need for the stack variable and only the variable inside the frame object is used. One example is the "Slow Clone" which is called by the scheduler once a steal occurs and as such it receives the frame with the state of the function. Instead of copying the variables from the frame structure into local stack variables, it can work directly with the variables from the frame object – so it will not declare the variables on the stack as the Cilk procedure did. In this case, the initialization of the stack variable in the Cilk procedure is translated into initialization of the corresponding variable in the frame object (at the beginning of the function). But some stack-based variables are not required to be saved in the frame and the declaration of these remains exactly as in the original Cilk procedure. The problem here is that a C function starts with declaration of variables and then proceeds with the function body. In the function body, there cannot be any new variable declarations (without declaring a new block). But the translated initialization of the frame object variable is not a declaration and hence it ends the declarative part of the function and begins the function body. So if the next variable was not transformed and remains a standard C definition then it will cause the compiler to generate an error since the declaration of the new variable is considered to be in the function body. A solution was required to allow us to mix the definitions of automatic variables with the initialization of those variables that exist only in the frame object (because it is imperative to maintain the initialization order in case the variables are dependent on one another). In Figure 3.1 we see a simple Cilk procedure to be transformed to C code. The original translation appears in Figure 3.2.

As we can see in Figure 3.2, the generated code used a `gcc` extension to ANSI C which allows "Statements and Declarations in Expressions" (excerpt from `gcc` online documentation: "A compound statement enclosed in parentheses may appear as an expression in GNU C. This allows you to use loops, switches, and local variables within an expression. Recall that a compound statement is a sequence of statements surrounded by braces; in this construct, parentheses go around the braces.").

Figure 3.3 show an ANSI-C solution to the same problem. The solution is to declare the same stack-based variable that was originally declared in the Cilk procedure and previously removed in the generated C code (because the equivalent variable in the frame object was used instead). This allows the initialization of the variable to remain exactly as is written in the Cilk procedure. The value is then copied onto the frame object using *`memcpy()`* which allows the copying of any object size and hence can handle any primitive or complex data type. In order for this *memcpy()* call to be valid in the declarative part of the function its return value is assigned to yet another temporary variable which is never used later on and hence will be removed by the optimizer. At first it might seem to be an overkill to initialize an automatic variable and then copy it into the equivalent frame object variable using a function call but this code relies on the fact that when using *memcpy()* for small objects, the optimizer removes the call to *memcpy()* (which is an intrinsic function and not an actual function call) and replaces it with simple assembly *move* instructions. On Intel Pentium III using gcc, copying up to 3 memory words with *memcpy()* (32 bit each) is translated to the required number of *move* instructions. Similar behavior was seen on Microsoft Visual C++ compiler and it will probably be similar when using other high end optimizing compiler (for which Cilk might be ported in the future) since most of them support such simple *libc* functions as intrinsic functions. Because the original automatic variable is never used apart from being copied, it is easy for the optimizer to remove it altogether and use only the frame variable. For simple variables and arrays it was tested that the gcc optimizer initializes the frame object variable directly without bothering with the stack-based temporary variable at all. With MIPS-Pro this was not the case – the *memcpy()* is implemented such that it assumes that the source and destination buffers may overlap and it causes the compiler to miss the full optimization potential in some cases (this can be fixed using the "-OPT:memcpy_cannot_overlap=ON" command-line switch which tells the compiler that *memcpy()* can assume that the source and destination buffers do not overlap). Using the OPT:memcpy_cannot_overlap optimization switch should not cause trouble because most other compilers assume that the source and destination buffers do not overlap and hence require portable software not to assume that they can overlap (for the case of possible overlapping there's the *memmove()* ANSI C function). With this switch turned on, MIPS-Pro yields similar behavior to that of gcc for this case.

Later on, a different solution was suggested by Bradley C. Kuszmaul of MIT who is the maintainer of the Cilk project. His solution can be seen in Figure 3.4 – it is both portable and efficient since it has no function call and it doesn't need to rely on optimization capabilities of the compiler other than simple ones such as removing an unused variable, etc. For arrays, we still use *memcpy()* as a solution but it is not common to find large stack-based arrays and even less common to find them initialized upon declaration. So there's no real performance issue here.

```
int _cilk_func (void)
{
  int num = 1;
  int _cilk_temp0 = ({ a_type_t _cilk_temp1 =
a_type_const;
         _cilk_frame->scope1.a_type = _cilk_temp1;
         _cilk_temp0;
         }
  );
  long    time;


  < statements … >
} /* end of function */
```

Figure 3.2: The original C code that was generated by *cilk2c* from the Cilk procedure in Figure 3.1. It uses the "statement as expression" gcc syntax which causes a compilation error with other compilers.

```
int _cilk_func (void)
{
  int _cilk_temp1 = 1;
  void *_cilk_temp0 =
      memcpy (&_cilk_frame->scope1.num,
              &_cilk_temp1,
              sizeof (_cilk_temp1));
  a_type_t _cilk_temp3 = a_type_const;
  void *_cilk_temp2 =
      memcpy (&_cilk_frame->scope1.a_type,
              &_cilk_temp3,
              sizeof (_cilk_temp3));
  long    time;


  < statements … >
} /* end of function */
```

Figure 3.3: The first ANSI-C code that was generated by `cilk2c` from the Cilk procedure in Figure 3.1.

```
int _cilk_func (void)
{
  int _cilk_temp0 = 1;
  void *_cilk_temp1 =
      ((_cilk_frame->scope1.num = _cilk_temp0),
       &_cilk_temp1);
  a_type_t _cilk_temp2 = a_type_const;
  void *_cilk_temp3 =
      ((_cilk_frame->scope1.a_type = _cilk_temp2),
       &_cilk_temp3);
  long    time;


  < statements … >
} /* end of function */
```

Figure 3.4: The final ANSI-C code that was generated by `cilk2c` from the Cilk procedure in Figure 3.1.

3.4 PERFORMANCE COMPARISONS

This section presents performance comparisons of various programs and the benefits that come with the support for the native compiler and libraries. The first comparison is made for all of the deterministic Cilk example programs (those available with the distribution). Each of these programs is compiled as is without changing any of its code to exploit the specific machine on which the measurements were made, with the gcc compiler and also with the MIPS-Pro compiler. Each program is then executed 5 times and the average of each program execution time is used for the comparison.

Because we measure only the changes due to the different compiler, we run all programs using a single processor although they are written as parallel programs. The details of hardware on which the measures were taken are:

- Machine: SGI Origin 2000
- CPU: 300 MHz IP27
- L1 Cache Size: 32KB for Instructions, 32KB for Data
- L2 Cache Size: 8 MB unified instruction/data
- Total Main Memory: 8 GB
- Node Memory : 512 MB

All tests were executed such that no swap is used.

The compilers that were used were:

- gcc 2.95.2 (the newer 3.3.4 version is available on the host but Cilk fails to compile with it because of missing supporting libraries. This problem is also seen on other platforms such as Solaris with new gcc installations and highlights the fact the gcc might have trouble on some systems while the vendor tools are more likely to perform well out of the box).

- MIPS-Pro 7.4

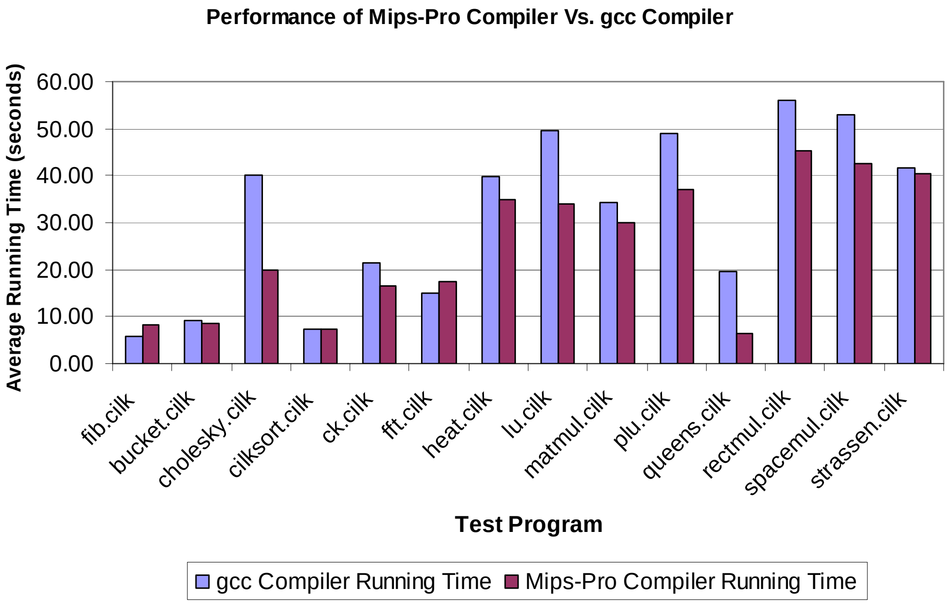

Figure 3.5: Comparison of running times for the example Cilk programs when compiled with MIPS-Pro vs. those compiled with gcc.

As the above graph shows, most test programs perform better when they are compiled with MIPS-Pro. The next diagram shows the relative gain of the native compiler.

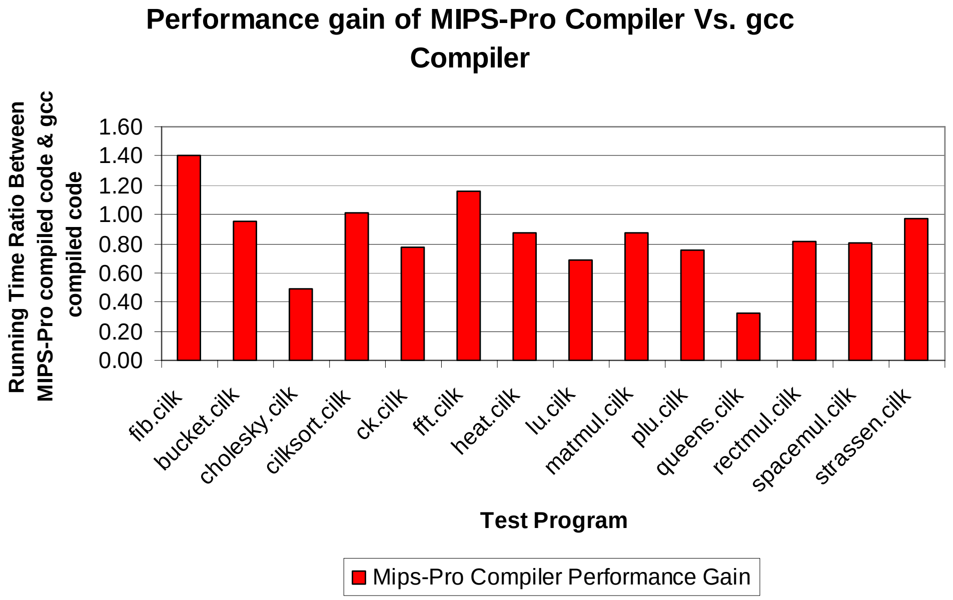

Figure 3.6 The ratio between the running time of the program compiled with MIPS-Pro and the program compiled with `gcc`. A value < 1 indicate that the program compiled with MIPS-Pro yields higher performance.

The second test case measures the benefits of the native compiler/libraries support for parallel code executions. In this case I have improved 2 of the example programs that came with the Cilk distribution. Both of these examples solve the matrix multiplication problem which is a well studied problem.

- The first program (matmul.cilk) solves the matrix multiplication problem by a simple recursive algorithm which divides one of the 3 dimensions of the matrices into 2 halves and initiates the 2 sub-computations in parallel. When the matrices are small enough the recursion ends and the multiplication is executed in one of 2 ways:

  - Using three simple nested loops as in $C_{i,j} = \Sigma(A_{i,j} * B_{j,k})$
  - Call the Native BLAS library (SCSL) that is available from SGI.

- The second program solves the matrix multiplication problem by using the strassen algorithm which has a lower complexity than the simple textbook solution. In this algorithm we have 3 phases (additions, multiplications and then additions again) and all three phases are implemented with parallel Cilk code in order to achieve scalability with a large number of processors. When the matrices are small enough, the recursion ends and the multiplication is executed in one of 2 ways:
  - Implement recursive algorithm which divides each computation into 8 sub-computations again and again until the matrices are small enough to compute them with a three nested loops as in $C_{i,j} = \Sigma(A_{i,j} * B_{j,k})$.
  - Call the Native BLAS library (SCSL) that is available from SGI.

Both of these programs (each with its 2 flavors for the base case) are compared with the performance of the SGI Scientific Library (SCSL) which is written by SGI specifically for their machines and hence should achieve higher performance. SCSL computes the matrix multiplication using the OpenMP standard which allows the use of multiple processors in a DSM machine.

Performance Results:

First is the comparison of the performance of the OpenMP version vs. a simple textbook solution that uses SCSL for the base case (the version which doesn't use SCSL at all has very low performance and was removed).

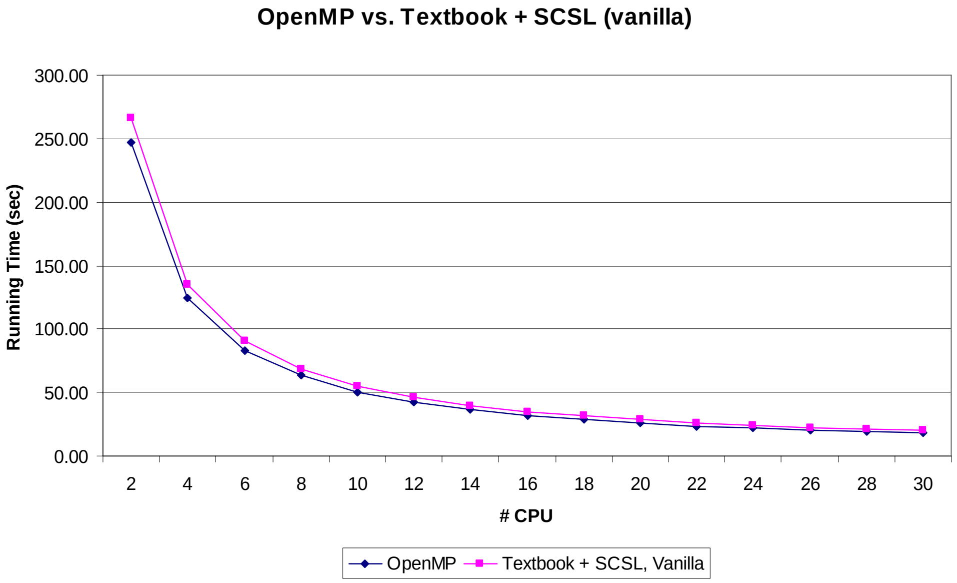

Figure 3.7 The running time of the matmul.cilk program which uses SCSL for base case compared with the OpenMP SCSL library.

These results show that the use of SCSL together with the efficient scheduling of Cilk yields similar performance to those achieved with the OpenMP version that is written by the machine vendor for any number of processors.

Second is the comparison of the performance of the OpenMP version vs. an implementation of the strassen algorithm that uses SCSL for the base case.

These results show that the use of SCSL together with the more scalable algorithm and the efficient scheduling of Cilk yields better performance than those achieved with the OpenMP version that is written by the machine vendor for any number of processors.

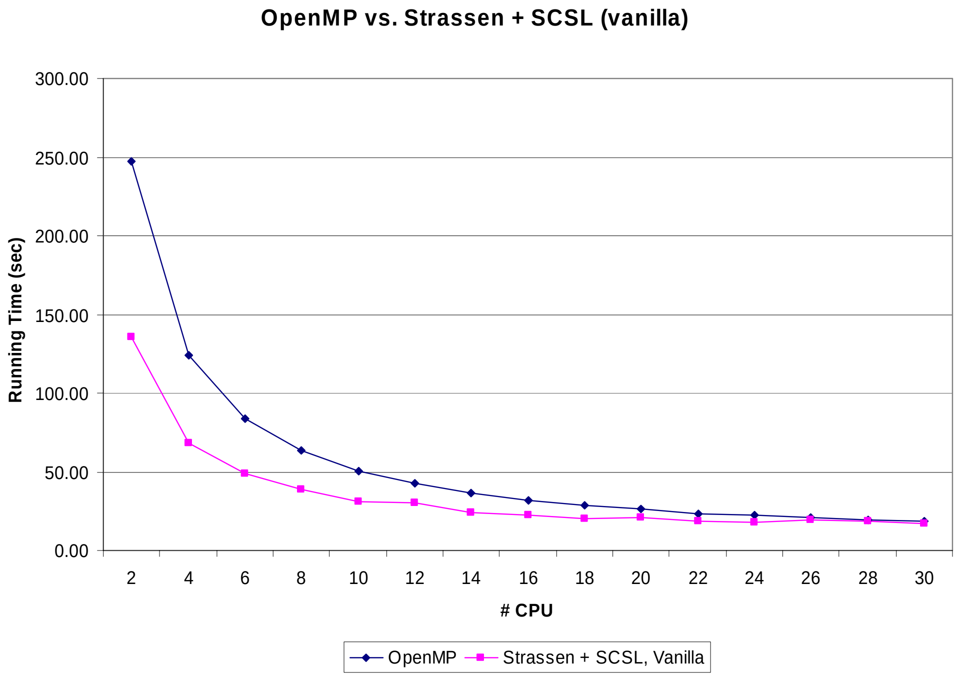

Figure 3.8 The running time of the strassen.cilk program which uses SCSL for base case compared with the OpenMP SCSL library.

The Cilk based code has several additional advantages including:

- Cilk uses the standard pthreads multithreading model rather than the proprietary sproc model. SGI documentation indicates that these two cannot be mixed together and hence one solution does not fit all. The improved Cilk solution fits well for commercial and open-source software which uses the standard pthreads model for portability.

- Because of the use of fine-grain scheduling, when the Cilk based matmul is used as a building block for a larger system (e.g.: matmul is a step in a single iteration of a cholesky factorization algorithm) the system has the potential for even higher utilization because unlike OpenMP, with Cilk we don't need every BLAS operation to have a barrier at the end of it. One example is when the higher levels of the computation are parallel and hence multiple matmul operations might be performed in parallel. In the OpenMP model, the matmul operation would have a barrier and hence all processors either work on that operation or remain unused.

The next graph shows that the use of Cilk together with SCSL yield highly scalable code in terms of processors in addition to achieving high performance of the serial code (mostly from SCSL) for the Textbook algorithm. The strassen algorithm is less scalable and yet yields even better running time results at the end of the day. Comparing both of these algorithms show that the lack of scalability stems from the algorithm and its implementation rather than from the Cilk system. This scalable and efficient solution is a winner for scientific computing.

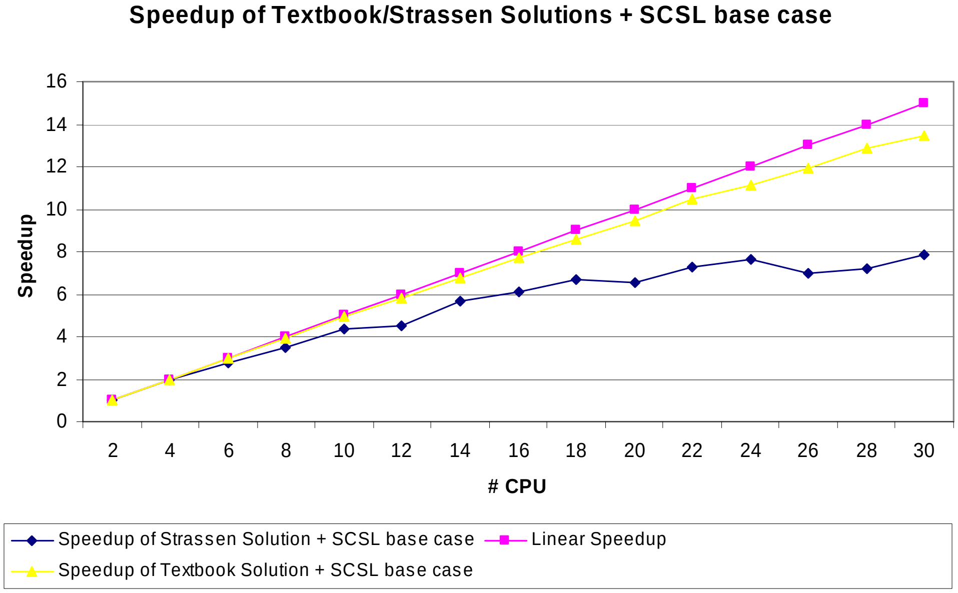

Figure 3.9 The speedup of the matmul.cilk program which uses SCSL for base case compared with the speedup of the OpenMP SCSL library.

3.5 PORTABILITY TO OTHER COMPILERS

In order to check that the final code was indeed portable in general rather than adapted to the SGI/Irix platform, I've tried to take the code to a Linux 2.4 system and compile the Cilk example programs with the latest Intel C compiler which is known for its high performance code generation and might be a viable candidate for usage in the development of high performance applications.

To compile the examples programs, runtime libraries and `cilk2c` translator took only 2-3 hours of programming and debugging time. Most of the time was required to figure out that some command line switches should be added /changed and this required the driver program to support the parsing of these new command line switches. Other than that, most programs ran out of the box with the following performance comparison to the `gcc` compiler.

The details of hardware on which the measures were taken are:

- Machine: Intel Pentium 4 CPU
- CPU Speed: 1600MHz, Stepping 10
- L1 Instruction Cache: 12 KB
- L1 Data Cache: 8KB

- L2 cache Size: 256 KB

- Memory: 512 MB

- Input Test cases did not exhaust the physical memory.

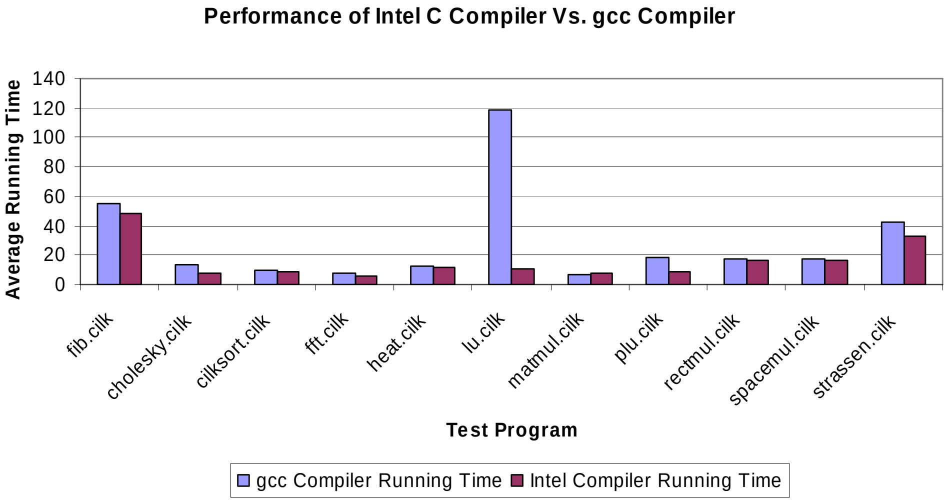

Figure 3.10 Performance Comparison of example programs which are compiled with gcc and with the Intel C Compiler.

As the above diagram shows, most test programs perform better when they are compiled with the Intel Compiler. The next diagram shows the relative gain of the native compiler in order to see more clearly the gain.

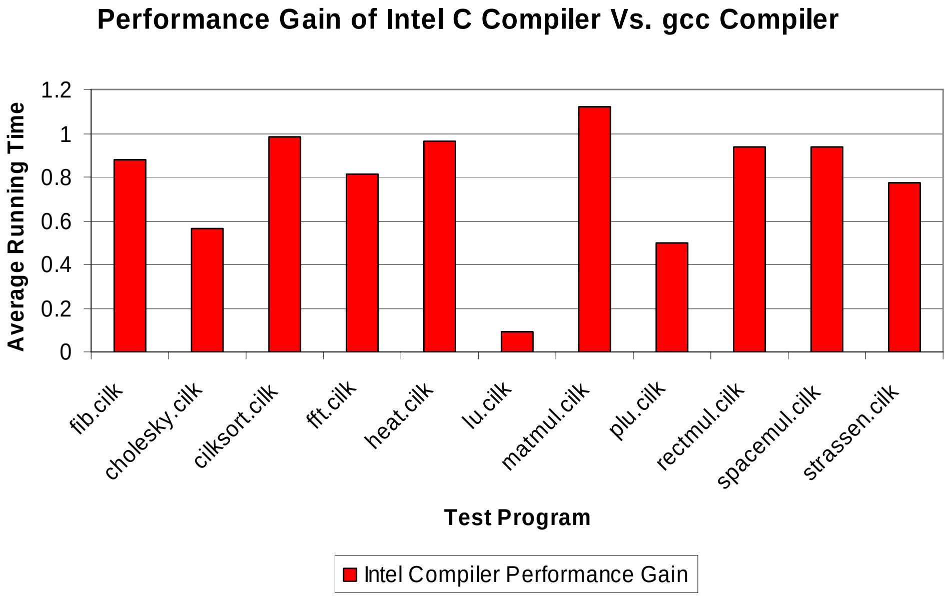

Figure 3.11 The ratio between the running times of a program that is compiled with the Intel C Compiler and the same program that is compiled with `gcc`. A value < 1 indicate that the Intel Compiler yields higher performance.

Now that Cilk operates with 3 different compilers and 4 Operating Systems its quite clear that moving to a new OS/compiler which supports the GNU build system (`autoconf, configure, make`, etc) should be fairly easy.

Chapter 4

NUMA AWARE WORK-STEALING SCHEDULER

This chapter describes the work done in order to improve the Cilk scheduler so that it takes the NUMA (CC-NUMA is the same in the context of this chapter) architecture into account when steal attempts are made. In a NUMA machine, a processor can access different parts of the system memory in the same way it access its local memory. However, the processor pays the penalty in terms of the time it takes to access the remote memory as well as increasing the load on the communication links that connect the system nodes (so at some point these overloaded links might become the bottleneck). For a multithreaded program to scale from an SMP system to a NUMA system we need that the memory allocation scheme would be such that a memory allocation request made by a processor will always attempt to locate a local memory for that processor. This will ensure that as long as processors allocate the memory they use, the memory access is local and no contention on memory modules or communication resources arise.

In Cilk, this becomes more complicated because we do not know which worker thread executes a Cilk thread, when are threads stolen, which processor steals from which other processor, which processors of the system participate in the execution of the Cilk program, etc. When a worker thread T_1 steals work from worker T_2, it uses memory allocated by T_2 – assuming T2 allocated memory on its local node, that memory now becomes remote for T_1 and as the 2 threads might be executing on any physical processor it might cause T_1 to use memory that is on the farthest end of the system. The goal of the NUMA aware scheduler is to improve the runtime scheduler so that it is more suitable for NUMA systems thereby achieving better scalability in terms of processors as well as providing APIs to Cilk programmers so that they will also be able to take this information into account and improve the performance of their implementation.

In order to allow easy comparison of the different schedulers and open the door for future ones, a small set of operations was defined to be required from the scheduler implementation. These include:

1. Global initialization of the scheduler.
2. Per-worker scheduler initialization.
3. Victim selection.
4. Per-worker scheduler termination.
5. Global termination of the scheduler.

The victim selection API is the most important since it implements the policy by which the scheduler selects from where to steal. Current Cilk scheduler implements this by selecting a victim from among the other workers in a uniform random distribution. On a NUMA machine, this victim selection policy is not ideal because we want to maximize the locality of memory. We want to keep the memory that we use close to the CPU and assuming that a thread manages to allocate most of its memory on the local node, stealing from a nearby CPU is preferable to stealing from a remote CPU since the memory that was local to the allocating CPU will be relatively close to the stealing CPU.

Using this separation API, 3 schedulers were implemented. The first one merely wraps the original Cilk scheduler (victim CPU is selected by uniform random distribution). It allows a reference implementation to which a new scheduler can be compared and it is termed as the "vanilla" scheduler. The second one takes into account the distance between processors. When a processor requests a victim, it selects one of the closest processors to the requesting processor. If that steal attempt fails, it will select a processor that is next in distance. This continues to increase the distance until we select the most distant group of processors. If stealing from the most distant group of processors fails, we wrap and start all over again with the group of closest processors. Selecting from increasingly distant processors ensure that we cannot get into a situation in which distant processors have lots of work but we always attempt to steal from closer processors and therefore we fail again and again burning cycles for nothing. The problem with this scheduler is that it undermines important theoretical issues of the original Cilk work stealing scheduler which is heavily based on the fact that **every** processor has a chance of being selected (probability plays an important role). However, the approach of selecting a closer processor sound reasonable and it should be compared with.

To address the theoretical issues of the Cilk scheduler, a third scheduler was implemented. This one also prefers closer processors over distant processors but only by giving higher probability for closer processors to be selected as victims. This means that in every victim selection request, every processor still has the chance of being selected and the theory of work stealing schedulers can be adapted for this non-uniform distribution. This is similar to the work that was done for the Distributed Cilk scheduler by Keith H. Randall in his Ph.D. Thesis [6]. In Distributed Cilk, the scheduler was changed to prefer stealing from local processors (on an SMP machine that is participating in the Network wide computation) rather than stealing from a processor that is accessed over the network. The scheduler is analyzed to show that if that preference is made by a "local bias" strategy (instead of stealing randomly with uniform distribution it steals with a biased distribution) rather than "maximally local" strategy (always still from a processor on the local SMP if there is work on that SMP) then we can retain the provably-good properties of the work-stealing scheduler.

In order to be able to take into account the distances between the processors of a NUMA machine we need to identify the topology of that system. Topology refers to various issues such as:

1. Processors on the same memory controller might share external cache and they surely share the same physical memory. So stealing from another processor on the same memory controller should be of highest probability.

2. Processors on the same board use the same physical memory but not necessarily the same memory controller and its associated cache. Stealing from such a processor doesn’t require the memory access to go over the communication link resulting faster access times and lower link utilization. Lower link utilization enables higher scalability in terms of CPUs.

3. Processors with different number of routers/communication-links between them. The more links and routers that a memory access must travel, the longer it takes to get the reply and the more bandwidth is used of the available system communication bandwidth.

All of this information needs to be identified. Identifying this information is made in a platform dependent manner since there is no standard OS APIs for these services.

Since this work was carried out on SGI machines running Irix OS, a utility program was written to extract the information from the OS. This program generates a matrix of size N * N (N = Number of system processors) that indicate the distance between every 2 processors. What really matters is that we need a more distant processor to have a larger value in the distance matrix so that we can assign increasing probability of selection to processors with lower distance.

The tool for SGI machines (called `sgi_numa_explorer`) uses the "/hw" file system to build a graph in which vertices are devices (CPU, memory, routers, etc) and edges are connecting 2 vertices if data can travel between these 2 devices (memory bus, communication link). The tool uses the Dijkstra algorithm from the Boost library [9] in order to find the shortest path from a processor to every other processor (this is usually the path used by the hardware to access the remote memory node). The distance from processor I to processor J indicate the distance of the processor on node I from the memory that is installed in node J. the distance matrix that is generated by this tool is written to a file that is later read by the Cilk runtime scheduler when it initializes (the tool can be incorporated into the Cilk runtime library and in that case, every processor can perform its own Dijkstra algorithm to compute distance of every other processor from it in parallel).

When trying to maintain the computation close to the memory it is using, it is much simpler when the worker threads are bounded to specific processors. On a small SMP this is not required because the OS scheduler will usually schedule a worker thread on the same processor it used, if possible and hence cache use is optimized. Memory is no issue because the same physical memory is used by all the processors. Cilk programs running this way adapt quite nicely even for varying number of available processors as the workload on the machine changes over time.

When aiming for large NUMA machines, programs are executed in one of 2 common methods:

- Using batch systems in which processors are usually allocated to a job from its beginning to its end.

- Using a standard shell (Interactive mode in SGI lingo). In this case the threads compete for processors with other programs and hence they usually migrate over time to more and more processors.

Moreover, it might be important to select specific processors such as those with faster processors, those with more local memory on board, those close to the graphics engine, etc.

The NUMA schedulers implemented as part of this thesis assume that a set of processors is allocated for the Cilk program. The specific processors can be specified at command line or they can be found by the scheduler when the worker threads begin execution and are allocated physical processors by the OS. In any case, once the processors are known, the worker threads are bound to these processors (one worker thread per processor) and from that moment till the end of the execution, we have a bidirectional mapping of a processor ID with a worker thread ID. This information is used in many ways such as:

1. Knowing the processor on which a thread is running without querying the OS saves expensive system calls.

2. Having each thread maintaining its scheduling information in its local memory so that it is as close as possible and no contention arises between different worker threads while they make steal attempts.

3. Allowing the Cilk programmer to query the processor on which it is running enables algorithm implementation optimizations such as allocating memory buffers on physical memory that is on a specific board, telling whether a memory is local to a worker thread and at what distance it is, etc. although the documentation of the Irix OS memory management system specifies that when using multiple threads, each one allocates memory on its own local board and only when this is not possible it allocates memory on other boards, simple test programs show that this is not exactly the case (at least when using pthreads) and hence further optimizations can be made to the memory access patterns.

Once we have for every processor I the distance vector of every other processor from I and we know that a worker thread with ID I' will always be bound to the same processor I, we are missing only the weights of the different distances. For the new scheduler we allow the programmer to specify the weights for the different distances so that their effect can be tested on the specific system on which the program is supposed to run. Such values will be dependent on the machine architecture, the ratio of memory access latency between local and non local memory, the memory access pattern of the specific algorithm, cache hit ratio, etc. Once the programmer sets the weights for each distance the scheduler needs to define the probability function that will yield the required ratio between the processors.

Assuming we have a random number generator that yields a uniform random distribution we would like to implement a probability distribution function that is dependent on the weights of the processors and the number of processors at each distance that is fairly efficient since it is computed at runtime upon every steal attempt that is executed. Assume we have the processors at distances 0,1, … (D-1) with weights w_0, w_1, …w_{D-1}. Assume also that we have p_i processors at distance i. The total weight of all processors (except myself, which is at distance 0, since we don't want that a processor will select itself as a victim) is:

$$W = \sum_{i=1..D} (w_i * p_i)$$

The total weight of a distance group g is $W_g = \sum_{i \in g} (w_i * p_i)$

It follows that $W = \sum_{i=1..D} W_g$

In order to select a certain victim processor we need to draw a random number in the range of [0 ... W-1] using a uniform random distribution (if we cannot have the range as we want, we can always draw the random value from the entire range of 32-bit integer, divide by the range and use the remainder). We then need to map that value into a specific processor. We can look at it as if processors are ordered in increasing distance order and every processor occupies a range equals to the weight of its distance groups. The ranges of the processor induce a set of segments with total length of W. Every distance group g has processors with a total length of W_g. We can prepare a vector with the values in which we move from one distance group to another at the initialization of the scheduler. So given a random value r we need to scan the vector and find the range that contains r (for a small number of discrete distances a simple linear search will suffice. For larger configurations we can use binary search). Assume that the range of group g which contains r begins at T_g. We can now find the specific processor within group g by computing $(r - T_g) / w_g$ because all processors of group g have the same weight of w_g.

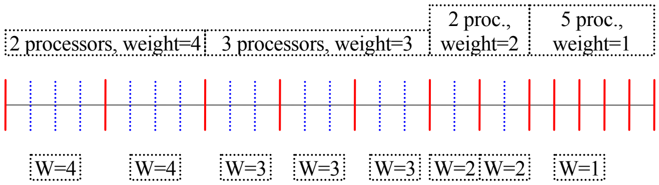

Figure 4.1: every processor has its range as a segment and all segments together define the range from which we draw a uniformly random number variable.

So going from a uniformly distributed variable into a specific distribution based on the weights of distances is made in two simple steps. This yields the next processor to steal from.

After completing the NUMA aware scheduler, I've redesigned the two matrix multiplication programs (simple Textbook algorithm, called `matmul.cilk`, and the Strassen algorithm, called `strassen.cilk`) so that they take advantage of the new scheduler. The amount of time that was required to code, test and tune each of these programs was too high to do the same work for the entire collection of examples programs within the Cilk distribution.

The following graph shows the performance of the `matmul.cilk` program with increasing number of processors. Each of the 3 colors is for a different scheduler. The graph show that the NUMA-aware schedulers were not as helpful as thought at first. Analyzing the algorithm suggests that the reason for this is that the base case of the recursion is such that the entire block of matrix fits into the L1 cache (for high performance of the serial code) and hence the cache hit-ratio of the actual computation (which consumes most of the running time) is so high that the latency of a distant memory doesn't affect the computation very much. The same explanation fits the next graph which describes the strassen algorithm since the difference is in the recursion of the algorithm but they both use the same base case (serial code) and enjoy the extremely high L1 cache hit ratio (~99%).

I conjecture from these results that further research is required in order to experiment with other algorithms that have intrinsically lower cache hit ratio and would be able to reveal the benefits of the NUMA-aware scheduler. If an algorithm can be adapted to use the NUMA topology knowledge (e.g.: use per CPU private memory pool for memory management) then further performance improvements might be possible.

Another cause is the physical memory placement policy of the Irix OS which undermines the foundation of the NUMA-aware schedulers. This issue is discussed in the next chapter.

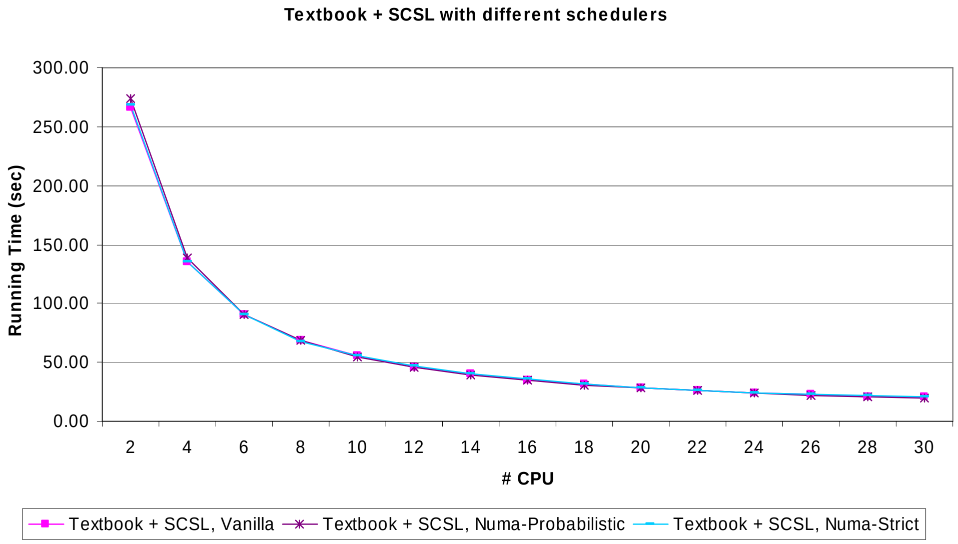

Figure 4.2: The performance of matmul when using different schedulers.

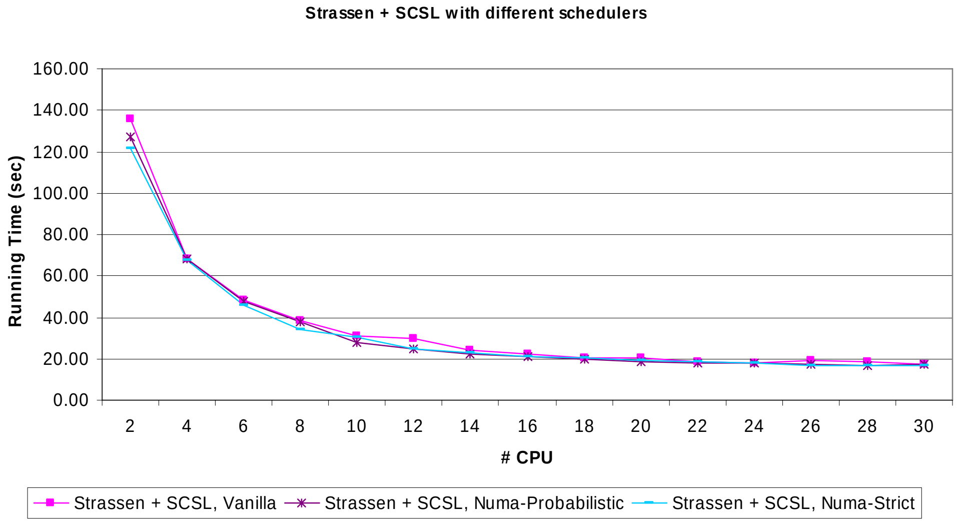

Figure 4.3: The performance of strassen when using different schedulers.

Nevertheless, the NUMA aware schedulers still shows some improvement as can be seen in the next graph. The graph shows only the graph portion for 16 processors and more in order to be able to use a much smaller scale and get a clearer view of the performance when using many processors.

The highest performing combination is the implementation of the Strassen algorithm with the SCSL library for the serial code using the NUMA-strict scheduler with a running time of 16.80 seconds using 30 processors. Compared to the performance of the OpenMP code from the vendor that runs for 18.33 seconds we have improved by close to 10 percent.

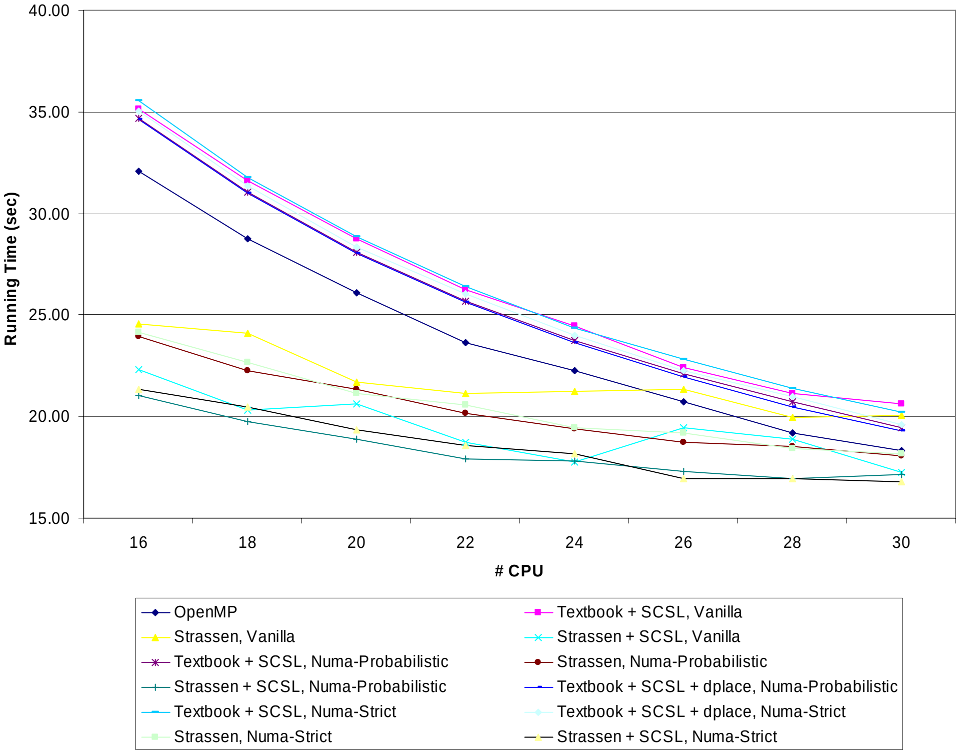

Figure 4.4: Comparing all the high performing programs and schedulers.

Chapter 5

EXPLICIT MEMORY PLACEMENT

This chapter describes the use of explicit memory placement in a parallel Cilk program and checks what are the performance gains that can be achieved by using it. At first, this issue was not on the work plan because Cilk was designed such that as long as the OS memory manager allocates the memory as close as possible to the requesting CPU (usually local to the executing node) the locality should be maximized. To make use of this locality, Cilk programs usually allocate the memory close to the point that the memory is required so that the default memory allocator has the chance to optimize the memory placement and use the closest possible physical memory. When this assumption was checked against the documentation from SGI, it was found that SGI designed Irix to behave just like that.

However, things turned out to be not as they are documented. The issue was checked after the tests of the NUMA-aware schedulers (described in chapter 4) did not yield the expected results, even when using a large number of processors where the locality is an important aspect. After writing some test programs to check this assumption it was clear that the system did not behave quite as it is documented for pthreads based parallel programs (which is the model used by Cilk).

5.1 CHECKING IRIX MEMORY AFFINITY

The fear that actual memory placement was not as documented required a definitive answer as it was at the heart of the NUMA-aware schedulers. The program "`irix_pthreads_mem_affinity.c`" was written in order to test the default placement that is implemented by Irix when executing a pthreads program. The intention is to check how many memory allocation requests are placed on the same node on which the thread is running and how many are placed on a different node, thereby requiring access through a router. The program uses 8 pthreads; each is executing 100 memory allocations. The test system was a 30 processor Origin 2000 machine.

To make things simple for the OS memory manager the program is written such that:

1. Every pthread binds itself to its own CPU as the first thing it does so that every pthread uses a different CPU throughout the test. This means that the pthread is never migrated to another CPU and hence deciding on optimal placement of its memory allocation requests should be obvious when plenty of memory is available on all nodes.

2. Each allocation is of 10KB and is aligned on page size (which is 16KB). The request for alignment of each and every buffer on page size ensures that every allocation will start off a page boundary and so it cannot use memory pages which are already allocated and placed. This means that the memory manager decides the specific physical placement for every allocation request – all placements of previous allocation requests are irrelevant.

3. Every allocated buffer is maintained until the program terminates (i.e.: it is not freed). This ensures that we will not allocate a buffer that was already used and hence its physical placement was set in the previous allocation. Following this lets the memory manager decide on every allocation for a physical placement of the memory pages.

4. The total memory allocated by each pthread is roughly 100*16KB = 1.6MB. This is negligible compared with the 300+ Megabytes of free memory that were available on every node when the tests were executed. So there is no issue of allocating memory on another node because there is no free memory on the local node.

When running the test program on a Origin 2000 machine with 30 processors (2 CPU per node, 8192 MB of memory), the following results were obtained (Each row describes the statistics gathered by one pthread):

Thread ID	Number of allocations on local memory	Number of allocations on non-local memory
1.	20	80
2.	3	97
3.	19	81
4.	18	82
5.	28	72
6.	12	88
7.	22	78
8.	5	95
Summary	127 (16%)	673 (84%)

Figure 5.1: Local and remote memory placement of 100 unique memory allocations (per pthread).

These results show that on average, only 16 percent of the allocation requests are physically placed on the memory node that is local to the CPU that requested the memory while all the other requests are placed on other memory nodes. This is very bad for locality - it shows that the "first touch" principle coupled with "Memory Locality Domains" does not ensure that memory allocations are placed on local memory even when sufficient memory is available. Running the tests several times yields similar results.

From these statistics it is obvious that explicit memory placement has the potential to improve the performance of pthreads based programs and hence Cilk programs which are based on pthreads. These statistics also suggest an explanation of why do all schedulers perform quite the same, since the primary principal on which the NUMA-aware scheduler was based (always allocate local memory) did not hold and in fact it was behaving almost randomly, thereby removing any gain from the biased steal attempts.

The following sections describe the work carried out to explore Explicit Memory Placement and Page Migration in Cilk before surveying the topic on other parallel models. As some of the conclusion of the survey on OpenMP suggest, these results might have been guessed without doing that work but the OpenMP topic was not surveyed early enough.

5.2 MEMORY PLACEMENT EFFECT ON THE NUMA-AWARE SCHEDULER

The problem of bad memory locality placement was highly acute to the NUMA-aware scheduler because the whole idea was to place the threads and then attempt to steal from close by threads for which their local memory is close to the thief - But when the close by neighbor actually allocates memory on a distant node of the system then stealing from it yields no benefit in comparison to the vanilla scheduler.

As a result, a solution was required to allow a Cilk program (which uses the pthreads model rather than the native sthreads model, which is based on sproc, of SGI) to allocate memory on a specific node. This can be coupled with information that the Cilk run-time library has to map worker threads to physical processors and allow the Cilk programmer to have knowledge of the CPU on which the worker thread is running. The programmer can then allocate the memory buffer on the node it is executing. Moreover, such a library will enable large data structures to be physically distributed among multiple nodes thereby further increasing the scalability of the system in cases of huge data sets. Such data structure might use too much memory on one node (hence causing all following allocations from that node to be directed to other nodes) and can also reduce the overall performance because all processors will access the huge data structure on a single physical node and overload the communication links which connect the system nodes with the memory module to which all nodes access.

The Irix documentation points to a vendor supplied library called `dplace` which implements explicit memory placement in a manner that is agnostic to the thread model being used. This makes the library suitable for use in Cilk programs. The use of this library is possible only because of the possibility to compile and link Cilk programs with the vendor compiler which was the first step I've accomplished in this thesis.

5.3 THE DPLACE LIBRARY

The `dplace` library implements the highest level API for physical placement of pages on nodes (other options are to use the Memory Locality Domain APIs which are more complex). The `dplace` library accepts commands regarding memory placement. The primary 2 commands that are used in order to place memory are:

1. `"place range <START> to <END> on memory <NODE>"`.

   This command instructs the OS memory manager to place the specified range of addresses (full memory pages) on the specific memory NODE. This command has effect only if the address range was not physically placed by this process before hand. Once a process places a page on a specific node, the command cannot be used to move the memory page to another node. To do that we need the next command.

2. `"migrate range <START> to <END> to memory <NODE>"`.

   This command instructs the OS memory manager to move the specified range of addresses (full memory pages) from the current node on which they physically reside to the specific memory

NODE. This command should be used when the memory range was already assigned to a memory node and hence it must be migrated.

Using both of these `dplace` commands enables the writing of a memory allocation function that allocates memory with a requested size and physical distribution so that it operates successfully no matter whether the memory it has allocated from the C runtime library (which allocated from the OS memory manager) was used by the process before (and hence it was already placed on a certain node and should be migrated from it to the requested node) or not (hence it should be placed on the specific requested node).

An important thing to note when using the `dplace` library functions in a program is that when we request the `dplace` library to place/migrate a page, it does not take effect immediately. Newly allocated memory-pages are physically placed only when they are first touched (either read or write) and this event occurs only when the page is mapped into the process address space. So placing a memory page and then checking where it is placed without touching it can yield unpredictable results. If the page was already used by the process earlier than this allocation is not the first touch and hence the page needs to be migrated. Migration will happen, again, only when the page is touched and hence checking where the page location is will yield its previous location if not touched since the new allocation was made.

5.4 EXPLICIT MEMORY PLACEMENT IN CILK

In response to the bad locality of memory placement and by using the `dplace` library, the following was implemented in order to solve the problem:

1. A function called “`malloc_place`()” was written to allocate a memory block with specific physical memory layout. This is used to distribute the input matrices over all nodes that participate in the computation. This has the effect that the memory access requests spread over all nodes and hence do not overwhelm a single node for which the memory controller would not be able to keep up. The "`malloc_place`()" function is a memory allocation routine similar to "`malloc`()" but after the memory is allocated it guarantees a physical distribution of the pages across physical nodes as specified by the call parameters. The allocated buffer will have linear addressing while making it possible to select the physical node for every memory page. The memory should be freed as usual by the "`free`()" C runtime routine.

2. A new keyword was added to the Cilk language called “`Self_CPU`”. It has similar usage as the Cilk keyword “`Self`” which evaluate to the worker ID that is executing the code. The new keywords, however, evaluates to the physical CPU which is executing the thread that executes the code. Using this keyword the code can use the “`malloc_place`()” function to allocate memory on the node that executes it.

3. A new API was added to the Cilk runtime library with which every Cilk program links – the API enables the Cilk program to retrieve the list of processor IDs which are participating in the computation. Knowing the

processor IDs enables a worker thread to allocate a memory buffer and request a specific placement.

With these 3 new features, the final versions of matmul.cilk and strassen.cilk are implemented. These versions use the malloc_place() API to allocate memory. This should have improved their performance as follows:

1. The fact that the memory is allocated on the local node should increase the performance of the programs when they use the NUMA-aware scheduler.
   Note however that the actual performance is not easily predictable as the dplace tool documentation indicates because the migration of memory pages is not so cheap, while the L1 + very large L2 (8 MB on the test machine) caches hide much of the access latency of remote memory.

2. The initial memory for the 3 matrices (C=A*B) is evenly spread across all the nodes that participate in the parallel computation. This means that the memory access to the matrices will be spread to all nodes rather than to just a single one. This is highly important for the matmul.cilk program which uses no temporary buffers – the output matrix is computed directly from the input matrices and so all processors read from A and B and write to C throughout the computation.

We first look at the performance of the matmul program. This program uses no temporary buffers and hence uses no memory allocation within the recursive algorithm. This means that the performance of the "malloc_place()" API has no effect on the performance of this code.

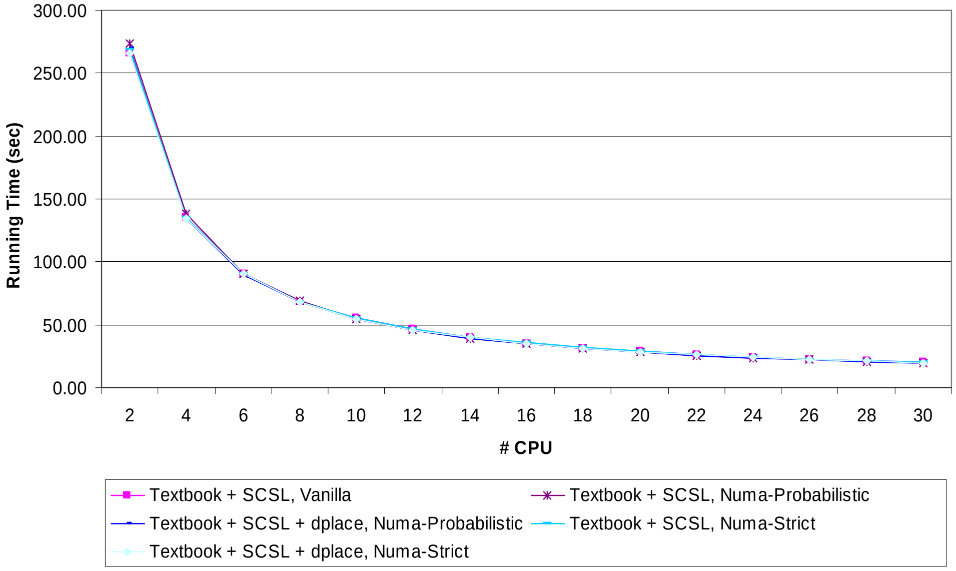

Figure 5.2: Performance of matmul w/o the use of dplace on the various schedulers.

From the graph above, it seems that there is no performance difference between the 3 schedulers, whether dplace was used or not. In this case, the extremely high cache hit ratio is the only explanation to this behavior. Using hardware counters we see that with a 30 processor test of the strassen algorithm, the test completed in 69.6 seconds with about 99% L2 data cache hit ratio for each processor. With such a high L2 cache hit-rate, the placement of the memory is insignificant. These are executions that use the standard memory allocation which has the bad locality as described previously.

With this low penalty of cache misses it is no wonder that the addition of new scheduler and explicit memory placement do not affect the overall performance too much. This further emphasize that additional research is required with parallel algorithms that have intrinsically lower cache hit ratio than matrix multiplication, one such algorithm is implemented by the `strassen.cilk` program.

We next turn to look at the performance of the strassen algorithm in which the nature of the algorithm requires temporary storage in each new activation frame of the recursive algorithm and hence this code is affected by the performance of the "`malloc_place()`" API and the initial memory placement of the operating system.

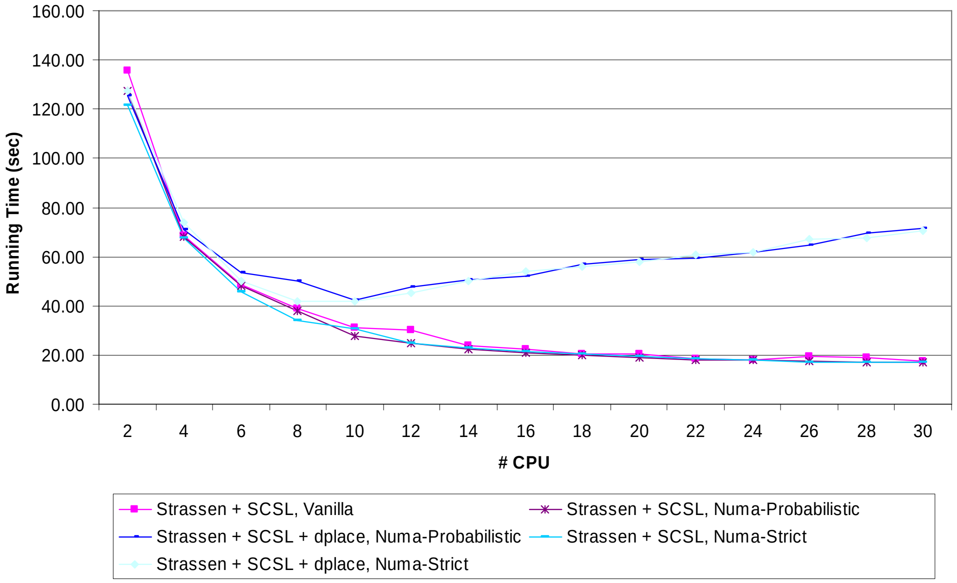

Figure 5.3: Performance of strassen w/o the use of `dplace` on the various schedulers.

Looking at the performance of the strassen implementation which used `dplace`, it is obvious that something is definitely wrong. Theoretically, `dplace` is supposed to improve performance, especially when large number of processors are used but instead we see the opposite as it is very slow compared to the same code which uses simple `malloc`(). Note that both versions use the same (SCSL library) base case.

After testing the performance of the placement library it has become obvious that when large buffers are allocated, it takes the OS kernel a lot of time to place the pages on the requested nodes. This means that for a program which makes many/large allocations (such as the strassen implementation which requires large temporary storage); the cost of the placement exceeds the possible gain (with the high cache hit ratio). Note that this migration cost occurs because we usually allocate memory which is not on the requested (local) node and hence the memory must be migrated to fulfill the request. If a different memory manager is integrated into the Cilk system (e.g. Hoard) then this overhead might be eliminated altogether by managing the memory of each node in a separate pool and allocating from the proper pool as requested by the memory allocation. Such a memory manager would obviously use `dplace` for placement but it will do so very rarely when it needs to increase the pool size rather than upon every memory allocation request.

The matmul program which is a simple textbook solution uses no temporary storage and as such it can only benefit from the placement since the placement time is counted on the initialization part of the program rather than on the actual computation of the result. This version gains a slight performance improvement but it is so small because of the high cache hit ratio of the L1 and L2 caches which mitigate the latency of distant memory.

One more set of performance tests was executed using the parallel cholesky factorization code that is found in the TAUCS library (see http://www.cs.tau.ac.il/~stoledo/taucs/). this code shows high serial performance and also pretty good scalability (originally tested on SGI Origin 3000 system) but once too many processors are used, its performance drops significantly. It is an algorithm with much more complex memory access pattern and intrinsically lower cache hit ratio.

The same code was tested with the NUMA aware version of Cilk in order to check whether the NUMA aware scheduler and the memory placement capabilities can improve the scalability of the implementation. The code was tested on an input matrix that was used in the original performance tests in [21] called “Threaded Connector/contact problem“. Because the authors used the SGI Origin 3000 system and this thesis was executed on the SGI Origin 2000 system, the tests were also executed with the original code in order to have a reference.

The following graph shows the performance of the original code found within TAUCS version 2.2 using the 3 schedulers that are now supported by Cilk.

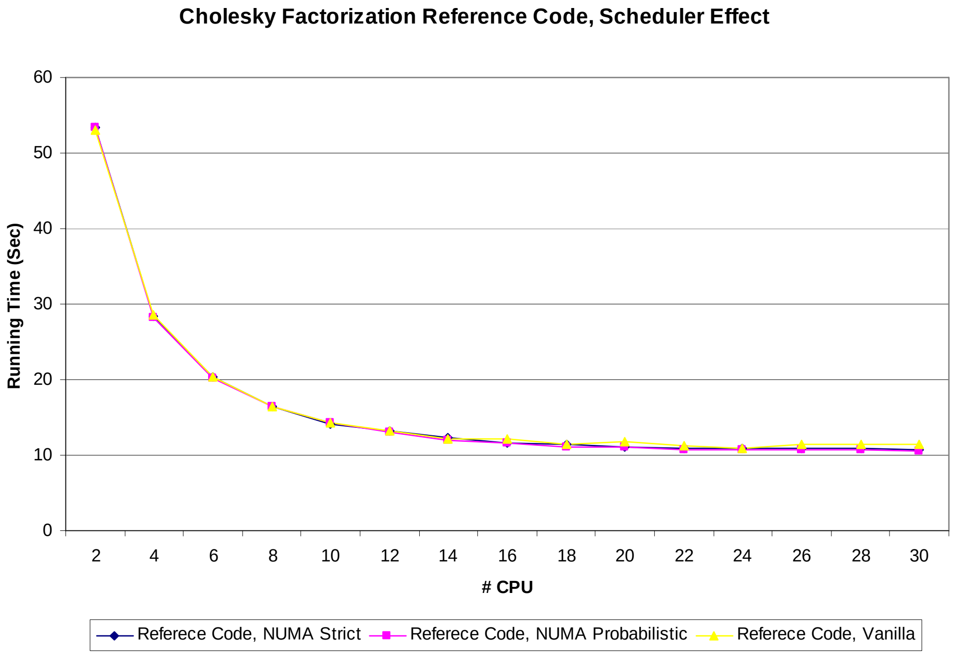

Figure 5.4: Performance of cholesky factorization with different schedulers.

The original code shows performance that improve quite well when using up to 12 processors but unlike the graph that is seen in figure 2.7, the performance do not drop when adding more processors but rather simply does not improve. Keeping in mind that the Origin 3000 is more scalable than the Origin 2000 (4 processors per system board instead of only 2, double bandwidth per NUMALink, etc) it was expected that the performance drop would occur with even less processors but this has not happened. This behavior is seen with both the original code and the improved code that uses memory placement to distribute the matrices over all the nodes that participate in the computation.

The following graph in figure 5.5 shows the performance of the cholesky factorization code after it was improved to distribute the input matrix over all the nodes that participate in the computation. This enabled us to verify the second speculation that was made in [21] for the reason of the performance drop. The speculation is that the allocation of the matrix using a single memory allocation places it on a single memory node and this will overload the memory subsystem when too many processors attempt to access the matrix.

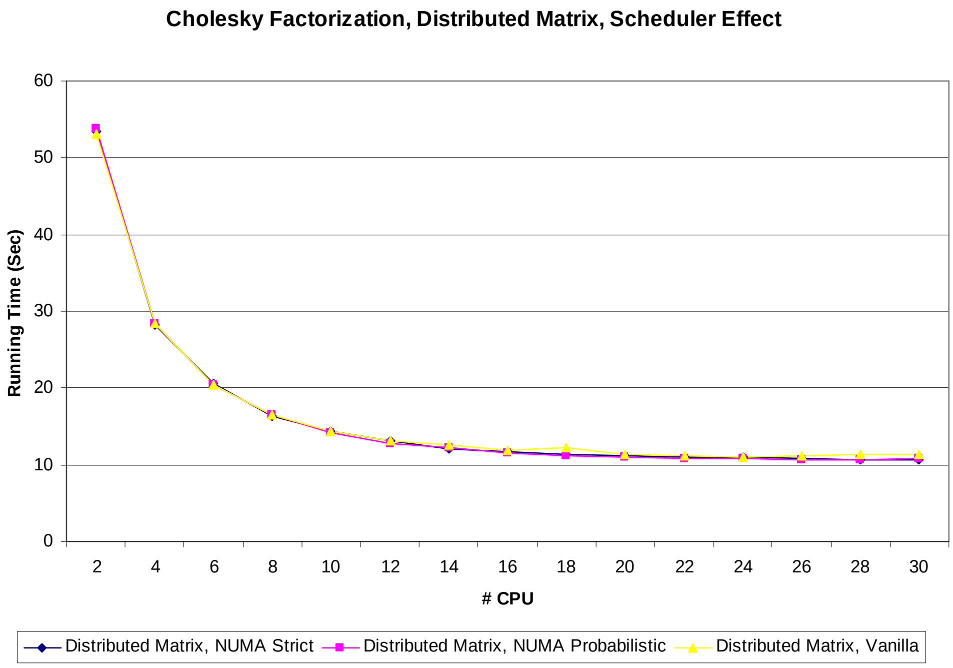

Figure 5.5: Performance of cholesky factorization with distributed matrix and different schedulers.

The graph turned out to be quite the same and means that neither the scheduling policy, not the distribution of the matrix solves the bottlenecks of the cholesky factorization algorithm that is found in TAUCS.

One thing to note about these performance measures is that they were executed slightly different than in [21]. In [21] the authors wanted to compare the performance of their implementation with the performance of other implementations and hence measured performance of the cholesky factorization code (which takes most of the execution time) and added the running time of the symbolic analysis phase. In this thesis, I was interested in the scalability and performance of the parallel Cilk code and hence did not add the running time of the symbolic analysis which is not parallelized (using Cilk or anything else) and have no use here.

The research of possible optimization that would be available with the integration of a parallel scalable memory manager such as Hoard is left for future work.

Chapter 6

TEST PROGRAMS USED TO UNDERSTAND/VERIFY SYSTEM BEHAVIOR

This chapter describes some of the test programs which were written in order to understand how the Irix OS implements various issues and to check the performance changes on select operations.

6.1 VIRTUAL ADDRESS -> PHYSICAL ADDRESS

The Irix OS manages the distributed memory of a large NUMA machine (such as Origin 2000 and Origin 3000 series) and presents a shared memory programming model. The OS maintains a 64-bit address space and assigns a process with an address space of 2^31 (for 32-bit programs) or 2^40 (for 64-bit programs). When the physical address is presented by the CPU to the HUB that connects it to the system, the hub uses some of the high–order bits to identify the node on which the required memory page is at. It then directs the request over a communication link to that node. The HUB on that node will receive the request and service it from its locally accessed memory.

The *va2pa* routine (implemented in the `va2pa.c` file) is a routine which is given as input an address. The routine returns the node on which that memory address is physically stored. This information is not usually required when programming a NUMA machine but it was used heavily in test programs in order to find out how to force a certain physical distribution of allocated memory (it can be found at [29]).

6.2 THE UT_MALLOC_PLACE.C PROGRAM

This program demonstrate how to allocate a buffer with size that is a multiple of the system page size and have a specific physical distribution of the buffer (which has linear addressing) over the requested physical nodes.

The program uses the "`malloc_place`()" function for the memory allocation. It contains code to demonstrate how to achieve some physical distributions. This can be used to test that memory is allocated as requested (of course it will not have proper physical placement in case memory is not available on the requested node but it can be found on another node).

6.3 THE DPLACE TOOL

The term "*dplace*" stands for 2 different things:

- The name of a stand alone development tool which accepts a binary program to execute and a placement file that specifies the system nodes to use, processors to execute the program threads, memories on which the memory pages are physically placed, etc. Because the tool accepts the placement commands together with the binary program, the commands cannot be adapted to dynamic behavior of the program (e.g.: a dynamically allocated memory buffer has an address range that is unknown in advance and hence that memory buffer cannot be affected by these commands).

- The name of a library with which any program can link. The library exports 2 API functions. One accepts the name of a file containing commands to be executed – this can be used for implementing the stand alone tool using this library. The second accepts one command at a time. These APIs can be used from within a program by inserting calls to the *dplace* library from within the code. Such commands can be generated dynamically and are the ones used for the implementation of the "`malloc_place`()" function.

An example of a *dpalce* commands file for the stand alone tool is the "dplace_spec_1.txt" file which can be used with some Cilk binary that is instructed to use 2 worker threads (the file addresses 3 threads but one is the main thread created by the OS, not a Cilk worker thread). Using the standard UNIX "ps –T" we can see the processor which executes the program and verify it is the same as we've requested.

An example pthreads program that demonstrates the capabilities of the *dplace* library (actually used during development to make sure the behavior of the library is well understood) is "`dplace_spec_1.c`". The program generates commands to place the pthreads on specific system nodes, allocate memory with a specific physical distribution and then change the memory placement to yet another physical distribution. this demonstrate the use of dynamic memory in a typical program in which the memory is allocated and freed such that in the C runtime layer and the OS memory management layer, the memory is actually recycled and reused.

6.4 WHICH PROCESSOR IS EXECUTING A THREAD

The program "get_executing_cpu.c" demonstrates how to find the processor ID which is executing the specified thread. This information doesn't seem to be available anywhere on the web and it is document in hope that it will help others who need to use this information.

6.5 BINDING PTHREDS TO PROCESSORS

The program "thread_cpu_binding.c" demonstrates how to create pthreads, bind them to specific processors and while they are running, query the OS about the processor executing every pthread, making sure that it is the one we've asked to bind to.

The program shows the use of a non standard pthreads attribute "PTHREAD_SCOPE_BOUND_NP" which is available in the pthreads implementation from SGI in order to allow binding of threads to processors.

Chapter 7

CONCLUSION

In this work I have attempted to improve the scalability and performance of Cilk programs on NUMA machines using a simple observation that the work stealing policy should take the NUMA architecture into consideration when it selects the victim from which the work stealing is attempted. The assumption is that every processor allocates memory and the memory will be placed on the local node. This means that we should prefer to steal work from nearby processors because its memory will be closer and the latency to access it is shorter. The implementation required preliminary phases in order to be able to use the vendor libraries later on with which the NUMA aware features can be implemented. These phases include making Cilk compliant with ANSI C and enabling compilation and link with non `gcc` compilers. These phases are worth on their own because the freedom to select out of multiple compilers can be used to generate higher performance code and indeed tests that were made showed substantial performance gained by using advanced compilers (e.g.: SGI MIPS Pro, Intel C Compiler). However, the scalability tests of the NUMA aware scheduler which is the main theme of this thesis did not yield the expected improvements. I conjecture that the main reason for that is a limitation of the Irix OS when allocating memory – it does not allocate the memory on the local node. This undermines the basics of memory locality and an attempt was made to overcome this problem using memory migration techniques. However, this feature had yet another limitation on its bandwidth – it was so slow that it was very hard to justify the time it takes to migrate/place the memory. These two problems have major effect on the NUMA aware scheduler and brought this work into a dead end.

These limitations can be solved either by improvements in the Irix OS or by integrating a memory manager that is capable of maintaining separate memory pools per system node. The first is out of the hands of such research and the second introduced work that was out of scope for this thesis. It was decided to leave it for future work.

Chapter 8

APPENDIXES

APPENDIX A – INLET TRANSFORMATIONS

```
cilk int test_abort_aux(int d, int b, int seek_item, int arg)
{
  int i;
  int done;
  int *expected_value;
  int *expected_child;

  inlet void catch(int res, int index) {
    int k;

    if (done)
      Cilk_die("bug --- done != 0\n");
    if (!expected_child[index])
      Cilk_die("unexpected child\n");
    expected_child[index] = 0;

    if (res == -2) {
      done = 1;
      for (k = 0; k < branch; k++)
        expected_child[k] = 0;
      abort;
    } else {
      if (res != expected_value[index])
        Cilk_die("wrong return value\n");
    }
  } /* end of inlet*/

  expected_value = Cilk_alloca(branch * sizeof(int));
  expected_child = Cilk_alloca(branch * sizeof(int));

  for (i = 0; i < branch; i++)
    expected_child[i] = 0;

  done = 0;

  if (d == depth) {
    if (b == seek_item)
      return -2;
    return arg - 200;
  }
  for (i = 0; i < branch; i++) {
    expected_child[i] = 1;
    expected_value[i] = (Cilk_rand() & 0xFFFFF) + 42;
    catch(spawn test_abort_aux(d + 1, b * branch + i, seek_item,
                               expected_value[i] + 200), i);
    if (done)
      break;
  }

  sync;

  for (i = 0; i < branch; i++)
    if (expected_child[i] != 0)
      Cilk_die("child did not return\n");

  if (done)
    return -2;
  else
    return arg - 200;
} /* end of Cilk procedure */
```

Figure A.1: A Cilk procedure using inlet from the testall.cilk example (this example comes with the installation and running it verifies that various features of the language are working as expected

```
int
test_abort_aux (CilkWorkerState * const _cilk_ws,
                int d,
                int b,
                int seek_item,
                int arg)
{
  struct _cilk_test_abort_aux_frame *_cilk_frame;
  CILK2C_INIT_FRAME (_cilk_frame,
                     sizeof (struct _cilk_test_abort_aux_frame),
                     _cilk_test_abort_aux_sig);
  CILK2C_START_THREAD_FAST ();
  {
    int i;
    int done;
    int *expected_value;
    int *expected_child;
#undef CILK_WHERE_AM_I
#define CILK_WHERE_AM_I IN_FAST_INLET
# 33
    void catch (int res, int index)
    {
      int k;

      if (done)
        Cilk_die_external (_cilk_ws->context, "bug --- done != 0\n");;
      if (!expected_child[index])
        Cilk_die_external (_cilk_ws->context, "unexpected child\n");;
      expected_child[index] = 0;

      if (res == -2)
      {
        done = 1;

        for (k = 0; k < branch; k++)
          expected_child[k] = 0;
        /* abort */ ;
      } else {
        if (res != expected_value[index])
          Cilk_die_external (_cilk_ws->context,
                             "wrong return value\n");
      }
    } /* end of inlet */

    expected_value = Cilk_alloca (branch * sizeof (int));
    expected_child = Cilk_alloca (branch * sizeof (int));

    for (i = 0; i < branch; i++)
      expected_child[i] = 0;

    done = 0;

    if (d == depth)
    {
      if (b == seek_item)
      {
        int _cilk_temp0 = -2;
        CILK2C_BEFORE_RETURN_FAST ();
        return _cilk_temp0;
      }
      {
        int _cilk_temp1 = arg - 200;
        CILK2C_BEFORE_RETURN_FAST ();
        return _cilk_temp1;
      }
    }
    for (i = 0; i < branch; i++)
    {
      expected_child[i] = 1;
      expected_value[i] = (Cilk_rand () & 0xFFFFF) + 42;
      {
        int _cilk_temp2;
        _cilk_frame->scope2.index = i;
        _cilk_frame->header.entry = 1;
        _cilk_frame->scope0.d = d;
        _cilk_frame->scope0.b = b;
        _cilk_frame->scope0.seek_item = seek_item;
        _cilk_frame->scope0.arg = arg;
        _cilk_frame->scope1.i = i;
        _cilk_frame->scope1.done = done;
        _cilk_frame->scope1.expected_value = expected_value;
        _cilk_frame->scope1.expected_child = expected_child;
```

```
        CILK2C_BEFORE_SPAWN_FAST ();
        CILK2C_PUSH_FRAME (_cilk_frame);
        _cilk_temp2 =
        test_abort_aux (_cilk_ws, d + 1, b * branch + i, seek_item,
        expected_value[i] + 200);
        {
          int __tmp;
          CILK2C_XPOP_FRAME_RESULT (_cilk_frame, 0, _cilk_temp2);
        }
        CILK2C_AFTER_SPAWN_FAST ();
        catch (_cilk_temp2, i);
      }

      if (done)
        break;
    }

    CILK2C_AT_SYNC_FAST ();

    for (i = 0; i < branch; i++)
      if (expected_child[i] != 0)
        Cilk_die_external (_cilk_ws->context,
                           "child did not return\n");

    if (done)
    {
      int _cilk_temp3 = -2;
      CILK2C_BEFORE_RETURN_FAST ();
      return _cilk_temp3;
    }
    else
    {
      int _cilk_temp4 = arg - 200;
      CILK2C_BEFORE_RETURN_FAST ();
      return _cilk_temp4;
    }
  }
} /* end of procedure */
```

Figure A.2: The original code that was generated by `cilk2c` to implement inlets using inner functions.

```
#undef CILK_WHERE_AM_I
#define CILK_WHERE_AM_I IN_FAST_INLET
# 993
void
_cilk_test_abort_aux_catch_inlet_fast (int res,
                                       int index,
struct _cilk_test_abort_aux_frame   *_cilk_frame,
CilkWorkerState * const _cilk_ws)
{
  int k;
  if (_cilk_frame->scope1.done)
    Cilk_die_external (_cilk_ws->context, "bug --- done != 0\n");

  if (!_cilk_frame->scope1.expected_child[index])
    Cilk_die_external (_cilk_ws->context, "unexpected child\n");
  _cilk_frame->scope1.expected_child[index] = 0;

  if (res == -2)
  {
    _cilk_frame->scope1.done = 1;

    for (k = 0; k < branch; k++)
      _cilk_frame->scope1.expected_child[k] = 0;
      /* abort */ ;
  }
  else
  {
    if (res != _cilk_frame->scope1.expected_value[index])
      Cilk_die_external (_cilk_ws->context, "wrong return value\n");
  }
} /* end of inlet */

#undef CILK_WHERE_AM_I
#define CILK_WHERE_AM_I IN_FAST_PROCEDURE
# 986
int
test_abort_aux (CilkWorkerState * const _cilk_ws,
                int d,
                int b,
                int seek_item,
                int arg)
{
  struct _cilk_test_abort_aux_frame *_cilk_frame;
  CILK2C_INIT_FRAME (_cilk_frame,
                     sizeof (struct  _cilk_test_abort_aux_frame),

  CILK2C_START_THREAD_FAST ();
  {
    int i;
    int done;
    int *expected_value;
    int *expected_child;
/*------- Fast inlet as inner function was removed ------*/
# 1015
    expected_value = Cilk_alloca (branch * sizeof (int));
    expected_child = Cilk_alloca (branch * sizeof (int));

    for (i = 0; i < branch; i++)
      expected_child[i] = 0;

    done = 0;

    if (d == depth)
    {
      if (b == seek_item)
      {
        int _cilk_temp110 = -2;
        CILK2C_BEFORE_RETURN_FAST ();
        return _cilk_temp110;
      }
      {
        int _cilk_temp111 = arg - 200;
        CILK2C_BEFORE_RETURN_FAST ();
        return _cilk_temp111;
      }
    }
    for (i = 0; i < branch; i++)
    {
      expected_child[i] = 1;
      expected_value[i] = (Cilk_rand () & 0xFFFFF) + 42;
      {
        int _cilk_temp112;
        _cilk_frame->scope2.index = i;
        _cilk_frame->header.entry = 1;
        _cilk_frame->scope0.d = d;
        _cilk_frame->scope0.b = b;
        _cilk_frame->scope0.seek_item = seek_item;
        _cilk_frame->scope0.arg = arg;
        _cilk_frame->scope1.i = i;
        _cilk_frame->scope1.done = done;
        _cilk_frame->scope1.expected_value = expected_value;
        _cilk_frame->scope1.expected_child = expected_child;
        CILK2C_BEFORE_SPAWN_FAST ();
        CILK2C_PUSH_FRAME (_cilk_frame);
        _cilk_temp112 =
        test_abort_aux (_cilk_ws, d + 1, b * branch + i,
                        seek_item, expected_value[i] + 200);
        {
          int __tmp;
          CILK2C_XPOP_FRAME_RESULT (_cilk_frame, 0, _cilk_temp112);
        }
        CILK2C_AFTER_SPAWN_FAST ();
        _cilk_test_abort_aux_catch_inlet_fast (_cilk_temp112, i,
                                               _cilk_frame, _cilk_ws);
# 1031
  /* Restore variables from frame */
# 1031
        {
          d = _cilk_frame->scope0.d;
          b = _cilk_frame->scope0.b;
          seek_item = _cilk_frame->scope0.seek_item;
          arg = _cilk_frame->scope0.arg;
          i = _cilk_frame->scope1.i;
          done = _cilk_frame->scope1.done;
          expected_value = _cilk_frame->scope1.expected_value;
          expected_child = _cilk_frame->scope1.expected_child;
        }
      }

      if (done)
        break;
    }

    CILK2C_AT_SYNC_FAST ();
    for (i = 0; i < branch; i++)
        if (expected_child[i] != 0)
          Cilk_die_external (_cilk_ws->context,
                             "child did not return\n");

    if (done)
    {
      int _cilk_temp113 = -2;
      CILK2C_BEFORE_RETURN_FAST ();
      return _cilk_temp113;
    }
    else
    {
      int _cilk_temp114 = arg - 200;
      CILK2C_BEFORE_RETURN_FAST ();
      return _cilk_temp114;
    }
  }
} /* end of Cilk procedure */
```

Figure A3: The new code that is generated by *`cilk2c`* to implement inlets using standard (global scope) functions.

BIBLIOGRAPHY

[1] Cilk 5.3 Reference Manual

[2] Robert D. Blumofe, Christopher F. Joerg, Bradley C. Kuszmaul, Charles E. Leiserson, Keith H. Randall, and Yuli Zhou. Cilk: An efficient multithreaded runtime system. In Proceedings of the Fifth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP), pages 207-216, Santa Barbara, California, July 1995.

[3] Siddhartha Chatterjee, Alvin R. Lebeck, Praveen K. Patnala, Mithuna Thottethodi Recursive Array Layouts and Fast Matrix Multiplication

[4] Robert D. Blumofe. Executing Multithreaded Programs Efficiently. PhD thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, September 1995.

[5] Robert D. Blumofe 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] Keith H. Randall, "Cilk: Efficient Multithreaded Computing", 1998.

[7] Christopher F. Joerg. The Cilk System for Parallel Multithreaded Computing. PhD thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, January 1996.

[8] Silicon Graphics man page for MIPS Pro compiler.

[9] http://www.boost.org/

[10] Field-testing IMPACT EPIC - research results in Itanium 2, John W. Sias, Sain-zee Ueng, Geoff A. Kent, Ian M. Steiner, Erik M. Nystrom, Wen-mei W. Hwu, 31st Annual International Symposium on Computer Architecture, 2004.

[11] http://www.hoard.org/ - hoard is a highly scalable memory manager for multithreaded/multiprocessor systems.

[12] D. Lenoski, J. Laudon, T. Joe, D. Nakahira, L. Stevens, A. Gupta, and J. Hennessy. The DASH prototype: Logic overhead and performance. IEEE Transactions of Parallel and Distributed Systems, pages 41-61, vol. 4, no. 1, January 1993.

[13] J. Kuskin, D. Ofelt, M. Heinrich, J. Heinlein, R. Simoni, K. Gharachorloo, J. Chapin, D. Nakahira, J. Baxter, M. Horowitz, A. Gupta, M. Rosenblum, and J. Hennessy. The Stanford FLASH multiprocessor. In Proceedings of the 21st International Symposium on Computer Architecture, pages 302-313, April 1994.

[14] A. Agarwal, R. Bianchini, D. Chaiken, K. L. Johnson, D. Kranz, J. Kubiatowicz, B.-H. Lim, K. Mackenzie, and D. Yeung. The MIT Alewife machine: Architecture and performance. In Proceedings of the 22nd International Symposium on Computer Architecture, pages 2-13, June 1995.

[15] Topics in Irix Programming, Document Number 007-2478-008, 1996 - 2000, Silicon Graphics, Inc.

[16] M.Sc Thesis, by Nathan Robertson, 2002.

[17] Dimitrios S. Nikolopoulos, Theodore S. Papatheodorou Constantine D. Polychronopoulos, Jes´us Labarta and Eduard Ayguad. Is Data Distribution Necessary in OpenMP?

[18] M. Marchetti, L. Kontothanassis, R. Bianchini and M. Scott. Using Simple Page Placement Policies to Reduce the Cost of Cache Fills in Coherent Shared-Memory Systems. Proceedings of the 9th International Parallel Processing Symposium, pp. 480–485. Santa Barbara, CA, April 1995.

[19] Dimitrios S. Nikolopoulos, Theodore S. Papatheodorou, Constantine D. Polychronopoulos, Jes´us Labarta and Eduard Ayguad. Case for User-Level Dynamic Page Migration.

[20] L. Noordergraaf and R. Van der Pas. Performance Experiences on Sun's Wildfire Prototype. In Proc. Of Supercomputing 99, November 1999.

[21] Dror Irony, Gil Shklarski, Sivan Toledo. Parallel and fully recursive multifrontal sparse Cholesky. In Future Generation Computer Systems (20) 2004, pg. 425-440.

[22] http://homepages.cwi.nl/~robertl/mash/numaflex the Origin 2000/3000 architecture and a survey of SMP/DSM acronyms and slang.

[23] Silicon Graphics Inc. Origin 2000 and Onyx2 Performance Tuning and Optimization Guide. http://techpubs.sgi.com, 1999

[24] V. Soundararajan, Flexible Use of Memory for Replication/Migration in Cache-Coherent DSM Multiprocessors. In Proc. Of the 25th International Symposium on Computer Architecture, pages 342-355, June 1998.

[25] B. Verghese, S. Devine, A. Gupta and M. Rosenblum. Operating System Support for Improving Data Locality on CC-NUMA Compute Servers. In Proc. Of the 7th International Conference on Architectural Support for Programming Languages and Operating systems, pages 279-289, October 1996.

[26] J. Laudon and D. Lenoski, The SGI Origin: A ccNUMA Highly Scalable Server. In Proc. Pf the 24th International Symposium on Computer Architecturem pages 171-181, June 1997.

[27] E. Hagersten and M. Koster. Wildfire: A Scalable Path for SMPs. In Proc. Of the 5th International Symposium on High Performance Computer Architecture, pages 172-181, January 1999.

[28] D. Jiang and J. P. Singh. Scaling Applications Performance on a Cache-Coherent Multiprocessor. In Proc. Of the 26th International Symposium on computer Architecture, pages 305-316, May 1999.

[29] http://techpubs.sgi.com/library/tpl/cgi-bin/getdoc.cgi/0650/bks/SGI_Developer/books/OrOn2_PfTune/sgi_html/apc.html