UNIVERSITY OF CALIFORNIA, SAN DIEGO

Using a Proxy to Enable Communication Overlap with Computation

A thesis submitted in partial satisfaction of the
requirements for the degree Master of Science
in Computer Science & Engineering

by

Stephen Lau

Committee in charge:

Professor Scott B. Baden, Chairperson
Professor Larry Carter
Professor William G. Griswold

2003

The thesis of Stephen Lau is approved:

Chair

University of California, San Diego

2003

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ABSTRACT OF THE THESIS

Using a Proxy to Enable Communication Overlap with Computation

by

Stephen Lau
Master of Science in Computer Science & Engineering
University of California, San Diego, 2003
Professor Scott B. Baden, Chair

Parallel computation with distributed memory has two key phases: local computation on the nodes of the system, and communication between these nodes. While communication speeds have increased with technological improvements, the level of growth can not match the rapid increase of speed of processors leading to a rising imbalance in the cost of communication relative to computation. Overlap of communication with computation is thus desirable as it helps to amortize the cost of communication latency by hiding it during periods of computation.

Typically, nodes in a parallel machine communicate via message passing libraries such as MPI. However, MPI is not always able to realize overlap, even with the asynchronous routines provided. The MPI standard does not mandate when a message has to be delivered, and can often delay transmission of messages arbitrarily. Also, larger messages may exceed system pre-set synchronous message limits thus delaying the return of control to the sender and increasing the perceived overhead of the message passing system.

We present a hybrid model of computation combining the two most common paradigms of parallel communication: message passing, and shared memory, to enable a Proxy thread running on a dedicated CPU on an clustered SMP system to maximize overlap of communication with computation.

We gathered results from two SMP clusters; *LeMieux* with 4-way nodes, and *Blue Horizon* with 8-way nodes.

Chapter I

Introduction

There have been many trends for design of parallel computers. Historically, parallel computers have been composed of highly specialised and expensive unique supercomputers. However, this proved to be too expensive to build, leading to designs attempting to use cheaper components. Multi-computers, composed of networks of single processor computers have been on one end of the spectrum (Figure I.1). Another design used multi-processor computers which were single machines with multiple CPUs within the machine. This led to the next obvious trend which was to combine the two single tier hierarchies and use clusters of multi-processor computers to form a multi-tier hierarchy (Figure I.2) [12] [11].

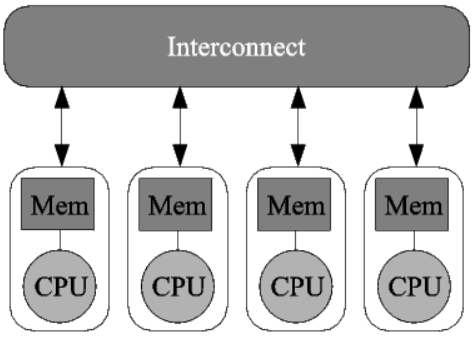

Figure I.1: Single-tier Computer System

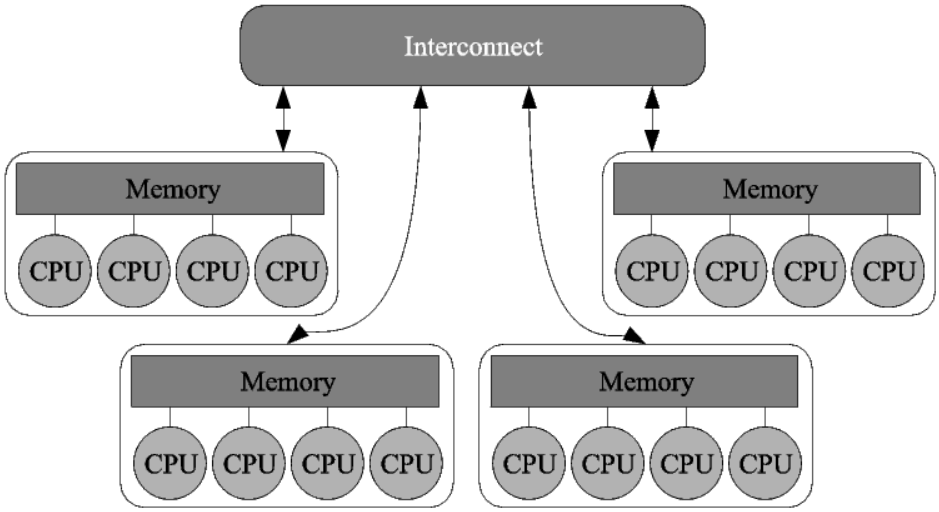

Figure I.2: Multi-tier Computer System

The relative cost of computational power has fallen with Moore's Law making clusters of workstations or multi-processor computers more feasible and cost-effective. Commodity-off-the-shelf computers and workstations, due to high sales volumes, and more rapid technological innovation provide superior cost-performance ratios as compared to more traditional parallel machines or micro-computers. However, the level of advancement in communications technology has not kept up with the level of growth of computation, leading to a disparity in computation vs. communication.[14] The growth of computational power is quickly out-pacing the growth of communication, leading to a highest cost for communication within parallel algorithms. Hardware solutions have attempted to remedy this by off-loading some computational load of communication protocols to on-board processors on the network interfaces themselves, but in addition to being expensive, they often can't off-load basic software protocol overheads. With the decreasing costs of symmetric multi-processing (SMP) machines, clusters with high computational power are becoming more and more available at cheaper costs while communications costs remain at a higher premium.

This is a significant concern in clusters of multi-processor computers, as

they typically are limited to one network input/output device per node. In multi-computers, where each node has only one processor, the network I/O device is more capable of handling the communication load generated by the single processor. Multi-processors, which have no external network, don't have a communications load problem as each processor is connected to every other processor through the machines internal bus. However, with clusters of SMPs, the network I/O device must be shared between all the processors within the node. Each node can have 4, 8, 16, or even more processors per node. As more processors are added, the costliness of communications rises.

Mult-tier architectures attempt to save on communication costs by aggregating processors onto a node. For example, given a system which has 512 processors total, a 128 node arrangement where each node has 4 processors might be slightly faster in that its available bandwidth per processor is higher than that of a 64 node 8-way arrangement. However, as more nodes are added to the interconnect, the cost of the switch becomes more expensive, sometimes prohibitively so. Multi-tiered computers must balance the trade-off of cost of the interconnect versus the available communication capabilities for each node on the interconnect. While narrowing the width of the node by decreasing the number of processors per node will increase communications performance, the increase in cost will be counter to the whole point of using commodity component clusters!

I.A The need for overlap

The most common means of implementing communication between nodes of a multi-computer system is by message passing. Message passing software libraries such as MPI[22] (Message Passing Interface - the industry standard for message passing) often do not provide an easy way to eliminate communications software overheads. Their basic message passing routines are synchronous resulting in blocks of computation with global synchronisation points where all clients

sync up and communicate. This model of parallel computation is called *Bulk Synchronous Parallelism* (BSP) [4]. They were also called *Loosely Synchronous* problems by G. C. Fox [13]. Smaller messages can often be sent immediately without requiring a synchronous receive call to be posted prior (referred to as *Rendezvous* mode communications) but once they exceed the *Eager Limit*[1] set by the system, then the synchronous receive call must be called on the receiving side before the send call will return control back to the caller [2] . MPI gets around this by offering non-blocking asynchronous routines in an attempt to provide overlap of communication with computation by allowing the user to setup or "post" communication calls that will return quickly, allowing the program to continue on into a period of computation while the system asynchronously handles the communication in the background. However, even these aren't foolproof. The MPI standard doesn't mandate or even set a ceiling for when a message must be delivered. It can be held and delayed for an arbitrary amount of time before delivery. [22] Overlap strategies involving these non-blocking communication calls necessitate network interface polling [20] [18] which compromises performance by forcing the system to periodically context-switch to handle the system-level polling calls. At a more basic level, it is also possible that the message layer library (MPI) may not be optimised to the hardware platform correctly and may not be able to take advantage of any intelligent asynchronous communication capabilities the hardware or network interface may have.

The cost of message passing protocols such as MPI can be broken down into two basic components: *overhead* and *transmission time*. The overhead is the cost of the message startup: essentially, all the time involved aside from the actual message transmission. This includes components such as data serialisation,

[1] The system's defined *Eager Limit* is the largest point-to-point message which will be accepted by a synchronous communication-send if no matching receive call has been posted prior, allowing the synchronous send to return control back to the caller immediately similar to an asynchronous send.

[2] http://www.cs.unb.ca/acrl/training/general/programming_ibm_power3/sld048.htm
http://www.lanl.gov/orgs/cic/cic8/para-dist-team/MSGPASS/MPI/...
IMPLEMENTATION_ISSUES/CHARACTERISTICS/unexpected.html

protocol stack startup, memory buffer packing, etc. The transmission time is the actual network transfer cost of sending the complete message. The overhead is mainly limited by on-node resources such as the computational ability to quickly pack data and get the data to the network interface; it is not limited by bandwidth or the interconnect itself, unlike the transmission time which is governed almost solely by the bandwidth of the interconnect. In asynchronous calls such as those provided by the immediate mode MPI calls, only the transmission time can be overlapped. The overhead cost of calling the basic communications primitives can not be avoided simply. Overlap using asynchronous calls does however achieve the goal of overlapping the perceived communication time with computational periods thus amortizing the cost of the network transfer out among the computation time period.

This is all of course dependent on the MPI library's ability to realise overlap using asynchronous communication to begin with. In our Results chapter we provide data showing one of our machine's MPI library's inability to attain overlap using MPI's Imode asynchronous communication primitives.

Different classes of problems lend themselves to overlap differently. Independent, or embarrassingly parallel computation classes can be rewritten quite easily to utilise overlap as iterations or datasets are relatively independent of each other. However, problems where iterations and/or data are dependently intertwined, or problems where frequent periodic synchronisation points are required are often harder to achieve overlap with.

I.B Enabling Overlap via a Proxy

Proxies are useful as a intermediary communications manager. They allow nodes to focus on being computational clients, and they can help amortize the cost of communications on multi-processors [17]. In addition, because clusters of multiprocessors are composed of SMP nodes, proxies give an opportunity to

achieve overlap on machines that may not support communication overlap via conventional asynchronous calls.

Consider a multicomputer with four-way SMP nodes; one can set aside one CPU per node as the proxy for the remaining three CPUs. Rather than having each CPU incur its own network I/O costs for each Send/Receive of data, the proxy (typically the first of the SMP CPUs, for no necessary reason) incurs all costs. In a system of single-processor nodes, using conventional overlap techniques (such as asynchronous message passing calls) allow the transmission time to be hidden (as noted above), however the costs associated with the overhead of the messaging system can not be hidden.

The usefulness and practicality of a proxy becomes more evident on clusters where the nodes are symmetric multi-processing (SMP) machines. Using one of the CPUs on an SMP node to manage all the communication will allow these overhead costs to be incurred by the proxy leaving the other CPUs free to continue processing, thus eliminating a serial portion of the communication code. The downside to this is, of course, the loss of one of the CPUs from the total computational power of the node if using a dedicated CPU for the proxy. If sharing a CPU between a proxy and compute client, thrashing of that CPU and its cache may occur leading to disproportionate run-times for that compute client forced to share its CPU with the proxy. If running a synchronous algorithm, this may in turn slow down the other CPUs forced to synchronise with the shared compute client.

While we are interested chiefly in proxies as a means of obtaining better communications overlap, they can also be used as a means of giving protected access to a single network I/O interface as mentioned in the paper by Lim, Heidelberger, Pattnaik, and Snir.[17]. This is often an important consideration, as many libraries such as MPI are not guaranteed to be thread-safe. MPICH (a free implementation of MPI popular under Linux and BSD clusters)[3] for exam-

[3]MPICH - A Portable MPI Implementation can be found at

ple is not thread safe. Use of a proxy allows all MPI communications routines to be called from the proxy thread, maintaining protected access to the network interface. IBM's implementation of MPI[2], for instance, is not guaranteed to be thread-safe. This protected access also leads to higher throughput communication, as the single proxy thread saves the network I/O from being shared and switched between multiple other threads.

I.C Plan and Summary of Results

The work in this thesis will extend work done by Fink[11] and Baden[3]. We attempt to show that for certain classes of problems, using a communication proxy will prove advantageous over asynchronous immediate mode MPI calls in enabling overlap of communication with computation. For the two applications we implement, a partial differential equation solver (RedBlack3D), and matrix-matrix multiplication (SUMMA), we observed mixed results. We observed that getting the best performance out of problem retooled to use a proxy required balancing between the size of the problem as well as the node configuration used to solve the problem. Smaller problem sizes tend to not provide enough communication, and larger problem sizes tend to provide too much computation, thus making a proxy beneficial for some classes of problems, but not in the general case.

---

http://www-unix.mcs.anl.gov/mpi/mpich/

Chapter II

Motivating Applications

II.A Finite Difference Method

The first application is called *RedBlack3D*, a solver which solves a partial differential equation (Poisson's equation) in three dimensions. It uses a successive over-relaxation method, Gauss-Seidel's method to solve the discrete equation using a seven-point nearest-neighbour (in all three dimensions) stencil.

The basic strategy of RedBlack3D, not taking into account the multi-tier hierarchy of the computer, is to divide the grid a flat uniform distribution chunks of data evenly distributed across all nodes. Figure II.1 shows an example distribution of data across four nodes. The RedBlack3D algorithm is divided into two phases:

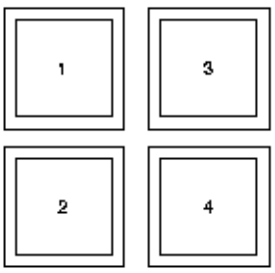

Figure II.1: The single-tier synchronous solution, partitioned in a flat uniform distribution across 4 nodes, showing the halo region of ghost cells for each node

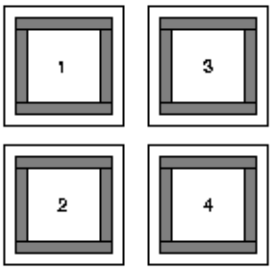

Figure II.2: The single-tier asynchronous solution, partitioned in the same flat uniform distribution across 4 nodes, showing the same halo region of ghost cells. The shaded region denotes the outer annulus which is dependent upon the completion of the asynchronous communication.

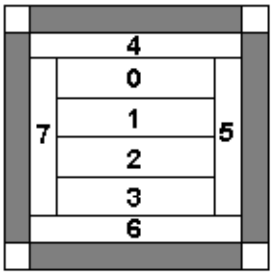

Figure II.3: The multi-tier solution, partitioned at the node level on a 4-way SMP, showing the halo as the shaded region. The annulus is depicted in the outer regions labelled 4-7. The interior is labelled as regions 0-3, one for each processor on the node.

```
ST_Relax()
{
    initiate ghost cell communication
    wait for ghost cell communication completion
    serialRelax(U[proc_entire])
}
```

Figure II.4: Pseudo-code for the RedBlack3D single-tier relaxation kernel

```
MT_Relax()
{
    initiate ghost cell communication
    serialRelax(U[proc_inner], rhs[proc_inner])
    wait for ghost cell communication completion
    serialRelax(U[proc_annulus])
}
```

Figure II.5: Pseudo-code for the RedBlack3D multi-tier relaxation kernel

```
MT_Relax()
{
    initiate ghost cell communication
    parallel for (each processor in node)
        serialRelax(U[node_inner], rhs[node_inner])
    wait for ghost cell communication completion
    parallel for (each processor in node)
        serialRelax(U[node_annulus], rhs[node_annulus])
}
```

Figure II.6: Pseudo-code for the RedBlack3D proxy relaxation kernel

computation and communication.

In the synchronous, non-overlapped version single-tier version, the sub-problems are defined solely as one region per node, with a ghost cell region bordering the sub-problem. This ghost cell region is copied during the communication phase from process that owns the neighbouring sub-problem. The computation phase then computes and updates the local sub-problem, and sends out the border region to its neighbours so they can update their own ghost regions.

In the asynchronous, overlapped single-tier version, the sub-problem is further divided into regions that depend on the ghost cells (called the *annulus*), and the region that is independent of any ghost cell updates (called the *inner* region). The two phases are then further sub-divided so that a first communication phase is started where asynchronous communication is initiated to receive the ghost cells from its neighbours. Local computation is then done on the inner region. After the inner computation is completed, the process waits to ensure the asynchronous communication has completed; upon completion, the computation updates the annulus region. After the annulus region has completed, it is sent to its neighbours so they can update their ghost cell values.

In the asynchronous, overlapped multi-tiered proxy version, the sub-problems are defined at the node level rather than the processor level. The annulus and inner regions are defined the same way as in the asynchronous single-tier version, except now the annulus region is wrapped around the entire node, rather than just the processors. The phases are the same in that a communication phase is first run initiating asynchronous communication on the ghost cells from the node's neighbours. The local computation divides up the inner region into p smaller disjoint sub-blocks (not necessarily uniform) which are mapped to the number of processors within the node. The processors compute and update their own sub-blocks. Upon completion, the algorithm waits to ensure ghost cell updates have completed before dispatching the processors to compute and update the annulus region and communicate the updated values to the node's neighbours. Figure II.3

shows the partition and distribution of data at the node level, depicting the annulus and partitioning of the inner region.

In the MPI native non-blocking asynchronous model, communication is flattened out into the single-tier model, which ignores the node-level communication. All communication is done processor to processor. An MPI geometry is defined which blocks all the processors on a node into a grid, thus minimizing off-node communication. However, the proxy model maps more closely onto the multi-tier model, eliminating processor-level communication. In the proxy model, processors do not communicate to other processors off node. All communication off-node is handled at the node level by the proxy thread. The non-overlapped model will of course tend to have the highest communication cost as the processes wait for communication to complete before continuing on to the computational update phase. The overlapped single-tier model attempts to overlap the communication with the local computation. Communication in this single-tier overlapped model will occur at two speeds as communication between two neighbouring processes residing within the same node should be fast as it does not have to go out to the interconnect while two processes residing on different nodes will be slowed by the interconnect. Communication in the multi-tier proxy model should be more optimal, as it eliminates excessive annulus region partitioning because the annulus is defined at the node level rather than the processor level.

II.B Matrix-matrix Multiplication

The second application implemented was Van de Geijn's *SUMMA* (Scalable Universal Matrix Multiplication Algorithm) [15] [1]. SUMMA is a fast matrix to matrix multiplication algorithm which allows for fast scalable matrix multiplication. More importantly, the SUMMA algorithm is very easy to re-write to enable overlap. Essentially, SUMMA uses broadcasting of data within row and column groups to more efficiently compute the product, as shown in Figures II.7-II.9. The

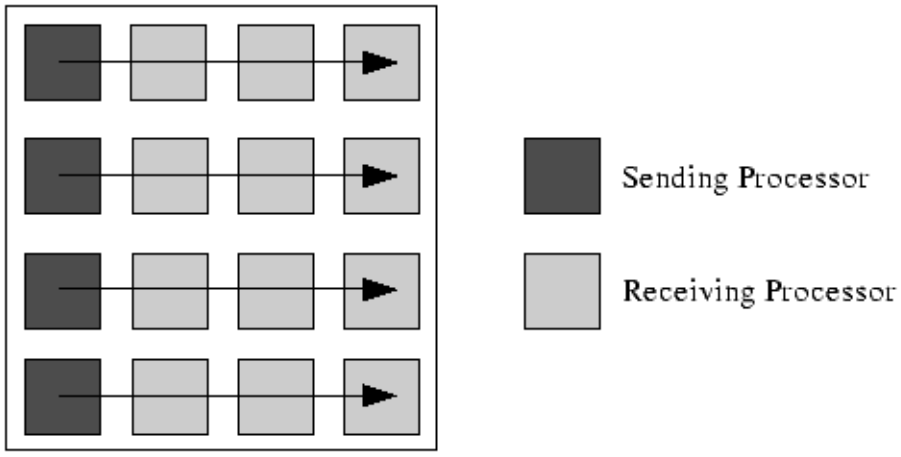

Figure II.7: Phase 1 - All processors in the same column group as the root node broadcast their sub-matrices of A to all the nodes in their same row group (i.e.: laterally along the mesh)

local sub-problems are computed using the BLAS level 3 `dgemm` matrix-multiply kernel which is generally provided as part of the system libraries.

A detailed description of the algorithm can be found in [15]. A brief overview is necessary here to explain how overlap is achieved, however. The processors are partitioned into a $r\ x\ c$ mesh of nodes, with the two-dimensional matrices overlaid on top. In Figures II.7-II.9, a 4x4 grid of processors is overlaid on the matrix to partition it up into 16 equally sized sub-matrices. Each sub-matrix is of dimension i rows by j columns. For our purposes, since we're operating on a uniform square matrix, $i = j$.

The matrix-matrix multiplication is then executed as a sequence of rank-1 updates, where each node loops over the common dimension k (given that A=$m\ x\ k$ and B=$k\ x\ n$). The iteration runs in panels, processing several k together to benefit from the use of locality and stride. Within the loop (iterating with i from 0 to k as mentioned), the algorithm works in three phases. During phase 1 (Figure II.7), all processors in the i^{th} column broadcast a copy of their **A** sub-matrix to all the other

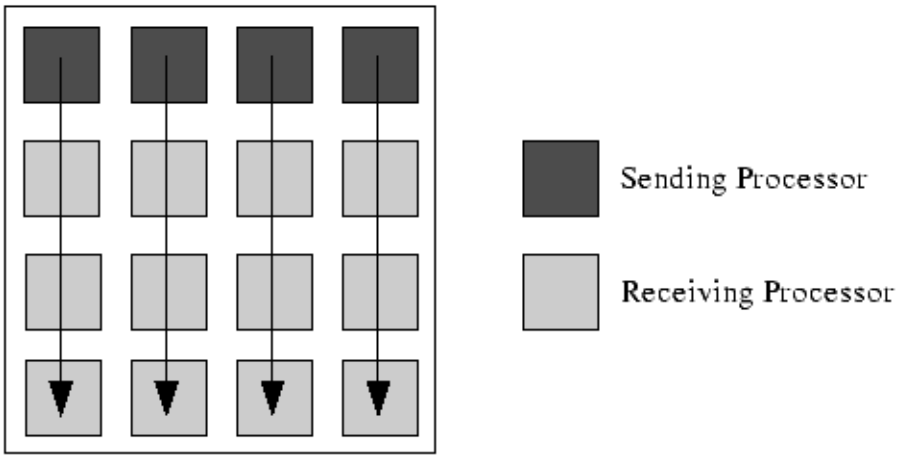

Figure II.8: Phase 2 - All processors in the same row group as the root node broadcast their sub-matrices of B to all the nodes in their same column group (i.e.: longitudinally along the mesh)

processors in the same row group (e.g.: a lateral broadcast along the rows of the mesh). The receiving processors receive this sub-matrix into a local buffer R. In phase 2 (Figure II.8), all processors in the i^{th} row broadcast a copy of their **B** sub-matrix to all the other processors in the same column group (e.g.: a longitudinal broadcast along the columns of the mesh). The receiving processors receive this sub-matrix into a local buffer S. In the third and final phase (Figure II.9)e, each processors does its local `dgemm` operation. The root node (the node which owns both the i^{th} row and column) does a `dgemm` of the two local sub-matrices of A and B. The nodes that broadcast laterally do a `dgemm` of their sub-matrix of A with the received sub-matrix of B in S (from phase 2). The nodes that broadcast longitudinally do a `dgemm` of their sub-matrix of B with the received sub-matrix of A in R (from phase 1). The remaining nodes in the mesh do a local `dgemm` of the received sub-matrix of A in R (from phase 1) with the received sub-matrix of B in S (from phase 2).

For the synchronous model, the algorithm follows a basic model where for

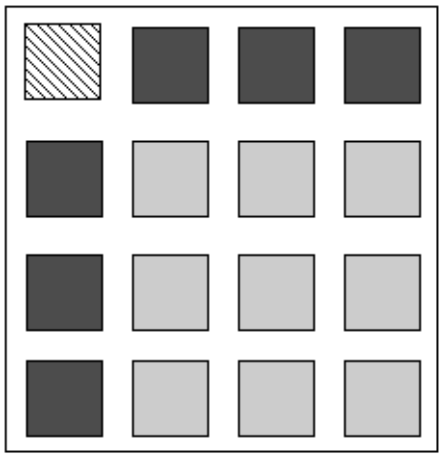

Figure II.9: Phase 3 - Local `dgemm` phase where the root node (hatched) multiplies the two sub-matrices of A and B it owns together. The neighbours (shaded) in its row group matrix-multiply their sub-matrix of B with the received sub-matrix of A from Phase 1. The neighbours (shaded) in its column group matrix-multiply their sub-matrix of A with the received sub-matrix of B from Phase 2. The other nodes matrix-multiply both the received A and B sub-matrices.

$C_{mn} = 0$
for $i = 0$ to $(k - 1)$
{
broadcast A_{mi} to members of row group
broadcast B_{in} to members of column group
$C_{mn} +=$**dgemm**(A_{mi} x B_{in})
}

Figure II.10: Pseudo-code for the SUMMA operation for $C = AB$

each iteration, the row broadcast is done first, followed by the column broadcast - both done using synchronous communication as the data being received is necessary for the dgemm sub-matrix multiplication. The `dgemm` is then run, and the next iteration is then started.

Enabling overlap is relatively straightforward on the SUMMA algorithm due to the clean separation and relative independence of communication within each loop iteration. The very first iteration is dependent upon data already being there, so the R and S buffers are filled synchronously for this first iteration. Once that is complete however, for each iteration i, asynchronous communication is started on the $i+1$ iteration before starting the `dgemm` operation on the i sub-matrix multiplication. When the `dgemm` is complete, the node waits on the asynchronous communication for $i+1$ to ensure that all the data necessary for the $i+1$ iteration is received properly. Once the data transfer is complete, the next iteration $i+1$ is started, thus asynchronously starting the communication for the $i+2$ iteration and so on, so forth. This makes for a relatively easy to overlap algorithm.

Chapter III

Implementation

III.A Overview

We propose using a dedicated "Proxy" thread running as a dedicated thread running on a dedicated CPU of an SMP node. This proxy then manages communication for that entire node. We hope that by using a hybrid communication model incorporating both shared memory threads and message passing we can enable better performance, and achieve easier overlap.

III.B What does the proxy provide?

III.B.1 Network device sharing and protection

As mentioned in the introduction, clusters of multi-processor nodes typically only have one network interface device per node. Uni-processor machines have exclusive access to the network device allowing for unlimited use. However, this is wasteful in that the network device isn't always in use. There are inevitably periods of non-use within many parallel algorithms.

In an SMP node, we can fill in the periods of non-use by enabling the other processors to share the network device making expensive communications

interfaces more cost-effective by bringing up their utilisation percentage. The extra threads create more communication, which can then be multiplexed by the proxy thread onto the network device.

However, if all processors share the network interface, then contention for the single hardware device rises. While some implementations of MPI are thread-safe, it is not guaranteed that all are. Also, even if they are thread-safe, there is a context-switching overhead incurred when the scheduler must manage permission control of the network interface between multiple processors or threads that request access to it. The system must manage access to the network device with some sort of mutual exclusion. With the proxy however, all communication is managed by the single proxy thread. This frees up the other threads from having to need to switch access to the network interface, allowing one thread to retain exclusive access and control of the interface. This eliminates any protection or locking issues, as well as eliminating the context-switch overhead mentioned above.

III.B.2 A dedicated communications manager

A proxy achieves better performance by allowing overlap, as mentioned. It allows a separate proxy thread to handle communication so that computational clients running in other threads don't waste cycles on communications routines, and can instead devote their processing power to computational cycles. Regular asynchronous routines provide ways of hiding the latency of communications. However, protocol overheads can not be amortized away with these asynchronous routines. A proxy allows for reduced latency, as well as eliminating the protocol overheads thus resulting in increasing effective bandwidth. Falsafi and Wood show [10] that a proxy running on a dedicated CPU can give a benefit for light-weight communications protocols in applications that exhibit high amounts of communication, i.e.: where communication is a bottle-neck.

By eliminating communication portions of the algorithm, and allowing other threads to focus solely on computation, the dedicated proxy thread helps

eliminate context overheads incurred between switching from computation to communication management. Because all the management is done by a dedicated thread on its exclusive processor, the other threads running on the remaining processors achieve better cache locality by constantly staying within their computational phases without having to worry about communications management.

An analogy can be made between a group of n engineers. While having all n engineers do development work is one solution; in practice, it is almost always better to have one manager to manage the team of engineers taking care of busy work such as budgeting, scheduling, etc. Essentially, taking care of the *busy-work* freeing up the other team members to focus on the engineering development necessary to complete the project. If all n engineers have to deal with their own paperwork involving balancing budgets, or scheduling meetings, then they are less effective as compared to having the one manager to manage them, even with the loss of the manager as a potential engineer.

III.C Models of communication

There are four permutations of communication models an algorithm can be classified into. The issue of single-tier vs. multi-tier deals with how closely the algorithm is aware of the architecture of the system. In single-tier formulations, the algorithm sees the system merely as a one-level forest of nodes, thus flattening out any sense of a hierarchy (if one exists). Multi-tier formulations are more aware of the system architecture, and thus can take advantage of system enhancements and locality issues that the single-tier formulation may not. The issue of making an algorithm single-tier or multi-tier aware is a balance between portability and performance. Single-tier algorithms are more portable since any architecture can be flattened to a single-tier representation. While some performance may be lost in the discarding of system-awareness, it can often be compensated for by hardware or system-level enhancements (such as making the under-lying communication li-

brary multi-tier aware, yet still retaining a single-tiered API). However, multi-tier algorithms which can map themselves to the system architecture better can have a better idea of how to partition data, and optimize the distribution of data in order to minimize communication. Since this is done at the algorithm level, any system-level enhancements done at the library level can not help out as much.

Synchronous versus asynchronous communication deals with how the communication between nodes is negotiated. In a synchronous communication model, the receiver must be aware and ready to receive a communication before the sender can complete and return control back to the caller (as mentioned in the introduction). In an asynchronous communication model, the sender can initiate a sending communication before the receiver is even aware that it will be receiving data. The control returns back to the sender who can go ahead and complete other tasks. When it wants to know when the communication has been completed, it will *wait* on the communication - thus making it synchronous in that the wait will not return until the communication has completed.

Overlap is achieved by using either asynchronous communication in a single-tier model or multi-tier model. Overlap is also achievable in the multi-tier model by using a dedicated proxy thread using either synchronous or asynchronous communication.

Our data in this thesis compares three models: single-tier synchronous, single-tier asynchronous, and multi-tier using a dedicated thread.

III.C.1 Message passing

Message passing toolkits such as MPI provide an explicit, clear-cut means of doing both inter and intra node communication. On the MPI implementations running on LeMieux and Blue Horizon, each process is spawned as its own heavy-weight process, rather than as a thread. This models the node environment more as a flat single-tier system, as it abstracts away the SMP layer, treating every process (and thus CPU) as members of a flat system of nodes. On an improperly

designed implementation of a message passing layer, it's possible that this may result in excessive network usage due to the hiding of the SMP hierarchy. On both LeMieux and Blue Horizon, the MPI runtime is designed to short-circuit the network for intra-node messages, i.e.: any messages going from one MPI process to another MPI process residing on the same node will shortcut the network interface and stay within the node. However, this still results in a deep memory copy from the address space of the first MPI process to the second. On Blue Horizon, the point to point transmission time of two processors on-node is 12μs, versus 140μs for an off-node message sent using MPI synchronous primitives. On LeMieux, the point to point transmission time of two on-node processors is 12μs, with an essentially identical time of 13μs for off-node messages. These times are for point to point messages consisting of one double. For longer messages of 4096 doubles, LeMieux takes approximately twice as long (79μs on-node, versus 154μs off-node).

Message passing toolkits' main advantages are in its more explicit control flow, as compared to shared memory toolkits thus leading to easier development and subsequent debugging. While shared memory systems are convenient for intra-node communication, distributed shared memory (DSM) systems abstract away the distribution of memory across remote nodes, resulting typically in worse performance. Higher performance is possible, if "hints" are provided, or if the shared data is laid out in a specific manner in which to minimise coherence transactions. However, to achieve the best performance, it is often necessary to provide as much control information as one would need to program in a message-passing environment [18]

III.C.2 Shared memory

Other parallel processing toolkits such as OpenMP or POSIX threads (pthreads) support a shared memory method of sharing information between processes. Shared memory provides an easier interface than the message-passing paradigm, as it is simply an extension of the serial memory programming interfaces

with which the programmer is already familiar with. However, shared memory implementations to allow off-node access are often complex or complicated, and have longer latencies or higher overheads than the more simple message passing toolkits - hence the popularity of MPI vs. threads. Shared memory libraries are often faster for intra-node communication however, as shown by the thread packages' behaviour on both LeMieux and Blue Horizon. Threads are spawned as light-weight processes, or threads. This enables them to have shared pages of memory mapped between threads.

As mentioned above, however, using distributed shared memory for remote node memory access often results in poor performance. While semantics make for more convenient usage, the costs are often hidden in the ease of use. Abstracting away the "remote-ness" of data merely allows the programmer to conveniently ignore where that data resides, resulting in mediocre or sub-par performance as compared to message passing environments which force the programmer to address where the data actually resides.

Typically, parallel algorithms are implemented using either message passing (on multi-computers, or clusters composed of a network of single-processor nodes), or shared memory (on multi-processors, or a single computer composed of multiple processors).

The shared memory paradigm does not map easily to a distributed network of multi-processors. The overhead of attempting to maintain shared memory semantics overlaid on top of (typically) a message passing communications layer between nodes causes significant overheads, despite the benefits of maintaining an easy to use programming paradigm and syntax [18]. However, within threads running within the same node on a multi-processor machine such as the SMP nodes utilized in this work, shared memory often provides better performance and lower overheads than message passing toolkits. Sharing pages of memory between light-weight threads, and allowing the threads to maintain their own protection model eliminates many overheads as compared with more heavy-weight message passing

processes. Because the threads share the same address space, context-switching overheads between threads are significantly reduced as compared to processes as well.

Often threads are able to give benefits when parallel code done solely with MPI suffers from load balancing, or memory limitations [21] [8]. Load balancing can be hard to setup and manage in a system of processes communicating with messages. For example, a master-worker job queue can be hard to manage with solely message passing. Workers may take too larger a job slice, leading to load imbalance. On the other hand, taking too little work leads to frequent and excessive communication. Shared memory allows for finer grain load balancing as workers can take work as they need it from a shared data area without explicit messages. Memory limitations can often be involved when spawning multiple heavy-weight processes on one node as each process must take its own stack and incur its own overhead. Lightweight threads working with a shared data area often enjoy lower overheads leading to lower memory requirements.

III.C.3 POSIX threads vs. OpenMP threads

In our findings, we compared using POSIX threads (pthreads) versus OpenMP threads. The main differences lie in the scheduling and placement of the respective threads on dedicated or shared CPUs. Placement of both OpenMP and POSIX threads is controlled by the operating system, with a job queue associated with each of the CPUs within a node. The management of the queues is controlled at the operating system level. Working with the Pittsburgh Supercomputing Centre support staff, it was determined that differences in placement over time in response to system load varied between pthreads and OpenMP threads.

OpenMP allocates a fixed number of threads as determined by the `OMP_NUM_THREADS` environment variable upon initialisation of the `omp parallel` pragma directive. Typically, if the number of threads matches the number of CPUs on a node (in the case of LeMieux, this is four), then the threads are usually

scheduled onto separate and dedicated CPUs, regardless of the potential workload.

The POSIX threads package however will not move a newly instantiated thread to its own processor unless the current processor has a high enough workload to mandate migrating a thread to a new CPU. If the total aggregate workload is not enough to warrant another processor, then the pthreads package will run the multiple threads on a smaller subset of physical processors forcing two or more threads to share a processor. The idea behind this is to improve cache accesses/misses for multiple threads that may be sharing memory.

While initial behaviour of both packages is different (i.e.: initial placement of a newly created thread), the prolonged behaviour of both thread packages resulted in similar scheduling policies. Both the OpenMP and pthreads packages tend to bind threads to a processor where they remain in place as long as workload is consistent. The only exception is if the workload is trivial, in which case OpenMP tends to keep the threads running on separate CPUs, whereas the POSIX threads will coalesce onto a smaller subset of CPUs.

In addition, there is the issue of portability. OpenMP is a thread library that has been standardized on by HPC (high-performance computing) vendors, with open support by the likes of HP/Compaq, IBM, etc. There are many thread libraries out there, such as pthreads, but OpenMP is the supported and accepted standard. Even if it is available, the pthreads package does not often exhibit standard behaviour. The POSIX standard provides a standardized interface, but does not dictate portable behaviour such as scheduling policies.

The choice was made to use the OpenMP threads package as it provided more explicit dedicated CPU scheduling during the initialisation phase. Since we know that our computational phases are designed to maximize CPU utilization, we know that our threads will need dedicated processors. The POSIX threads will migrate the threads to their own dedicated processor eventually, however, OpenMP saves us the cost of migrating the thread from one processor to another.

III.C.4 Utilizing a hybrid model

While the MPI routines on both LeMieux and Blue Horizon use shared memory to by-pass and short circuit going out to the network, we believe that a hybrid approach of both message passing (using MPI) and shared memory (using OpenMP threads) will provide better performance.

The message passing paradigm maps well to heavy-weight processes; and one can indeed run multiple processes on an SMP machine, mapping one processor per processor. However, using a message passing library to go out onto the network to communication from one processor to another processor residing on that same node is extremely wasteful. MPI implementations on both LeMieux and Blue Horizon short circuit the network by using shared memory to pass data between two communicating processors within the same node.

Other work on SMP machines has often used heavy-weight processes instead of the more light-weight threads we use in this work [17] [24]. While creating one process per physical processor does provide more traditional message-passing semantics as well as ease of porting, we feel that the benefits of utilizing a threaded model outweigh the benefits of using processes. A threaded model provides reduced switching/scheduling and copying overheads[18], allowing for faster context-switches. Utilizing a threaded model also allows for the usage of shared memory - a feature critical to efficient queue management, as we use shared memory to achieve fast intra-node communication.

However, the explicit control and semantics of messages allow for more optimal and cost-effective inter-node communications. A hybrid approach of using OpenMP threads to manage on-node communication using shared memory data structures, and MPI message passing routines to manage inter-node data transfer will bring the benefits of both paradigms, utilizing the benefits of each individual communication model to help offset the cons of the other.

Essentially, going to the hybrid multi-tier model creates another layer in the communication control levels. Previously with message-passing or shared

memory, there was just the *Collective* and *Processor* levels of communication control. With a hybrid multi-tier model, we now have a new middle-level, *Node* level control.

In addition, by consolidating the communications to one proxy thread we decrease the amount of processes communicating on the network interface for the node. It has been shown that when fewer processes share an interface, the contention decreases allowing for lower latency and higher bandwidth to the interface. [9]. In addition, by consolidating messages, we eliminate any excess costs incurred by message start-up times.

III.C.5 Scheduling the proxy thread

One issue with scheduling a proxy communicator thread is whether or not to dedicate a CPU in the SMP node to run the proxy thread, or whether the thread should share that CPU with a computational client. We adopt the terms *dedicated* to indicate the proxy thread running on its own dedicated CPU, and *shared* to indicate that the proxy shares a CPU with another computational client.

In the dedicated scenario, a CPU which can be used to run computational threads is taken off to handle the proxy thread. This obviously reduces the peak performance capability in terms of raw computational power of the application as a whole. However, this added cost must be weighed against the potential benefits of off-loading the communication overhead out of the remaining computational threads onto the proxy thread. A dedicated scheduling scheme seems to be more applicable for applications that are very communication-intensive and exhibit another communication load to make the loss of computational power worth the cost.

In the shared scenario, the hope is that communication isn't enough to load the entire CPU that the proxy thread is running on, and that by sharing that CPU among both the proxy thread and another computational client, we can achieve the best of both worlds. Practice seems to indicate that this isn't optimal,

as it is only practical if the communication load isn't truly intensive enough to warrant using a proxy thread in the first place. The potential thrashing and cache-invalidation due to context-switching of the threads reduces the performance so that the performance is not worth running a proxy thread at all.

III.D Programming with the Proxy

Any programmer familiar with OpenMP will be familiar with how to spawn OpenMP threads. To use a proxy communicator, the developer must first isolate the communications routines and rewrite the algorithm to separate communication-dependent data from communication-independent data.

The basic structure of an algorithm written to utilize a proxy thread looks like the pseudo-code shown in Figure III.2.

A library was developed which followed a similar model, by running a thread full-time that would accept MsgSend/MsgRecv primitives in an attempt to utilize a proxy while maintaining MPI/message-passing semantics with minimal change, but performance was not optimal. A proxy requires that communication and computation patterns be well known and partitioned in advance, rather than communicating messages as they occur. To this end, a good interface that might be useful in the future would be to maintain a data structure separate from the main computational structure that describes the work-set data. I.e.: it would give indexes and ranges partitioning a dataset seen in one of the nodes in Figure II.1 and describing it so it looks like the dataset seen in Figure II.2 from Chapter II. A suitable interface that we would build given time would look like the code in Figure III.2.

Indeed, this interface is similar to the `MotionPlan` and `FloorPlan` structures which are a part of the KeLP library [11] [12].

```
main()
{
   for each iteration {
#pragma omp parallel
      if (thread_id = 0) {
            asynchronous send ghost cell region data
            asynchronous recv ghost cell region data
            wait for asynchronous sends to complete
            trigger computational thread
      } else {
            do communication-independent computation
            wait for trigger from proxy (thread_id 0)
            complete computation
      }
   }
}
```

Figure III.1: Pseudo-code for utilizing a Proxy communicator thread

```
main()
{
    ProxyThread proxy
    DataStructure data
    DataCommunicationPlan plan

    describe/partition data into plan
    for each iteration {
        proxy.StartIteration()
        do communication-independent computation
        proxy.WaitForCompletion()
        complete computation
    }
}
```

Figure III.2: Pseudo-code for a future Proxy library interface

Chapter IV

Results

IV.A Testbed & Architectural Details

The proxy was implemented on two systems. The first, LeMieux, is a 750 node cluster located at the Pittsburgh Supercomputer Centre. Each node is an HP/Compaq AlphaServer ES45 system composed of a 4-way SMP (symmetric multi-processor) configuration of four 1GHz Alpha 21264C EV68 processors. Each processor has a peak floating-point capability of 2 gigaflops. Each processor has separate 64 Kbyte write-back L1 data and instruction caches, with 2-way set associativity and a cache line size of 64 bytes. The L2 cache is a 8 megabyte dual data rate cache, enabling it to perform two data fetch operations per clock cycle. It is a unified data and instruction write-back cache, also with 2-way set associativity. [1] The nodes are connected with a Quadrics QSNet interconnect, a high performance network interconnect designed for low-latency, high-bandwidth communication with an advertised peak bandwidth of 340 MB/sec, and a process to process latency of 2 μs (5 μs for MPI messages). Each node has 4 Gigabytes of shared main memory, with a memory cross-bar switch capable of switching 8 gigabytes/second. [2]

[1] http://www.psc.edu/machines/tcs/lemieux.html#optimization

[2] http://h18002.www1.hp.com/alphaserver/es45/

The nodes run Compaq Tru64 Unix v5.1A, and provide parallel programming toolkits in the form of MPI, OpenMP, and Shmem. The Fortran compiler used was the `f90` Fortran90 compiler. LeMieux provides both GNU and HP compilers. The code for this thesis was compiled using the HP `cxx` C++ compiler. The Fortran code was compiled with the following options: `-fno-second-underscore -ff90 -fugly-complex`, while the C++ code was compiled with the following options: `-O3`

The second system used was Blue Horizon, a 144 node system located at the San Diego Supercomputer Centre. Each node is an 8-way IBM Power3 system consisting of eight 375MHz Power-3 processors. Each node has 4 Gigabytes of shared main memory, with each processor having 1.5 GB/sec bandwidth to memory. Each processor has an 8MB 4-way set associative L2 cache, and a 64KB 128-way set associative L1 cache. Both the L2 and L1 cache have a cache line size of 128 bytes. The nodes are interconnected with IBM's fast proprietary Colony switch which has a measured MPI bandwidth of 350MB/sec, and a latency of 17μs

The nodes run AIX 4.3 with the IBM MPI and OpenMP implementations. Native IBM compilers are used, with xlC_r and xlf_r used for C++ and Fortran code respectively. The compilers are wrapped in the MPI mpCC, and mpF compiler wrappers. The C++ code was compiled with the following options: `-O3 -qstrict -qarch=auto -qtune=auto -qsmp=omp:noauto` while the Fortran code was compiled with the `-O3 -qstrict -qarch=auto -qtune=auto -qsmp=omp:noauto -u -qnosave -q64` flags.

IV.B Performance Side Effects

We believe the hybrid model will provide the benefits of both message passing and shared memory with hopefully none of the deficiencies. However, there are some side effects that will be caused by using a proxy thread. Because one of the processors has been taken out of the compute pool to dedicate toward running

communication, this necessarily causes the computation time on the remaining computational threads to increase (i.e.: the other three CPUs on LeMieux, and the other seven CPUs on Blue Horizon).

We can model an expected level of performance as follows:

Let T_s = total running time in the synchronous model

$C_t < T$ = time spent communicating

p = number of CPUs per node

s = number of CPUs dedicated to running a proxy per node

Then the theoretical time on the proxy model T_p is defined as:

$$T_p = max((T_s - C_t) * \frac{p}{p-s}, C_t)^3$$

E.g.: we are scaling the computational time by the ratio given that we are losing s processors to run proxy communicators. Assuming one would utilize all the p CPUs on a node we can expect to lose $\frac{s}{p}$ computational performance. For example, with Blue Horizon, each node has eight processors. Assuming each CPU is utilized to its full extent (100%), we expect to see a $\frac{1}{8} = 12.5\%$ rise in computational time for the seven other computational threads when used with a proxy thread running dedicated on the eighth processor. Similarly, with LeMieux, we expect to see a $\frac{1}{4} = 25\%$ rise in computational time on the remaining three computational threads. This performance model was originally derived by Fink [11] and Baden [3].

In addition, since all the CPUs within a node share the same main memory, problems with shared memory contention arises. As the node grows wider (e.g.: more processors sharing the same bus) this shared memory contention cost grows geometrically. In machines with two or four CPUs per node (such as LeMieux),

[3]It is the maximum of the two expressions since there are cases in communication-intensive algorithms where the communication time spent by the proxy thread dwarfs the computational time spent by the other threads, thus making the overall time the same as the proxy communication time.

this is generally not too much of a problem. As the configurations grow wider though, this can become detrimental - especially on machines such as Blue Horizon which is 8-way, or another NERSC machine Seaborg which is 16-way. This is often alleviated by having multiple ports to memory, allowing CPUs to simultaneously access memory without being bottlenecked. On Blue Horizon, each group of four CPUs within a node share the same memory port. For our case, since we are running one proxy thread which deals mostly with communication and doesn't have the same memory bandwidth requirements as the computational threads, this presents a load imbalance problem in that three computational threads would share one port, while the remaining port would be shared among the other four computational threads.

Communication-wise, though we noted in Section III.C.4 that aggregating communications onto one thread (the case where s=1 in the performance model presented) can result in lower latency with higher bandwidth it must also be noted that in problems where the chief constraint is bandwidth (as opposed to CPU, or memory bounds), often the aggregate bandwidth is lower than when using multiple processes to communicate [9].

One other side effect must be noted in even using simple overlap using MPI asynchronous calls. It is common for asynchronous communications to cause an increase in the computational time, even without the proxy. This can be attributed to a few components. The asynchronous transfer of data could cause network interrupts, causing the system to context-switch and possibly use time to manage the transfer of communication or lose cache locality in the servicing of the communication. Additionally, the network interface must get the data back into memory which places an additional burden on the memory system decreasing the available memory bandwidth for the computational threads.

We also noted that IBM's MPI implementation on Blue Horizon seems to be unable to overlap communication using MPI's simple non-blocking asynchronous calls. Indeed, it actually causes the communication time to *increase*.

Model	Total (s)				Per Iteration (ms)		
	Total	Comm.	Wait	Ideal Proxy	Total	Relax	Comm.
Sync	2.95	1.07	0.88	2.36	178	133	70
Async	3.60	1.72	1.63	2.25	225	133	108

Table IV.1: RedBlack3D on Blue Horizon, Average Times for 4 nodes @ 16 iterations, N=400) showing IBM's MPI implementation's lack of ability to realise overlap when using immediate mode asynchronous communication

IV.B.1 Optimized Hardware

Clusters can have a varying level of interconnects, ranging from regular Ethernet up to highly specialized networks such as IBM's Colony, or the Quadrics QsNet interconnect (used on Blue Horizon and LeMieux respectively). Both are designed for high bandwidth, and low latency communications. In the case of the Quadrics interconnect, the network interface *Elan* is as intelligent as some nodes in previous supercomputers! Its core consists of a flexible and powerful dedicated co-processor which can be programmed as the user wishes. As such, Compaq's MPI implementation on LeMieux is highly optimized to exploit the power of *Elan*. This extra co-processors is powerful enough to be thought of as a communications proxy in its own right employing its own cache, CPU, and memory. This may have a side effect in that optimizing the algorithm for use of a software proxy may attempt to circumvent the very usefulness of having the *Elan* interface, and may result in adverse effects as the *Elan* co-processor, having been programmed by the MPI library to respond favourably to a single-tier communication model, attempts to handle the abnormal communications pattern of our multi-tier asynchronous proxy communication model.

IV.C Redblack3D

IV.C.1 Blue Horizon

We first ran some small benchmarks on problems to evaluate the ability to achieve overlap using base MPI immediate mode asynchronous calls. From the results shown in Table IV.1, we observed that the MPI implementation on Blue Horizon was unable to realize overlap using the asynchronous communication primitives. Because of this, comparisons are made chiefly between the synchronous single-tier and proxy multi-tier models. Before we ran experiments, it was helpful to see how the RedBlack3D kernel would scale thread-wise. These results would then give us a ballpark figure as to what sub-problem size per node would be most optimal. This enables us to see how well the RedBlack3D kernel parallelises across multiple threads accessing the memory sub-system, as well as what overheads (if any) the thread layer would create. To examine how the RedBlack3D computational kernel scaled, we ran the kernel itself on one node only with no communication. We ran it varying from 1 to 8 threads, with the results shown in Table IV.2. These results show that at a smaller problem size per node (240^3), the problem scales poorly due to the size of the problem. It's just too small to effectively parallelize across 8 processors. At 360^3 we see the problem start to scale more effectively. At 4 threads, the running time has decreased from 30.658s to 7.888s, a speedup of 3.886 - very close to the linear speedup value of 4 we would hope to achieve. At 7 threads, we attain a speedup of 6.316. This is the value we are most interested in, since we will be taking off 1 thread to run the communication proxy. At 480^3 we see that the problem starts to scale much less. At 7 threads, we have attained a speedup of 4.935 - hardly the linear speedup value of 7 we would hope to achieve. The graph in Figure IV.1 shows the speedup values plotted in Table IV.2. Since this is run only on one node, eliminating off-node communication, we can observe that the lack of speedup at 7 threads on the larger sized problem is contributed to mostly by the overhead of the thread system.

Problem Size	Number of OpenMP Threads							
	1	2	3	4	5	6	7	8
240	4.94	2.67	2.10	1.94	1.56	1.32	1.21	1.04
360	30.66	15.25	10.27	7.89	6.35	5.33	4.85	4.14
480	44.20	21.51	16.63	15.35	12.25	10.29	8.96	8.01

Table IV.2: Running time in seconds, showing the effect of varying threads on the RedBlack3D kernel to observe scaling of the memory system and thread system overhead on Blue Horizon

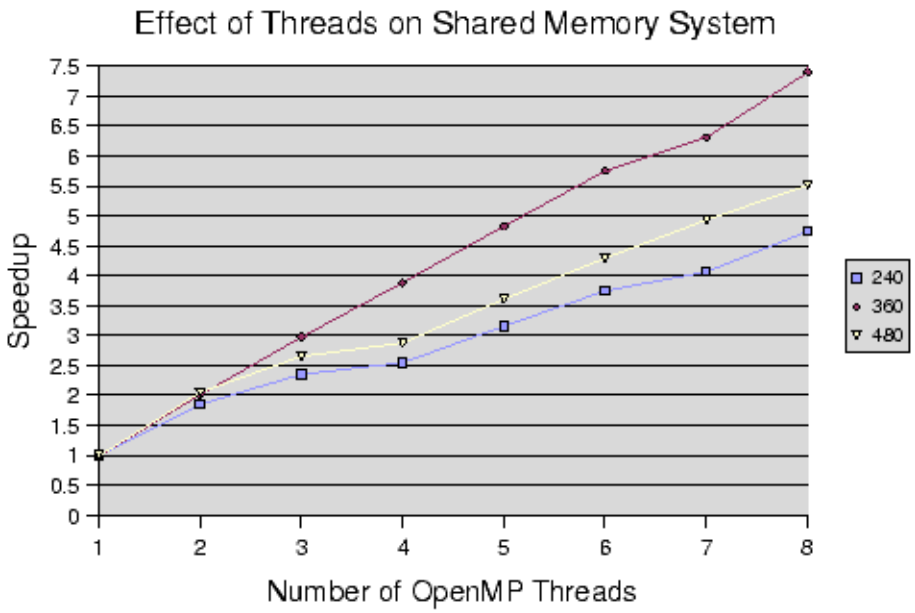

Figure IV.1: Speedup on the RedBlack3D kernel for Blue Horizon

Problem Size	Threaded	MPI HWP	
	Total	Total	Comm
120	0.10	0.08	0.03
240	1.06	1.11	0.18
320	2.51	2.62	0.38
360	4.81	3.93	0.87
400	4.87	5.11	0.72
480	8.36	9.10	0.95
540	11.94	n/a	n/a
600	16.33	n/a	n/a

Table IV.3: RedBlack3D on Blue Horizon, MPI HWP vs. OpenMP LWP (communication on the proxy is always zero for 1 node

The plot in Figure IV.1 shows that to obtain close to linear speedup, the problem size per node must be great enough that there is enough work to parallelize, but not too great as to cause saturation of the bandwidth to the shared main memory system. At our problem size of N=800^3 running over eight nodes, this equates to a per-node sub-problem size of N=400^3, which obtains reasonably close to linear speedup.

We also experimented with comparing the performance of running eight MPI heavy-weight processes per node (1 per CPU) communicating using the MPI message-passing via shared memory implementation versus running one MPI process per node spawning eight light-weight OpenMP threads. Essentially, we are benchmarking message passing at a single-tier level versus the threaded model, with the intent of measuring thread system overheads. This also gives an idea of how the processor sub-problem strides and geometry affect the performance.

This benchmark shows more clearly, and justifies the use of OpenMP to manage intra-node communication, by eliminating the off-node communication. From this data shown in Table IV.3, we can see that the proxy outperforms the synchronous model slightly at $N = 240^3$, and more so at $N = 480^3$. At $N = 480^3$ the communication cost for the synchronous model has started to take up an reasonable fraction of the running time at almost $\frac{1}{9}$ of the total time - a cost

the proxy model does not have to bear. However, at 360^3, the single-tier solution gives an 18% performance increase over the multi-tier, despite being slower at both the smaller and larger problem sizes of 240^3 and 480^3. We believe this is due to what is essentially bad luck in hitting a poor stride for the sub-problem. With the multi-tier model, the threads operate on a single large matrix which creates larger memory strides in accessing subsequent rows. This can cause some TLB conflicts between the multi-processors as they try to access the shared memory. At the single-tier model, since each process has its own isolated sub-problem, the per-process sub-problem is $\frac{1}{8}^{th}$ the size of the multi-tier model's single large matrix. This allows for each processor on the node to have smaller memory strides which are more optimal.

We noticed an interesting problem at $N = 600^3$ where the process crashed when trying to allocate enough memory for 8 subproblems of $\frac{600^3}{8}$, but it succeeded when creating a single 600^3 matrix and sharing over 8 lightweight OpenMP threads. The node has 4GB of memory, and a problem of 600^3 (assuming 8 byte doubles) will come very close to filling that since the node has to allocate both the main matrix and the right hand side matrix. The lower overhead of creating LWP threads seems to enable a larger working-space versus the heavier MPI processes, though this is a small example.

Table IV.4 summarises results typical of a run of RedBlack3D for 8 iterations, on 8 nodes (64 processors), with a problem size of 800^3 using double-precision floating point; the table shows the total time spent communicating, the total time of the run, the percentage of time spent communicating as well as the total time spent waiting for communication to finish. For the synchronous and asynchronous versions, the percentage of time spent communicating is the total time for overhead of communications and the time to wait for that communication to finish (latency). On the proxy, since the proxy thread initiates communication on behalf of the node, the overhead experienced for communication primitives on the other threads is zero, and the communication time consists solely of the time spent

Model	Total (s)				Per Iteration (ms)		
	Total	Comm.	Wait	Ideal Proxy	Total	Relax	Comm.
Sync	11.81	3.58	2.61	10.51	738	557	224
Proxy	10.20	0.53	0.53		637	606	33

Table IV.4: RedBlack3D on Blue Horizon - Average Times, 8 nodes @ 16 iterations, N=800

in the computational thread waiting for communication to finish. There is still overhead involved in terms of thread synchronization. For the synchronous and asynchronous runs, the performance model described in the previous section was applied to determine an ideal proxy performance time. Table IV.4 also shows the breakdowns of iterations for each model, showing the time spent in computation vs. communication.

The jobs were run with sufficient iterations to ensure any "warm-up" effects were amortized out, and to make sure runs lasted at least a consistent amount of a few seconds to ten seconds. Times were reported as the average of 16 runs, with outliers discarded. We defined outliers as runs where the total time was at least 20% slower than the average of the other runs. Wall clock times were observed with `MPI_Wtime()`.

For the 800^3 case, we observed that the relaxation time (computational phase of the RedBlack3D algorithm) increased by about 8.8% when going from the synchronous algorithm to utilizing the proxy. Since we took off one CPU to run the proxy, we predict that the system loses about 12.5% of its computational power, so our actual observed value of 8.8% is close to this predicted value. The communication time decreased considerably by about 85%, bringing the total (per iteration) time down by about 13.7%. It is interesting to note the relaxation time didn't go up as much as we predicted, this suggests that the problem is memory-bandwidth constrained, rather than CPU constrained.

Given the performance of the synchronous case, the ideal proxy performance (modelled in the performance model discussed above) should have been

Model	Total (s)				Per Iteration (ms)		
	Total	Comm.	Wait	Ideal Proxy	Total	Relax	Comm.
Sync	12.41	4.40	3.43	10.26	775	558	274
Proxy	12.89	1.21	1.21		806	740	76
Multi-tier	14.30	3.20	3.20		894	704	201

Table IV.5: RedBlack3D on Blue Horizon, Average Times, 16 nodes @ 16 iterations, N=1008

10.5s. Our observed running time was 10.2s, thus showing that the shared memory model, in addition to eliminating part of the communication costs, helped speedup other parts of the algorithm, most likely related to the cost savings when using shared memory access. We eliminated 1.51s off the synchronous time of 11.8s, which can be attributed to the difference between eliminating the communication offset by the increased relaxation computational time for an overall speedup of about 13.6%.

We next attempted to scale the workload up to 16 nodes, by maintaining roughly the same sub-problem size per node of 400^3 doubles. This makes for an aggregate problem size of N=1008^3. However, at this problem size we see the proxy performance being unable to match the performance of the baseline synchronous model. Our results which can be seen in Table IV.5 show that while the relaxation time per iteration for the synchronous model (running at a 4x4x8 geometry) remains constant at around 560ms per iteration, the relaxation time for the proxy model jumps from 600ms to 740ms. It was surprising to see the relaxation time jump up this much; upon taking a closer look at the runs, it seems part of the problem can be laid down to cache locality. The 800^3 problem allowed us to run at a geometry of 1x2x8 (in the $X\ x\ Y\ x\ Z$ dimension order), which breaks the problem up into 3D sub-matrices of 800x400x200. At the 1008^3 problem, we tried a few different geometries with the best results (Table IV.5 being in the 1x4x4 geometry which gives us a sub-matrix size of 1008x252x252. The matrix is laid out such that x is the nearest dimension, with the y-dimension being off by a stride

of x, and the z-dimension being off by a stride of $x * y$. Therefore, the best cache locality comes from having a longer x-partition, hence the need to partition with 1 processor along the x-axis. This leaves a $Y\ x\ Z$ geometry of 1x16, 2x8, 4x4, 8x2, or 16x1. Since the node divides the sub-problem internally along the y-axis, it would be more beneficial to avoid over-partitioning of the y-dimension, which leaves either 1x16, 2x8, or 4x4 as the optimal $Y\ x\ Z$ geometry. The 1x16 and 2x8 partitions leave a Z-dimension of size either 63 or 126, which is not enough to benefit from cache pre-fetching. This leaves 1x4x4, which while it gets the best performance of the proxy geometries, is unable to match the synchronous performance due to the smaller sizes of the y-dimension partitioning. At 1x4x4, each node gets a 252-row block in the y-dimension, which means each processor works on roughly a $\frac{252}{7} = 36$ row sub-block which does not enjoy the same cache benefits as the smaller problem which has a y-dimension block of $\frac{400}{7} = 57$ which allows for more local computation to be performed.

Our results obtained were similar to the ones obtained by Baden and Shalit on the same machine [5]. With 8 nodes at a problem size of 800, they achieved total iteration times of 626ms when using the multi-tier variant with overlap, proxy, and irregular partitioning while ours achieved 637s. Their single-tier synchronous version also achieved comparable performance at 732ms while ours achieved 738ms.

We attempted a run at 24 nodes, but saw continued performance degradation along the same vein as in the 16 node case. Our 24 node runs completed in 13.69s on the synchronous single-tier implementation, whereas the proxy finished in a slightly slower 15.13s. This was still however, faster than the asynchronous MPI implementation.

We implemented a multi-tier model without using overlap and without using a proxy thread (similar to Baden and Shalit's *MT(k)+!OVLP* model) to ensure that the proxy was not causing unnecessarily high overheads. Our relaxation times compared similarly (shown in the last row of Table IV.5, with the multi-tier

w/o overlap or proxy model getting relaxation times of about 700ms. The proxy's relaxation time is slightly higher due to the loss of a computational thread. This result seems to indicate that the high spike increase in relaxation time is not due to the proxy thread or overlap. Instead, it seems to be due to some sort of overhead penalty incurred by the shared memory OpenMP thread system. Part of this may be due to the memory stride of the sub-problem in going to a multi-tier aware model. The single-tier model splits up the sub-problems to a smaller size allowing for smaller strides thus increasing the performance and locality (by reducing the number of misses) of the cache as well as the TLB.

It seems that the proxy is hard pressed to get performance out of more than the eight nodes we ran on. At larger problems we run into cache problems due to non-optimal geometry partitioning, as well as increasing the impact of losing computational threads. At smaller problems, we run into the problem where there is not enough computation to amortize the communication costs away. There is a balance that must be struck between smaller problems which too little computation and larger problems with too much computation.

IV.C.2 LeMieux

The MPI implementation on LeMieux is supposedly capable of realizing overlap in the baseline asynchronous MPI model, so we were able to benchmark some more interesting results. Due to there being four processors per node, versus the eight in each Blue Horizon node, we expected the proxy to have a more significant impact in the loss of computation since there would now be a loss of 25% aggregate computational power versus the 12.5% cost for a Blue Horizon node.

We performed similar kernel benchmarks on LeMieux as we did on Blue Horizon in order to see how well the memory system would scale. We suspect similar results to Blue Horizon, as the node configurations are similar. While LeMieux has four processors per node versus Blue Horizon's eight, the port to memory is similarly half (LeMieux's processors share one port to memory, similar

Problem Size	Number of OpenMP Threads							
	1		2		3		4	
240	3.36	na	1.98	1.70x	1.57	2.14x	1.49	2.26x
360	12.92	na	7.42	1.74x	5.69	2.27x	5.00	2.58x
480	30.08	na	17.16	1.75x	13.40	2.25x	12.01	2.51x

Table IV.6: Running time (in seconds) and speedup showing the effect of varying threads on the RedBlack3D kernel to observe scaling of the memory system and thread system overhead on LeMieux

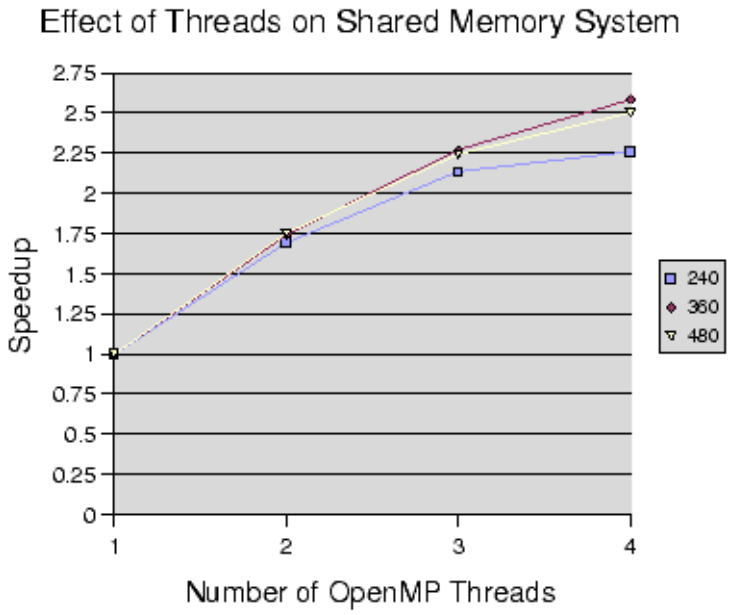

Figure IV.2: Speedup on the RedBlack3D kernel for LeMieux

Problem Size	Multi-tier		Single-tier		
	Total	Relax/iter	Total	Comm	Relax/iter
304	2.89	.36	3.29	0.48	.35
360	4.94	.62	5.54	0.89	.59
424	8.09	1.05	8.95	1.06	1.01
456	10.34	1.31	11.41	1.61	1.21
480	12.02	1.49	13.31	1.84	1.38

Table IV.7: RedBlack3D on LeMieux, MPI HWP vs. OpenMP LWP (communication on the proxy is always zero for 1 node, 8 iterations. 304^3 corresponds to 768^3 over 16 nodes, and 456^3 corresponds to 1152^3.

to Blue Horizon's configuration where each set of four has its own port to main memory). Since RedBlack3D is more constrained by memory, we expected the results to be similar given similar memory subsystem performance.

The data in Table IV.6 shows that the nodes on LeMieux seem to peak at around a 2.6 times speedup. Going to 4 threads on one node versus 1 thread only offers a 2.2 to 2.6 times speedup for any of the three sample problem sizes. At the proxy configuration (3 threads), the best speedup attained was at 360^3 where we attained a 2.27 times speedup going from 12.924s to 5.686s. Since off-node communication is eliminated in this benchmark, these results seem to verify that the problem seems to be more memory bound, rather than CPU bound thus constraining the speedup of having multiple threads processing the data.

We again ran the same MPI single-tier versus OpenMP multi-tier comparison on a single node in order to better understand the issues of how local problem stride and geometry might affect performance of the RedBlack3D kernel. These results can be seen in Table IV.7. We took our later problem sizes of 768^3 and 1152^3 run over 16 nodes, and scaled them down to roughly the single-node sub-problem sizes of 304^3 and 456^3. We also ran at a couple intermediate values (360^3 and 424^3) as well as a slightly larger problem size of 480^3. The table shows the total runtime for the multi-tier (threaded) model, and the relaxation time per iteration. It also shows the total runtime for the single-tier (MPI synchronous)

model along with the total time spent communicating, as well as the relaxation time per iteration. From these results, one can see that the computational time on the multi-tier threaded model is slightly higher than the single tier model. In the multi-tier model, the data set is allocated as a single shared large three-dimensional matrix, leading to longer strides than the smaller sub-problems when allocated per process in the single-tier model. The shorter strides prove beneficial to the single-tier solution in reducing the relaxation times. However, overall, the total run-times tend to be smaller for the multi-tier threaded model due to the elimination of intra-node communication.

Our results of running RedBlack3D in synchronous, asynchronous, and proxy modes can be seen in the following tables. Table IV.8 shows the run-times when a problem size of 800^3 is run on 8 nodes (4 processors each) for 16 iterations, the same configuration we ran on Blue Horizon. Table IV.9 shows the performance when run for 256 iterations on sixteen nodes at both 768^3 and 1152^3.

At 8 nodes at 800^3, we see a surprising result: asynchronous version running slower in both computation AND communication in comparison to the synchronous version. It appears that the cost of running both computation and communication simultaneously for affects the node's performance very negatively. This seems to be an indicator that overlap using regular asynchronous MPI communication primitives (Isend/Irecv) is not possible, an observation we noted with Blue Horizon. With the proxy model however, we were able to eliminate almost entirely the communication wait time, decreasing it from 17% to a negligible 0.2%. This increases the computation time per iteration by about 10%. The increased computation time directly leads to the fact that there is almost a zero wait time as the loss of 25% aggregate computational power causes the increased computation to completely overlap the communication. Overall though, a positive result is attained resulting in an overall speedup of 5.13%.

For the 16 node configuration, at the smaller problem size 768^3, we see that the proxy, while out-performing regular MPI synchronous code, does not out-

Model	Total (s)					Per Iteration (ms)		
	Total	Comm.	Comm. %	Wait	Ideal	Total	Relax	Comm.
Sync	16.56	2.81	16.9%	2.51	18.73	1034	885	178
Async	17.18	3.48	20.2%	3.38	18.40	1073	963	217
Proxy	15.71	0.04	0.3%	0.04		981	977	2

Table IV.8: RedBlack3D on LeMieux, Average Times, 8 nodes @ 16 iterations, N=800

Size	Model	Total (s)			Per Iteration (ms)		
		Total	Comm.	Comm. %	Total	Relax	Comm.
768^3	Sync	114.78	26.62	23.2%	448	365	104
	Async	110.96	9.80	8.8%	433	406	38
	Proxy	112.64	4.65	4.1%	440	421	18
1152^3	Sync	383.48	72.14	18.8%	1500	1250	282
	Async	390.96	59.43	15.2%	1530	1340	232
	Proxy	378.57	16.86	4.4%	1480	1430	65

Table IV.9: RedBlack3D on LeMieux, Average Times, 16 nodes @ 256 iterations, varying N

perform the baseline MPI asynchronous code. This result seems to indicate that overlap IS indeed possible, though only at a slight margin.

At the larger problem size of 1152^3, the proxy does start to out-perform, as it barely edges out the asynchronous model by 3.1%. What is interesting is that at that same problem size, the asynchronous model actually runs slower than the synchronous model. It seems at that point the cost of both managing overlap while simultaneously computing takes its toll on the computational threads too much causing the overall system to run slower than its base synchronous model.

Size	Model	Total (s)			Per Iteration (ms)		
		Total	Comm.	Comm. %	Total	Relax	Comm.
384^3	Sync	4.74	2.49	52.5%	19	10	0.97
	Async	4.55	2.28	50.1%	18	11	0.89
	Proxy	6.17	0.78	12.6%	24	23	0.3
768^3	Sync	59.83	17.02	28.5%	234	180	66
	Async	54.28	7.14	13.2%	212	192	28
	Proxy	57.03	3.25	5.7%	223	218	12
1152^3	Sync	197.75	49.07	24.8%	772	631	192
	Async	191.12	31.19	16.3%	747	651	122
	Proxy	190.58	10.19	5.3%	744	722	39
1536^3	Sync	475.63	88.37	18.6%	1860	1570	345
	Async	490.51	122.72	25.1%	1920	1540	479
	Proxy	454.01	23.75	5.2%	1770	1710	92

Table IV.10: RedBlack3D on LeMieux, Average Times, 32 nodes @ 256 iterations, varying N

At any problem size, however, it can certainly be seen that the proxy model saves communication costs, as it brings down the communication times considerably in each problem size. However, it appears that the cost of taking away 25% of the available computational power on these 4-way SMP nodes does not off-set the benefits of having a dedicated communications manager. The QSNet interconnect backend is more than capable of quickly delivering communications across the network - enough to warrant having all available processors on the node compute, rather than removing one to run a dedicated communications proxy.

Table IV.10 shows our results when run on a 32 node configuration. At 32 nodes, for the smaller problem sizes, we again see the proxy not being able to deliver faster performance. It significantly reduces the communication percentage, but it isn't until we get to a problem size of 1152^3 that we start to see the proxy out-perform the synchronous and asynchronous models. At 1152^3 the proxy just barely squeezes by at 3.67% faster than the synchronous model, and 0.3% faster than its asynchronous model. At a problem size of 1536^3, we see slightly better performance with the proxy performing 4.5% faster than the synchronous model, and 7.4% faster than the asynchronous model. The asynchronous model gives us an interesting result at that problem size. The communication time jumps up incredibly, making the total running time longer than the synchronous model. The results for the asynchronous model were consistently the best at this geometry configuration (4x8x4 in the X x Y x Z axis). At other geometry configurations, the problem ran in comparable communication times to the synchronous model, but the computation time increased dramatically. At this 4x8x4 configuration, the communication time spiked higher than any other geometry, but the overall running time was the best of all the asynchronous runs.

A similar trend was shared between both the 16 and 32 node cases where as the problem sizes got larger, the proxy implementation starts to perform better relative to the synchronous or asynchronous single-tier implementations. From examining the computation times per iteration on the 1152^3 and 1536^3 problem

sizes for the 32 node case we can see that the proxy's computational speed improves by a couple of percentage points going to the larger problem size. At the 1152^3 case, the difference between the asynchronous single-tier and proxy is 71ms, or 9.8% of the proxy time. At the 1536^3 case, the difference between the synchronous and proxy is 140ms, or 8.2% of the proxy time. This shows the proxy's computational compabilities are more efficient at the larger problem size, thus showing a speedup compared to the smaller problem sizes. Essentially, at the larger problem size, the proxy's impact on the remaining computational threads decreases.

Both the 16 and 32 node results seem to indicate that the proxy provides marginally better results at large enough problem sizes. Any larger than 1536^3 is both unreasonably large (for a finite difference method), as well as detrimental in terms of losing the proxy thread. Problem sizes larger than 1536^3 require enough computation that it isn't desirable to dedicate a proxy thread. At certain problem sizes and node configurations, the proxy thread can achieve overlap where the baseline MPI asynchronous code is unable to.

IV.C.3 Summary of RedBlack3D

Our analysis of RedBlack3D on both Blue Horizon and LeMieux is that while performance gains can be made, they are more often than not, small. With the exception of the 13.6% speedup we obtained at a problem size of 800^3 on 8 nodes on Blue Horizon, most of the gains made were small (less than 6%). For this kernel, as we moved to larger node configurations, the gains made became non-existant. At the smaller problems (both in node configuration and in problem size), communication plays a higher percentage of the total problem and justifies dedicating a thread to manage communication. Scaling up to larger problems on larger node configurations seem to incur overheads due to the shared memory OpenMP thread system. The threads allow for the creation of a larger shared sub-problem per node, but this large stride seems to cause a higher cost. In addition, it is possible that the OpenMP implementation may be adding extra overhead in

terms of thread and loop synchronization. In addition, at the larger configurations, the loss of computational power and the constraints of the geometry partitioning lead to less beneficial returns.

IV.D SUMMA

SUMMA presents an interesting problem in that it is relatively straightforward to overlap. Contrary to RedBlack3D which requires repartitioning each node's sub-problem into regions that do or do not require communication to have been completed for that iteration before proceeding with computation, SUMMA allows for easy overlap by allowing for a clean and intuitive separation of an iteration's communication from computation. The formulation of the algorithm allows for any given iteration i to start communicating the data needed for the $(i+1)^{th}$ iteration while working on the computational steps for calculating the result of the i^{th} iteration.

The SUMMA communications requirements are also interesting in that nodes communicate in broadcasts rather than staggered/offset point to point messages (i.e.: all communicating processes send their messages at once, rather than being staggered). The algorithm communicates by broadcasting along row and column groups which could lead to increased communication saturation of the interconnect given enough nodes and large enough messages as discussed in Section II.B.

In our experiments we deviated slightly from the published SUMMA algorithm by utilizing a restricted *panel* [4] parameter. The panel decreases the granularity of the communications by limiting the messages to a certain size, rather than communicating an entire sub-matrix. By restricting the size of the message, it enables overlap to be better achieved since the computation can start earlier while communication of the subsequent messages occurs. Through our experiments we observed that the synchronous and asynchronous typically reached their best run-

times at a panel size of 100 or 128 elements. The proxy model, due to its utilization of shared memory within a node typically enjoyed its best performance at a larger panel size of 256.

IV.D.1 Local Matrix-Matrix Multiplication

The larger matrices in SUMMA are divided up into smaller sub-matrices per process (on the single-tier variants), or per node (on the multi-tier variant). These sub-matrices are multiplied using the local BLAS level 3 `dgemm` function. On LeMieux, the `dgemm` operation used in our implementation is Goto's "Fast `dgemm` for Alpha systems", which is optimised specifically for the Alpha 21264 processor used within each node of LeMieux. HP/Compaq also provide a `cxmlp` library which has an optimised CXML `dgemm` routine intended for parallel execution using OpenMP threads across a node.

On Blue Horizon, we utilize the BLAS `dgemm` operation provided by the optimised IBM ESSL library. For the parallel SMP kernel, we use the analogous `dgemm` operation provided by the `ESSL-SMP` library which is optimised for parallel execution using OpenMP threads across the node, similar to LeMieux. The parallel ESSL (PESSL) library, also provided by IBM, can be used; however, PESSL was designed for parallel execution on distributed data across multiple-nodes, and is thus more appropriate for a inter-node distributed partition of data. Since we use `dgemm` as a local (on-node) operation only, it was more appropriate to use the ESSL-SMP library instead.

By default, the MPI implementations lay out the processors per node straight across per row, as shown in Figure IV.3 for an 8 node, 8 CPUs per node configuration as one might see on Blue Horizon. Consecutively numbered processors are in the same node, in groups of eight (i.e.: processors 0-7 are in one node, processors 8-15 are in a second node, and so on so forth). However, this is not an optimal layout. Since the SUMMA algorithm broadcasts along row and column groups, this layout causes unbalanced broadcasts as all row broadcasts remain

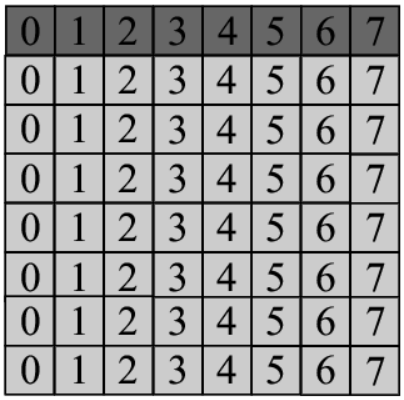

Figure IV.3: SUMMA processor partitioning, 1x8 processor configuration per node for an 8x8 node partitioning

on-node, while each column broadcast creates seven off-node communication calls causing a total cost of:

$$7_{off} + 7_{on} \tag{IV.1}$$

Alternative layouts which may be more optimal can be seen in Figureẽfim:summa-cpulayout8x8-alt for an 8x8 node layout. The first re-balances row broadcasts to be one on-node, and six off-node communication calls, while column broadcasts cost three on-node, and three off-node for a total cost of:

$$10_{off} + 4_{on} \tag{IV.2}$$

The second layout is just a transpose of the first, with the same total cost.

Our last layout is for a 4x16 node configuration, with a 2x4 processor partitioning which results in a row broadcast cost of 12 off-node and 3 on-node communication calls. The column broadcast costs 4 off-node and 1 on-node communication calls for a total cost of:

$$14_{off} + 4_{on} \tag{IV.3}$$

While the overall total cost of the default layout for an 8x8 partitioning is the most balanced, it causes the cost of column broadcasts to be disproportional

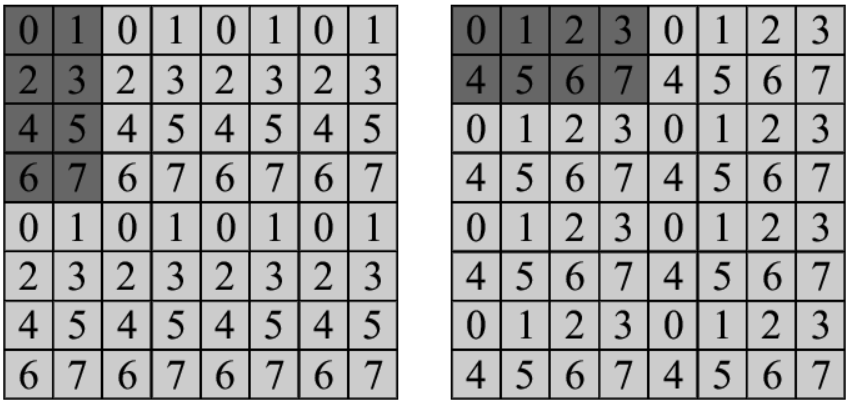

Figure IV.4: SUMMA processor partitioning, 4x2 & 2x4 processor configurations per node an 8x8 node partitioning

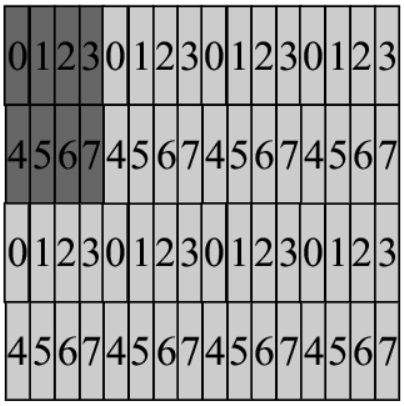

Figure IV.5: SUMMA processor partitioning, 2x4 processor configuration per node an 4x16 node partitioning

Problem Size	Number of OpenMP Threads							
	1	2	3	4	5	6	7	8
1000^3	1.511	0.745	0.508	0.392	0.323	0.276	0.247	0.213
1500^3	5.120	2.646	1.783	1.339	1.087	0.922	0.839	0.729
2000^3	12.195	6.326	4.291	3.260	2.585	2.171	1.889	1.675

Table IV.11: Effect of varying threads on the SUMMA kernel to observe scaling of the memory system on Blue Horizon

Problem Size	Sync	MT
1000^3	0.25	0.22
1500^3	0.79	0.72
2000^3	1.70	1.66

Table IV.12: SUMMA on Blue Horizon, MPI HWP vs. OpenMP LWP

to the cost of row broadcasts. The 2x4 processor configuration allows for slightly more balanced costs.

IV.D.2 Blue Horizon

We first ran a similar benchmark to the RedBlack3D application where we ran the SUMMA kernel at varying numbers of threads to examine how well the kernel would scale across a single SMP node using a threaded kernel. The results shown in Table IV.11 show that the threaded model on a single node scales increasingly well as the problem size increases with the 1000^3 problem achieving a 6.86 times speedup, the 1500^3 achieving 7.11 times, and the largest 2000^3 problem achieving a 7.36 times speedup. The speedups are plotted in Figure IV.6

To compare, we also ran the same size problems on the single-tier kernel on a single node with 8 processes. Our results, as compared to the thread kernel with 8 threads from Table IV.11 can be seen in Table IV.12. The results show both kernels exhibiting similar performance on the single node, with the multi-tier threaded code slightly outperforming the single-tier synchronous code by a few percentage points.

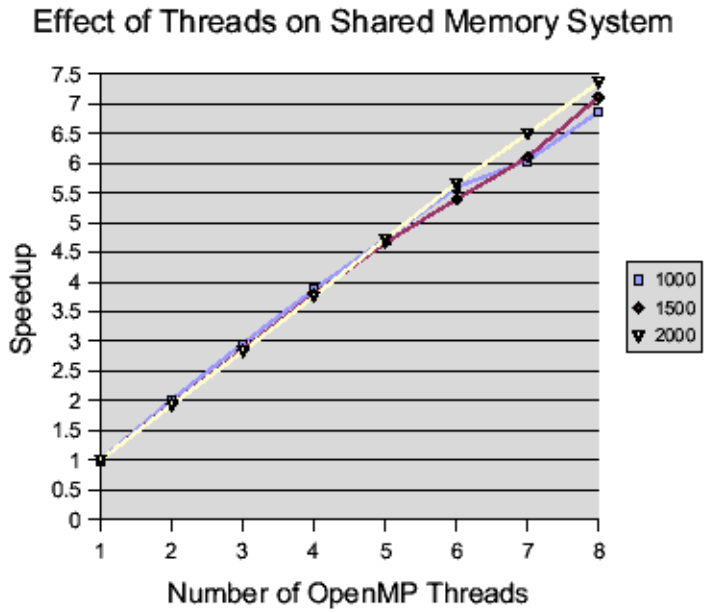

Figure IV.6: Speedup on the SUMMA kernel varying threads for Blue Horizon

During our experimentation with SUMMA on Blue Horizon, we ran four models. Single-tier synchronous (no overlap), single-tier asynchronous (overlap), multi-tier with overlap, and multi-tier with proxy enabled overlap. Through running these experiments, we observed that the multi-tier with proxy enabled overlap did not seem to be functioning properly. Because of our desire to use the `esslsmp` library's optimized `dgemm` kernel, we coded the algorithm as in Figure IV.7.

Because the `esslsmp's dgemm` also uses OpenMP threads, we had to first spawn two threads. One becomes the proxy thread, while the other becomes the master computational thread. This computational thread then runs the `esslsmp` provided `dgemm` which internally spawns 7 child threads (one for each CPU which isn't running the proxy thread). This requires the use of OpenMP's support for nested threading, which unfortunately, the IBM SP2 implementation of MPI doesn't seem to support. Thus, varying the number of threads set in the master computational thread does not affect the `dgemm` kernel whatsoever. The multi-tier without proxy version is essentially the same code with the key differ-

```
parallel_DGEMM_()
{
        spawn 2 threads
        if (thread 0)
                setup asynchronous communication for next iteration
        else
                set OpenMP to use 7 threads, and enable nested threads
                local_ESSLSMP_DGEMM()
}
```

Figure IV.7: Pseudo-code for the SUMMA parallel `dgemm` when the proxy is enabled

ence being that the asynchronous communication setup is done in serial code before any threads are spawned, thus eliminating the need for nested thread spawning (as it removes the outer parallel section). This same multi-tier code responds and behaves as expected when varying the `OMP_NUM_THREADS` variable to change the number of threads the OpenMP system will utilize.

After some investigation, it turns out that the IBM MPI implementation on Blue Horizon is indeed incapable of supporting nested parallelism with OpenMP [4]. The compiler syntactically supports the full OpenMP specification, but the C compiler reports "(W) Option -qsmp=nested_par may cause behavior that is different from the one described in OpenMP API Specification." when nested parallelism is enabled.

The actual behaviour, instead of spawning a second level tree of threads, instead simply reuses any available threads. In addition, it turns out that the reused/nested threads aren't even guaranteed to run in parallel, as the behaviour might be such that the threads run serialized instead.

[4] As stated by the IBM C manual, http://www.csit.fsu.edu/ burkardt/pdf/sc094960.pdf

Problem Size	Single-tier		Multi-tier
	Synchronous	Asynchronous	
3000^3	0.94	0.95	1.20
4000^3	1.98	2.12	2.53
8000^3	15.75	17.59	17.93

Table IV.13: Summary of SUMMA run-times on Blue Horizon, over 8 nodes

We implemented and ran the code to enable multi-tier overlap with a proxy by spawning 8 threads initially, and then keeping 6 of them idle in the hopes that the `dgemm` call would use their CPUs anyway; however, this added even more additional overhead and resulted in performance degradation.

Experimenting with the panel size, our best results for the single-tier variants were with the panel size of 100. For the smaller 3000^3 size, the results with a panel size of 64 were the same as with 100, but at all other sizes, a panel size of 100 outperformed 64. Similarly for the multi-tier variants, we observed best performance at a panel size of 100, with 128 occasionally netting the same run-times.

We executed runs for 8 nodes at different processor configurations. Our best performance obtained was universally at the 8x8 node configuration (i.e.: the whole problem size is split with 8 nodes per edge), and with a CPU partition per node of 4x2 (i.e.: the 8 CPUs per node were laid out in 4 rows, and 2 columns as seen in Figure IV.4.

Our results when running the different variants can be seen in Table IV.13. From these results, we see that the single-tier asynchronous code is unable to achieve any better performance through overlap, thus validating our results we obtained with RedBlack3D: the MPI implementation on Blue Horizon is unable to realize performance gains through overlap. The multi-tier code achieved similar results to the asynchronous code, again failing to achieve overlap. Part of this failure can be attributes to the overhead of spawning extra threads and having them idle. The multi-tier code should theoretically be able to achieve reasonable

performance due to the significant decrease in communication (i.e.: no intra-node broadcasts of data required).

While the results are inconclusive, the results presented in the next section on LeMieux show that overlap can be realized if the `dgemm` kernel is fine-tuned to expose more of its internal parallelism. If the `dgemm` kernel were more amenable to being run in a hybrid OpenMP/MPI environment, and could take advantage of the multi-tier layout, it might result in better overlap.

IV.D.3 LeMieux

On LeMieux, we used Kazushige Goto's excellent *Fast `dgemm` for Alpha* kernel which is comparable, and in some cases, often faster than Hewlett-Packard's reference CXML (Compaq eXtended Math Library) library optimized for parallel SMP nodes (cxmlp). In particular, it let us explicitly set the number of threads we wished to dedicate to the local `dgemm` kernel with the `STATABO_NUMTHREADS` environment variable.

We ran the same benchmark kernel to examine the scalability of the SUMMA kernel across one of LeMieux's 4-way nodes, with our results shown in Table IV.14. At a problem size of 2000^3 we see almost perfect linear scaling going to two threads with a speedup of 1.98. At three threads we also obtain very close to linear speedup with a speedup of 2.87. At four threads we start to incur slowdown, most likely due to either memory contention issues or large memory strides as we obtain a (still respectable) 3.52 times speedup. The results are similar at the smaller problem size of 1500^3 with similar speedups of 1.96, 2.83, and 3.41 respectively as we increase the number of threads. At 1000^3 we see decent speedups when going to 2 and 3 threads (1.96 and 2.84 times speedup respectively). At 4 threads however, the speedup barely increases to 2.91. These results seem to indicate that the SUMMA kernel scales well across the memory subsystem, with the caveat that there must be enough work for the threads to compute on, with 1000^3 being the lower bound on the per node sub-problem.

Problem Size	Number of OpenMP Threads			
	1	2	3	4
1000^3	1.08	0.55	0.38	0.37
1500^3	3.59	1.83	1.27	1.05
2000^3	8.64	4.37	3.01	2.45

Table IV.14: Effect of varying threads on the SUMMA kernel to observe scaling of the memory system on LeMieux

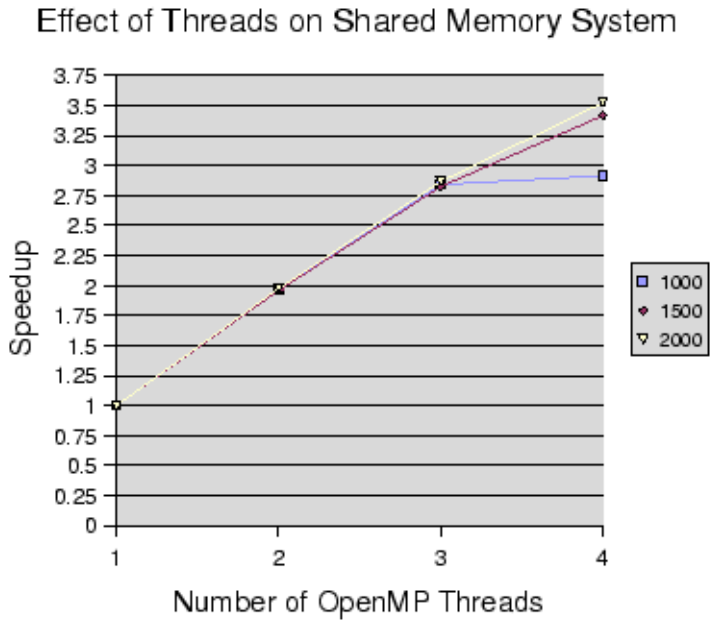

Figure IV.8: Speedup on the SUMMA kernel varying threads for LeMieux

Model	Total Runtime	Communication Time
Proxy	2.45	
Sync	2.40	0.12
Async	2.3	0.18

Table IV.15: SUMMA on LeMieux, MPI HWP vs. OpenMP LWP (communication on the proxy is always zero for 1 node at N=2000, panel=128

We also ran the same pure MPI vs. Proxy comparison on one node to examine the performance of the MPI message passing implemented in shared memory messages versus OpenMP shared memory data sharing. The MPI message passing implementation benefits by varying the panel size in order to block, with our experiments showing the optimal panel size (for this single node experiment) to be 128. For comparison, we fixed the Proxy to run at the same panel size - but in truth, this is an unfair comparison. Because the Proxy's OpenMP threads operate on a shared data structure, they achieve best performance when there is limitation on the panel size. From looking at Table IV.15 it seems that the pure MPI implementation out-performs the Proxy. However, these results belie the true advantages of using shared memory. When the Proxy model is set to an unrestricted panel length, the runtime for our sample 2000^3 problem goes down to a blazingly-fast 0.11s. This is a reflection of the shared memory benefits of using OpenMP, rather than a reflection of the proxy itself - since indeed, there is no off-node communication for the proxy thread to manage. Setting it to an unrestricted panel size allows the CPUs to more fully utilize cache locality. The MPI implementation still benefits from restricting a panel limitation due to the need to communicate data, as limiting the panel size allows the application to decrease the granularity of the communicated messages to the other processes. Since no data needs to be communicated in the shared memory implementation, there is no need to restrict the panel limit.

Results for running at varying problem sizes on both 16 and 32 node configurations are show in Table IV.16 and Table IV.17. At 16 nodes, our results show that asynchronous overlap of communication is possible, though the best results are with the asynchronous baseline MPI version with the results netting as much as a 30% performance speedup for the 5000^3 case. The proxy still shows good speedup as compared to the synchronous case however, thus showing it is still definitely capable of realizing overlap.

At 32 nodes we see similar results, with both the MPI asynchronous

Problem Size	Synchronous			Asynchronous			Proxy		
	Total	Comp	Comm	Total	Comp	Comm	Total	Comp	Comm
3000^3	0.804	0.526	0.275	0.554	0.471	0.075	0.687	0.650	0.037
4000^3	1.766	1.301	0.450	1.270	1.124	0.127	1.473	1.411	0.062
5000^3	3.493	2.808	0.670	2.396	2.226	0.118	2.732	2.636	0.096
8000^3	10.33	8.624	1.655	9.384	8.912	0.386	10.18	10.085	0.096

Table IV.16: SUMMA, on LeMieux @ 16 nodes

Problem Size	Synchronous			Asynchronous			Proxy		
	Total	Comp	Comm	Total	Comp	Comm	Total	Comp	Comm
8000^3	5.88	4.31	1.54	4.69	4.29	0.37	5.49	5.48	0.01
10000^3	12.31	9.85	2.34	10.43	9.84	0.58	10.24	10.15	0.09

Table IV.17: SUMMA, on LeMieux @ 32 nodes

and proxy out-performing the MPI synchronous implementation by 20% and 6.6% respectively at a problem size of 8000^3. At the larger 10000^3 problem, the proxy marginally outperforms the asynchronous implementation with a 16.8% (versus the asynchronous's speedup of 15.2% over the MPI synchronous implementation).

On both the 16 and 32 node runs, we see that the asynchronous communication times cost very little in terms of the overall total runtime. The proxy can therefore do little to improve on these times, as the asynchronous implementation manages to achieve good overlap with the MPI I_mode communication calls. Nevertheless, the proxy is able to squeeze that extra 1.6% improvement over the MPI asynchronous implementation.

IV.D.4 Summary of SUMMA

The results from SUMMA clearly show that overlap is both attainable, and relatively easily achievable when the inter-node communication library (MPI in this case) permits asynchronous communication due to the simplicity and ease of overlapping the algorithm. Despite each iterations dependence on the iteration prior to it, the results show that with a large enough problem size, and enough communication to warrant the loss of a computational thread, the speedup gains are worth the investment. Similarly to RedBlack3D, a balance must be struck between ensuring the problem is large enough that the communication time required by the proxy can be overlapped entirely by the computational time and ensuring that there is also enough communication in order to warrant taking one CPU away from the computational resources to dedicate to handling the communication. Due, in small part, to the aggregation of communication broadcasts onto one thread, we only generate enough communication on the proxy model when we reach a large number of nodes. In our results, we didn't typically start to see consistent speedup until 32 nodes on LeMieux.

Our results with SUMMA also illustrated a dependence parallel algorithms have on the system libraries upon which they call. For Blue Horizon, the

results were inconclusive due to the limitations of the IBM ESSL-SMP `dgemm` kernel in conforming to a new model of parallelism. Given a more tunable library, one that is more amenable to having the number of threads be a variable parameter (rather than a all-or-one threads model), we suspect more significant performance gains on Blue Horizon could be obtainable.

Chapter V

Related Work

V.A Project Inspiration

The proxy was based off a previous project written in Professor Scott Baden's CSE 262 (Software and Systems Support for Parallel Computation)[1] course. It was originally written in a manner which supported Valkyrie, a Linux system running MPI (tested with the MPICH implementation) on a sixteen node, two-way SMP cluster of COTS (Commodity Off The Shelf) systems of PCs. It runs SDSC's Rocks cluster management tools. Valkyrie's model of thread spawning supports LWP (light-weight processes) as threads. Using POSIX thread calls allows a newly instantiated thread (created with `pthread_create()`) to be spawned onto the second CPU. Any memory allocated on the stack before the thread is spawned, or any heap memory allocated is shared between the th'reads. The proxy was written such that when each MPI HWP (heavy-weight process) starts, it allocates shared queues for outgoing and incoming data before spawning a new LWP onto the second CPU to be the computational client. Whenever the client has data to be sent, it is pushed onto the outgoing work queue. The proxy slept on a semaphore protecting access to the work queue; when the semaphore goes 'up', the proxy starts sending the outgoing data.

[1] This project was written by Craig Donner, Cyrus Jam, and Stephen Lau

Originally, an effort was made to attempt to port this code-base directly to the new architectures of LeMieux and Blue Horizon. It was however ultimately decided, for performance reasons, that an entire re-write from the ground up would be better.

V.B Other work

Any previous work in the area of using communications proxies will inevitably come down to work done by Lim, Heidelberger, Pattnaik, and Snir [17]. While their goal was primarily concerned with protecting access to a shared network interface, it laid the groundwork for further development with regard to gaining faster and more cost-effective performance.

Falsafi and Wood did similar work [10] in determining whether dedicated or shared protocol processors produced more effective performance on NOW (Networks of Workstations) using SMP nodes. Their work differed in that they were determining whether more optimal performance could be obtained from having a dedicated protocol processor versus having the different computational threads share the role of being a protocol processor. A protocol processor is similar in idea to a proxy in that it handles communication for that node. A 'floating' protocol processor was a model where each thread would handle the communication management, depending on their idleness. I.e.: if the threads were performing a bulk-synchronous computation, then the first to finish would immediately start communication for the next stage, rather than having one fixed communication management thread. Their work determined that a dedicated protocol processor benefited lighter-weight protocols more than heavy-weight protocols (i.e.: protocols with smaller end-to-end latencies). They also found that a dedicated, or fixed, protocol processor was more beneficial as compared to a 'floating' protocol processor for applications where communication was the main bottleneck.

Steven Lumetta has also done similar work [18] [19] [20] involving multi-

protocol *Active Messages* (i.e.: using both shared memory and message passing) to increase communications performance across clusters of multi-processors or SMPs. The work done there was also different in that each thread or processor within a node could still communicate with any other processor on any other node. As far as each thread is concerned, the cluster is still flattened out as a single-tier. The multi-tier awareness is limited to the communication system, rather than being brought into the algorithm itself. Lumetta's work also involved more lower-level access to a more intelligent NIC, while our work should remain portable to any NIC, as it simply builds on top of other already-existing communications layers such as MPI and OpenMP.

Much work has been done in describing advantages and disadvantages of hybrid MPI and OpenMP approaches to managing communications. [9] [6] [8] [7] [21] The results are mixed, though the generally consensus is that mixed-mode hybrid MPI/OpenMP applications can deliver performance for specific algorithms at certain problem sizes and/or geometries, but often the gains are either not applicable to the general case, or not worth the effort put in to perform the non-trivial optimizations.

Some of the work in this thesis was derived from work done by Scott Baden with Stephen J. Fink [3] [4] and Daniel Shalit [5]. Fink was instrumental in the development of KeLP and its support for multi-tiered parallel computer architectures. His PhD thesis [11] on KeLP provided an application runtime infrastructure which allows the programmer to model parallel algorithms around data, rather than having to concentrate on explicit movement of the data. Much work has been done by Baden and Fink on KeLP (and its successor KeLP2) which provides many benefits of exploiting multi-tier hierarchies in addition to overlap of communication with computation [12]. Some of the strategy in this thesis for overlapping the SUMMA algorithm was derived from Agarwal, et. al in [1] as well as Stephen J. Fink's SUMMA algorithm using KeLP2 in [3].

The RedBlack3D kernel used was the Fortran numerical kernel developed

by Baden and Fink. The original work in developing the strategy for overlapping RedBlack3D came from Sohn and Biswas [23].

The SUMMA algorithm used was Van de Geijn's as presented in [15] with some overlapping techniques described by Agarwal, Gustavson, and Zubair in [1]. The `dgemm` kernel used on Blue Horizon was provided by IBM's ESSL (SMP) library, while Goto's "Fast DGEMM for Alpha" was used as the `dgemm` kernel on LeMieux [16].

Chapter VI

Conclusion

Overlap is not as simple as just moving the critical path out of communication. With today's hardware, communication performance has improved since prior work regarding proxies and other multi-tier formulations for achieving overlap. Now programmers must contend with the adverse affects of thread library overhead, as well as the negative impact of irregular geometries and other computational side effects on facets of performance such as cache locality, TLB locality, and poor memory strides.

In addition, an inherent difficulty in realizing overlap when using off-the-shelf numerical libraries such as ESSL, or CXML is the inability to use the numerical operations in ways the author did not intend. E.g.: ESSL-SMP is optimized to use all available processors on a node as threads when doing a local `dgemm` operation. It doesn't cope as well when trying to run with one less processor as with the proxy. Often, one may want to get access to the raw serial numerical kernel within the operation in order to use it in more flexible ways such as ours.

The dependence upon system libraries which provide both intra and inter-node communication such as OpenMP and MPI was also highlighted through this work. This work utilized both types of communication libraries in its hybrid model, and the inability of an implementation to support such communication models as nested parallel threads, and asynchronous communication significantly hinders

work seeking to unify the two models.

Perhaps one reason for the inabilities or failings of system libraries to provide an expected behaviour (such as nested threads or asynchronous communication) is due inadvertently to the system designers. The designers or architects often envision a paradigm of use for their machine, and may significantly optimize for that model at the cost of hurting other approaches. It may not be the IBM MPI implementation's lack of ability to realize overlap with asynchronous communication, instead, it may be that the synchronous communication is so optimized as to render the need for faster asynchronous communication primitives unnecessary.

In addition, on clusters of multi-processors such as LeMieux and Blue Horizon, the nodes communicate via an intelligent fast interconnect such as QsNet/Quadrics or Colony. The nodes' network interfaces (such as Elan, in the case of LeMieux) have now reached a point at which they are extremely robust and flexible, certainly more so than the dedicated co-processors of yesterday. The combination of these intelligent network interfaces, the high-bandwidth and low-tolerance interconnect itself, and an appropriately written system library such as MPI optimized to exploit the performance potential of the interconnect is often enough to squeeze enough performance. The dedicated co-processor on the network interface essentially acts as a proxy for the machine, thereby often making a specifically software-programmed proxy not worth the development effort involved. While such a tailored approach can provide performance gains, it is usually for a small class of problems and is not scalable without sufficient effort.

Nonetheless, our work showed that a proxy can help deliver potential performance gains on certain classes of machines where overlap may not be possible using conventional asynchronous communications (such as the MPI implementation on Blue Horizon). This shows that a proxy can deliver performance portability in that overlap is *generally* achievable. While it may not outperform conventional asynchronous techniques on every platform, it is usually capable of achieving overlap and thus has the potential to deliver better performance relative to a conven-

tional synchronous communication model (dependent upon the software flexibility of libraries and such being used). The proxy method can often then provide a more portable method of achieving overlap. The proxy is able to realize overlap on a system that could not do I-mode communication (IBM's Blue Horizon) while still achieving better performance relative to the synchronous model without inducing performance penalties on a system that could realize better performance through I-mode communication (Compaq's LeMieux). For the general broad case, a proxy can help achieve more portable performance as compared to using strictly I-mode asynchronous communication.

Chapter VII

Appendix A: Proxy API Definition

While our implementation was custom-written for the RedBlack3D and SUMMA algorithms and integrated into their code, we envision a suitable library API would look like the following when the proxy code is extracted into a general library.

```
class ProxyThread {
private:
    /* v_handles is where the communication handler should store the
MPI_Request communication handles/descriptors so that waitComm can find
them */
    MPI_Request** v_handles;
    int num_threads;
public:
    int init(int num_comp_threads, int (*commFn)(int iteration));
    int startComm(int iteration);
    int waitComm(int iteration);
};
```

the Proxy can then be utilized by a client (doing a 10 iteration computation) as follows:

```
#include "ProxyThread.h"
DataStructure data; int handleCommunication(int iteration)
{
        /* communication handler to do communication for a given iteration */
        v_handles[iteration] = Send(data[myrank][iteration],
                                                num_threads%iteration);
}
int main(int argc, char** argv)
{
        ProxyThread proxy;

        int num_cpus = sysconf(_NUM_AVAILABLE_CPUS);
        int iter;

        proxy.init(num_cpus-1, handleCommunication);
        for (iter=0; iter<=10; iter++)
        {
                proxy.startComm(iter);
                doInnerComputation();
                proxy.waitComm(iter);
                doAnnulusComputation();
        }
}
```

Bibliography

[1] R.C. Agarwal, F. Gustavson, and M. Zubair. A high-performance matrix multiplication algorithm on a distributed memory parallel computer using overlapped communication. *IBM Journal of Research and Development*, 38(6):673–681, 1994.

[2] Yukiya Aoyama and Jun Nakano. *RS/6000 SP: Practical MPI Programming*. IBM International Technical Support Organization, 1999.

[3] Scott B. Baden and Stephen J. Fink. Communication overlap in multi-tier parallel algorithms. In *Proceedings of the 1998 ACM/IEEE conference on Supercomputing*, pages 1–20. IEEE Computer Society, 1998.

[4] Scott B. Baden and Stephen J. Fink. A Programming Methodology for Dual-tier Multicomputers. In *IEEE Transactions on Software Engineering*, pages 212–226. IEEE Computer Society, 2000.

[5] Scott B. Baden and Daniel Shalit. Performance Tradeoffs in Multi-tier Formulation of a Finite Difference Method. In *Proceedings of the International Conference on Computational Science*, 2001.

[6] Franck Cappello and Daniel Etiemble. MPI versus MPI+OpenMP on the IBM SP for the NAS Benchmarks. In *Proceedings of SuperComputing 2000*, pages 51–51, 2000.

[7] Franck Cappello and Olivier Richard. Intra node parallelization of mpi programs with openmp, 1998.

[8] Franck Cappello, Olivier Richard, and Daniel Etiemble. Investigating the performance of two programming models for clusters of SMP PCs. In *Proceedings of the 6th IEEE Symposium on High-Performance Computer Architecture*, pages 349–359, 2000.

[9] Edmond Chow and David Hysom. Assessing Performance of Hybrid MPI/OpenMP Programs on SMP Clusters, 2001.

[10] Babak Falsafi and David A. Wood. Scheduling Communication on an SMP Node Parallel Machine. In *Proceedings of the IEEE Third International Symposium on High Performance Computer Architecture*. IEEE Computer Society, 1997.

[11] Stephen J. Fink. *A Programming Model for Block-Structured Scientific Calculations on SMP Clusters*. PhD thesis, University of California at San Diego, June 1998.

[12] Stephen J. Fink and Scott B. Baden. Runtime Support for Multi-Tier Programming of Block-Structured Applications on SMP Clusters. In *Proceedings of the International Scientific Computing in Object-Oriented Parallel Environments Conference*, pages 1–8, 1997.

[13] Geoffrey C. Fox, Roy D. Williams, and Paul C. Messina. *Parallel Computing Works!* Morgan Kaufmann, 1994.

[14] Antonio Augusto Frohlich. "SMP PCs: A Case Study on Cluster Computing".

[15] R. A. Van De Geijn and J. Watts. SUMMA: Scalable universal matrix multiplication algorithm. *Concurrency: Practice and Experience*, 9(4):255–274, 1997.

[16] Kazushige Goto and R.A. Van De Geijn. On Reducing TLB Misses in Matrix Multiplication. In *ACM Transactions on Mathematical Software*, 2003.

[17] Beng-Hong Lim, Philip Heidelberger, Pratap Pattnaik, and Marc Snir. Message Proxies for Efficient, Protected Communication on SMP Clusters. In *Proceedings of the IEEE Third International Symposium on High Performance Computer Architecture*, pages 116–127. IEEE Computer Society, 1997.

[18] S. S. Lumetta. *Design and Evaluation of Multi-Protocol Communication on a Cluster of SMP's*. PhD thesis, University of California at Berkeley, November 1998.

[19] S. S. Lumetta and D. E. Culler. Managing Concurrent Access for Shared Memory Active Messages. In *Proceedings of the International Parallel Processing Symposium*, Orlando, Florida, April 1998.

[20] S. S. Lumetta, A. M. Mainwaring, and D. E. Culler. Multi-Protocol Active Messages on a Cluster of SMP's. In *Proceedings of SC97: High Performance Networking and Computing*, San Jose, California, November 1997.

[21] Lorna Smith and Mark Bull. Development of mixed mode mpi / openmp applications. *Scientific Programming*, 9(2-3/2001):83–98, 2001. Presented at Workshop on OpenMP Applications and Tools (WOMPAT 2000), San Diego, Calif., July 6-7, 2000.

[22] Marc Snir, Steve Otto, Steven Huss-Lederman, David Walker, and Jack Dongarra. *MPI: The Complete Reference*. The MIT Press, 1996.

[23] A. Sohn and R. Biswas. Communication studies of dmp and smp machines, 1997.

[24] Kwan-Po Wong and Cho-Li Wang. Push-pull messaging: A high-performance communication mechanism for commodity smp clusters. In *Proceedings of the IEEE International Conference on Parallel Processing*, page 12. IEEE Computer Society, 1999.