Computer Science Technical Report

Predicting Remote Reuse Distance Patterns in Unified Parallel C Applications

by

**Steven Vormwald, Steven Carr,
Steven Seidel and Zhenlin Wang**

**Computer Science Technical Report
CS-TR-09-02**

December 18, 2009

**Department of Computer Science
Houghton, MI 49931-1295
www.cs.mtu.edu**

ABSTRACT*

Productivity is becoming increasingly important in high performance computing. Parallel systems, as well as the problems they are being used to solve, are becoming dramatically larger and more complicated. Traditional approaches to programming for these systems, such as MPI, are being regarded as too tedious and too tied to particular machines. Languages such as Unified Parallel C attempt to simplify programming on these systems by abstracting the communication with a global shared memory, partitioned across all the threads in an application. These Partitioned Global Address Space, or PGAS, languages offer the programmer a way to specify programs in a much simpler and more portable fashion.

However, performance of PGAS applications has tended to lag behind applications implemented in a more traditional way. It is hoped that cache optimizations can provide similar benefits to UPC applications as they have given single-threaded applications to close this performance gap. Memory resuse distance is a critical measure of how much an application will benefit from a cache, as well as an important piece of tuning information for enabling effective cache optimization.

This research explores extending existing reuse distance analysis to remote memory accesses in UPC applications. Existing analyses store a very good approximation of the reuse distance histogram for each memory access in a program efficiently. Reuse data are collected for small test runs, and then used to predict program behavior during full runs by curve fitting the patterns seen in the training runs to a function of the problem size. Reuse data are kept for each UPC thread in a UPC application, and these data are used to predict the data for each UPC thread in a larger run. Both scaling up the problem size and the increasing the total number of UPC threads are explored for prediction. Results indicate that good predictions can be made using existing prediction algorithms. However, it is noted that choice of training threads can have a dramatic effect on the accuracy of the prediction. Therefore, a simple algorithm is also presented that partitions threads into groups with similar behavior to select threads in the training runs that will lead to good predictions in the full run.

*This work is partially supported by NSF grant CCF-0833082.

CHAPTER 1

Introduction

1.1 Motivation

High performance computing is becoming an increasingly important part of our daily lives. It is used to determine where oil companies drill for oil, to figure out what the weather will be like for the next week, to design safer buildings and vehicles. Companies save millions of dollars every year by simulating product designs instead of creating physical prototypes. The movie industry relies heavily on special effects rendered on large clusters. Scientists rely on simulations to understand nuclear reactions without having to perform dangerous experiments with nuclear materials.

In addition, the machines used to carry out these computations are becoming drastically larger and more complicated. The Top500 list claims that the fastest supercomputer in the world has over 200000 cores [1]. It is made up of thousands of six-core opteron processors in compute blades networked together. The compute blades can each be considered a computer in its own right, working together with the others to act as one large supercomputer. This clustering model of supercomputer is now the dominant force in high performance computing.

Traditionally, applications written for these clusters required the programmer to explicitly manage the communication needs of the program across the various nodes of the cluster. It was thought that the performance needs of such applications could only be met by a human programmer carefully designing the program to minimize the necessary communication costs. This approach to programming for supercomputers is quickly becoming unwieldy. The productivity cost of requiring the application programmer to manage and tune an application's communication for these complex systems is simply too high.

Partitioned global address space languages, such as Co-Array Fortran [2] and Unified Parallel C [3], attempt to address these productivity concerns by building a shared memory programming model for programmers to work with, delegating the task of optimizing the necessary communication to the language implementor.

While these languages do offer productivity improvements, implementations haven't been able to match the performance of more traditional message passing setups. Various approaches to catching up have been tried. The UPC implementation from the University of California Berkeley [4] uses the GASNet network library [5], which attempts to optimize communication using various methods such as message coalescing [6]. Many implementations try to split synchronous operation into an asynchronous operation and a corresponding wait, then spread these as far apart as possible to hide communication latency. These optimizations can

lead to impressive performance gains, but there is still a performance gap for some applications.

One approach hasn't been commonly used in implementations is software caching of remote memory operations. Caching has been used to great effect in many situations to hide the cost of expensive operations. Programmers are also accustomed to working with caches, since they are so prevalent in today's CPUs. As Marc Snir pointed out in his keynote address to the PGAS2009 conference [7], programmers would like to see some kind of caching in these languages' implementations. He demoed a caching scheme implemented entirely in the application. However, it is desirable that the caching be done at the level of the language implementation to avoid forcing the programmer to deal with the complexities of communication that these languages were designed to hide.

This research takes an initial look at the possibility of using existing algorithms for single threaded applications designed to predict patterns in the reuse distances for memory operations to predict patterns in the reuse distances for remote memory operations in Unified Parallel C applications. It is hoped that this information could be used to tune cache behavior for caching remote references, and to enable other optimizations that rely on this information and have been successfully used with single-threaded applications to work with multi-threaded UPC applications.

1.2 Thesis Outline

The rest of this document is organized as follows. Chapter 2 gives a broad background in instruction-based reuse distance analysis and Unified Parallel C. Chapter 3 introduces the instrumentation, test kernels and models used to predict remote memory behavior. Chapter 4 shows the prediction results obtained. Finally Chapter 5 summarizes the results, looks at ways this prediction model can be used, and possible future work to overcome some of this model's weaknesses.

CHAPTER 2

Background

2.1 Parallel Computation

When one wishes to solve problems faster, there are generally only three things to do. First, one can try to find a better algorithm to solve the problem at hand. While this can lead to massive performance benefits, most common problems already have known optimal solutions. Second, one can increase the rate at which a program executes. If a program requires the execution of 10000 instructions, a machine that can execute an instruction every millisecond will finish the program much sooner than one that can only execute an instruction every second. Finally, one can execute the program in parallel–solving multiple pieces of the problem at the same time. Just as having multiple checkout lanes speed consumers through their purchases, having the ability to work on multiple pieces of a problem can speed its solution.

There are many different ways to think about solving a problem in parallel. Flynn's taxonomy classifies parallel programming models up along two axes, one based on the program's control, the other based on its data [8, 9]. This creates four classes of programs: single instruction single data, multiple instruction single data, single instruction multiple data, and multiple instruction multiple data. These are henceforth referred to by the acronyms SISD, MISD, SIMD, and MIMD respectively.

The SISD model is the classic, non-parallel programming model where each instruction is executed serially and works on a single piece of data. While modern architectures actually do work in parallel, this remains the most commonly used abstraction for software developers.

The MISD model is widely considered nonsensical, as it refers to multiple instructions being executed in parallel operating on the same data. While this usually makes little sense, the term has been used to refer to redundant systems, which use parallelism not to speed up a computation, but rather to prevent program failures.

The SIMD model sees widespread use in many special purpose accelerators. In this model, a single instruction is executed in parallel across large chunks of data. The most well-known use of this is probably in the graphics processing units, or GPUs, that have become standard on modern personal computers. Many architectures also include SIMD extensions to their SISD instruction set that enable certain types of applications to perform much better.

The MIMD model is the most general, where multiple instructions are run in parallel, each on different data. This model is perhaps the most commonly used

abstraction for software developers writing parallel applications, as this model is used in the threading libraries included with many operating systems.

2.1.1 SPMD Model

The Single Program, Multiple Data, or SPMD model of parallel programming is a subset of the MIMD model from Flynn's taxonomy. The MIMD model can be thought of as multiple SISD threads executing in parallel that have some means of communicating amongst themselves. The SPMD model is a special case where each thread is executing the same program, only working with different data. The threads need not operate in lock-step, nor all follow the same control-flow through the program.

2.1.2 Communication and the Shared Memory Model

Flynn's taxonomy describes how a problem can be broken up and solved in parallel. It does not specify how the various threads of execution that are being run in parallel communicate. From a programmer's perspective, there are two major models of parallel communication, message passing and shared memory.

The message passing model, as its name suggests, requires that threads send messages to one another to communicate. In general, these messages must be paired in that the sender must explicitly send a message and the receiver must explicitly receive that message.

In the shared memory model, all threads share some amount of memory, and communication occurs through this shared memory. This naturally simplifies communication as the programmer no longer needs to explicitly send data back and forth between threads. The programmer is also no longer responsible for ensuring threads send messages in the correct order to avoid deadlocks, nor to check for errors in communication.

However, as memory is a shared resource, the programmer must be aware of the consistency that the model allows. In the most simple case, there is a strict ordering on all memory accesses across all threads, which is easy for the programmer to understand, but is usually quite expensive for the implementation to enforce. There are various ways of relaxing the semantics to improve the performance of the program by permitting threads to see operations occur in different orders, eliminating unnecessary synchronization.

2.1.3 Partitioned Global Address Space

While the shared memory programming model offers the programmer many advantages, it is often difficult to implement efficiently on today's large distributed systems. One major difficulty comes from the fact that it is usually orders of magnitude more expensive to access shared memory that is off-node than it is to

access on-node memory. If the programmer has no way of differentiating between on-node and off-node memory, it becomes difficult to write programs that run efficiently on these modern machines.

Partitioned Global Address Space, PGAS, languages try to address this problem by introducing the concept of affinity [10]. The shared memory space is partitioned up among the threads such that every object in shared memory has affinity to one and only one thread. This allows programmers to write code that takes advantage of the location of the object.

2.1.4 Unified Parallel C

Unified Parallel C, henceforth UPC, is a parallel extension to the C programming language [3]. It uses the SPMD programming model, where a fixed number of UPC threads execute the same UPC program. Each UPC thread has its own local stack and local heap, but there is also a global memory space that all threads have access to. As UPC is a PGAS language, this global memory is partitioned amongst all the UPC threads.

It is important to note that accesses to shared memory in UPC applications do not require any special libraries or syntax. Once a variable is declared as shared, it can be referenced just as any local variable, at least from the programmer's point of view. In many implementations, including MuPC, these accesses are directly translated into runtime library calls that perform the read or write as necessary.

For example, the program in Figure 2.1 prints hello from each UPC thread, records how many characters each thread printed, and exits with *EXIT_FAILURE* if any thread had an error (printf() returns a negative value on errors).

One important performance feature of UPC is the ability to specify that memory accesses use relaxed memory consistency. The default, strict, memory consistency requires that all shared memory accesses be ordered, and that all threads see the same global order of memory accesses. In particular, if two threads write to the same variable at the *same* time, all threads will see the same written value after both threads have occurred. Consider the code segment in Figure 2.2.
For threads other than thread 0, there are only three possible outputs at the end of the code: $a = 0, b = 0$ or $a = 1, b = 0$ or $a = 1, b = 2$.

By contrast, relaxed memory consistency provides no such guarantee. Different threads may see operations occur in different orders. Consider the same code segment using relaxed variables instead of strict ones shown in Figure 2.3.
For threads other than thread 0, there are now four possible outputs at the end of the code: $a = 0, b = 0$ or $a = 1, b = 0$ or $a = 1, b = 2$ or $a = 0, b = 2$. The additional value, $a = 0, b = 2$ is permitted because the relaxed semantics allow threads other than thread 0 to see the assignment to sb occur before the assignment to sa, while the strict semantics require they occur in program order.

Since implementation is allowed to reorder relaxed operations, it is also permit-

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

/* Each UPC thread has 1 element of this array. */
shared [1] int printed[THREADS];

int main()
{
  int i, exit_value = EXIT_SUCCESS;

  /* Initialize local part of printed array to 0. */
  printed[MYTHREAD] = 0;

  /* Wait for everyone to finish initialization. */
  upc_barrier( 1 );

  /* Record the number of characters printed. */
  printed[MYTHREAD] = printf("Hello from UPC thread %d of %d.\n",
                             MYTHREAD, THREADS);

  /* Wait for everyone to finish printing. */
  upc_barrier( 2 );

  /* Verify all the threads printed something. */
  for(i=0;i<THREADS;++i)
  {
    if(printed[i]<0) exit_value = EXIT_FAILURE;
  }

  exit(exit_value);
}
```

Figure 2.1. Hello World in UPC

ted to cache the values without having to worry about keeping the caches coherent until a strict access or collective occurs. Despite this capability, relatively few UPC implementations cache remote accesses, and those that do use relatively simple caches. At the time of this writing, only the MuPC reference implementation from Michigan Technological University [11] and the commercial implementation from Hewlett Packard [12] are known to implement caching of remote references.

2.1.5 MuPC

MuPC is a reference UPC implementation that is built on top of MPI and POSIX threads. It currently supports Intel x86 and x86-64 based clusters running Linux, as well as alpha clusters running Tru64. Each UPC thread is implemented as a single OS process using two pthreads, one for managing communication, the other to run the UPC program's computation. The compile script first translates the UPC code into C code with calls into the MuPC runtime library to handle communication and synchronization. This is then compiled with the system MPI compiler and the resulting binary can be run as if it were an MPI program.

```
strict shared int sa=0, sb=0;
int la=0, lb=0;

if(MYTHREAD==0)
{
  sa=1;
  sb=2;
}
else
{
  lb=sb;
  la=sa;
  printf("Thread %d: a = %d, b = %d\n",la,lb);
}
```

Figure 2.2. Example of Strict Semantics in UPC

```
relaxed shared int sa=0, sb=0;
int a=0, b=0;

if(MYTHREAD==0)
{
  sa=1;
  sb=2;
}
else
{
  b=sb;
  a=sa;
  printf("Thread %d: a = %d, b = %d\n",a,b);
}
```

Figure 2.3. Example of Relaxed Semantics in UPC

MuPC implements a cache for remote references in the communication thread. The cache is divided into THREADS−1 sections, one for each non-local UPC thread. The size of the cache is determined by the user via a configuration file. The user can also change the size of a cache line. The defaults setup a 2mb cache with 64-byte cache lines.

2.2 Reuse Distance Analysis

Reuse distance is defined as the number of distinct memory locations that are accessed between two accesses to a given memory address. This information is generally used to determine, predict, or optimize cache behavior.

Forward reuse distance answers the question "How many distinct memory locations will be accessed before the next time this address is accessed?". It scans forward in an execution, counting the number of memory locations accessed until the given address is found. This information can be useful for determining whether or not caching should be performed for a memory reference, among other things.

Backward reuse distance answers the question "How many distinct memory locations were accessed since the last time this address was accessed?". It scans backward in an execution, counting the number of memory locations accessed until the given address is found. This information can be useful for determining whether or not a memory reference should be prefetched, among other things.

```
A[1] = 1;
A[2] = 2;
A[1] = 3;
A[2] = 4;
A[3] = 5;
A[1] = 6;
A[2] = 7;
```

Figure 2.4. Reuse Distance Example

For example, consider the second reference to $A[2]$ in the short code segment in Figure 2.4. Because only $A[1]$ was accessed since the last access to $A[2]$, the backward reuse distance is 1. The forward reuse distance is 2, because both $A[1]$ and $A[3]$ are accessed before $A[2]$ is accessed again.

There is also a distinction between temporal reuse and spatial reuse. Temporal reuse refers to reuse of a single location in memory, as in the example above. Spatial reuse considers larger sections of memory, as the cache in many systems pulls in more than one element at a time. Assuming the cache lines in the system can hold two array elements and that $A[1]$ is aligned to the start of a cache line, the backwards spatial reuse distance of the second access to $A[2]$ is 0, because their were no intervening accesses to different cache lines since the last access. The forward reuse distance is 1 however, because $A[3]$ does not share a cache line with $A[1]$ and $A[2]$.

2.2.1 Instruction Based Reuse Distance

Analyses generally create histograms of either the forward or backward reuse distances encountered in a program trace. The histograms are generally associated either with a particular memory address or with a particular memory operation. When associated with a memory operation, the data are referred to as *instruction based reuse distances* [13, 14].

Studying the reuse distances associated with operations can provide many useful insights into an application's behavior. For example, the maximum reuse distance seen by any operation can tell how large a cache will be beneficial to the application. *Critical instructions*, operations that suffer from a disproportionately large number of the cache misses in a program, can be identified as well [14].

It is generally expensive to record the reuse distances over an entire program execution exactly, as there can be trillions of memory operations encountered.

However, it has been shown that highly accurate approximations of the reuse distance can be stored efficiently using a splay tree to record when memory locations are encountered [15]. This information can then be used to create histograms describing the reuse distances for a given application.

2.2.2 Predicting Reuse Distance Patterns

input: the set of memory-distance bins B
output: the set of locality patterns P

```
for each memory reference r {
  P_r = ∅; down = false; p = null;
  for (i = 0; i < numBins; i++)
    if (B_r^i.size > 0)
      if (p == null ‖(B_r^i.min − p.max > p.max − B_r^i.min)‖
          (down&&B_r^{i−1}.freq < B_r^i.freq)) {
        p = new pattern; p.mean = B_r^i.mean;
        p.min = B_r^i.min; p.max = B_r^i.max;
        p.freq = B_r^i.freq; p.maxf = B_r^i.freq;
        P_r = P_r ∪ p; down = false;
      }
      else {
        p.max = B_r^i.max; p.freq += B_r^i.freq;
        if (B_r^i.freq > p.maxf) {
          p.mean = B_r^i.mean; p.maxf = B_r^i.maxf;
        }
        if (!down && B_r^{i−1}.freq > B_r^i.freq)
          down = true;
      }
    else
      p = null;
}
```

Figure 2.5. Pattern-formation Algorithm

Using profiling data, it has been shown that the memory behavior of a program can be predicted by using curve fitting to model the reuse distance as a function of the data size of an application [14].

First, patterns are identified for each memory operation in the instrumented training runs. Histograms storing the reuse data for memory operations are used to identify these patterns. For each bin in the histogram, a *minimum* distance, *maximum* distance, *mean* distance, and *frequency* are recorded. Then, adjacent bins are merged using the algorithm in Figure 2.5. The *locality patterns* for an operation are defined as the sets of merged bins. Finally, the prediction algorithm uses curve fitting with each of a memory operation's patterns in two training runs to predict the corresponding pattern in the predicted run [14].

2.2.3 Prediction Accuracy Model

There are two important measures of the reuse distance predictions. First is the *coverage*, which is defined as the percentage of operations in the reference run that can be predicted. An operation can be predicted if it occurs in both training runs, and all of its patterns are *regular*. A pattern is *regular* if it occurs in both training runs and the reuse distance does not decrease as the problem size grows.

The *accuracy* then is the percentage of *covered* operations that are predicted *correctly*. An operation is predicted correctly if the predicted patterns exactly match the observed patterns, or they overlap by at least 90%. The overlap for two patterns A and B is defined as

$$\frac{A.max - \max(A.min, B.min)}{\max(B.max - B.min, A.max - A.min}$$

The overlap factor of 90% was used in this work because this factor worked well in prior work with sequential applications [14].

CHAPTER 3

Predicting Remote Reuse Distance

3.1 Instrumentation

In order to get the raw cache reuse data for the predictions, it was important to have a working base compiler from which to add instrumentation to generate the raw cache reuse data. The MuPC compiler and runtime was used for the data collection, with a number of modifications to support runtime remote reuse data collection.

Initially, it was necessary to update MuPC to add support for x86-64 systems to ensure MuPC continues to function in the future, as well as to support our new cluster. Because the vender-provided MPI libraries were 64-bit only, it was not possible to simply use the 32-bit support in the OS. Therefore, the MuPC compiler was modified to generate correct code for the new platform. The bulk of this work was merely increasing the size of primitive types and enabling 64-bit macros and typedefs in the EDG front-end.

The other changes were all to support recording cache reuse in UPC programs. First, generic instrumentation was added to many of the runtime functions. These allow a programmer to register functions that get called whenever a particular runtime function is called. This enables a programmer to inspect the program's runtime behavior. To test this instrumentation, a simple function was written to create a log of all remote memory accesses, recording the operation (put or get), the remote address, the size of the access, and the location in the program source that initiated the access.

Once this functionality was working, existing instrumentation for the Atom simulator [16] was modified for use with MuPC. This code uses splay trees to store cache reuse data per instruction. Since it was originally designed to work with cache reuse in hardware, associating the reuse data with an instruction works fine. However, for this project, there is no simple *instruction* to associate the reuse data with, as remote accesses are complicated operations. It was finally decided that the return address (back into the application code) would work as a substitute, since it would produce the desired mapping back to the application source.

Additionally, the Atom instrumentation had to be modified to deal with differing sizes of addresses. In particular, shared memory addresses are a struct in MuPC, containing a 64-bit address, a thread, and a phase. The phase was not important to this research, but both the thread and the address were. To properly store these values, the instrumentation was modified to store addresses as 64-bit values instead of 32-bit values, and the thread was stored in the upper six bits of the address, since they were unused due to memory alignment.

Unfortunately, the implementation does require modifications to the source program. The modifications are quite small, and can easily be disabled with the preprocessor. The macros and global variables shown in Figure 3.1 were defined in the *upc.h* header.

```
/*
 *  Added to support tracing remote accesses.
 */
extern char *__mupc_trace_func;

char *__mupc_get_func_name();

#define TRACE_FUNC \
        char *__mupc_trace_func_prev; \
        __mupc_trace_func_prev=__mupc_trace_func; \
        __mupc_trace_func=__mupc_get_func_name()

#define TRACE_FUNC_RET __mupc_trace_func=__mupc_trace_func_prev

/** Macros for tracing.  Must be used exactly once per program! **/
#define MUPC_TRACE_NONE \
        void (*__mupc_trace_init)() = NULL;

#define MUPC_TRACE_FILE \
        void __mupc_trace_init_file(); \
        void (*__mupc_trace_init)() = __mupc_trace_init_file;

#define MUPC_TRACE_RD \
        void __mupc_trace_init_rd(); \
        void (*__mupc_trace_init)() = __mupc_trace_init_rd;
```

Figure 3.1. Added MuPC Macros

These macros setup information in global variables that is used by special functions that wrap calls into the MuPC runtime. The calls save the return address of the call and the name of the function that it was called from so they can be recorded by the instrumentation.

The macros TRACE_FUNC and TRACE_FUNC_RET should be used at the beginning and end of all functions with remote accesses. While these macros are not strictly necessary, they enable the instrumentation to track the function name that an operation originated from without having to figure it out from the return address.

The MUPC_TRACE_* macros initialize the tracing code. The MUPC_TRACE_RD macro configures the tracing to store per instruction shared memory reuse data. It must be included exactly once in the program's source.

During an instrumented run, all remote accesses are logged, and each place there is a call into the runtime gets associated with a reuse distance histogram. Barriers and strict accesses cause the last use data to be dropped to force all later references to act as if no addresses had yet been seen. When the program completes, these histograms are written out to disk, one file per thread.

3.2 Test Kernels

Since there are no standard benchmark applications for UPC, a small number of kernels were written or modified from existing applications to model program behavior. These are described below.

3.2.1 Matrix Multiplication

Matrix multiplication is a well studied problem in parallel computation. Each index (i, j) in the resulting matrix is defined as the sum of the products of the elements of row i from the first matrix with the corresponding elements of column j from the second. A simple UPC implementation to solve $C = A * B$ where A, B, and C are $N \times N$ matrices is shown in Figure 3.2.

```
for(int i=0;i<N;++i) {
    upc_forall(int j=0;j<N;++j;&C[i][j]) {
        C[i][j] = 0;
        for(int k=0;k<N;++k) {
            C[i][j] += A[i][k] * B[k][j];
        }
    }
}
```

Figure 3.2. Matrix Multiplication in UPC

The Matrix Multiplication kernel simply multiplies two large (square) arrays together. The nodes are arranged in a 2-d grid, and each node has an NxN block of the array. The multiplication is performed using a naive implementation with a triple-nested loop, where each thread calculates its portion of the final matrix, working a block at a time. The problem size in this kernel is the local size of the three matrices. In testing, this kernel was run with 4, 9, and 16 threads with 4 elements per thread up to 262144 elements per thread, in increasing powers of 4. The complete source for the kernel used can be found in Appendix A.1.

3.2.2 Jacobi

The Jacobi method is an iterative method that finds an approximate solution to a series of linear equations. While it does not find exact solutions, it can give a solution that is within a desired delta of the exact solution for most problems. However, its most important feature is its numerical stability, which is critical when working with inexact values. Since the floating point format most commonly used to represent real numbers is inexact, this stability makes the Jacobi method quite useful in a variety of applications.

The Jacobi kernel simulates a Jacobi iterative solver for a large array distributed in a block cyclic fashion. The kernel only simulates the remote access pattern for a Jacobi solver, it does not actually attempt to solve the generated

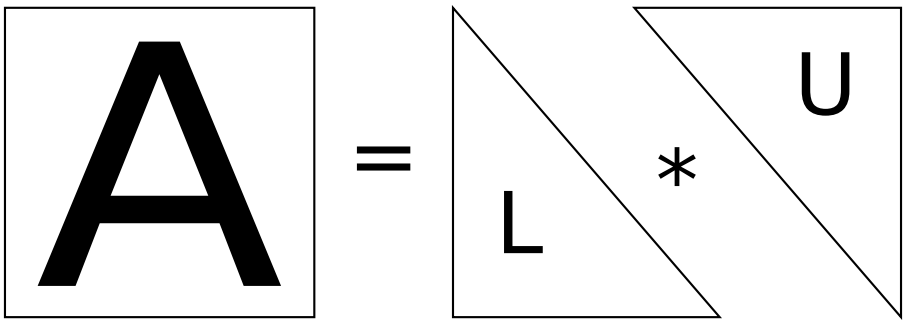

Figure 3.3. LU Decomposition of a Matrix

array. Like the Matrix Multiplication kernel, every thread performs essentially the same task. The problem size in this kernel is the number of iterations the solver runs for. In testing, this kernel was run with 2 through 24 threads, with iteration counts as a power of 2 up to 8192. The complete source for the kernel used can be found in Appendix A.2.

3.2.3 LU Decomposition

LU decomposition is another important operation for many scientific applications. It decomposes a matrix into the product of a lower triangular matrix with an upper triangular matrix, as shown in Figure 3.3. This can be used to find both the determinant and inverse of a matrix, as well as to solve a system of linear equations.

The LU kernel, which comes from the test suite from Berkeley's UPC compiler [4] based on a program from Stanford University [17], performs an LU-decomposition on a large array that is distributed across nodes in a block cyclic fashion. Square blocks of $B \times B$ ($B = 8, 16, 32$ were tested) are distributed to each thread until the entire array has been allocated. The problem size is the total size of the array. In testing, this kernel was run with 2 through 36 threads, with problem sizes from 1024 elements to 16777216 elements. The complete source for the kernel used can be found in Appendix A.3.

3.2.4 Stencil

Stencil problems are problems where elements of an array are iteratively updated based on the pattern of its neighbors. A common example of this is John Conway's Game of Life, where the life or death of a cell at a given time step is determined by the life or death of neighboring cells in the previous time step. The stencil describes which neighboring cells are used to update a given cell. This is often used in engineering when modeling the flow of heat or air.

The stencil kernels are a family of kernels that apply a 2-d or 3-d stencil to an array. These kernels were added as an example of a class of problems where threads displayed differing behavior based on the logical layout of threads. The problem size for these is the total size of the array the stencil operates over. These kernels were developed later than the others and did not have as many test runs as the other kernels due to time and hardware constraints. Therefore, exhaustive results are not available for these kernels, results are available only for the four and eight point 2d stencil kernels.

While initially instrumenting the stencil code, it was observed that in some cases, there were additional unexpected remote memory accesses that did not match up to the theoretical communication behavior of the programs.

```
void stencil(int i, int j)
{
    A(i,j) += A(i-1,j);
    A(i,j) += A(i+1,j);
    A(i,j) += A(i,j-1);
    A(i,j) += A(i,j+1);
    A(i,j) /= 5.0;
}
```

Figure 3.4. Failing Stencil Code

Assuming that $A(i, j)$ is local to the calling thread, one would expect that the *stencil* function in Figure 3.4 generates at most four operations (calls to the runtime) that could be remote accesses. Without that assumption, at most fourteen operations could be remote accesses. The code generated actually has at most twenty-three operations that could be remote accesses. This was causing the thread-scaling prediction to fail, as the pattern of which of these operations is used varies with thread count.

The culprit was determined to be the '+=' operator and its interaction with a shared variable. Changing the stencil code sample from Figure 3.4 to the that shown in Figure 3.5 eliminates the superfluous operations.

This fixed the problem because the UPC to C translator uses nested conditional operators to check for local accesses when working with shared variables. This nesting caused multiple accesses to be generated for a single source access when there are multiple accesses to shared variables in a statement. Since this is a limitation of the MuPC compiler and common practice is to use local temporaries to avoid multiple remote accesses, the stencil code was updated to use the latter form. The complete source for the kernel used can be found in Appendix A.4.

```
void stencil(int i, int j)
{
        double t;

        t = A(i,j);

        t += A(i-1,j);
        t += A(i+1,j);
        t += A(i,j-1);
        t += A(i,j+1);
        t /= 5.0;

        A(i,j) = t;
}
```

Figure 3.5. Corrected Stencil Code

3.3 Thread Partitioning

Since a prediction is being performed for each UPC thread in a run, the choice of which thread's data are used to train becomes important. If the number of threads is kept constant, the obvious choice is to use the same thread's data for the training. However, that doesn't work when the number of threads increases. Therefore, it was necessary to come up with a way of selecting the threads used in the predictions.

In searching for a suitable algorithm for partitioning threads for training, it was noted that all of the kernels tested worked with large square arrays, and the communication pattern was based on the distribution of these arrays. Because the communication pattern is based on the geometric layout of the data, an algorithm was used that matches the training data with a pattern describing this geometric layout, and then chooses threads for prediction based on it. As a special exception, thread 0 is assumed to be used for various extraneous tasks, such as initialization, and is therefore always predicted with thread 0 from each of the training runs.

The algorithm is split into three parts. The threads in the two training runs are partitioned into groups based on their communication behavior, as shown in Figure 3.6. It is assumed that data for each thread in each training run has associated with it the pattern data, represented as t.patterns, the instructions encountered (patterns are associated with an instruction) represented as t.inst. Each thread's patterns are tested against those in the existing groups. The thread is included in the group if all the patterns are present in both, and there is at most 5% difference between the min, max, freq, and mean values for the thread and the average of the values for all the threads in the group. Each group tracks the number of members and the running arithmetic average min, max, freq, and mean of each pattern. Once the patterns are generated, they are associated with the run as T.groups.

Then these groups are matched against a function describing the expected

input: the set of training runs containing pattern data for each thread in the run
output: the set of training runs is updated with the set of groups G

```
for each training run T
  T.groups = ∅
  for each thread t ∈ T
    t.group = NULL
    for each group g ∈ T.groups
      if patterns_match(g,t)
        t.group = g
        break
    if t.group == NULL
      t.group = new group
      t.group.vals = NULL
      t.group.inst = t.inst
      t.group.patterns = t.patterns
      t.group.nthr = 1
      T.groups = T.groups ∪t.group
    else
      for each pattern p_g ∈ t.group.patterns
        let p_t be the corresponding pattern in t.patterns
        p_g.min = (p_g.min*t.group.nthr+p_t.min)/(t.group.nthr+1)
        p_g.max = (p_g.max*t.group.nthr+p_t.max)/(t.group.nthr+1)
        p_g.freq = (p_g.freq*t.group.nthr+p_t.freq)/(t.group.nthr+1)
        p_g.mean = (p_g.mean*t.group.nthr+p_t.mean)/(t.group.nthr+1)
      t.group.nthr = t.group.nthr+1
```

Figure 3.6. Partitioning Algorithm

input: a group g and a thread t
output: *true* if the patterns of t match those of g, *false* otherwise

```
if g.inst≠ t.inst return false
for each pattern p_g ∈ g.patterns
  let p_t be the corresponding pattern in t.patterns
  if no such p_t return false
  if (p_g.min−p_t.min)/max(p_g.min,p_t.min)> 0.05 return false
  if (p_g.max−p_t.max)/max(p_g.max,p_t.max)> 0.05 return false
  if (p_g.freq−p_t.freq)/max(p_g.freq,p_t.freq)> 0.05 return false
  if (p_g.mean−p_t.mean)/max(p_g.mean,p_t.mean)> 0.05 return false
return true
```

Figure 3.7. *patterns_match()* Function

partitioning of threads, as shown in Figure 3.8. The pattern function is used to generate the set of values associated with threads in the group. The training run matches the pattern function if the set of values associated with each group is disjoint from ever other group. Additionally, a map mapping values with one of the threads associated with it (the lowest if the threads are tested in ascending order) is kept for each training run, for later use when picking which threads to use in the prediction.

input: the set T of training runs with the associated group assignments
output: *true* if training runs match the pattern, *false* otherwise

```
for each training run T_i
  for each thread t ∈ T_i
    t.group.vals = t.group.vals∪f(t, T.numthreads)
    if !M_i.containsKey(f(t, T.numthreads))
      M_i.insert(f(t, T.numthreads),t)
  for each unordered pair of groups g_1, g_2 ∈ T.groups
    if (g_1.vals∩g_2.vals) ≠ ∅ return false
return true
```

Figure 3.8. Matching Algorithm

Unfortunately, the pattern itself is not automatically detected, but was chosen because the kernels tested all used a square thread layout. The pattern function used is shown in Figure 3.9. Note that the pattern partitions a square into regions based on geometric properties. Because the algorithm merges regions based on observed behaviors, the pattern actually defines a large number of subpatterns, which are automatically matched by the algorithm. The ability to automatically generate a pattern function for a given problem would increase the generality of this analysis greatly, an important avenue for future work. However, it is possible to create general patterns that match a wide variety of problems by utilizing the subsetting inherent to this algorithm.

Finally, if both training runs match the pattern, threads are predicted with threads from the training runs who share the same group as determined by the matched function. The algorithm selecting the pairs is shown in Figure 3.10. It uses the maps generated while matching the pattern to choose threads in the training runs with the same value returned by the pattern function.

As an example, consider using equation 3.1 as a pattern to match against the LU kernel run with 16 threads in the test dataset, 25 threads in the train dataset, and 36 threads in the reference dataset with a fixed per-thread data size. In this case, the threads with data on the diagonal of the array have to do extra work. Equation 3.1 returns 0 for threads on the diagonal and 1 for threads not on the diagonal, assuming the threads are laid out in a square grid. This should perfectly match the thread layout of the LU kernel when run with a number of threads that

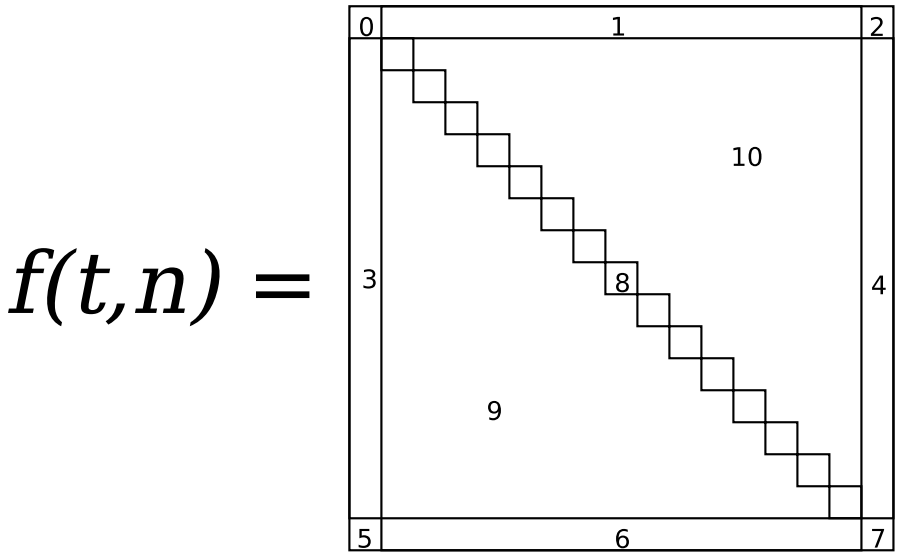

Figure 3.9. Pattern Function

input: the set of threads T_{pred} in the prediction run
input: for each training run, a map M_i mapping a return value from the pattern $f(t, T)$ to a thread in the training run that generates that return value
output: for each thread in the prediction run, a pair of threads from the training runs to use

for each thread $t_{\text{pred}} \in T_{\text{pred}}$
$t.\text{preds} = (M_0.\text{get}(f(t_{\text{pred}}, T_{\text{pred}}.\text{numthreads})), M_1.\text{get}(f(t_{\text{pred}}, T_{\text{pred}}.\text{numthreads})))$

Figure 3.10. Training Thread Selection Algorithm

is a perfect square, such as in this example. For clarity, the matching of reuse patterns is simplified such that threads with data on the diagonal perfectly match only other threads with data on the diagonal, and likewise for threads without data on the diagonal.

$$f(t,n) = \begin{cases} 0 & \text{if } \lfloor \frac{t}{\sqrt{n}} \rfloor = (t \bmod \sqrt{n}) \\ 1 & \text{o.w.} \end{cases} \tag{3.1}$$

First the threads in the test and train datasets are sorted into groups. In both cases there are two groups, those threads that have data on the diagonal – and therefore have extra work, and those that don't. For the test dataset, the groups are $g_0 = \{t_5, t_{10}, t_{15}\}$ and $g_1 = \{t_1, t_2, t_3, t_4, t_6, t_7, t_8, t_9, t_{11}, t_{12}, t_{13}, t_{14}\}$. For the train dataset, the groups are $g_0 = \{t_6, t_{12}, t_{18}, t_{24}\}$ and $g_1 = \{t_1, t_2, t_3, t_4, t_5, t_7, t_8, t_9, t_{10}, t_{11}, t_{12}, t_{13}, t_{14}, t_{15}, t_{16}, t_{17}, t_{19}, t_{20}, t_{21}, t_{22}, t_{23}\}$. As noted earlier, t_0 is excluded as a special case.

Next, for each group, the set *vals* of results of $f(t_i, n)$ where n is the number of threads is computed for each t_i in the group. This gives g_0.vals $= \{0\}$ and g_1.vals $= \{1\}$ for both the test and train datasets. Because g_0.vals $\bigcap g_1$.vals is empty, the pattern is matched in both datasets.

Ref	Test	Train	Ref	Test	Train
0	0	0	18	1	1
1	1	1	19	1	1
2	1	1	20	1	1
3	1	1	21	5	6
4	1	1	22	1	1
5	1	1	23	1	1
6	1	1	24	1	1
7	5	6	25	1	1
8	1	1	26	1	1
9	1	1	27	1	1
10	1	1	28	5	6
11	1	1	29	1	1
12	1	1	30	1	1
13	1	1	31	1	1
14	5	6	32	1	1
15	1	1	33	1	1
16	1	1	34	1	1
17	1	1	35	5	6

Table 3.1. Thread Grouping for 36-Thread Reference Dataset

Since the pattern was matched, it can be used to select the pairs of threads

used for prediction of the reference dataset. For each thread in the reference dataset, a pair of threads from the test and train datasets are chosen based on which set they are in. Since threads are grouped by behavior, it doesn't matter which thread in a set is used for the prediction. In the solution shown in table 3.1, the lowest thread in the set is used for the predictions.

CHAPTER 4

Prediction Results

To determine whether or not it would be possible to model the behavior of remote memory accesses, the four kernels were instrumented and run with a number of different data sizes and numbers of threads. All tests were run with the instrumented MuPC compiler on a 24-node dual dual core opteron cluster with an infiniband interconnect.

Due to hardware and software limitations, the testing was restricted to a maximum of 48 UPC threads. In practice, any more than 24 threads ran quite slowly, thus there are relatively few results with more than 24 threads. While it is recognized that these are relatively small systems in the world of high performance computing, it is believed that the results would hold as the problem and thread size increases since the problems encountered were due to either changes in data layout or using training data from runs that were too small.

4.1 Problem Size Scaling

As expected, the prediction accuracy was very high when holding the number of threads constant and just increasing the problem size. The prediction followed the same pattern as shown in earlier work, which makes sense as there is very little to distinguish size scaling with constant threads from size scaling with one thread as far as the prediction is concerned.

However, problems can arise when the growth of the problem size causes the distribution of shared data to change, which in turn causes the communication pattern between threads to change. This behavior is seen in the LU kernel, where prediction accuracy and coverage drop steeply in a few cases because the distribution of the array changed.

Table 4.1. Problem Size Scaling Accuracy Distribution

Kernel	Predictions	Minimum	Average	Maximum
Matrix Multiplication	1547	3.08%	96.98%	100.00%
Jacobi Solver	14234	99.87%	100.00%	100.00%
LU Decomposition, 8x8	7027	6.36%	94.20%	100.00%
LU Decomposition, 16x16	5809	8.54%	95.91%	100.00%
LU Decomposition, 32x32	5839	0.00%	96.29%	100.00%

Table 4.2. Problem Size Scaling Coverage Distribution

Kernel	Predictions	Minimum	Average	Maximum
Matrix Multiplication	1547	53.45%	99.45%	100.00%
Jacobi Solver	14234	38.80%	100.00%	100.00%
LU Decomposition, 8x8	7027	0.00%	76.82%	100.00%
LU Decomposition, 16x16	5809	0.00%	63.17%	100.00%
LU Decomposition, 32x32	5839	0.00%	52.97%	100.00%

Tables 4.1 and 4.2 show that both coverage and accuracy are quite high in most cases. In the matrix multiplication and Jacobi kernels, the low minimum accuracies are seen only when training with very small problem sizes. Of the 1612 predictions on the matrix multiplication kernel where the accuracy is less than 60%, 1610 of them occur when the smallest training size is 256 or less, 1332 when the smallest training size is 16 or less. Likewise, the low minimum coverage percentages are seen only when the training sizes are small enough that some operations disappear.

The LU decomposition kernel is a bit more problematic. Consider the results in Table 4.3. It charts the percentage of predictions that have greater than 80, 90, and 95% accuracy for each of the three blocking factors tested, first for all predictions, then only for those where the coverage was 100%.

Table 4.3. LU Problem Size Scaling Accuracy by Coverage

Kernel	Predictions	>80% Acc	>90% Acc	>95% Acc
8x8, All Predictions	7027	56.07%	50.39%	33.56%
16x16, All Predictions	5809	50.20%	49.63%	40.37%
32x32, All Predictions	5839	47.58%	45.83%	45.71%
8x8, 100% Coverage	3503	96.97%	85.64%	53.04%
16x16, 100% Coverage	2486	97.30%	96.46%	74.82%
32x32, 100% Coverage	1844	88.72%	85.36%	84.98%

The large difference between predictions with 100% coverage and others stems from the behavior of the kernel. The data distribution amongst the threads is determined by the problem size, thus changing the problem size changes the data distribution. The data distribution in turn determines which remote memory operations a thread encounters, which also changes when the the problem size changes. This results in low coverage. The altered data distribution also changes the behavior of a couple remote memory operations as the number of threads encountering

the operation changes. This has the effect of reducing the accuracy of the prediction for those operations.

It is clear that this model is restricted to using training data from runs that have similar communication patterns as the run being predicted. An interesting question for future work is whether or not a model of the data distribution can be used to model how the scaling will affect the applications communication pattern, and if that can in turn be used to enable high coverage and prediction accuracy when the data distribution does change.

4.2 Thread Scaling

Table 4.4. Thread Scaling Accuracy Distribution by Kernel

Kernel	Predictions	Minimum	Average	Maximum
Matrix Multiplication	61056	3.08%	97.75%	100.00%
Jacobi Solver	103600	94.02%	99.87%	100.00%
LU Decomposition, 8x8	862683	12.85%	94.71%	100.00%
LU Decomposition, 16x16	346905	19.57%	91.73%	99.98%
LU Decomposition, 32x32	94600	13.11%	82.49%	99.31%
2d Stencil	18000	100.00%	100.00%	100.00%

Table 4.5. Thread Scaling Coverage Distribution by Kernel

Kernel	Predictions	Minimum	Average	Maximum
Matrix Multiplication	61056	27.62%	99.29%	100.00%
Jacobi Solver	103600	55.66%	91.26%	100.00%
LU Decomposition, 8x8	862683	0.00%	41.73%	100.00%
LU Decomposition, 16x16	346905	0.00%	37.43%	100.00%
LU Decomposition, 32x32	94600	0.00%	20.96%	98.56%
2d Stencil	18000	0.00%	57.99%	100.00%

Tables 4.4 and 4.5 show the prediction results for varying the number of threads, and predicting using all possible pairs of training threads from the training data available. Both coverage and accuracy are quite high in most cases. In the matrix multiplication and Jacobi kernels, the low minimum accuracies are seen only when training with very small problem sizes, the same that occurs when scaling the problem size. These results are for exhaustively predicting every thread

in the reference set with every possible combination of threads in the two training sets. The high accuracy and coverage in the matrix multiplication and Jacobi kernels in these tables indicate that the prediction works well regardless of the threads chosen for training.

The prediction coverage and accuracy on the LU kernel is much more dependent on the choice of threads used for the training however. Consider the accuracy of the prediction for thread 9 of 25, when using threads from runs with 4 and 16 threads for the training. The prediction accuracy by training pairs is shown in Figure 4.1.

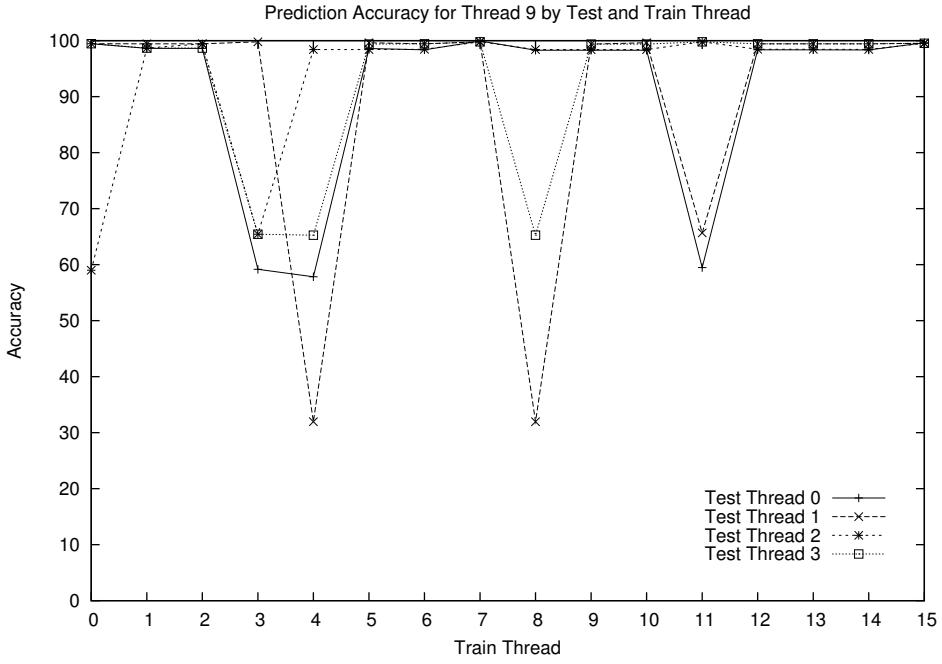

Figure 4.1. LU 32x32 4-16-25 Thread Scaling Example Thread

For most of the pairs, the prediction accuracy is quite good. However, there are a number of pairs that results in terrible accuracy. This is a result of the behavioral differences between threads in the LU kernel. Since certain threads (those that contain blocks on the diagonal) have to do additional communication, and thread 9 when run with 25 threads is not one of them, pairs where both threads are on the diagonal show dramatic decreases in accuracy.

However, these results show that if the threads can be partitioned in such a way that threads with similar behaviors are in similar groups, high accuracy predictions can be made. Thus consider what happens when partitioning the training threads as described in Section 3.3 with the pattern shown in Figure 3.9.

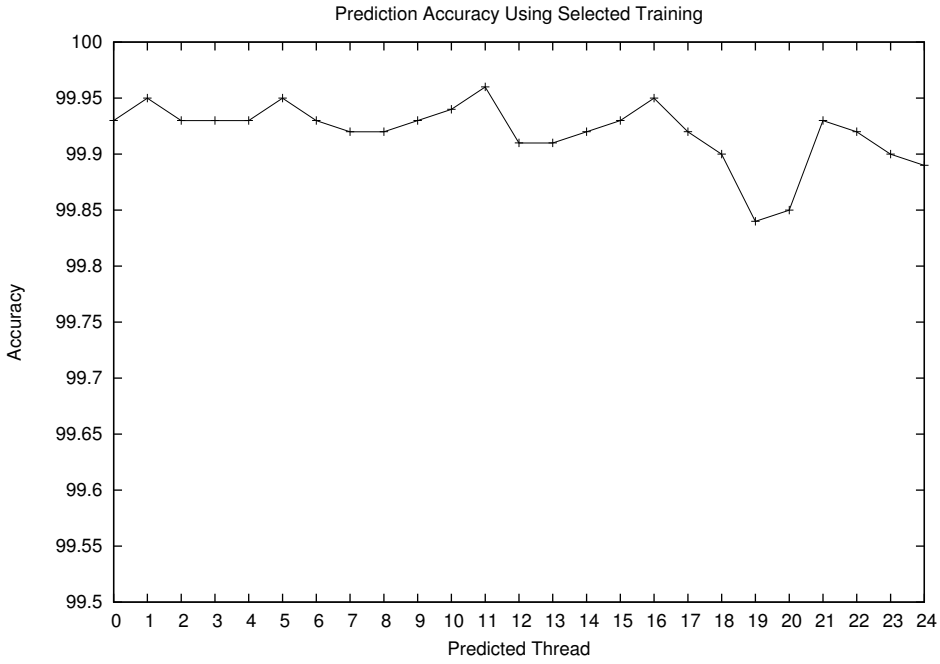

Figure 4.2. LU 32x32 4-16-25 Accuracy Using Thread Selection Algorithm

The equation for Figure 3.9 is a complicated function that simply partitions a square into groups based on geometric location. Each corner is in its own group, the edges (minus the corners) are each in their own groups. The diagonal, and the upper and lower triangles also have their own groups. Figure 4.2 shows the accuracy results when using the thread partitioning to select training pairs.

As expected, by predicting threads on the diagonal with threads also on the diagonal, it is possible to avoid the pits seen in Figure 4.1. However, it is desirable that the pattern that is matched not be specific to the LU kernel. Thus, the square 2-d stencils were used to verify that the pattern would also work for an application with a very different communication pattern than the LU kernel.

Like the LU kernel, the data distribution of the shared array determines the control flow through the program in the stencil kernels. Unlike the LU kernel, threads on the corners and edges exhibit differing behavior. This is due to lacking communication on one or more sides of the stencil. In turn, this causes low prediction coverage, if threads along the edges are used to predict for threads in the middle–thus the low minimum and average coverage values in Table 4.5. The accuracy of covered operations is not impacted, however, because the skipped operations have minimal impact on the reuse patterns. This is shown by the very high accuracy of predictions seen in Table 4.4.

Table 4.6. Thread Scaling Accuracy Distribution with Partitioning

Kernel	Minimum	Average	Maximum
Matrix Multiplication	92.38%	99.44%	100.00%
Jacobi Solver	94.69%	99.87%	100.00%
LU Decomposition, 8x8	51.07%	91.24%	99.86%
LU Decomposition, 16x16	56.26%	80.36%	99.33%
LU Decomposition, 32x32	48.95%	91.67%	99.31%
2d Stencil	100.00%	100.00%	100.00%

Table 4.7. Thread Scaling Coverage Distribution with Partitioning

Kernel	Minimum	Average	Maximum
Matrix Multiplication	100.00%	100.00%	100.00%
Jacobi Solver	100.00%	100.00%	100.00%
LU Decomposition, 8x8	97.45%	98.79%	99.90%
LU Decomposition, 16x16	97.56%	98 97%	100.00%
LU Decomposition, 32x32	96.34%	98.26%	98.56%
2d Stencil	100.00%	100.00%	100.00%

Using thread partitioning, the threads in the corners, on each edge, and in the middle are separately grouped. This provides 100% coverage for all threads, as threads in the training runs that skip operations are used to predict for threads that will also skip those instructions due to the data distribution.

Tables 4.6 and 4.7 show the results of using the partitioning algorithm presented to select training threads for all kernels. As expected, the coverage and accuracy both show marked improvement for all kernels. The only unexpected result is the decline in the average accuracy for the lu kernels. On closer inspection, this because the results in Table 4.4 are padded by the large number of combinations that match threads not on the diagonal. Additionally, because the number of predictions is so much smaller, the minimums have more weight.

CHAPTER 5

Conclusions

In summary, it has been shown that it is possible to predict the remote reuse distance for UPC applications with a high degree of accuracy and coverage, though there are a number of important limitations.

First, it is necessary to choose training data that match behavior of the desired prediction size to achieve high accuracy and coverage. Changes in the data distribution caused by increases in the problem size or number of threads can cause significant drops in both accuracy and coverage.

Choice of training threads is also critically important for prediction when scaling up the number of threads. The prediction results can vary from extremely poor to excellent merely by the choice of which threads were used for the prediction. It is therefore necessary to match threads' behaviors in the training data to patterns that predict which threads will perform similarly in the scaled-up runs.

5.1 Applications

One promising application of this research is automatically adjusting cache parameters such as size and thread affinity of cache lines. This includes things like bypassing the cache for operations that are likely to result in a cache miss, or disabling the cache entirely for applications that won't make much use of it. Consider the code segment in Figure 5.1. Assume *do_something* is a function that performs no communication.

```
void example_sub( shared float *p1, shared float *p2, int len )
{
  int i,j;
  upc_forall(i=0;i<len;++i;p1+i) {
    for(j=0;j<len;++j) {
      do_something(p1[i],p2[j],i,j);
    }
  }
}
```

Figure 5.1. Sample UPC Function

Since elements of *p2* are accessed in order, repeatedly for each element of *p1*, this particular segment of code would work well with a cache large enough to hold at least *len* elements. This work could be used to predict how large a cache is needed for the application to cache all the elements this function sees based on the reuse distance patterns seen.

Another option is warning the user about operations that show poor cache performance, perhaps as part of a larger performance analysis/profiling tool. Going back to the previous code, assume this time that the maximum size the cache can grow to in an implementation is *CACHE_LEN*. If the prediction indicates that *len* is likely to be larger than this, it might warn the user that the accesses to elements of *p2* are likely to result in cache misses because the cache is too small. Then the user could change the code, perhaps to something like the code in Figure 5.2 to take better advantage of the cache, or perhaps simply change the compiler options used to tell the compiler to do so automatically.

```
void example_sub( shared float *p1, shared float *p2, int len )
{
  int i,j,k;
  for(k=0;k<len/CACHE_LEN;++k) {
    upc_forall(i=0;i<len;++i;p1+i) {
      for(j=0;j<CACHE_LEN;++j) {
        do_something(p1[i],p2[k*CACHE_LEN+j],i,k*CACHE_LEN+j);
      }
    }
  }
  upc_forall(i=0;i<len;++i;p1+i) {
    for(j=0;j<len%CACHE_LEN;++j) {
      do_something(p1[i],p2[k*CACHE_LEN+j],i,k*CACHE_LEN+j);
    }
  }
}
```

Figure 5.2. Sample UPC Function Tuned for Cache Size

Inserting prefetches prior to operations that are likely to result in a cache miss, and likewise avoiding putting in unnecessary prefetches, would also be a good use of these predictions. Since network congestion can cause dramatic performance penalties on large clusters, avoiding unnecessary communication can be just as important as requesting data before it is actually needed.

5.2 Future Work

This research shows that it is possible to predict the remote reuse distance behavior of UPC applications. There are a number of weaknesses that should be addressed in future work. Foremost among these is the ability to model the data distribution of an application, and use that model to avoid problems such as are seen with the LU kernel where changing the data distribution between training runs causes poor accuracy. Since the data distribution is known at runtime, it should be possible to store it and use it to tune the prediction by modeling how the distribution will change with an increase in problem size.

Another weakness of this research is that the function used as a pattern during thread partitioning must be chosen manually. As the thread grouping is largely

based on the data distribution, it seems natural to expect that a model of how data distribution changes when the number of threads grows would also enable a better partitioning of threads for improved prediction accuracy. Another possibility is looking at the program in a more abstract fashion, choosing a pattern based on the type of problem being solved. A list of such abstract problem types, and the associated communication patterns, such as Berkeley's Dwarfs [18] could be used as a starting point.

Since UPC is meant to increase productivity on large systems, it will also be necessary to improve the scalability of this work. In particular, storing reuse patterns for every thread in the two training sets, and predicting for every thread generated a large amount of data even for the relatively small test applications used. If this were to be used in a production environment, there would need to be some way of compressing the data or skipping threads whose patterns are similar to another thread's. This could perhaps extend into exploring a *global view* of cache behavior, where the data is not kept on a per-thread basis, but rather taken over all the threads in an application.

Finally, this work only explored temporal reuse. Spatial reuse, where "nearby" data is pulled in along with requested data provides quite a bit of performance for many serial applications. It is likely that it would work similarly for many UPC applications. The same prediction scheme used for temporal reuse was shown to work well for spatial reuse in serial applications as well. However, spatial locality in UPC shared memory can be cross thread or on the same thread, depending on how the application steps through memory and the way the data is laid out.

LIST OF REFERENCES

[1] TOP500.Org. "ORNLs Jaguar Claws its Way to Number One, Leaving Reconfigured Roadrunner Behind in Newest TOP500 List of Fastest Supercomputer." Nov. 2009. [Online]. Available: http://top500.org/lists/2009/11/press-release

[2] R. Numwich and J. Reid, "Co-Array Fortran for parallel programming," Rutherford Appleton Laboratory, Tech. Rep. RAL-TR-1998-060, 1998.

[3] The UPC Consortium. "UPC language specification, v1.2." June 2005. [Online]. Available: http://www.gwu.edu/~upc/docs/upc_specs_1.2.pdf

[4] University of California Berkeley. "The Berkely UPC Compiler." 2002. [Online]. Available: http://upc.lbl.gov

[5] University of California Berkeley. "GASNet Communication System." 2002. [Online]. Available: http://gasnet.cs.berkeley.edu/

[6] W.-Y. Chen, C. Iancu, and K. Yelick, "Communication optimizations for fine-grained upc applications," in *PACT '05: Proceedings of the 14th International Conference on Parallel Architectures and Compilation Techniques*. Washington, DC, USA: IEEE Computer Society, 2005, pp. 267–278.

[7] M. Snir, "Shared memory programming on distributed memory systems," 2009, keynote address at PGAS 2009. [Online]. Available: http://www2.hpcl.gwu.edu/pgas09/tutorials/PGAS_Snir_Keynote.pdf

[8] M. J. Flynn, "Very high-speed computing systems," in *Proceedings of the IEEE*, vol. 54, no. 12. IEEE Computer Society, Dec. 1966, pp. 1901–1909.

[9] M. J. Flynn, "Some copmuter organizations and their effectiveness," *IEEE Transactions on Computers*, vol. C-21, no. 9, pp. 948–960, Sept. 1972.

[10] B. Carlson, T. El-Ghazawi, R. Numrich, and K. Yelick. "Programming in the Partitioned Global Address Space Model." 2003. [Online]. Available: http://crd.lbl.gov/UPC/images/b/b5/PGAS_Tutorial_sc2003.pdf

[11] J. Savant and S. Seidel, "MuPC: A Run Time System for Unified Parallel C," Michigan Technological University, Tech. Rep. CS-TR-02-03, Sept. 2002. [Online]. Available: http://upc.mtu.edu/papers/CS.TR.2.3.pdf

[12] Hewlett Packard. "HP Unified Parallel C."

[13] C. Fang, S. Carr, S. Onder, and Z. Wang, "Reuse-distance-based miss-rate prediction on a per instruction basis," in *Proceedings of the 2nd ACM Workshop on Memory System Performance*, June 2004, pp. 60–68.

[14] C. Fang, S. Carr, S. Onder, and Z. Wang, "Instruction based memory distance analysis and its application to optimization," in *In Proceedings of the 14 th International Conference on Parallel Architectures and Compilation*, 2005.

[15] C. Ding and Y. Zhong, "Predicting whole-program locality through reuse distance analysis," *SIGPLAN Not.*, vol. 38, no. 5, pp. 245–257, 2003.

[16] A. Srivastava and A. Eustace, "Atom: A system for building customized program analysis tools." ACM, 1994, pp. 196–205.

[17] Stanford University. "Parallel dense blocked LU factorization." 1994.

[18] University of California Berkeley. "The Landscape of Parallel Computing Research: A View From Berkeley." [Online]. Available: http://view.eecs.berkeley.edu/wiki/Main_Page"

APPENDIX

Test Kernel Sources

A.1 Matrix Multiplication

```
#include <stdlib.h>
#include <stdio.h>
#include <math.h>
#include <assert.h>
#include <upc.h>
#include <upc_relaxed.h>

#ifdef MUPC_TRACE_RD
MUPC_TRACE_RD
#endif

shared [] int * shared [1] * A;
shared [] int * shared [1] * B;
shared [] int * shared [1] * C;

#define arr_idx(arr,i,j) *((arr)[((i)/N)*n+((j)/N)]+((i)%N)*N+((j)%N))

/* Initialize will set this to sqrt(THREADS), the number
 * of columns and rows of THREADS.  Threads are laid out
 * as a n x n array.  Each thread locally has an NxN element.
 */
int n,N;

void initialize(char *argv)
{
  int i,j,k;

#ifdef TRACE_FUNC
  TRACE_FUNC;
#endif

  /* Seed the random number generator. */
  srand(MYTHREAD);

  /* Verify that the number of threads is a square. */
  n = (int) floor(sqrt((double)THREADS));
  assert((n*n)==THREADS);

  /* Read in the size of the local elment. */
  N = atoi(argv);
  if(MYTHREAD==0) printf("Using local blocks of size %dx%d\n",N,N);

  /* Allocate memory for the arrays. */
  A = (shared [] int * shared [1] *)
    upc_all_alloc(THREADS,sizeof(shared [] int *));
  B = (shared [] int * shared [1] *)
    upc_all_alloc(THREADS,sizeof(shared [] int *));
  C = (shared [] int * shared [1] *)
    upc_all_alloc(THREADS,sizeof(shared [] int *));
  assert((A!=NULL)&&(B!=NULL)&&(C!=NULL));

  A[MYTHREAD] = (shared [] int *)(((shared [1] int *)
    upc_all_alloc(THREADS,N*N*sizeof(int)))+MYTHREAD);
```

```
  B[MYTHREAD] = (shared [] int *)(((shared [1] int *)
    upc_all_alloc(THREADS,N*N*sizeof(int)))+MYTHREAD);
  C[MYTHREAD] = (shared [] int *)(((shared [1] int *)
    upc_all_alloc(THREADS,N*N*sizeof(int)))+MYTHREAD);

  /* Fill in arrays A and B. */
  upc_forall(i=0;i<THREADS;i++;i)
  {
    assert(A[i]!=NULL);
    assert(B[i]!=NULL);
    assert(C[i]!=NULL);

    for(j=0;j<N;j++)
    {
      for(k=0;k<N;k++)
      {
        *(A[i]+j*N+k) = i;
        *(B[i]+j*N+k) = i;
        *(C[i]+j*N+k) = 0;
      }
    }
  }

#ifdef TRACE_FUNC_RET
  TRACE_FUNC_RET;
#endif
}

void print_array( shared [] int * shared [1] *A )
{
  int i,j;

#ifdef TRACE_FUNC
  TRACE_FUNC;
#endif

  /* Only thread 0 prints.  Everyone else just returns. */
  if(MYTHREAD!=0) return;

  for(i=0;i<n*N;i++)
  {
    printf("\t%d",i);
  }
  putchar('\n');
  for(i=0;i<n*N;i++)
  {
    printf("%d",i);
    for(j=0;j<n*N;j++)
    {
      //printf("\t%d",*(A[(i/N)*n+(j/N)]+(i%N)*N+(j%N)));
      printf("\t%d",arr_idx(A,i,j));
    }
    putchar('\n');
  }
  putchar('\n');

#ifdef TRACE_FUNC_RET
  TRACE_FUNC_RET;
#endif
}

void calc_block(int idx)
```

```
{
  int i,j,k,rowt,colt;

#ifdef TRACE_FUNC
  TRACE_FUNC;
#endif

  rowt=(MYTHREAD/n)*n+idx;
  colt=(MYTHREAD%n)+(idx*n);

  for(i=0;i<N;i++)
  {
    for(j=0;j<N;j++)
    {
      for(k=0;k<N;k++)
      {
        *(C[MYTHREAD]+i*N+k) += (*(A[rowt]+i*N+j))*(*(B[colt]+j*N+k));
      }
    }
  }

#ifdef TRACE_FUNC_RET
  TRACE_FUNC_RET;
#endif
}

void mult_kernel()
{
  int i,j;

  upc_forall(i=0;i<THREADS;i++;i)
  {
    for(j=0;j<n;j++)
    {
      calc_block(j);
    }
  }
}

int main(int argc,char **argv)
{
  /* Must have 1 argument - size of N. */
  if(argc!=2) exit(EXIT_FAILURE);

  /* Initialize arrays. */
  initialize(argv[1]);

  upc_barrier(0);

  /* Print out A and B. */
  print_array(A);
  print_array(B);

  upc_barrier(1);

  /* Compute C=A*B. */
  mult_kernel();

  upc_barrier(2);

  /* Print out the result. */
  print_array(C);
```

```
  return 0;
}
```

A.2 Jacobi Solver

```
#include <stdlib.h>
#include <stdio.h>
#include <math.h>
#include <assert.h>
#include <upc.h>
#include <upc_relaxed.h>

#ifdef MUPC_TRACE_RD
MUPC_TRACE_RD
#endif

/* Default number of unknowns per thread. */
#ifndef N
#define N 100
#endif

#define SIZE N*THREADS

shared [N] double A[SIZE][SIZE];
shared [N] double X[2][SIZE];
shared [N] double B[SIZE];
shared [N] double D[SIZE];
double maxD;

//double epsilon;
long int MAX_ITER;

void initialize(char *argv)
{
  int i,j;

#ifdef TRACE_FUNC
        TRACE_FUNC;
#endif

  /* Seed the random number generator. */
  srandom(MYTHREAD);

  /* Read in the desired epsilon. */
  //epsilon = atof(argv);
  MAX_ITER = atoi(argv);
  //if(MYTHREAD==0) printf("Using epsilon = %g\n",epsilon);
  if(MYTHREAD==0) printf("Using MAX_ITER = %ld\n",MAX_ITER);

  /* Fill in A and B. Initialize X[i] to B[i]. */
  upc_forall(i=0;i<SIZE;i++;&B[i])
  {
    X[0][i]=X[1][i]=B[i]=
         ((double) THREADS)*(((double) random())/((double) RAND_MAX));

    for(j=0;j<SIZE;j++)
    {
      A[j][i]=((double) random())/((double) RAND_MAX);
      if(j==i) A[j][i]+=(double) SIZE;
    }
  }

#ifdef TRACE_FUNC_RET
  TRACE_FUNC_RET;
#endif
}
```

```
unsigned int jacobi_kernel()
{
  unsigned int iter;
  int i,j;
  double sum;

#ifdef TRACE_FUNC
        TRACE_FUNC;
#endif

  for(iter=0;iter<MAX_ITER;iter++)
  {
    /* Update X[] */
    upc_forall(i=0;i<SIZE;i++;&B[i])
    {
      sum=0.0;
      for(j=0;j<i;j++)        sum+=A[i][j]*X[iter%2][j];
      for(j=i+1;j<SIZE;j++) sum+=A[i][j]*X[iter%2][j];
      X[(iter+1)%2][i]=(B[i]-sum)/A[i][i];
    }

    upc_barrier;

    /* Compute maximum deltas on each thread. */
    upc_forall(i=0;i<SIZE;i++;&D[i])
    {
      sum=0.0;
      for(j=0;j<SIZE;j++) sum+=(A[i][j]*X[(iter+1)%2][j]);
      D[i]=fabs(sum-B[i]);
    }

    upc_barrier;

    /* Check for termination. */
    for(maxD=i=0;i<SIZE;i++) maxD=(D[i]>maxD)?D[i]:maxD;
    if(MYTHREAD==0) fprintf(stderr,"maxD: %8g\n",maxD);
    //if(maxD<epsilon) return iter;
  }

#ifdef TRACE_FUNC_RET
  TRACE_FUNC_RET;
#endif

  return iter;
}

void print_A()
{
  int i,j;

#ifdef TRACE_FUNC
        TRACE_FUNC;
#endif

  if(MYTHREAD!=0) return;

  puts("A[][]:");
  for(i=0;i<SIZE;i++)
  {
    for(j=0;j<SIZE;j++)
    {
```

```
      printf("\t%8g",A[i][j]);
    }
    putchar('\n');
  }
  putchar('\n');

#ifdef TRACE_FUNC_RET
  TRACE_FUNC_RET;
#endif
}

void print_B()
{
  int i;

#ifdef TRACE_FUNC
       TRACE_FUNC;
#endif

  if(MYTHREAD!=0) return;

  puts("B[]:");
  for(i=0;i<SIZE;i++)
  {
    printf("\t%8g\n",B[i]);
  }
  putchar('\n');

#ifdef TRACE_FUNC_RET
  TRACE_FUNC_RET;
#endif
}

void print_X(int iter)
{
  int i;

#ifdef TRACE_FUNC
       TRACE_FUNC;
#endif

  if(MYTHREAD!=0) return;

  puts("X[]:");
  for(i=0;i<SIZE;i++)
  {
    printf("\t%8g\n",X[iter%2][i]);
  }
  putchar('\n');

#ifdef TRACE_FUNC_RET
  TRACE_FUNC_RET;
#endif
}

int main(int argc,char **argv)
{
  int iter;

  /* Must have 1 argument - desired precision. */
  if(argc!=2) exit(EXIT_FAILURE);
```

```
  /* Initialize arrays. */
  initialize(argv[1]);

  upc_barrier(1);

  /* Print out the randomly generated A and B. */
  print_A();
  print_B();

  upc_barrier(2);

  /* Find X such that AX=B. */
  iter = jacobi_kernel();

  upc_barrier(3);

  /* Print out the result. */
  print_X(iter);

  return 0;
}
```

A.3 LU Decomposition

```
/*************************************************************************/
/*                                                                       */
/*  Copyright (c) 1994 Stanford University                               */
/*                                                                       */
/*  All rights reserved.                                                 */
/*                                                                       */
/*  Permission is given to use, copy, and modify this software for any   */
/*  non-commercial purpose as long as this copyright notice is not       */
/*  removed.  All other uses, including redistribution in whole or in    */
/*  part, are forbidden without prior written permission.                */
/*                                                                       */
/*  This software is provided with absolutely no warranty and no         */
/*  support.                                                             */
/*                                                                       */
/*************************************************************************/

/*************************************************************************/
/*                                                                       */
/*  Parallel dense blocked LU factorization (no pivoting)                */
/*                                                                       */
/*  This version contains one dimensional arrays in which the matrix     */
/*  to be factored is stored.                                            */
/*                                                                       */
/*  Command line options:                                                */
/*                                                                       */
/*  -nN : Decompose NxN matrix.                                          */
/*  -pP : P = number of processors.                                      */
/*  -bB : Use a block size of B. BxB elements should fit in cache for    */
/*        good performance. Small block sizes (B=8, B=16) work well.     */
/*  -s  : Print individual processor timing statistics.                  */
/*  -t  : Test output.                                                   */
/*  -o  : Print out matrix values.                                       */
/*  -h  : Print out command line options.                                */
/*                                                                       */
/*  Note: This version works under both the FORK and SPROC models        */
/*                                                                       */
/*************************************************************************/

#include <stdio.h>
#include <math.h>
#include <stdlib.h>
#include <sys/time.h>
#include "upc.h"
#include "lu.h"
//MAIN_ENV

#define MAXRAND                         32767.0
#define DEFAULT_N                         128
#define DEFAULT_P                           1
#define DEFAULT_B                          16
#define min(a,b) ((a) < (b) ? (a) : (b))

#ifdef MUPC_TRACE_RD
MUPC_TRACE_RD
#endif

//Everthing in globalmemory corrspond to shared types
shared double t_in_solve[THREADS];
shared double t_in_mod[THREADS];
shared double t_in_bar[THREADS];
shared double t_in_fac[THREADS];
```

```
shared double completion[THREADS];
shared struct timeval rf;
shared struct timeval  rs;
shared struct timeval done;
shared int id;
upc_lock_t *idlock;

/*
struct GlobalMemory {
  //double *t_in_fac;

  shared double *t_in_solve;
  shared double *t_in_mod;
  double *t_in_bar;
  double *completion;
  struct timeval starttime;
  struct timeval rf;
  struct timeval rs;
  struct timeval done;
  int id;
  //BARDEC(start)
  //LOCKDEC(idlock)
  upc_lock_t *idlock;
};
*/

//shared struct GlobalMemory *Global;

struct LocalCopies {
  double t_in_fac;
  double t_in_solve;
  double t_in_mod;
  double t_in_bar;
};

shared int n;           /* The size of the matrix */
shared int block_size; /* Block dimension */
int nblocks;                /* Number of blocks in each dimension */
int num_rows;               /* Number of processors per row of processor grid */
int num_cols;               /* Number of processors per col of processor grid */
//double *a;                  /* a = lu; l and u both placed back in a */
shared double *a;
//double *rhs;
shared double *rhs;
int *proc_bytes;            /* Bytes to malloc per processor to hold blocks
                               of A*/
int test_result = 0;        /* Test result of factorization? */
int doprint = 0;            /* Print out matrix values? */
int dostats = 0;            /* Print out individual processor statistics? */

void InitA();
void SlaveStart();
void OneSolve(int, int, shared double *, int, int);
void lu0(shared double *,int, int);
void bdiv(shared double *, shared double *, int, int, int, int);
void bmodd(shared double *, shared double*, int, int, int, int);
void bmod(shared double *, shared double *, shared double *, int, int, int, int);
void daxpy(shared double *, shared double *, int, double);
int BlockOwner(int, int);
void lu(int, int, int, struct LocalCopies *, int);
double TouchA(int, int);
void PrintA();
```

```
void CheckResult();
void printerr(const char *);

#define CLOCK(x) gettimeofday(&(x), NULL)

float calc_time(struct timeval tp_1st, struct timeval tp_2nd) {

  float diff = (tp_2nd.tv_sec-tp_1st.tv_sec) * 1000000.0 +
    (tp_2nd.tv_usec-tp_1st.tv_usec) ;
  return diff / 1000000.0;
}

int main(int argc, char* argv[])

{
  int i, j;
  int ch;
  double mint, maxt, avgt;
  double min_fac, min_solve, min_mod, min_bar;
  double max_fac, max_solve, max_mod, max_bar;
  double avg_fac, avg_solve, avg_mod, avg_bar;
  int proc_num;
  struct timeval start;

#ifdef TRACE_FUNC
       TRACE_FUNC;
#endif

  if(MYTHREAD==0){n=DEFAULT_N;block_size=DEFAULT_B;}

  CLOCK(start);

  if (!MYTHREAD) {

    while ((ch = getopt(argc, argv, "n:p:b:cstoh")) != -1) {
      switch(ch) {
      case 'n': n = atoi(optarg); break;
      case 'b': block_size = atoi(optarg); break;
      case 's': dostats = 1; break;
      case 't': test_result = !test_result; break;
      case 'o': doprint = !doprint; break;
      case 'h':
  printf("Usage: LU <options>\n\n");
  printf("options:\n");
  printf("  -nN : Decompose NxN matrix.\n");
  printf("  -bB : Use a block size of B. BxB elements should fit in cache \
for \n");
  printf("          good performance. Small block sizes (B=8, B=16) work well.\n");
  printf("  -c  : Copy non-locally allocated blocks to local memory before \
use.\n");
  printf("  -s  : Print individual processor timing statistics.\n");
  printf("  -t  : Test output.\n");
  printf("  -o  : Print out matrix values.\n");
  printf("  -h  : Print out command line options.\n\n");
  printf("Default: LU -n%1d -p%1d -b%1d\n",
         DEFAULT_N,DEFAULT_P,DEFAULT_B);
  exit(0);
  break;
      }
    }

    printf("\n");
```

```
    printf("Blocked Dense LU Factorization\n");
    printf("     %d by %d Matrix\n",n,n);
    printf("     %d Processors\n", THREADS);
    printf("     %d by %d Element Blocks\n",block_size,block_size);
    printf("\n");
  }

  upc_notify;

  num_rows = (int) sqrt((double) THREADS);
  for (;;) {
    num_cols = THREADS/num_rows;
    if (num_rows*num_cols == THREADS)
      break;
    num_rows--;
  }
  nblocks = n/block_size;
  if (block_size * nblocks != n) {
    nblocks++;
  }

  if(!MYTHREAD) {
    printf("     num_rows = %d\n",num_rows);
    printf("     num_cols = %d\n",num_cols);
    printf("     nblocks = %d\n",nblocks);
    printf("\n");
    printf("\n");
  }

  upc_wait;

  //a = (double *) G_MALLOC(n*n*sizeof(double));
  a = (shared double *) upc_all_alloc(n*n, sizeof(double));

  //rhs = (double *) G_MALLOC(n*sizeof(double));
  rhs = (shared double*) upc_all_alloc(n, sizeof(double));

  //Global = (struct GlobalMemory *) G_MALLOC(sizeof(struct GlobalMemory));
  /*
    Global->t_in_fac = (double *) G_MALLOC(P*sizeof(double));
    Global->t_in_mod = (double *) G_MALLOC(P*sizeof(double));
    Global->t_in_solve = (double *) G_MALLOC(P*sizeof(double));
    Global->t_in_bar = (double *) G_MALLOC(P*sizeof(double));
    Global->completion = (double *) G_MALLOC(P*sizeof(double));
  */

/* POSSIBLE ENHANCEMENT:  Here is where one might distribute the a
   matrix data across physically distributed memories in a
   round-robin fashion as desired. */

  //BARINIT(Global->start);
  //LOCKINIT(Global->idlock);
  idlock = upc_all_lock_alloc();
  //Global->id = 0;
  if (MYTHREAD == 0)
    id = 0;

  //Fork off code is unnecessary due to spmd model
  /*
  for (i=1; i<P; i++) {
    CREATE(SlaveStart)
  }
```

```
*/

InitA();
if (MYTHREAD == 0 && doprint) {
  printf("Matrix before decomposition:\n");
  PrintA();
}


//SlaveStart(MyNum);
SlaveStart();

upc_barrier;
//WAIT_FOR_END(P-1)

if (MYTHREAD == 0) {
  if (doprint) {
    printf("\nMatrix after decomposition:\n");
    PrintA();
  }

  if (dostats) {
    maxt = avgt = mint = completion[0];
    for (i=1; i<THREADS; i++) {
if (completion[i] > maxt) {
  maxt = completion[i];
}
if (completion[i] < mint) {
  mint = completion[i];
}
avgt += completion[i];
    }
    avgt = avgt / THREADS;

    min_fac = max_fac = avg_fac = t_in_fac[0];
    min_solve = max_solve = avg_solve = t_in_solve[0];
    min_mod = max_mod = avg_mod = t_in_mod[0];
    min_bar = max_bar = avg_bar = t_in_bar[0];

    for (i=1; i<THREADS; i++) {
if (t_in_fac[i] > max_fac) {
  max_fac = t_in_fac[i];
}
if (t_in_fac[i] < min_fac) {
  min_fac = t_in_fac[i];
}
if (t_in_solve[i] > max_solve) {
  max_solve = t_in_solve[i];
}
if (t_in_solve[i] < min_solve) {
  min_solve = t_in_solve[i];
}
if (t_in_mod[i] > max_mod) {
  max_mod = t_in_mod[i];
}
if (t_in_mod[i] < min_mod) {
  min_mod = t_in_mod[i];
}
if (t_in_bar[i] > max_bar) {
  max_bar = t_in_bar[i];
}
if (t_in_bar[i] < min_bar) {
```

```
      min_bar = t_in_bar[i];
    }
    avg_fac += t_in_fac[i];
    avg_solve += t_in_solve[i];
    avg_mod += t_in_mod[i];
    avg_bar += t_in_bar[i];
        }
        avg_fac = avg_fac/THREADS;
        avg_solve = avg_solve/THREADS;
        avg_mod = avg_mod/THREADS;
        avg_bar = avg_bar/THREADS;
      }
      printf("------------------------------PROCESS_STATISTICS\n");
      printf("----------------Total-------Diagonal------Perimeter-------Interior--------\
Barrier\n");
      printf("-Proc----------Time----------Time----------Time------------Time----------\
Time\n");
      printf("----0-----%10.6f-----%10.6f-----%10.6f-----%10.6f-----%10.6f\n",
       completion[0],t_in_fac[0],
       t_in_solve[0],t_in_mod[0],
       t_in_bar[0]);
      if (dostats) {
        for (i=1; i<THREADS; i++) {
    printf("--%3d-----%4.6f-----%4.6f-----%4.6f-----%4.6f-----%4.6f\n",
           i,completion[i],t_in_fac[i],
           t_in_solve[i],t_in_mod[i],
           t_in_bar[i]);
        }
        printf("--Avg-----%10.6f-----%10.6f-----%10.6f-----%10.6f-----%10.6f\n",
         avgt,avg_fac,avg_solve,avg_mod,avg_bar);
        printf("--Min-----%10.6f-----%10.6f-----%10.6f-----%10.6f-----%10.6f\n",
         mint,min_fac,min_solve,min_mod,min_bar);
        printf("--Max-----%10.6f-----%10.6f-----%10.6f-----%10.6f-----%10.6f\n",
         maxt,max_fac,max_solve,max_mod,max_bar);
      }
      printf("\n");
      printf("------------------------------TIMING_INFORMATION\n");
      //printf("Start time                                  : %16d\n",
      //       starttime);
      //printf("Initialization finish time                  : %16d\n",
      //       rs);
      //printf("Overall finish time                         : %16d\n",
      //       rf);
      printf("Total time with initialization -----: %4.6f\n",
       calc_time(start, rf));
      printf("Total time without initialization --: %4.6f\n",
       calc_time(rs, rf));
      printf("\n");

      if (test_result) {
        printf("------------------------------TESTING_RESULTS\n");
        CheckResult();
      }

    }

#ifdef TRACE_FUNC_RET
    TRACE_FUNC_RET;
#endif

    return 0;
}
```

```
void SlaveStart()

{

/* POSSIBLE ENHANCEMENT:  Here is where one might pin processes to
   processors to avoid migration */

  OneSolve(n, block_size, a, MYTHREAD, dostats);
}


void OneSolve(n, block_size, a, MyNum, dostats)

shared double *a;
int n;
int block_size;
int MyNum;
int dostats;

{
  unsigned int i;
  struct timeval myrs, myrf, mydone;
  struct LocalCopies *lc;

#ifdef TRACE_FUNC
       TRACE_FUNC;
#endif

  lc = (struct LocalCopies *) malloc(sizeof(struct LocalCopies));
  if (lc == NULL) {
    fprintf(stderr,"Proc %d could not malloc memory for lc\n",MyNum);
    exit(-1);
  }
  lc->t_in_fac = 0.0;
  lc->t_in_solve = 0.0;
  lc->t_in_mod = 0.0;
  lc->t_in_bar = 0.0;

  /* barrier to ensure all initialization is done */
  //BARRIER(Global->start, P);
  upc_barrier;

  /* to remove cold-start misses, all processors begin by touching a[] */
  TouchA(block_size, MyNum);

  //BARRIER(Global->start, P);
  upc_barrier;

/* POSSIBLE ENHANCEMENT:  Here is where one might reset the
   statistics that one is measuring about the parallel execution */

  if ((MyNum == 0) || (dostats)) {
    CLOCK(myrs);
  }

  lu(n, block_size, MyNum, lc, dostats);

  if ((MyNum == 0) || (dostats)) {
    CLOCK(mydone);
  }
```

```
  //BARRIER(Global->start, P);
  upc_barrier;

  if ((MyNum == 0) || (dostats)) {
    CLOCK(myrf);
    t_in_fac[MyNum] = lc->t_in_fac;
    t_in_solve[MyNum] = lc->t_in_solve;
    t_in_mod[MyNum] = lc->t_in_mod;
    t_in_bar[MyNum] = lc->t_in_bar;
    completion[MyNum] = calc_time(myrs, mydone);
  }
  if (MyNum == 0) {
    rs = myrs;
    done = mydone;
    rf = myrf;
  }

#ifdef TRACE_FUNC_RET
  TRACE_FUNC_RET;
#endif
}


void lu0(a, n, stride)

shared double *a;
int n;
int stride;

{
  int j;
  int k;
  int length;
  double alpha;

#ifdef TRACE_FUNC
        TRACE_FUNC;
#endif

  for (k=0; k<n; k++) {
    /* modify subsequent columns */
    for (j=k+1; j<n; j++) {
      a[k+j*stride] /= a[k+k*stride];
      alpha = -a[k+j*stride];
      length = n-k-1;
      daxpy(&a[k+1+j*stride], &a[k+1+k*stride], n-k-1, alpha);
    }
  }

#ifdef TRACE_FUNC_RET
  TRACE_FUNC_RET;
#endif
}


void bdiv(a, diag, stride_a, stride_diag, dimi, dimk)

shared double *a;
shared double *diag;
int stride_a;
int stride_diag;
int dimi;
```

```
int dimk;

{
  int j;
  int k;
  double alpha;

#ifdef TRACE_FUNC
        TRACE_FUNC;
#endif

  for (k=0; k<dimk; k++) {
    for (j=k+1; j<dimk; j++) {
      alpha = -diag[k+j*stride_diag];
      daxpy(&a[j*stride_a], &a[k*stride_a], dimi, alpha);
    }
  }

#ifdef TRACE_FUNC_RET
  TRACE_FUNC_RET;
#endif
}


void bmodd(a, c, dimi, dimj, stride_a, stride_c)

shared double *a;
shared double *c;
int dimi;
int dimj;
int stride_a;
int stride_c;

{
  int i;
  int j;
  int k;
  int length;
  double alpha;

#ifdef TRACE_FUNC
        TRACE_FUNC;
#endif

  for (k=0; k<dimi; k++)
    for (j=0; j<dimj; j++) {
      c[k+j*stride_c] /= a[k+k*stride_a];
      alpha = -c[k+j*stride_c];
      length = dimi - k - 1;
      daxpy(&c[k+1+j*stride_c], &a[k+1+k*stride_a], dimi-k-1, alpha);
    }

#ifdef TRACE_FUNC_RET
  TRACE_FUNC_RET;
#endif
}


void bmod(a, b, c, dimi, dimj, dimk, stride)

shared double *a;
shared double *b;
```

```
shared double *c;
int dimi;
int dimj;
int dimk;
int stride;

{
  int i;
  int j;
  int k;
  double alpha;

#ifdef TRACE_FUNC
        TRACE_FUNC;
#endif

  for (k=0; k<dimk; k++) {
    for (j=0; j<dimj; j++) {
      alpha = -b[k+j*stride];
      daxpy(&c[j*stride], &a[k*stride], dimi, alpha);
    }
  }

#ifdef TRACE_FUNC_RET
  TRACE_FUNC_RET;
#endif
}


void daxpy(a, b, n, alpha)

shared double *a;
shared double *b;
double alpha;
int n;

{
  int i;

#ifdef TRACE_FUNC
        TRACE_FUNC;
#endif

  for (i=0; i<n; i++) {
    a[i] += alpha*b[i];
  }

#ifdef TRACE_FUNC_RET
  TRACE_FUNC_RET;
#endif
}


int BlockOwner(I, J)

int I;
int J;

{
  return((I%num_cols) + (J%num_rows)*num_cols);
}
```

```
void lu(n, bs, MyNum, lc, dostats)

int n;
int bs;
int MyNum;
struct LocalCopies *lc;
int dostats;

{
  int i, il, j, jl, k, kl;
  int I, J, K;
  //double *A, *B, *C, *D;
  shared double *A, *B, *C, *D;
  int dimI, dimJ, dimK;
  int strI;
  //unsigned int t1, t2, t3, t4, t11, t22;
  struct timeval t1, t2, t3, t4, t11, t22;
  int diagowner;
  int colowner;

#ifdef TRACE_FUNC
        TRACE_FUNC;
#endif

  strI = n;
  for (k=0, K=0; k<n; k+=bs, K++) {
    kl = k+bs;
    if (kl>n) {
      kl = n;
    }

    if ((MyNum == 0) || (dostats)) {
      CLOCK(t1);
    }

    /* factor diagonal block */
    diagowner = BlockOwner(K, K);
    if (diagowner == MyNum) {
      A = &(a[k+k*n]);
      lu0(A, kl-k, strI);
    }

    if ((MyNum == 0) || (dostats)) {
      CLOCK(t11);
    }

    //BARRIER(Global->start, P);
    upc_barrier;

    if ((MyNum == 0) || (dostats)) {
      CLOCK(t2);
    }

    /* divide column k by diagonal block */
    D = &(a[k+k*n]);
    for (i=kl, I=K+1; i<n; i+=bs, I++) {
      if (BlockOwner(I, K) == MyNum) {  /* parcel out blocks */
        il = i + bs;
        if (il > n) {
          il = n;
        }
```

```
        A = &(a[i+k*n]);
        bdiv(A, D, strI, n, il-i, kl-k);
      }
    }
    /* modify row k by diagonal block */
    for (j=kl, J=K+1; j<n; j+=bs, J++) {
      if (BlockOwner(K, J) == MyNum) {  /* parcel out blocks */
        jl = j+bs;
        if (jl > n) {
          jl = n;
        }
        A = &(a[k+j*n]);
        bmodd(D, A, kl-k, jl-j, n, strI);
      }
    }

    if ((MyNum == 0) || (dostats)) {
      CLOCK(t22);
    }

    //BARRIER(Global->start, P);
    upc_barrier;

    if ((MyNum == 0) || (dostats)) {
      CLOCK(t3);
    }

    /* modify subsequent block columns */
    for (i=kl, I=K+1; i<n; i+=bs, I++) {
      il = i+bs;
      if (il > n) {
        il = n;
      }
      colowner = BlockOwner(I,K);
      A = &(a[i+k*n]);
      for (j=kl, J=K+1; j<n; j+=bs, J++) {
        jl = j + bs;
        if (jl > n) {
          jl = n;
        }
        if (BlockOwner(I, J) == MyNum) {  /* parcel out blocks */
          B = &(a[k+j*n]);
          C = &(a[i+j*n]);
          bmod(A, B, C, il-i, jl-j, kl-k, n);
        }
      }
    }
    if ((MyNum == 0) || (dostats)) {
      CLOCK(t4);
      lc->t_in_fac += calc_time(t1, t11);
      lc->t_in_solve += calc_time(t2, t22);
      lc->t_in_mod += calc_time(t3, t4);
      lc->t_in_bar += calc_time(t11,t2) + calc_time(t22, t3);
    }
  }

#ifdef TRACE_FUNC_RET
  TRACE_FUNC_RET;
#endif
}
```

```
//void InitA(double *rhs)
void InitA()

{
  int i, j;

#ifdef TRACE_FUNC
        TRACE_FUNC;
#endif

  srand48((long) 1);
  for (j=0; j<n; j++) {
    for (i=0; i<n; i++) {
      a[i+j*n] = (double) lrand48()/MAXRAND;
      if (i == j) {
  a[i+j*n] *= 10;
      }
    }
  }

  upc_forall (j=0; j<n; j++; j) {
    rhs[j] = 0.0;
  }
  for (j=0; j<n; j++) {
    upc_forall (i=0; i<n; i++; i) {
      rhs[i] += a[i+j*n];
    }
  }

#ifdef TRACE_FUNC_RET
  TRACE_FUNC_RET;
#endif
}


double TouchA(bs, MyNum)

int bs;
int MyNum;

{
  int i, j, I, J;
  double tot = 0.0;

#ifdef TRACE_FUNC
        TRACE_FUNC;
#endif

  for (J=0; J*bs<n; J++) {
    for (I=0; I*bs<n; I++) {
      if (BlockOwner(I, J) == MyNum) {
        for (j=J*bs; j<(J+1)*bs && j<n; j++) {
          for (i=I*bs; i<(I+1)*bs && i<n; i++) {
            tot += a[i+j*n];
          }
        }
      }
    }
  }

#ifdef TRACE_FUNC_RET
  TRACE_FUNC_RET;
```

```
#endif

  return(tot);
}


void PrintA()
{
  int i, j;

#ifdef TRACE_FUNC
        TRACE_FUNC;
#endif

  for (i=0; i<n; i++) {
    for (j=0; j<n; j++) {
      printf("%8.1f ", a[i+j*n]);
    }
    printf("\n");
  }

#ifdef TRACE_FUNC_RET
  TRACE_FUNC_RET;
#endif
}


void CheckResult()

{
  int i, j, bogus = 0;
  double *y, diff, max_diff;

#ifdef TRACE_FUNC
        TRACE_FUNC;
#endif

  y = (double *) malloc(n*sizeof(double));
  if (y == NULL) {
    printerr("Could not malloc memory for y\n");
    exit(-1);
  }
  for (j=0; j<n; j++) {
    y[j] = rhs[j];
  }
  for (j=0; j<n; j++) {
    y[j] = y[j]/a[j+j*n];
    for (i=j+1; i<n; i++) {
      y[i] -= a[i+j*n]*y[j];
    }
  }

  for (j=n-1; j>=0; j--) {
    for (i=0; i<j; i++) {
      y[i] -= a[i+j*n]*y[j];
    }
  }

  max_diff = 0.0;
  for (j=0; j<n; j++) {
    diff = y[j] - 1.0;
    if (fabs(diff) > 0.00001) {
```

```
        bogus = 1;
        max_diff = diff;
      }
    }
    if (bogus) {
      printf("TEST FAILED: (%.5f diff)\n", max_diff);
    } else {
      printf("TEST PASSED\n");
    }
    free(y);

#ifdef TRACE_FUNC_RET
  TRACE_FUNC_RET;
#endif
}


void printerr(const char *s)

{
  fprintf(stderr,"ERROR: %s\n",s);
}
```

A.4 Stencil

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <assert.h>
#include <upc.h>

#ifdef MUPC_TRACE_RD
MUPC_TRACE_RD
#endif

#define N      6
#define ITERS 100

shared [N] double a[N][THREADS*N];
shared [1] double dmax[THREADS];
int n;

#define A(i,j) a[(i)%N][((i)/N)*N*n+(j)]

void stencil(int i, int j)
{
  double t;

#ifdef TRACE_FUNC
        TRACE_FUNC;
#endif

  t = A(i,j);

#ifdef EIGHT_PT_STENCIL
  t += A(i-1,j-1);
  t += A(i-1,j);
  t += A(i-1,j+1);

  t += A(i,j-1);
  t += A(i,j+1);

  t += A(i+1,j-1);
  t += A(i+1,j);
  t += A(i+1,j+1);
  t /= 9.0;
#else
  t += A(i-1,j);
  t += A(i+1,j);
  t += A(i,j-1);
  t += A(i,j+1);
  t /= 5.0;
#endif

  A(i,j) = t;

#ifdef TRACE_FUNC_RET
  TRACE_FUNC_RET;
#endif
}

int main()
{
  int i, j, iter;

#ifdef TRACE_FUNC
```

```
        TRACE_FUNC;
#endif

  /* Ensure number of threads is a square. */
  n = (int) sqrt( (double) THREADS );
  assert( (n*n) == THREADS );

  /* Initialize A. */
  for(i=0;i<N;++i)
    for(j=0;j<N;++j)
      a[i][N*MYTHREAD+j] = (double) MYTHREAD;

  upc_barrier;

  for(iter=0;iter<ITERS;++iter)
  {
    /* Update all points with 4/8-pt stencil. */
    for(i=1;i<(N*n)-1;++i)
    {
      for(j=1;j<(N*n)-1;++j)
      {
        if(MYTHREAD==upc_threadof(&A(i,j)))
        {
          stencil(i,j);
        }
      }
    }

    /* Update dmax. */
    dmax[MYTHREAD] = (double) iter + 1.0;

    upc_barrier;

    /* Check for completion. */
    for(i=0;i<THREADS;++i)
      if(dmax[i]<0.0) goto end;
  }
end:
  return 0;
}
```