UNIVERSITY OF CALIFORNIA, SAN DIEGO

A Programming Model for Block-Structured Scientific Calculations on SMP Clusters

A dissertation submitted in partial satisfaction of the requirements for the degree Doctor of Philosophy in the Department of Computer Science and Engineering

by

Stephen Jason Fink

Committee in charge:

Professor Scott B. Baden, Chair
Professor Larry Carter
Professor William G. Griswold
Professor Sutanu Sarkar
Professor John Weare

1998
The dissertation of Stephen Jason Fink is approved, and it is acceptable in quality and form for publication on microfilm:

Chair

University of California, San Diego

1998
TABLE OF CONTENTS

Signature Page ........................................ iii
Table of Contents ......................................... iv
List of Figures .......................................... vii
List of Tables ......................................... x
Acknowledgments ....................................... xi
Vita, Publications, and Fields of Study .............. xiv
Abstract ........................................... xvi

1 Introduction .......................................... 1
  1. Motivation ...................................... 1
  2. SMP Cluster Hardware Characteristics ............. 5
     1. Inter-node Communication .................... 5
     2. Memory System Contention ................... 8
  3. Software Infrastructure ........................... 8
  4. Application Studies ............................. 10
  5. Organization of the Dissertation ................ 11

2 Programming Abstractions ............................ 12
  1. Introduction .................................. 12
  2. KeLP Programming Abstractions .................. 14
     1. Meta-Data Abstractions ..................... 15
     2. Control Flow Abstractions ................. 18
     3. Storage Model .............................. 19
     4. Data Motion ................................ 20
     5. Other Collective Operations ............. 25
  3. Programming Example ........................... 26
     1. Single-tier code .......................... 27
     2. Multi-tier code ........................... 31
     3. Multi-tier code with communication overlap 33
  4. Related Work ................................... 37
     1. Structural Abstraction .................... 37
     2. Software Support for Hierarchical Programming 37
     3. Single-Tier Parallel Languages ............ 40
     4. Libraries .................................. 42
LIST OF FIGURES

1.1 A single-tier multicomputer. .............................................. 3
1.2 A multi-tier multicomputer. ............................................ 3

2.1 Software migration path to an optimized multi-tier parallel code.... 15
2.2 The KeLP FloorPlan stores an array of Regions, each mapped to an integer. ............................................................ 17
2.3 The KeLP MotionPlan represents a set of block copy operations. ... 21
2.4 a) Pseudocode to generate a fillpatch MotionPlan M to fill in ghost cells for XArray x. b) The dark shaded regions represent ghost regions that are copied into the central Grid. ......................... 23
2.5 KeLP pseudo-code to generate a MotionPlan that fills in periodic boundary conditions along one dimension. ....................... 24
2.6 Main procedure for redblack3D example. .............................. 27
2.7 Single-tier KeLP code for redblack3D relaxation. .................... 30
2.8 Multi-tier KeLP code for redblack3D relaxation. ..................... 32
2.9 Multi-tier KeLP code with communication overlap for redblack3D relaxation. ...................................................... 34
2.10 a) A FloorPlan with six Regions represents the stencil data dependencies for a two-node SMP. b) KeLP code to setup the FloorPlan. 36

3.1 Software layers in the KeLP 2.0 implementation. .................... 55
3.2 KeLP code for the ring message-passing microbenchmark. ........ 62
3.3 KeLP code for the exchange message-passing microbenchmark. ..... 63
3.4 Performance of the ring microbenchmark on a) the AlphaServer cluster, b) the SparcStation cluster, and c) the T3e. ................. 64
3.5 Performance of the exchange microbenchmark on a) the AlphaServer cluster and b) the SparcStation cluster. ....................... 65
3.6 Performance of the exchange microbenchmark on the AlphaServer cluster with the flow control message optimization. .......... 66
3.7 Fortran 77 code for the local kernel used in the overlap efficiency experiments. ....................................................... 67
3.8 Performance results for the kernel for Figure 3.7 on a) the AlphaServers and b) the SparcStation cluster. .......................... 68
3.9 Performance results for the kernel for Figure 3.7 on a) the AlphaServers and b) the SparcStation cluster. .......................... 70
3.10 Overlap efficiency results on a) the AlphaServer cluster and b) the SparcStation cluster. .................................................. 72
3.11 Barrier synchronization performance on a) the AlphaServer cluster and b) the SparcStation cluster. ................................. 74
3.12 Reduction performance on a) the AlphaServer cluster and b) the SparcStation cluster. .......................... 75

4.1 Inheritance relationships between KeLP objects and the DOCK, FD-DSL, and dGrid DSLs. Arrows indicate inheritance. ....... 84
4.2 Multi-tier KeLP code for redblack3D relaxation, using the DOCK DSL to perform the intra-node partitioning. .......... 85
4.3 Main procedure for redblack3D example, using FD-API and DOCK facilities. ........................................... 88
4.4 Performance of single-tier redblack3D implementations on a) the AlphaServers, and b) the SparcStations. ............ 89
4.5 Redblack3D performance on one SMP node: a) AlphaServer b) SparcStation. The MPI code requires the number of processors to be a power of 2, so no MPI results reported for 3 processors. .... 90
4.6 Redblack3D performance on two SMP clusters, using all four processors per node: a) AlphaServers b) SparcStations. .... 91
4.7 A two-stage dimensional exchange algorithm used for ghost cell updates in 2D. a) Stage 1 b) Stage 2. The Figure only shows messages passed which affect the final state of the left edge of node 4. Other messages are omitted, but symmetric to those shown. ............. 96
4.8 Single-tier performance for the NAS-MG benchmark on a) the AlphaServers, and b) the SparcStations. .................. 99
4.9 NAS-MG performance on a single SMP: a) AlphaServer, b) SparcStation. The MPI code requires the number of processors to be a power of 2, so no results are reported for 3 processors. ......... 100
4.10 NAS-MG performance on a) the AlphaServer cluster and b) the SparcStation cluster. ................................. 101
4.11 Grid geometries used for the multi-block multigrid code. The colors denote the assignment of grids to nodes on a) two nodes, b) four nodes, and c) eight nodes. ............................... 106
4.12 Single-tier performance of the multi-block multigrid code on a) the AlphaServers and b) the SparcStations. ............ 107
4.13 Multi-block multigrid performance on a) one AlphaServer node and b) one SparcStation node. ......................... 110
4.14 a) Structure of the basic Multi-tier Multiblock relaxation loop. b) A modified version to reduce intra-node synchronization. .......... 111
4.15 Multi-tier performance of the multi-block multigrid code on a) the AlphaServer cluster and b) the SparcStation cluster ........ 114
4.16 Structure of the NAS-FT benchmark. .......................... 117
4.17 Structure of Agarwal et al.’s overlap algorithm for NAS-FT. .... 118
4.18 Single-tier FT performance on a) the AlphaServer cluster, and b) the SparcStation cluster. .................................................. 120
4.19 a) FT performance on one AlphaServer. b) FT performance on one SparcStation. .................................................. 120
4.20 a) FT performance on the AlphaServer cluster. b) FT performance on the SparcStation cluster. .................................. 121
4.21 a) The standard “ijk” matrix-multiply loop. b) With the “kij” ordering, the inner two loops perform an outer product operation. . . 125
4.22 Basic SUMMA algorithm. .................................................. 125
4.23 Blocked basic SUMMA algorithm ........................................ 126
4.24 Graphical depiction of one stage of the blocked SUMMA algorithm. 126
4.25 Overlapped blocked SUMMA algorithm. ............................. 127
4.26 Single-tier SUMMA performance on the AlphaServer cluster. ... 128
4.27 a) SUMMA performance on one AlphaServer node. b) SUMMA performance on the AlphaServer cluster. .............................. 129
4.28 Representation of a pipelined computational pattern. .............. 130
4.29 In each iteration of the outer loop, the ScaLAPACK LUD algorithm operates on the trailing submatrix \( A \), consisting of the union of \( A_{00}, A_{01}, A_{10}, \) and \( A_{11} \). .................................................. 133
4.30 High-level description of the ScaLAPACK LU algorithm. ........ 140
4.31 a) Single-tier LU Performance on the AlphaServers. b) Single-node LU performance on one AlphaServer. ............................ 141
4.32 High level description of the multi-tier LU algorithm that explicitly overlaps several activities. ................................. 142
4.33 Performance of the multi-tier LU codes on the AlphaServer cluster. 143
4.34 Performance of single-tier redblack3D implementations on the T3E. 144
4.35 Single-tier performance for the NAS-MG benchmark on the T3E. . 145
4.36 Single-tier performance for the multi-block multigrid benchmark on the T3E. .................................................. 146
4.37 Single-tier FT performance the T3E. .................................. 146
4.38 Single-tier SUMMA performance on the T3E. .......................... 147
1.1 Characteristics of some single-tier multicomputers. $t_s$ is the time to send a one-byte message using MPI. $BW$ is the peak bandwidth observed for long messages. ........................................ 6

1.2 Characteristics of some SMP clusters. $t_s$ is the time to send a one-byte message between nodes using MPI. $BW$ is the peak bandwidth observed for long messages. FLOPS results in parentheses are per-node, others are per-processor. ........................................ 7

2.1 A brief synopsis of the KeLP data types. ......................... 16

3.1 The *smp++* functions. ........................................ 56

3.2 Software environments for all experiments in the dissertation. . 61

4.1 Problem sizes for the redblack3D experiments. ................. 89

4.2 Execution time break-down, in ms, for one iteration of redblack3D on the AlphaServer cluster. The column labeled *Comm* gives the time the application spends waiting for communication to complete. 94

4.3 Execution time break-down, in ms, for one iterations of redblack3D on the SparcStation cluster. The column labeled *Comm* gives the time the application spends waiting for communication to complete. 94

4.4 Execution-time breakdown, in seconds, for 10 V-cycles of the multi-tier KeLP NAS-MG codes on eight AlphaServers. ............... 102

4.5 Execution-time breakdown, in seconds, for 10 V-cycles with the multi-tier KeLP NAS-MG codes on four SparcStations. ........ 104

4.6 The number of grid points assigned to nodes with the Lake Superior multi-block geometry. ................................. 108

4.7 The number of ghost cells, per node, which require inter-node communication in the 2D multi-block multigrid experiments. .......... 109

4.8 Execution-time breakdown, in seconds, for 10 V-cycles of the multi-tier KeLP multiblock multigrid code on eight AlphaServers. For each code section, The Table reports the maximum time spent in that section by any node; therefore, the sum of the individual sections does not add up to the total running time. 112

4.9 Execution-time breakdown, in seconds, for 10 V-cycles of the multi-tier KeLP multiblock multigrid code on four SparcStations. For each code section, The Table reports the maximum time spent in that section by any node; therefore, the sum of the individual sections does not add up to the total running time. 113
4.10 Execution-time breakdown for 10 timesteps of the basic and overlapped multi-tier KeLP FT codes on the AlphaServer cluster. The Table reports the maximum time, in seconds, spent by any node in each section of the application.

4.11 Execution-time breakdown for 10 timesteps of the basic and overlapped multi-tier KeLP FT codes on the SparcStation cluster. The Table reports the maximum time, in seconds, spent by any node in each section of the application.

4.12 Execution-time breakdown, in seconds, for one iteration of the basic and overlapped multi-tier KeLP SUMMA codes on the AlphaServer cluster. The column labeled Comm gives the time the application code spends waiting for communication to complete.

4.13 Performance, in MFLOPS, of the ScaLAPACK LU code on the AlphaServers, using one MPI process per physical processor. $N$ is the number of nodes used, and $b$ is the block size used in the block-cyclic matrix decompositions.

4.14 Execution-time breakdown, in seconds, reported by processor 0 on each node for one iteration of the KeLP LU code on the AlphaServer cluster.
ACKNOWLEDGMENTS

I would like to thank my advisor, Scott Baden, for all his support, encouragement, and guidance over the past five years. In our many discussions, Scott's insight and advice have made an enormous contribution to this dissertation. I have enjoyed having the opportunity to work with him.

Additionally, I thank the members of my committee, Larry Carter, Bill Griswold, Sutanu Sarkar, and John Weare, for helpful suggestions and feedback. Professors Fran Berman and Jeanne Ferrante also gave invaluable assistance along the way, and their help is greatly appreciated.

Special thanks to the other members of the lab, whose friendship and support helped make my graduate career fun. In particular, Scott Kohn, Jenny Schopf, Silvia Figueira, Nick Mitchell, Kang-Su Gatlin, and Rich Wolski contributed many technical ideas and criticism. I greatly valued their input and discussions.

Thanks to Max Orgiy an, Jeff Howe, and Abdul Tabbara for suggestions and assistance with the the KeLP implementation. Thanks also to Matt Ruben for his friendship and occasional beignets.

My parents and sister helped make me the person I am, and certainly deserve most of the credit for any accomplishments of mine (just ask them!). Thanks to Mom, Dad, and Lo Lo, for tremendous love and support over the years.

Finally, I dedicate this dissertation to my favorite person, my wife Janet. Janet’s unwavering support and encouragement were my greatest resources during these past years. I think we make a great team, and I treasure our every day together.

Generous financial support for my education was provided by the Department of Energy Computational Science Graduate Fellowship Program. Computer time on the Digital Maryland AlphaServer was provided by NSF CISE Institutional Infrastructure Award CDA9401151 and a grant from Digital Equipment Corp.. Computer time on the Oregon State SparcStation cluster was graciously provided by Professor Michael Quinn. Computer time on the T3E was provided
by the National Partnership for Advanced Computational Infrastructure. Thanks also to Richard Frost, Alan Sussman, and Joel Saltz, for helping arrange computer access.
VITA

July 7, 1970  Born, Brooklyn, New York

1992  B.S. Computer Science
      Additional major in Mathematics
      Duke University

1994  M.S. Computer Science
      University of California, San Diego

1998  Ph.D. Computer Science
      University of California, San Diego

PUBLICATIONS


**FIELDS OF STUDY**

Major Field: Computer Science


Professor Scott B. Baden
ABSTRACT OF THE DISSERTATION

A Programming Model for Block-Structured Scientific Calculations on SMP Clusters

by

Stephen Jason Fink

Doctor of Philosophy in Computer Science
University of California, San Diego, 1998
Professor Scott B. Baden, Chair

Multi-tier parallel computers such as clusters of symmetric multiprocessors (SMPs) offer both new opportunities and new challenges for high-performance computation. Although these computer platforms can potentially deliver unprecedented performance for computationally intensive scientific calculations, realizing the hardware’s potential remains a formidable task. To achieve high performance, the programmer must coordinate several levels of parallelism and locality to match the hardware’s capabilities. Current programming languages and software tools do not directly facilitate this task, and the resultant difficulties hinder efficient implementations of scientific calculations on SMP clusters.

We present a concise set of programming abstractions that simplify implementation of efficient algorithms for block-structured scientific calculations on SMP clusters. The software infrastructure, KeLP, provides intuitive geometric mechanisms to help the programmer coordinate data layout, data motion, and parallel control. The KeLP constructs abstract away many low-level programming details of message-passing, thread management, synchronization, scheduling, and storage allocation. Nevertheless, KeLP still provides enough expressive power to implement effective multi-tier algorithms for a broad class of computationally-intensive scientific applications. Most importantly, the KeLP implementation adds
little overhead to lower-level primitives, and KeLP performance usually matches or exceeds performance for comparable programs using lower-level primitives.

The dissertation presents solutions to varied technical challenges in the realization of a concise, abstract, expressive, and efficient programming model for multi-tier computers. In particular, the dissertation extends the structural abstraction programming model to manage three levels of parallel control and data structures to match the multi-tier hardware. KeLP presents a new communication orchestration model which combines structural abstraction with ideas from the inspector/executor paradigm.

In application studies, we present new multi-tier algorithms for several applications, including multigrid, matrix multiplication, and dense matrix factorization. Experimental results on several platforms expose bottlenecks that limit performance and trade-offs for algorithmic and hardware design. Finally, this research has resulted in the KeLP 2.0 implementation, a C++ class library which has been used successfully in a number of computational science research projects.
Chapter 1

Introduction

1.1 Motivation

Modern high-performance computer systems can potentially deliver unprecedented performance for computationally intensive scientific applications. Unfortunately, for many applications, realizing the hardware's potential remains a formidable task. Most high-performance computer systems present a complex non-uniform memory hierarchy with several levels of parallelism and locality [6]. In order to use these systems efficiently, one must judiciously orchestrate parallelism and locality in the application to match the hardware resources. To this end, the programmer or compiler must navigate a complex landscape of performance tradeoffs and implementation details. These complexities hinder efficient parallel implementations, and discourage the use of high-performance parallel computers for scientific computation.

To address this problem, much research has considered parallel programming languages and libraries to help the programmer manage locality and parallelism. Notably, the Parallel Virtual Machine (PVM) [130] and Message-Passing Interface (MPI) [102] have emerged as standards for portable parallel programming with message-passing. With PVM or MPI, the programmer can build a portable code that will achieve reasonable performance on a variety of parallel computers.
However, with these libraries, the programmer must manage a plethora of low-level implementation details, such as message buffers, message tags, and global-to-local mappings.

Other programming models, such as High Performance Fortran (HPF) [78], provide higher-level abstractions for portable parallel programming. With higher-level abstractions, these languages reduce the complexity of parallel programs by hiding many of the low-level implementation details. However, higher-level languages usually match only limited problem domains, and with current technology, some programs do not perform as well as their message-passing counterparts [13].

Although current parallel programming languages and libraries represent a wide range of design choices, most share the same underlying model of a parallel computer, which we refer to as the *single-tier multicomputer model* (Figure 1.1).

A single-tier multicomputer consists of a number of *nodes*, where a node consists of one microprocessor along with a private local memory system. The nodes communicate over an interconnection network, either by passing messages or relying on hardware-supported shared-memory. This general model directly reflects the structure of many commercial distributed-memory parallel computers, such as the IBM SP-2 [2], Cray T3E [117], and networks of uniprocessor workstations [8]. Additionally, the single-tier multicomputer arises in various forms in formal parallel computation models, such as CTA [123], BSP [134], and LogP [53]. Research based on the single-tier multicomputer model has resulted in advances in algorithms, compiler technology, programming languages, and run-time libraries which make it possible to achieve reasonably good performance for many scientific applications on distributed-memory parallel computers.

Although many programming models target the single-tier multicomputer model, clusters of symmetric multiprocessors (SMPs) have recently emerged as widespread targets for high-performance scientific computation [138]. Figure 1.2 shows a representation of an SMP cluster as a *multi-tier multicomputer*. Unlike message-passing machines with uniprocessor nodes, SMP clusters present two coarse-grained levels of parallelism. Within a single SMP, multiple processors share
Interconnection Network

Figure 1.1: A single-tier multicomputer.

Interconnection Network

Figure 1.2: A multi-tier multicomputer.

ding a coherent address space and can exploit fast fine-grain communication through the shared memory system. Multiple SMPs communicate over a much slower interconnection network, typically via message-passing.

SMP clusters present a different set of performance tradeoffs than single-tier multicomputers. We have found that in many cases, a program that explicitly accounts for these tradeoffs will outperform a program that neglects to account for the multi-tier computer architecture. Multi-tier software which explicitly manages an SMP cluster’s two levels of parallelism and locality may outperform software
designed for the single-tier multicomputer model. In order to best utilize the multi-tier hardware, we must design programming models, algorithms, and implementation techniques that maximize performance on the target hardware, while still minimizing software complexity wherever possible. Previous single-tier parallel programming models such as MPI and HPF do not address these issues.

This dissertation introduces a programming model for multi-tier multicomputers and address implementation techniques and new algorithms tailored to the multi-tier architectural model. Together these techniques present a methodology for building efficient multi-tier programs that hides considerable detail while still delivering high performance.

We restrict our attention to a limited but important context: block-structured scientific calculations on clusters of space-shared, loosely-coupled SMPs which communicate via message-passing. We consider only dedicated hardware; that is, we focus on the performance of a single code running on reserved hardware, without interference from external users. With this limitation, we need not deal with scheduling decisions and resource contention between multiple independent parallel jobs [62, 125, 57].

Block-structured applications encompass those codes which give rise to (possibly multidimensional) rectangular, block data or numerical structures. This class of calculations includes numerical techniques such as finite difference methods for partial differential equations, Fast Fourier Transforms, and blocked algorithms for dense linear algebra.

We assume that SMP nodes communicate via message-passing, and that message-passing incurs high software overheads. We do not assume fast shared-memory access between nodes (eg. [94, 77, 115, 133, 28]), low-overhead messages [136, 107, 33] or an efficient software-based distributed-shared memory system [88, 40, 60]. We make no claims regarding the utility, performance, or economic feasibility of these specialized solutions. Indeed, tightly-coupled scalable shared memory systems may well pervade the high-performance computer market in coming years. However, whether or not tightly-coupled shared memory solu-
tions succeed, less specialized hardware solutions such as we consider will remain prevalent for several years to come.

Our techniques apply to technology widely available today: commodity networks of multiprocessor workstations, communicating with public-domain message-passing software over a commodity network. This hardware scenario hampers efficient execution for many codes, since inter-node communication may be quite expensive relative to local floating-point computation. Conceding to this reality, we must restrict our attention to algorithms with reasonably coarse-grain structures. However, applications which perform well in this “worst case” scenario should also perform well on more specialized, tightly-coupled hardware platforms.

1.2 SMP Cluster Hardware Characteristics

In order to maximize performance, we must consider the important hardware characteristics when designing a programming methodology. Several architectural features distinguish current SMP clusters from the previous generation of distributed-memory multicomputers. We briefly highlight two characteristics of SMP clusters that play a large role in application performance: inter-node communication and memory system contention.

1.2.1 Inter-node Communication

For many scientific applications, inter-node communication costs dictate the attainable efficiency of a parallel code.

One way to quantify communication costs considers the potential floating-point work that could be done in the time to communicate a message. Table 1.1 shows peak hardware rates and observed message-passing performance for several single-tier multicomputers. The message-passing results were obtained with a simple ping-pong MPI [102] program. For all three architectures, a message start takes the same time as several thousand FLOPS at peak speed. Long messages cost between one and eight peak FLOPs per byte transmitted.
Table 1.1: Characteristics of some single-tier multicomputers. \( t_s \) is the time to send a one-byte message using MPI. \( BW \) is the peak bandwidth observed for long messages.

<table>
<thead>
<tr>
<th>Machine (Processor)</th>
<th>Clock (MHz)</th>
<th>Peak MFLOPS</th>
<th>( t_s ) (( \mu \text{s} ))</th>
<th>( BW ) (MB/s)</th>
<th>Peak FLOPS ( *t_s ) (FLOP)</th>
<th>MFLOPS ( * BW^{-1} ) (MFLOP/MB)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Intel Paragon (i860XP)</td>
<td>50</td>
<td>75</td>
<td>51</td>
<td>72</td>
<td>3800</td>
<td>1.0</td>
</tr>
<tr>
<td>IBM SP-2 (POWER2)</td>
<td>67</td>
<td>267</td>
<td>52</td>
<td>34</td>
<td>14000</td>
<td>7.8</td>
</tr>
<tr>
<td>Cray T3E (Alpha 21164)</td>
<td>300</td>
<td>600</td>
<td>33</td>
<td>148</td>
<td>19800</td>
<td>4.1</td>
</tr>
</tbody>
</table>

Although communication costs are high on traditional multicomputers, hardware trends indicate that the relative cost of communication will be even worse on the next generation of SMP clusters. Table 1.2 shows the relative cost of communication for three SMP clusters. Comparing with Table 1.1, we see that the relative cost of communication exceeds the costs on single-tier multicomputers by roughly 1-2 orders of magnitude. In all cases shown, the overhead for starting a message costs more than 800,000 peak flops on an SMP node.

One oft-proposed strategy for masking communication costs is to overlap communication and computation. If communication can occur concurrently with computation, the program can keep processors busy with other activities while waiting for communication to complete. Culler et al. have proposed modeling communication delay from two sources: overhead and latency [53]. Overhead accounts for the time a processor spends executing instructions on behalf of communication activity. This includes buffer-packing and message-passing protocol activities. Latency describes the transmission delay across the network.

A single-processor multicomputer node can overlap latency with local computation, but cannot hide the message overhead. Unfortunately, overhead of-
<table>
<thead>
<tr>
<th>Machine</th>
<th>Processor</th>
<th>Clock (MHz)</th>
<th>Peak MFLOPS</th>
<th>( t_s ) (( \mu s ))</th>
<th>( BW ) (MB/s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>UMD Alpha Farm</td>
<td>Alpha 21064A(4)</td>
<td>275</td>
<td>275</td>
<td>781</td>
<td>6.30</td>
</tr>
<tr>
<td>Oregon State SparcStation Cluster</td>
<td>SuperSparc(4)</td>
<td>50</td>
<td>50</td>
<td>957</td>
<td>1.03</td>
</tr>
<tr>
<td>LLNL Alpha8400 Cluster</td>
<td>Alpha 21164(4)</td>
<td>300</td>
<td>600</td>
<td>408</td>
<td>8.88</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Machine</th>
<th>Peak FLOPS * ( t_s ) (FLOP)</th>
<th>Peak MFLOPS * ( BW^{-1} ) (MFLOP/MB)</th>
</tr>
</thead>
<tbody>
<tr>
<td>UMD Alpha Farm</td>
<td>( 2.15 \times 10^3 ) (8.6 \times 10^5)</td>
<td>44 (176)</td>
</tr>
<tr>
<td>Oregon State SparcStation Cluster</td>
<td>( 4.79 \times 10^4 ) (1.91 \times 10^5)</td>
<td>48 (192)</td>
</tr>
<tr>
<td>LLNL Alpha8400 Cluster</td>
<td>( 2.5 \times 10^6 ) (10^6)</td>
<td>68 (270)</td>
</tr>
</tbody>
</table>

Table 1.2: Characteristics of some SMP clusters. \( t_s \) is the time to send a one-byte message between nodes using MPI. \( BW \) is the peak bandwidth observed for long messages. FLOPS results in parentheses are per-node, others are per-processor.

ten causes most of the delay for coarse-grain structured applications [98]. This factor will severely limit the utility of communication overlap for single-tier multiprocessors without a dedicated message coprocessor.

Fortunately, the SMP node architecture naturally allows overlap of communication and computation. On an SMP node, both communication latency and overhead can be masked by devoting one processor to communication processing while others compute. Of course, implementing this strategy requires a programming model that can express overlap of communication and computation, as well as careful algorithm design and implementation.
1.2.2 Memory System Contention

In the single-tier multicomputer model, only one instruction stream runs on a node at a time. While that stream runs, it has dedicated access to all resources at that node. Other processors cannot impact the node’s performance, unless the node explicitly accesses a shared resource such as the network or disk.

SMP architectures introduce a new source of contention: the shared memory subsystem. Even if each instruction stream grabs a dedicated processor, all processors compete for access to the shared memory bus and memory banks. Additionally, overlapping communication and computation may contribute to memory system contention. Since off-node communication traffic may access shared memory modules and traverse a shared system bus, overlapped message-passing may slow down memory accesses by other processors at an SMP.

A shared memory programming style introduces several other factors. Shared memory allows different processors to cache the same location, leading to possible problems with cache-line invalidation patterns and false sharing. Additionally, the implementation must deal with potentially expensive synchronization mechanisms between processes at an SMP node.

1.3 Software Infrastructure

When implementing a high-performance parallel code, the programmer faces myriad tradeoffs concerning both low-level implementation details and high-level algorithmic decisions. To ease the programmer’s burden, programming languages and libraries can hide many low-level details of a parallel implementation (e.g. [78, 92, 54, 3, 45, 43, 110, 52, 44, 85, 97, 16]). A useful set of programming abstractions allows the programmer to express efficient algorithms at a high level, while the software infrastructure manages tedious low-level details.

Many software systems provide specific support for block-structured scientific calculations. Block structures arise in many scientific applications ranging from finite difference methods for partial differential equations [35] to blocked al-
gorithms for numerical linear algebra [48]. In these applications, computational structures arise as uniform rectangular arrays of data, which communicate in potentially irregular patterns.

Although many software systems support block-structured calculations on distributed-memory and shared-memory single-tier multicomputers, relatively few efforts have targeted multi-tier architectures such as SMP clusters. We present KeLP (Kernel Lattice Parallelism) a small set of programming abstractions to simplify implementation of efficient algorithms for block-structured scientific calculations on SMP clusters.

KeLP provides two types of objects: *meta-data* and *instantiator* objects. Meta-data objects represent the abstract structure of a calculation. Instantiator objects instantiate the structure represented by the meta-data objects.

KeLP objects help the programmer manage three types of parallel activity: data layout, data motion, and parallel control flow. KeLP’s parallel control flow objects provide three levels of control, corresponding to three levels of an SMP cluster’s memory hierarchy. KeLP’s data layout abstractions build on the *structural abstraction* programming methodology introduced in the LPARX programming system [92]. The data motion abstractions combine ideas from structural abstraction and inspector/executor communication analysis [3] with asynchronous execution in the multi-level control flow.

With the KeLP programming model, this dissertation contributes a new methodology for programming SMP clusters. KeLP explicitly exposes two levels of parallelism, locality, and data motion to the application programmer. Although the KeLP programmer must consciously attend to high-level algorithmic decisions, KeLP provides intuitive, concise abstractions to help the programmer implement efficient algorithmic decisions. We claim that KeLP’s abstractions provide an intuitive programming interface, while still allowing an efficient implementation. The KeLP programmer can express a wide variety of efficient and complex algorithmic policies, while the KeLP system handles most of the tedious and error-prone low-level details.
1.4 Application Studies

To evaluate the KeLP programming model, the dissertation examines six block-structured codes in detail. Three of the codes implement finite difference methods: a simple regular Poisson solver, the NAS multigrid benchmark [15], and a multi-block multigrid solver over an irregular domain. A fourth code, the NAS Fast Fourier transform benchmark [15], solves a 3D diffusion equation with FFTs. Finally, we consider two blocked dense linear algebra codes: SUMMA matrix multiplication [135] and ScaLAPACK’s right-looking blocked dense LU factorization algorithm [49].

Each code raises distinct issues regarding the KeLP programming model. For each code, we examine algorithmic techniques to improve performance on SMP clusters. In particular, we restructure each algorithm to explicitly overlap communication and computation. For each case, we evaluate the utility of KeLP primitives in expressing the basic and restructured algorithms.

With these codes, we present a experimental performance study to evaluate the efficiency of the KeLP implementation. We compare each code against single-tier message-passing implementations, to quantify performance benefits of the multi-tier KeLP programming model. We also address the efficacy of restructured algorithms to overlap communication and computation, and analyze hardware characteristics which limit application performance.

The results show that in most cases, multi-tier software implemented with KeLP matches or exceeds performance of equivalent naive MPI implementations. The KeLP abstractions match a wide range of algorithmic structures, including new algorithms designed for the multi-tier hardware. The experiences show that many implementation issues such as communication, scheduling, and synchronization may limit application performance if managed incorrectly. However, we demonstrate implementation techniques that successfully overcome or bypass many critical performance problems.
1.5 Organization of the Dissertation

The dissertation presents and evaluates a multi-tier programming model for block-structured applications on SMP clusters.

Chapter 2 presents the multi-tier KeLP programming model. The Chapter presents the KeLP programming abstractions to manage parallel control, storage, data motion, and other program aspects. The Chapter includes short programming examples that illustrate crucial aspects of the KeLP model.

Chapter 3 discusses the implementation of KeLP as a C++ class library, and discusses performance tradeoffs and limitations of the target hardware.

Chapter 4 presents detailed application studies using six KeLP applications. For each code, we present a restructured algorithm to overlap communication and computation.

Chapter 5 concludes the dissertation and suggests avenues for future investigation.
Chapter 2

Programming Abstractions

2.1 Introduction

We have developed Kernel Lattice Parallelism (KeLP), a small set of programming abstractions to simplify efficient implementation of block-structured scientific calculations on SMP clusters. KeLP provides mechanisms to help the programmer coordinate data layout, data motion, and parallel control flow with intuitive, high-level abstractions. The KeLP run-time system manages the tedious low-level implementation details such as message-passing, processes, threads, synchronization, and memory allocation.

Many parallel programming models hide even high-level details of parallelism and locality from the programmer. For example, data-parallel languages such as HPF [78] perform all inter-processor data motion implicitly, as dictated by the code’s data dependencies and the distributed array layouts. With this programming model, the programmer need not explicitly manage data motion, but instead leaves all data motion decisions to the compiler. While this model relieves the programmer from much responsibility, it also limits the programmer’s power to express implementation decisions. For example, in HPF, the programmer cannot explicitly express the overlap communication and computation. Instead, the data-parallel compiler must perform heroic program analysis to determine where,
when, and how to overlap communication and computation.

The KeLP model offers a different philosophy. KeLP explicitly exposes parallelism, locality, and data motion decisions to the programmer. The KeLP programmer assumes the power and responsibility to manage these high-level implementation decisions. The power of KeLP lies in intuitive, concise abstractions that represent the program structure. KeLP will not analyze program source code to make high-level restructuring and algorithmic decisions. The programmer must make these decisions. However, the KeLP primitives allow the programmer to implement the decisions with modest programming effort, as compared to lower-level primitives such as message-passing and threads.

KeLP supports block-structured scientific calculations, in which the underlying data structures have a rectangular geometry. For example, finite difference methods for the solution of partial differential equations carry an underlying structured finite difference mesh. Efficient algorithms for dense linear algebra often exhibit a blocked structure [7]. KeLP abstractions help manage the data structures, computational domains, and data motion for these applications.

KeLP also supports irregular and dynamic block-structured applications. For example, multiblock methods for PDEs operate over a collection of structured grids with potentially irregular couplings. Structured adaptive mesh refinement algorithms carry an irregular collection of locally structured grids. These applications entail dynamic and irregular communication between grids that arise at run-time and defy compile-time analysis.

To support these potentially complex and dynamic application structures, KeLP builds on structural abstraction, a programming methodology introduced in the LPARX programming system [92]. Under structural abstraction, first-class language objects represent the geometric structure of a block-structured parallel code. LPARX introduced structural abstraction objects to represent potentially irregularly block-structured data sets. KeLP inherits this facility from LPARX. Additionally, KeLP combines structural abstraction with an inspector/executor communication model [3] to manage data motion more efficiently. KeLP’s data
motion objects support asynchronous execution, providing a simple mechanism to overlap inter-node communication with other operations. KeLP also extends structural abstraction to coordination of several levels of parallel control.

To maximize performance on SMP clusters, many codes will exploit fast shared-memory hardware within each SMP node, while managing inter-node communication via message-passing. The KeLP programming model reflects this strategy. A complete multi-tier KeLP program contains three program levels: a collective level which executes on the entire machine, a node level that manages parallelism between SMP nodes, and a processor level that manages parallelism within a single SMP node. KeLP abstractions help manage each level independently where desired, and also help manage interaction between the levels where necessary.

When first designing a parallel implementation strategy, managing interactions between three parallel program levels may appear to be a daunting task. To address this concern, KeLP supports a step-by-step migration path to a multi-tier parallel implementation. The recommended migration path appears in Figure 2.1. First, the programmer builds a single-tier parallel code using KeLP abstractions. Then, the programmer can selectively parallelize individual sub-routines using a second level of parallel constructs. Finally, the programmer may choose to optimize the code by overlapping activities at both levels simultaneously.

The remainder of this Chapter proceeds as follows. Section 2.2 presents the KeLP programming abstractions. Section 2.3 presents some simple programming examples to illustrate the programming model. Section 2.4 discusses related work, and Section 2.5 analyzes the Chapter's contents and suggests directions for future work.

2.2 KeLP Programming Abstractions

KeLP provides a small set of objects to help the programmer coordinate parallel applications. All KeLP objects exist as first-class language objects, cre-
Figure 2.1: Software migration path to an optimized multi-tier parallel code.

ated at run-time. The KeLP abstractions fall into two categories: meta-data and instantiator. KeLP meta-data objects represent the abstract structure of some facet of the calculation. Instantiator objects instantiate program behavior based on information contained in meta-data objects.

Table 2.1 lists the KeLP programming abstractions. The following Sections present the abstractions in more detail.

2.2.1 Meta-Data Abstractions

KeLP provides five meta-data abstractions, the Point, Region, Map, FloorPlan, and MotionPlan. KeLP 2.0 implements each of these abstractions as a first-class C++ object. LPARX introduced the Point and Region abstractions; the others are new to KeLP.

The Point consists of a d-tuple of integers, representing a point in d-dimensional coordinate index space. For a Point P, P(i) denotes the ith element of the tuple. As in LPARX, KeLP provides element-wise arithmetic operations over Points. For example, for Points P1 and P2, P1 + P2 = (P1(0) + P2(0), P1(1) +
Table 2.1: A brief synopsis of the KeLP data types.

\[ P_2(1) \ldots P_1(d) + P_2(d) \]. Element-wise subtraction, multiplication and division are defined analogously.

The Region represents a rectangular subset of \( Z^n \); i.e., the Region represents a regular section with stride one. Thus, a pair of Points \([P_1, P_2]\) uniquely define a Region. For a Region \( R \), \( R.lwb \) denotes the first Point (the lower corner, \( P_1 \)), and \( R.upb \) denotes the second Point (the upper corner, \( P_2 \)). If, for \( 0 \leq i \leq d \), \( R.lwb(i) > R.upb(i) \), then we interpret \( R \) as the empty set. Extents \((R, i)\) returns the length of the Region along the \( i \)th axis; i.e., \( R.upb(i) - R.lwb(i) + 1 \).

To help the programmer manipulate Regions, KeLP inherits from LPARX a set of high-level geometric operations called the Region calculus. Three useful Region calculus operations are shift, intersect, and grow. For a Region \( R \) and a Point \( P \), \( shift(R, P) \) denotes \( R \) translated by the vector \( P \): \([R.lwb + P, R.upb + P]\). Intersection between two Regions is defined in the usual geometric way. Given a Point \( g \), \( grow(R, g) \) returns the Region that is equivalent to \( R \) padded with \( g(i) \) cells in the \( i \)th direction. If \( g(i) < 0 \), the Region is shrunk in the obvious way. Note
Figure 2.2: The KeLP FloorPlan stores an array of Regions, each mapped to an integer.

that all Region operations can be implemented efficiently with a small number of integer operations, and Regions are closed under shift, intersection, and grow.

The Map class implements a function $Map : \{0, \ldots, k-1\} \rightarrow Z$, for some integer $k$. For $0 \leq i < k$, and a Map $M$, $M(i)$ returns some integer. For $i \geq k$, $M(i)$ is undefined. The programmer can construct a Map by defining the function one value at a time with the setOwner operation; $M.setOwner(i, j)$ modifies the Map so $M(i) = j$. The Map forms the basis for node and processor assignments in KeLP partitioning.

The FloorPlan consists of a Map $M$ of size $k$ along with an Array of $k$ Regions. For a FloorPlan $F$, $F(i)$ denotes the $i$th Region of $F$, and $F.owner(i)$ denotes $M(i)$. The programmer may construct and manipulate FloorPlans one element at a time; $F.setRegion(i, R)$ sets $F(i) = R$ and $F.setOwner(i, j)$ sets $F.owner(i) = j$. The KeLP programmer can use the FloorPlan in a variety of contexts. For example, the FloorPlan can represent an irregular block data decomposition. Figure 2.2 shows a FloorPlan that represents an irregular partitioning of a rectangular domain into five Regions. Alternatively, a FloorPlan can represent distribution of work among processors of a single SMP. Yet another example is to use the FloorPlan to represent a section of a distributed array, which proves useful for block dense linear algebra algorithms.

The MotionPlan implements a first-class, user-level block communication schedule [3]. A MotionPlan is defined by a List of 4-tuples $< F, i, T, j >$, where
\( F \) and \( T \) are Regions, and \( i \) and \( j \) are integers. The programmer builds and manipulates MotionPlans one tuple at a time. Intuitively, each tuple represents a block data motion operation, discussed in more detail shortly in Section 2.2.4.

### 2.2.2 Control Flow Abstractions

Many previous programming models for parallel computing can be characterized as having two levels of control flow: a `collective` level, and a `node` level. For example, a SPMD MPI [102] program contains a number of collective operations, such as reductions, barriers and broadcasts, interspersed within a node level program. The node-level instructions form separate threads of control which execute independently. An HPF [78] program consists of a single flow of control, the collective level. However, the HPF LOCAL extrinsic environment allows an HPF program to drop into SPMD control flow. In other words, the program breaks into node-level control when it enters an extrinsic environment. Snyder clearly articulates the two-level approach in the XYZ program levels of the Phase Abstractions programming model [124].

The two-level control flow model matches single-tier architectural models, such as the Candidate Type Architecture [123], BSP [134], and LogP [53]. These architectural models do not represent the two levels of locality and parallelism of SMP clusters.

In contrast to the two levels of MPI and HPF, the multi-tier KeLP abstractions support three levels of control: a `collective` level, a `node` level, and a `processor` level. The collective level manages data layout and data motion among SMP nodes. The node level control stream manages activities at a single SMP node. Node-level control performs serial computation at an SMP node, and performs collective operations which apply only to the multiple processors at a single SMP node. In particular, the node-level control stream often controls partitioning and scheduling decisions for the multiple processors at the SMP node. The processor-level control stream executes a serial instruction stream on a single physical processor. In most cases, the processor level will invoke highly tuned serial
numeric kernels, such as the BLAS [56] or optimized Fortran and C code.

The KeLP program starts in the collective level, and descends to the node level and to the processor level through two KeLP constructs: the nodeIterator and procIterator. These iterators interpret the assignment of parallel loop iterations as stored in a Map meta-data object.

Given a Map $M$, a nodeIterator executes the $i$th loop iteration on node $M(i)$. Thus, each loop iteration forms an independent node-level instruction stream. From the node-level stream, the program descends to the processor level via a procIterator. Given a Map $M$, a procIterator executes the $i$th loop iteration on processor $M(i)$. Each procIterator iteration executes serially a single processor. At the processor level, the programmer may invoke numerical kernels in an extrinsic sequential language such as Fortran.

KeLP enforces an implicit barrier synchronization point at the end of a procIterator or nodeIterator loop. The procIterator synchronization point enforces a logical barrier among all processors at a node. The nodeIterator logically synchronizes all SMP nodes. A clever implementation may eliminate the barriers to improve performance. The KeLP 2.0 implementation enforces the procIterator barrier, but eliminates the nodeIterator barrier with inspector/executor communication analysis.

Note that KeLP structured loops contrast sharply with unstructured thread programming, where the programmer must explicitly manage synchronization between individual threads. CC++ [44] provides a programming model with both types of parallel control constructs. Like the CC++ structured parallel loops, the KeLP iterators simplify the expression of parallelism, but restrict the forms of parallel control flow available to the programmer.

2.2.3 Storage Model

The Point, Region, Map, FloorPlan, and MotionPlan meta-data may live at any of the three levels of control flow. For example, a program may create and write to a FloorPlan at either the collective level, node level, or processor
level. Meta-data can pass through the levels from the top down. For example, a FloorPlan written at the collective level may be read at the node level or processor level. However, meta-data cannot pass up the program levels. For example, the contents of a FloorPlan written at the processor program level are undefined at the node and collective levels.

The KeLP meta-data objects describe only the structure of the KeLP program. The KeLP Grid object holds the actual data of the application. A Grid is an array of objects of some type $T$, whose index space is a Region. For example, the Fortran array `real A(3:7)` corresponds to a one-dimensional Grid $A$ of `real` with $\text{region}(A) = [3,7]$.

While KeLP meta-data exists at all three program levels, KeLP Grid data exists only at the node level. A Grid lives in exactly one node's address space; a single Grid is not distributed across multiple nodes. The program can access the Grid data from the node-level instruction stream at that node, or from processor-level instruction streams nested at that node. The program cannot access Grid data from the collective level.

An XArray is an array of Grids, whose structure is represented by a FloorPlan. An XArray is a collective object; it must be created and modified from the collective program level. For an XArray $X$, $X(i)$ denotes the $i$th Grid in $X$. The Grids in an XArray may span arbitrary Regions of index space and even overlap. All Grids in a XArray must have the same dimensionality.

### 2.2.4 Data Motion

KeLP moves data between Grids via the MotionPlan and Mover classes. The MotionPlan implements a block communication schedule, similar to those incorporated in Multiblock PARTI [3]. However, whereas Multiblock PARTI supported only a small fixed number of regular communication patterns, KeLP utilizes structural abstraction to support more general block motion patterns.

A MotionPlan consists of a List of 4-tuples $< F, i, T, j >$, where $F$ and $T$ are Regions, and $i$ and $j$ are integers. The programmer builds a MotionPlan, one
Figure 2.3: The KeLP MotionPlan represents a set of block copy operations.

tuple at a time, using the copy operation. For a MotionPlan \( M \), \( M.copy(F, i, T, j) \) adds the tuple \(< F, i, T, j >\) to \( M \). Regions \( F \) and \( T \) must have the same size. Intuitively, this tuple represents the data motion “copy from Region \( F \) of Grid \( i \) into Region \( T \) of Grid \( j \)”. Figure 2.3 illustrates one possible MotionPlan as applied to a set of Grids.

In many cases, a specialized form of copy proves useful: \( copyOnIntersection \). Given two FloorPlans \( A \) and \( B \), two Regions, \( F \) and \( T \), and two integers \( i \) and \( j \), \( M.copyOnIntersection(A, F, i, B, T, j) \) adds the tuple \(< A(i) \cap F, i, B(j) \cap T, j >\) to \( M \).

In LPARX, Kohn demonstrated that using the Region calculus, the programmer can express a variety of communication patterns [92]. The same concise algorithms apply to the construction of KeLP MotionPlans. For example, many finite difference calculations fill in ghost cells with data from logically overlapping grids. Figure 2.4 shows the KeLP pseudo-code to build a MotionPlan to fill in ghost cells for a finite-difference stencil. Another common pattern fills in ghost cells across periodic boundary conditions. Figure 2.5 shows KeLP pseudo-code to
generate a MotionPlan to satisfy periodic boundary conditions along one dimension of an irregularly partitioned XArray.

Since the MotionPlan is a first-class object, the programmer can compose and manipulate MotionPlans to construct complex data motion patterns from simple building blocks. For example, many finite difference codes require ghost cells from both logically overlapping Grids and across periodic boundary conditions. The programmer can construct this pattern by composing the two MotionPlans constructed by Figures 2.4 and 2.5. With this mechanism, one may define a library of pre-defined MotionPlan generators which serve as building blocks for more complicated patterns. MotionPlan composition may also give better performance, since the composed MotionPlan may expose more opportunities for optimizations like message aggregation.

The first-class representation of data motion pattern has other advantages. The programmer can query a MotionPlan to examine its contents. Since the MotionPlan holds information about the program’s data dependencies, this information can be useful for making algorithmic decisions at run-time. Additionally, the programmer can modify a MotionPlan, one tuple at a time. With this facility, the programmer can optimize extant MotionPlans, as described in [64].

Note that a MotionPlan holds only a description of a data motion pattern. Building a MotionPlan does not move any data. That task falls to the Mover.

The Mover, a first-class executor [3] object, performs the data motion represented by a MotionPlan as a collective operation. The Mover object analyzes a MotionPlan and performs memory-to-memory copies and message-passing to effect the data motion pattern. As discussed in the next Chapter, the implementation can exploit the information in the MotionPlan to perform several communication optimizations.

Even with optimizations, we have assumed that slow inter-node communication presents a formidable obstacle to good performance. To achieve the best performance, a program must tolerate inevitable inter-node communication overhead and latency. To this end, KeLP Movers provide asynchronous execution. For
BuildFillGhostPattern(\textbf{XArray} X, \textbf{MotionPlan} M)
\begin{algorithmic}
\State \textbf{begin}
\For{each $i \in X$}
\State \textbf{Region} $I = grow(X(i), -1)$
\For{each $j \in X$}
\If{$(i \neq j)$}
\State \textbf{Region} $R = I \cap X(j)$
\State $M.\textbf{Copy}(X, i, R, X, j, R)$
\EndIf
\EndFor
\EndFor
\End
\end{algorithmic}

a)

b)

Figure 2.4: a) Pseudocode to generate a \textbf{fillpatch} MotionPlan $M$ to fill in ghost cells for \textbf{XArray} $x$. b) The dark shaded regions represent ghost regions that are copied into the central Grid.
BuildPeriodic(XArray X, MotionPlan M, Region Clip)
begin
for each $i \in X$
    Region $\text{Ghost}_R = X(i) \cap \text{shift}(\text{Clip}, [\text{extents}(\text{Clip}, 1), 0])$
    Region $\text{Ghost}_R = \text{shift}(\text{Ghost}_R, [-\text{extents}(\text{Clip}, 1), 0])$
for each $j \in X$
    if ($i \neq j$) {
        Region $R_D = I \cap \text{Ghost}_R$
        Region $R_S = \text{shift}(R_D, [\text{extents}(\text{Clip}, 1), 0])$
        M.Copy(X, j, R_S, X, i, R_D)
    }
end for
end for
end

Figure 2.5: KeLP pseudo-code to generate a MotionPlan that fills in periodic boundary conditions along one dimension.
a Mover $M$, $M$.start() begins asynchronous execution of a data motion pattern. 
$M$.wait() blocks until the data motion pattern completes. Individual nodes or processors may block on $M$.wait(), while other nodes or processors continue execution. Thus, the programmer may selectively block individual nodes or processors as dictated by the data dependencies of the application.

Since the Mover is a self-contained object, independent of the other KeLP classes, the programmer can extend and replace Mover objects to realize different communication operations. Using object inheritance, the programmer can derive a specialized objects by composing Movers or overriding a Mover's virtual functions. From a software engineering viewpoint, the ability to customize Movers provides a convenient interface to encapsulate communication that accompanies a numeric operator.

### 2.2.5 Other Collective Operations

Multi-tier KeLP also implements barriers, reductions, and broadcasts across and within SMP nodes.

KeLP provides two barrier primitives: one at the collective level and one at the node level. The node-level barrier synchronizes all processors at a single SMP node. The collective-level barrier performs a node-level barrier on each node, and then synchronizes all nodes in the cluster. KeLP barriers do not apply to asynchronous data motion by Movers; these operations will continue past a barrier.

Recall that meta-data, including stack variables, have either collective, node, or processor-level values. The broadcast transforms a variable from a node or processor level value to a collective value. For example, suppose $i$, an integer, has a processor-level value. That is, the value of $i$ was assigned inside a procIterator loop. Then the collective operation $\text{broadcast}(i, n, p)$ changes $i$ to be a collective value, using the value of $i$ as assigned at node $n$, processor $p$.

Reductions perform collective reduction operations analogously. KeLP provides multi-level reductions corresponding to the MPI reduce operations for standard data types. Additionally, the programmer can define custom reduction
operations, similar to the process provided by MPI.

2.3 Programming Example

To illustrate KeLP’s multi-tier programming constructs, we briefly discuss the implementation of redblack3D, a simple regular finite difference code. The program solves Poisson’s equation in three dimensions:

\[ \nabla^2 u = f \quad (2.1) \]

where \( u(x, y, z) \) is the unknown variable and \( f(x, y, z) \) is fixed. Redblack3D solves this equation over the unit cube, using Dirichlet boundary conditions on the six faces. We discretize the domain onto a structured finite difference mesh with \( N^3 \) points, and solve the equation by the iterative application of a second-order seven-point stencil relaxation.

We start by illustrating a single-tier KeLP implementation, appropriate for a distributed memory computer with uniprocessor nodes. Then, we show how to modify the code with KeLP constructs to build a multi-tier code. Finally, we optimize the code to overlap communication and computation for better performance.

For illustrative purposes, the dissertation’s code examples use an abstract KeLP syntax. Actual KeLP 2.0 C++ code is similar, but includes more detailed type names to specify dimensionality and parameterized types. For example, in the code examples, the KeLP statement

\[ \text{XArray X} \]

corresponds to the following actual KeLP 2.0 C++ syntax:

\[ \text{XArray2<Grid2<double>} > X; \]

For clarity, in a few cases, the dissertation’s code examples substitute English descriptions for simple operations.
main()
begin
  /* describe data partitioning */
  Region domain = [1 : N, 1 : N, 1 : N]
  Floorplan T = blockPartition(domain)
  for each i ∈ T, T.setRegion(i, grow(T(i), 1))
  /* instantiate storage */
  XArray u(T)
  XArray rhs(T)
  /* determine communication pattern */
  MotionPlan M
  BuildFillGhostPattern(u, M)
  /* solve the problem */
  integer rb = 0
  while (not converged) {
    Relax(u, rhs, M, rb)
    rb = (rb + 1) mod 2
  }
end

Figure 2.6: Main procedure for redblack3D example.

2.3.1 Single-tier code

As a first step towards building an optimized multi-tier redblack3D code, we first discuss the implementation of a single-tier parallel code. This first version will manage only one level of parallelism and locality.

We follow the usual SPMD implementation which employs a BLOCK data decomposition and carries additional ghost cells to buffer off-processor data. Each relaxation consists of two steps: (1) communicate with nearest neighbors to exchange ghost cell values, and (2) independently relax on the local portion of the global mesh.

Figure 2.6 shows the main routine for redblack3D. The main procedure
starts in the collective program level; there is a single logical thread of control. All nodes collectively execute statements in the main procedure.

First, we must set up distributed data structures to hold the $N \times N \times N$ finite difference grid. Before actually creating any storage, we use the Region and FloorPlan meta-data objects to describe the storage. We describe the global computational domain with a Region:

$$\textbf{Region} \ domain = [1 : N, 1 : N, 1 : N]$$

That is, we create a Region object called $domain$ that describes an $N \times N \times N$ box of indices.

Next, we must set up a FloorPlan to hold the structure of the distributed grids. For this example, we elide the details and assume the existence of a standard partitioning library. The library includes a subroutine $\text{blockPartition}$ which distributes a Region in HPF-style blocks over the nodes. Using this subroutine, we generate a FloorPlan that describes the block partitioning of Region $domain$ over the nodes:

$$\textbf{FloorPlan} \ T = \text{blockPartition}(domain)$$

The KeLP distribution includes a library called DOCK (Decomposition Object Collection for KeLP) that implements regular block partitioning. DOCK provides meta-data objects, derived from Regions and FloorPlans, to represent HPF-style block partitioning. In particular, the DOCK $\text{ Decomposition}$ object, derived from FloorPlan, automatically decomposes a Region into a block partitioning and maps the blocks across SMP nodes.

Following the usual parallel implementation strategy, we will pad each node’s local grid with a layer of ghost cells to buffer off-node data. To represent this change, we modify each element of the FloorPlan $T$, growing the Regions by one element in each direction. Recall that $T(i)$, the $i$th element of FloorPlan $T$, is a Region. To add a one-cell layer of ghost cells to $T(i)$, we invoke the Region calculus $\text{grow}$ operation. Thus,
\texttt{T.setRegion(i,\texttt{grow(T(i),1))}}

adds space for ghost cells to FloorPlan element $T(i)$.

Now that FloorPlan $T$ is properly initialized, we instantiate the actual storage by creating two XArrays, $u$ and $rhs$. We intend $u$ to hold the computed solution, and $rhs$ to hold the right-hand side vector for the Poisson equation. The XArray constructor

\texttt{XArray u(T)}

creates an XArray with one Grid for each element of $T$. The $i$th Grid of $u$, $u(i)$, has the domain $T(i)$ and exists on node $T.\text{owner}(i)$.

After instantiating storage, we set up the initial values for $u$ and $rhs$. This process is straightforward and not shown. Next, we will set up a MotionPlan $M$ to describe the data motion needed in this algorithm. As discussed earlier, the data motion pattern in this calculation must fill in ghost cells between adjacent grids. We have already described the algorithm to build this MotionPlan in Figure 2.4. Thus, we call the subroutine of Figure 2.4 to build the MotionPlan.

Finally, we iterate until convergence, relaxing the numerical solution in each iteration. Figure 2.7 shows single-tier KeLP subroutine \texttt{Relax} which implements the relaxation. This code works when limited to one processor per node.

The main procedure calls \texttt{Relax} from collective control flow, so \texttt{Relax} begins execution from collective control flow. Before performing numerical relaxation, the nodes must communicate to update ghost cell values on each node. Recall that we computed this communication pattern and stored it in MotionPlan $M$.

We create a Mover object to carry out the data motion pattern specified by $M$. The statement

\texttt{Mover mov(F, T, M)}

creates a Mover object $mov$. The constructor arguments specify $mov$ should copy data from XArray $F$, to XArray $T$, according to the pattern stored in MotionPlan.
Relax($\text{XArray} \ X$, $\text{XArray} \ rhs$, $\text{MotionPlan} \ M$, integer $rb$)

begin
    Mover mov($X$, $X$, $M$)
    mov.start()
    mov.wait()
    for (nodeIterator $ni(X)$; $ni$; $++ni$) {
        integer $n = ni()$
        serialRelax($X(n)$, $rhs(n)$, $X$.region($n$), $rb$)
    } 
end

Figure 2.7: Single-tier KeLP code for redblack3D relaxation.

$M$. In this example, we move data between different Regions of the same XArray $X$. Thus, in the example, $F = T = X$.

The Mover start call begins the data motion pattern asynchronously. In this first example, we do not attempt to overlap communication and computation. So, we immediately call the Mover wait call to block until the Mover’s data motion completes.

Having completed the ghost cell data motion, the program will now perform the serial relaxation kernel on each node. Each node will drop down to serial Fortran 77 to perform local relaxation. In order to perform local work, the program must drop from the collective control level to node-level control. This happens via the KeLP nodeIterator loop:

        for (nodeIterator $ni(X)$; $ni$; $++ni$) {

The nodeIterator constructor takes a Map, which specifies the number of loop iterations and the mapping of these iterations to nodes. In the example, we use the XArray $X$ as a Map (XArray is derived from Map). NodeIterator $ni$ will create one loop iteration for each element of $X$. Furthermore, $ni$ causes loop iteration $n$ to execute only on node $X.owner(n)$. If $X$ happens to assign more than one iteration to a node, then the loop execution order preserves loop-carried data dependencies.
That is, if $X.\text{owner}(i) = X.\text{owner}(j)$ and $i < j$, then iteration $i$ will complete before iteration $j$ starts.

Inside the loop, the program runs in node-level control, with one stream per SMP node. First, the program queries the nodeIterator to determine the number of the current iteration. Then, the program calls \texttt{serialRelax}, a serial numeric kernel to perform the relaxation. In practice, \texttt{serialRelax} will often call a local numeric kernel in Fortran 77.

### 2.3.2 Multi-tier code

The single-tier code just described will run on multicollectors with uniprocessor nodes. We can also run the single-tier code on an SMP cluster with $n$ nodes and $p$ processors per node. One possibility uses only one processor per SMP node. However, this scenario wastes the processing power of $p - 1$ processors per node. Alternatively, we can treat the SMP cluster as a “flattened” single-tier multicomputer with $np$ nodes.

The \texttt{redblack3D} code presents two immediate drawbacks for this flattened approach. First, the single-tier data structures waste storage. Using shared-memory hardware on each node, we need carry ghost cells only on a per-node basis, rather than on a per-processor basis. Secondly, the extra internal ghost cells incur extra data motion. These drawbacks may cause significant performance penalties for a code which uses high-order stencils with many layers of ghost cells. In many other scenarios, the programmer may wish to exploit shared memory hardware for faster intra-node algorithms, such as dynamic load balancing strategies between processors at a node.

The multi-tier KeLP model helps the programmer parallelize the node-level code explicitly. Figure 2.8 shows multi-tier KeLP code to implement the \texttt{Relax} subroutine with two levels of parallelism.

As in the previous example, \texttt{Relax} starts in collective control flow. The collective level updates ghost cell values by creating and invoking a Mover. Also as before, we drop to node-level control via the nodeIterator, with one iteration
Relax($\text{XArray } X$, $\text{XArray } rhs$, MotionPlan $M$, integer $rb$)
begin
    Mover mov($X, X, M$)
    mov.start()
    mov.wait()
    for (nodeIterator $ni(X)$; $ni$; $ni$) {
        integer $n = ni()$
        Floorplan $F = \text{IntranodeBlockPartition}(X.region(n))$
        for (procIterator $pi(F)$; $pi$; $pi$) {
            integer $p = pi()$
            serialRelax($X(n), rhs(n), F(p), rb$)
        }
    }
end

Figure 2.8: Multi-tier KeLP code for redblack3D relaxation.

per block of XArray $X$.

From the node-level loop, we now parallelize the relaxation on each block across the processors at an SMP node. Consider iteration $n$ of the loop, executing on SMP node $X.owner(n)$. Iteration $n$ must perform relaxation on Grid $X(n)$, which has domain $X.region(n)$. To parallelize this relaxation across the $P$ processors of SMP node $X.owner(n)$, we will partition $X.region(n)$ into $P$ blocks and assign each block to one processor.

We construct the intra-node partitioning by building a FloorPlan $F$ with $P$ elements. As before, we rely on a standard library of FloorPlan generators to break a Region into blocks. The library routine $\text{IntranodeBlockPartition}$ divides $X.region(n)$ into $P$ pieces and maps them to the processors so $F.owner(i) = i$. The KeLP DOCK library provides this functionality.

Having described the intra-node partitioning in FloorPlan $F$, we now drop to processor-level control via the procIterator. Similar to the nodeIterator described earlier, the procIterator executes one iteration for each element of $F$, and
executes iteration $i$ on processor $F.\text{owner}(i)$. The procIterator numbers processors from 0 on each node.

From the processor-level control, each iteration invokes the serial relaxation kernel independently. Each iteration $p$ passes $\text{serialRelax}$ the Region $F(p)$, which is the sub-domain of Grid $X(n)$ assigned to the iteration.

Note that in KeLP’s memory model, a Grid $G$ lives in an address space corresponding to an SMP node. The program can access the data of $G$ from node-level control or from processor-level control. Thus, if multiple procIterator iterations write to the same locations of $G$, race conditions occur. In redblack3D, each processor updates a Region of each Grid, but the Regions do not overlap, so no race conditions occur.

By default, KeLP enforces a logical synchronization point at the end of each procIterator and each nodeIterator loop. In the redblack3D code, the synchronization points ensures that all relaxation will complete before any part of the program proceeds to the next stage of the program. The programmer can relax the synchronization if desired, but must take care to ensure correctness.

### 2.3.3 Multi-tier code with communication overlap

The previous code example managed two levels of parallelism, exploiting fast shared-memory hardware within each SMP node while relying on message-passing between nodes. However, performance may still disappoint due to the high costs of message-passing. To improve performance further, we will restructure the $\text{Relax}$ subroutine to overlap communication and computation.

In order to overlap communication and computation in $\text{Relax}$, we will use the following strategy:

1. Asynchronously begin communication.
2. Perform local computation on the interior of each Grid.

---

\footnote{The current implementation enforces the intra-node synchronization with a barrier. However, no action is required to enforce a logical nodeIterator barrier, since the message-passing implementation enforces inter-node data dependencies implicitly.}
Relax($X$Array $X$, $X$Array $rhs$, MotionPlan $M$, integer $rb$)
begin

Mover mov($X$, $X$, $M$)

mov.start()

for (nodeIterator $ni(X)$; $ni$; ++$ni$) {

integer $n = ni()$

Floorplan $F = $IntranodeDepPartition($X$, region($n$))

for (procIterator $pi(F)$; $pi$; ++$pi$) {

integer $p = pi()$

integer $P = $number of processors at this SMP node

if ($p \geq P$) mov.wait()

serialRelax($X(n)$, $rhs(n)$, $F(p)$, $rb$)
}
}

end

Figure 2.9: Multi-tier KeLP code with communication overlap for redblack3D relaxation.

3. Wait for communication to complete.

4. Perform local computation on the annulus of each Grid.

Figure 2.9 shows the restructured code. As before, the collective level constructs a Mover and asynchronously starts communication. In the previous examples, the program immediately blocked on the Mover. In this case, we defer the synchronization point, and instead choose to begin local computation immediately.

Figure 2.10 illustrates the algorithm for a 2D grid, using a five-point stencil. With a five-point stencil, each grid point depends on its four neighbors. Grid points in the interior of the Grid do not depend on values from the Grid’s ghost cells. Thus, in each iteration, the program can relax on the interior points before incoming ghost cell values arrive. In contrast, the points on the annulus of the Grid depend on incoming ghost cells values for the five-point stencil, so communication must complete before the program can perform local computation.
on the annulus.

Suppose we have a two-processor SMP. Then we divide the interior of the Grid into two Regions, and assign each Region to a processor. Figure 2.10a divides the interior of the Grid into two Regions marked 0 and 1. These Regions do no depend on incoming ghost cell values, and can be updated before ghost cell communication completes. The surrounding annulus, Regions 2 through 5, must wait for incoming ghost cell values (shaded).

To manage this algorithm in KeLP, we store this dependency and partitioning information in a FloorPlan. Figure 2.10b presents the KeLP code to construct this FloorPlan for a 2D problem. For a $P$-processor SMP node, the routine generates a FloorPlan with $P + 4$ Regions. The first $P$ Regions divide the interior of the Grid among the $P$ processors. The remaining four Regions hold the annulus of the Grid, which we arbitrarily assign to processor 0. Analogous code generates a Floorplan with $P + 6$ Regions in 3D.

Returning to Figure 2.9, the Relax routine generates this FloorPlan $F$ from node-level control. With FloorPlan $F$ set up, the procIterator loop in Figure 2.9 proceeds as follows. The first $P$ (in the example, $P = 2$) domains do not wait on the Mover, and execute immediately, in parallel. However, the remaining domains depend on ghost cell values, and must wait until communication completes. The conditional Mover wait() call enforces this constraint.

Thus, we have incrementally constructed a optimized multi-tier parallel code that overlaps communication and computation. Other optimizations are possible, and each application may demand unique algorithmic techniques to maximize performance. However, the KeLP abstractions are sufficiently general to express optimized algorithms for many block-structured scientific calculations. In Chapter 4, we will examine further algorithmic techniques and KeLP coding issues in more detail.
Figure 2.10: a) A FloorPlan with six Regions represents the stencil data dependencies for a two-node SMP. b) KeLP code to setup the FloorPlan.
2.4 Related Work

2.4.1 Structural Abstraction

KeLP descends directly from the LPARX programming system [92]. KeLP inherits the Point, Region, Grid, and XArray abstractions directly from LPARX, although the notions of Grid and XArray have evolved. The most important changes in these classes involve the FloorPlan classes, which hold the structure of a KeLP XArray.

The FIDIL language [121] laid the groundwork for structural abstraction as employed by LPARX and KeLP. FIDIL provided general abstractions for representing structured and unstructured domains, and geometric operations over domains. FIDIL was targeted for serial computers and did not provide any notion of parallel control flow or data decomposition.

KeLP’s data motion model combines structural abstraction with concepts introduced in the Multiblock PARTI [3] runtime system. Multiblock PARTI supports regular block distributions for dynamic arrays, but does not directly support irregular block decompositions as in systems like KeLP. Multiblock PARTI provides two common communication patterns, one to fill in ghost regions and one that moves data over regular sections. KeLP differs from Multiblock PARTI in that KeLP exposes communication management to the programmer, who can then describe arbitrary communication patterns with high-level geometric operations. The Multiblock PARTI communication schedule is an opaque object over which the user has limited control. In contrast, KeLP exposes the schedule to the programmer as a first class mutable object, which may be manipulated, modified and interpreted according to the programmer’s needs.

2.4.2 Software Support for Hierarchical Programming

There have been a few projects that incorporate hierarchical abstractions into programming languages. There are two reasons why hierarchical abstractions
are desirable:

- The architecture has a hierarchical structure, as in SMP clusters.
- The application or algorithm entails hierarchical structures.

Although this dissertation focuses on architectural considerations, we also review application-centric techniques that may apply.

The Illinois Cedar architecture [133] was a prominent cluster-based NUMA shared memory architecture. To use Cedar for scientific applications, the Cedar Fortran language [58] included constructs to express multiple levels of parallelism and locality. Cedar Fortran variables are allocated with global, node, or processor scope. With these constructs, the programmer could implement a two-level array decomposition by hand. Cedar Fortran provided both two-level and flat parallel loops, allowing the programmer to program with either one or two levels of loop parallelism.

The pSather language is based on a cluster machine model for specifying locality [106]. pSather presents a two-level shared address space in the framework of a concurrent object-oriented model. On shared-memory machines, pSather supports a non-blocking asynchronous function call with specialized synchronization mechanisms. On distributed memory machines, asynchronous function calls can be made to remote clusters. Each cluster defines a distinct address space, and each object is assigned to one cluster. Additionally, pSather provides some facilities for allocating distributed objects over multiple clusters, and parallel iteration over coarse-grain distributed objects.

Sawdey et al. [120] have applied the Fortran-P programming model to grid-based applications on SMP clusters. In Fortran-P, a compiler translates serial grid-based code to explicitly threaded parallel code. Our approach differs in that KeLP exposes all partitioning and scheduling decisions to the programmer. Thus, KeLP presents a more complex, explicitly parallel model, but allows the programmer to express a wider class of algorithms and exert more control over the implementation.
Bader and J{\'{a}}{\acute{J}}{\acute{a}} have developed SIMPLE [14], a set of collective communication operations for SMP clusters. SIMPLE provides more general, lower-level primitives than multi-KeLP. For example, SIMPLE provides multi-tier versions of collective communication operations such as barriers, reductions, and gather/scatter. SIMPLE also provides macro directives to control data parallel loops, and directives to control locality on a node or processor basis. Although the syntax differs, these directives provide control flow equivalent to KeLP’s three-level control model. SIMPLE does not help with data decomposition or overlap of communication and computation. The next Chapter discusses the KeLP 2.0 implementation, which relies on collective operations similar to those provided by SIMPLE.

The HPC++ project [18] proposes language extensions and library facilities to facilitate parallel C++ implementations. The HPC++ model explicitly targets two levels of parallel programming for SMP clusters. For parallel programming in a single address space (HPC++ context), HPC++ provides lightweight thread objects similar to those of Java [12]. HPC++ also supports SPMD programming between multiple contexts. HPC++ provides many facilities to help manage communication and synchronization between address spaces. For example, HPC++ provides a global pointer abstraction, parallel versions of C++ Standard Template Library objects, remote function calls, and inter-context collective operations.

HPC++ borrows concepts from several previous object-based systems, including CC++ [44], ABC++ [11], and MPC++ [83]. In general, these concurrent object-oriented models that allow multiple concurrent member function invocations could support multi-tier programming. Many systems allow construction of two-level parallel programs, where the first level handles parallelism between objects and the second level handles parallelism within objects. Mapping these programs onto multi-tier parallel machines may provide a useful and efficient programming paradigm.

The NESL language [25] implements nested data-parallelism, a model which supports hierarchical parallelism and data structures through vectors of
vectors. NESL is an applicative language, and provides no constructs to control data decomposition or granularity of parallelism.

Other systems with hierarchical data structures include systems with support for grid hierarchies in adaptive mesh refinement [92, 64, 109, 116]. Many hierarchical programs arise from divide-and-conquer algorithms. Several task-oriented parallel languages [70, 27, 66, 75] support fork-join parallelism suitable for divide-and-conquer.

2.4.3 Single-Tier Parallel Languages

We now review prominent parallel programming languages that apply to block-structured applications, but do not explicitly address a multi-tier or hierarchical programming style.

Data-parallel languages support regular array-based calculations. The data-parallel approach is typified by High Performance Fortran (HPF) [78], a data-parallel Fortran standard defined by a group from academia and industry. HPF supports loop-based parallelism for F90 codes. The programmer can specify locality with distribution directives, which gives hints to the compiler as to how to distribute arrays across processors. HPFF has also defined the HPF_LOCAL extrinsic environment as a means to drop from HPF into SPMD code. HPF is well-suited for regular, static applications such as *jacobi2d* on single-tier multi-computers. It is not yet clear how HPF will perform on SMP clusters.

HPF builds on a large body of data-parallel languages, including Fortran D [67], Fortran 90D [32], Kali [100], CM Fortran [132], and Vienna Fortran [47]. There have also been several data-parallel extensions to C (e.g., [119]). Fx [74] combines the data-parallel HPF ideas with task-parallel constructs.

The ZPL language [97] provides powerful geometric semantics to express data-parallel computation. Like KeLP, ZPL relies on a Region abstraction to describe rectangular subsets of index space. The ZPL programmer uses Regions to specify iteration space masks, which proves useful for some types of boundary conditions. The ZPL compiler exploits the high level of abstraction of the array
syntax to generate efficient code. ZPL applies only to regular geometries and data parallel operations, although the programmer can drop outside the ZPL model for more complex operations. The ZPL language provides no constructs to control data distribution.

In general, data-parallel languages such as HPF and ZPL allow a more concise expression of data-parallel operations than KeLP. KeLP, in turn, gives the programmer much greater control over the implementation decisions. In our opinion, KeLP should not be considered a competitor to data-parallel languages. Rather, KeLP facilities might help implement the underlying run-time system for a data-parallel language.

The pC++ language [29] extends C++ with a few constructs to support object-oriented data parallel codes. The pC++ collection class implements a distributed set of C++ element objects. pC++ supports an object-parallel control flow model; the program invokes a member function in parallel on elements of a collection. Like KeLP, pC++ supports a shared name space, but does not implement a shared address space. For built-in distributed array and vector class, pC++ provides a run-time version of HPF array distributions, implementing alignments, templates, and virtual processor sets with C++ classes. KeLP’s DOCK library uses a similar strategy, although DOCK does not implement CYCLIC distributions.

The Split-C language [52] extends C with several constructs for parallel computation. Split-C supports a SPMD programming model with a global address space. The program can access either local memory locations or remote memory locations, using global pointers. The programming model reflects two modes of locality, as the programmer expects remote accesses to be slower than local accesses. To tolerate slow remote access, Split-C provides asynchronous (split-phase) get, put, and store operations. The language supports both fine-grain and bulk access to remote data. With these primitives, the programmer can control software pipelining and overlap communication and computation. Split-C supports automatic data layout for regular problems with spread arrays, which allow the
programmer to specify a variety of block and cyclic partitionings.

Since the Split-C primitives encourage fine-grain programming, Split-C performance may depend highly on the overhead for short messages [98]. Lim et al. [95] examined the performance of various Split-C programs on SMP clusters under several architectural assumptions. Their results emphasize the need for low-overhead messages for Split-C, and provide a promising software architecture to achieve this by reserving an idle SMP processor.

2.4.4 Libraries

Parashar and Browne [109] have developed the Hierarchical Dynamic Distributed Array / Distributed Adaptive Grid Hierarchy (HDDA/DAGH) program development infrastructure (PDI), an infrastructure to support structured adaptive mesh refinement calculations. As in structural abstraction, HDDA/DAGH separates the structure of a grid hierarchy from the instantiation of the data. Like KeLP and LPARX, the HDDA/DAGH PDI supports irregular collections of unstructured grids; however, while KeLP and LPARX support grids of arbitrary shape, orientation, and processor assignment, HDDA/DAGH limits refinement structures with a more restrictive spatial framework. Based on this framework, HDDA/DAGH implements a fixed load-balancing policy based on locality-preserving space-filling curves.

P++ is a data-parallel array class library, built on the serial A++ array library [110]. P++ performs delayed evaluation of array expressions, allowing the run-time system to recognize array operations that can occur in parallel. P++ supports dynamic run-time array distributions. Like LPARX/KeLP, P++ presents a SPMD programming model, which is appropriate for structured applications with some task parallelism, such as adaptive mesh refinement.

The design of the PETSc (Portable Extensible Tools for Scientific Computing) toolkit [16] incorporates many of the same ideas as used in KeLP. PETSc provides an object-oriented framework to encapsulate mathematical algorithms, particularly for the numerical solution of partial differential equations. PETSc
provides three basic categories of abstract data types: index sets, vectors, and matrices. Similar to structural abstraction, PETSc index sets represent the abstract domain of a data type and can be used to specify collective communication domains in gather/scatter operations. PETSc provides specialized implementations of the data types to efficiently support both dense and sparse structures. Also, PETSc overlaps communication and computation by providing asynchronous, high-level communication operations. Repeated patterns may be optimized with inspector/executor analysis. The PETSc implementation implements overlap with non-blocking message-passing calls, in contrast to the KeLP multi-threaded implementation.

Although PETSc provides careful implementations of the basic data structures, PETSc targets data-structure neutral design, where the programmer may substitute optimized implementations of the matrix and vector classes. It may be possible to construct dense matrix and vector data-types specialized for SMP clusters using KeLP, and use these data structures in PETSc’s algorithmic framework.

2.4.5 Parallel Programming Models

Every programming model relies on some abstract machine model.

Many parallel computation models roughly correspond the single-tier multicomputer model, pictured in Figure 1.1. Snyder’s Candidate Type Architecture (CTA) [123] provides a general machine model for a multicomputer with a central controller. Valiant’s Bulk-synchronous Parallel (BSP) model [134] considers a program as a sequence of coarse-grain supersteps, and provides an estimate of communication costs with a bandwidth parameter. The LogP [53] model provides parameters to characterize communication costs on a CTA-like machine, specifically communication latency, bandwidth, and overhead. LogGP [4] extends LogP with a parameter to model bandwidth for long messages.

None of these models address the two levels of parallelism and locality in SMP clusters. KeLP instead builds on more germane, hierarchical programming models.
KeLP’s hierarchical control model reflects on the machine conception of the Parallel Memory Hierarchy (PMH) model citealpern93:pmh. The PMH models a parallel computer as a tree of memory modules, and can represent all levels of the memory hierarchy from secondary storage to functional units. The PMH suggests a coding style where chores run concurrently at different levels of the memory hierarchy [5].

The Phase Abstractions [124] programming model describes a parallel program for a CTA in terms of X, Y, and Z levels. The X level corresponds to a serial program on a single processor. The Y level coordinates data decomposition, communication, and synchronization between processors. The Z level consists of a sequence of Y-level operations.

The top two levels of the KeLP control flow fit naturally into XYZ levels. The KeLP collective level corresponds to Snyder’s Y level, which manages data layout and data motion as collective operations. KeLP’s node-level control flow corresponds to Snyder’s X program level. The XYZ program levels do not capture KeLP’s third control flow level, the processor level. However, a straightforward hierarchical extension of the XYZ levels would cover the processor level. In general, it may be worthwhile to consider a recursive XYZ program model based on either nested CTAs or the PMH.

Gropp and Lusk have proposed a programming model taxonomy for SMPs and SMP clusters [73]. Their taxonomy classifies programming styles along three axes: address space, process scheduling, and heterogeneity of the model. It is not clear how a hybrid model like an XYZ program fits in Gropp and Lusk’s taxonomy.

Crandall et. al [51] report experiences with dual-level parallel programs on an SMP cluster. This work uses threads to control intra-node parallelism, and PVM [130] to manage inter-node parallelism. Their results with some simple applications and algorithms motivate further research into multi-tier programming models and environments.
2.5 Discussion

The KeLP programming abstractions define a new programming model for block-structured scientific calculations on SMP clusters. KeLP’s programming model leaves high-level algorithmic decisions and trade-offs to the programmer. The abstractions help the programmer implement these decisions, by allowing the programmer to coordinate the structure of the application with intuitive geometric primitives.

The KeLP model introduces new meta-data programming abstractions, hierarchical control flow constructs, storage model, and data motion primitives. We now discuss these contributions of the KeLP model, as well as limitations of the abstractions.

2.5.1 Geometric Abstractions

KeLP extends the *structural abstraction* programming methodology introduced in LPARX [92]. KeLP directly inherits the Point, Region, Grid, and XArray from LPARX. KeLP adds to this set the Map, FloorPlan, MotionPlan, and Mover abstractions.

These new abstractions add functionality and expressive power to the structural abstraction programming methodology. The Map and FloorPlan give KeLP meta-data the power to express data decompositions, partitioning decisions, and control flow assignments with simple, first-class objects. KeLP utilizes these abstractions in the context of a hierarchical control flow model, representing information for the collective, node, and processor program levels. We will demonstrate in Chapter 4 that with these primitives, the KeLP programmer can express a variety of multi-tier block-structured algorithms.

The MotionPlan and Mover provide a new methodology for representing and manipulating data motion, and will be discussed shortly.

KeLP’s abstractions suit block-structured applications; the Region represents only a rectangular section of space. With the Region as the fundamental
unit of geometry, KeLP does not naturally support applications without a block geometry. For example, KeLP does not support unstructured mesh calculations, sparse matrix representations, or tree-structured algorithms.

KeLP exploits the compact meta-data descriptions of block structures for efficiency. A KeLP implementation can replicate much of the meta-data without wasting much memory. Operations over KeLP meta-data, such as set intersections, involve only a few simple integer operations. With these properties, meta-data manipulation should not be the bottleneck in KeLP applications. In more irregular codes, it is not clear how to represent program structures compactly, making KeLP-like meta-data abstractions potentially very expensive.

2.5.2 Control Flow

KeLP introduces a new three-level loop-structured control flow model, based on the nodeIterator and procIterator. We have shown that these three naturally map to the architecture of an SMP cluster.

An alternative model could define arbitrary nested parallel loops, allowing an arbitrary number of parallel control flow levels. We chose to avoid this model due to implementation concerns. We believe it is straightforward to efficiently map KeLP’s control flow model to an SMP cluster, since KeLP’s machine model directly reflects the hardware structure. Recall that KeLP’s nodeIterator and procIterator classes directly map loop iterations to physical nodes and processors. If we were to divorce the hardware structure from the programming model, it raises many open questions regarding how to map the control flow onto the hardware. Solving these questions remains an open research issue.

2.5.3 Storage Model

Recall that the KeLP control flow model contains three program levels: the collective, node, and processor levels. While KeLP meta-data lives at all three program levels, KeLP Grid data lives only at the node level. An orthogonal
model would contain Grid data at all three program levels. That is, we could define KeLP to have collective-level Grids and processor-level Grids. A collective-level Grid could be created, written and read at the collective program level. A processor-level Grid could be created, written, and read at the processor-level.

We chose to avoid this more general model due to the difficulty in implementing collective Grids on distributed-memory hardware. A collective Grid, in effect, defines a shared-memory address space across all SMP nodes. Distributed shared-memory (DSM) machines such as the SGI Origin 2000 [93] and the HP/Convex Exemplar [34] provide a shared address space. However, clusters of workstation-like SMPs do not provide a shared address space, so a collective Grid implementation would demand a software-based DSM.

While software-based DSM systems have been presented (eg. [88, 40]), it is not clear how these systems will perform on SMP clusters. Examining three Grid levels remains an outstanding research issue.

### 2.5.4 Data Motion

KeLP’s data motion facilities, the MotionPlan and Mover, provide powerful new abstractions for storing and manipulating communication patterns. As discussed in Section 2.4.1, these classes incorporate concepts demonstrated in LPARX [92] and Multiblock PARTI [3].

We have assumed that inter-node communication is very expensive relative to local floating-point computation. The Mover’s asynchronous entry points reflect this assumption, giving the programmer the power to overlap communication and computation. This communication interface differs significantly from those in other parallel libraries and languages. MPI [102] allows the expression of communication overlap with non-blocking messages. However, with MPI’s interface, the programmer asynchronously starts a single message at a time. Split-C [52] provides asynchronous remote memory access, but also at a low-level.

In contrast to these approaches, the KeLP MotionPlan represents an arbitrary collective block-structured data motion pattern, potentially involving many
messages as part of a single unified pattern. With asynchronous Mover operations, the programmer can start and wait for completion of the entire pattern with a single program statement. For applications with complex data motion patterns, this simplified program interface significantly reduces program complexity compared to the more fine-grained approach. Of course, KeLP’s more powerful interface demands an efficient implementation strategy, as will be discussed in the next Chapter.

2.5.5 Future Directions

The KeLP abstractions currently target two levels of parallelism for SMP clusters. The model contains only one level of Grid data, corresponding to Grids that live in SMP node memory. One outstanding research question asks whether the KeLP abstractions prove useful for more general machine models.

For example, consider out-of-core block-structured calculations. One could extend KeLP’s storage model with a new type of Grid that lives not in node memory, but on disk. Accordingly, one could implement a Mover that moves data between an XArray that lives on disk and an XArray that lives in memory. With KeLP’s constructs, it would possible to coordinate a variety of calculations such as out-of-core FFT and matrix operations. Some outstanding research questions include:

- Should an extended KeLP model expose multiple parallel disks to the programmer, or present a single unified abstract disk?

- Can an implementation exploit KeLP’s collective representation of data motion to improve I/O performance?

- Are the current KeLP meta-data objects sufficient to guide the run-time system for efficient execution?

Another possible extension could represent Grids that live pinned in individual processor caches. This technique could be used to minimize cache conflicts
in limited-associativity caches, having the run-time system transparently interact with the operating system page-coloring policy [37].

Finally, interoperability with other programming models remains an outstanding KeLP issue. Merlin et al. have successfully incorporated KeLP with SHPF, a data-parallel HPF-like Fortran dialect [101]. Interaction with other systems, such as PETSc [16] and task-oriented systems [74, 66], remains an unresolved issue.
Chapter 3

Implementation

3.1 Introduction

KeLP aims to provide a convenient programming model to express high performance block-structured scientific calculations on SMP clusters. To realize high performance, the system must satisfy two requirements:

1. The programming abstractions must provide sufficient flexibility to express algorithms appropriate for the underlying hardware.

2. The system must efficiently implement the provided programming abstractions.

In this Chapter, we discuss the implementation techniques and tradeoffs considered to achieve the second requirement.

We have implemented the KeLP programming abstractions in KeLP 2.0, a C++ class library. First, we state assumptions about the underlying hardware and software environment. Next, we discuss the software layers in the implementation. Finally, we quantify the efficiency of the implementation using some simple microbenchmarks.
3.2 System assumptions

To guide implementation decisions, we rely on certain assumptions about the target hardware and system software environment.

As discussed in Section 1.2, we assume an $n$-node SMP cluster, where each SMP has $p$ processors. We rely on hardware support for shared memory operations within each SMP node. As such, we expect the hardware to provide relatively fast communication and synchronization between processors at each SMP node. In contrast, we assume that communication and synchronization costs between SMP nodes are several orders of magnitude more expensive.

The model admits single-tier multicomputers; i.e., SMP clusters with $p = 1$. The implementation should support this hardware with minimal performance sacrifice. Software support for multiprocessor nodes should not significantly slow down codes when using uniprocessor nodes.

In the system software environment, we rely on C++ and Fortran compilers that can generate multiprocessor-safe code. Thus, the compilers must support automatic allocation of local variables on the stack, rather than relying on static allocation of local variables. We rely on MPI for inter-node communication. We do not assume the MPI implementation is thread-safe, nor do we assume that intra-node MPI communication is efficient.

If the system provides a functional lightweight thread package, the KeLP 2.0 implementation uses the thread package to implement parallel control streams at an SMP node. However, we do not assume a lightweight thread package is available. Instead, if lightweight threads are not available, we assume a UNIX fork system call with mmap memory-mapped file support. With this scenario, we implement our own shared heap under KeLP control. We rely on some sort of system provided inter-process synchronization facilities, such as semaphores.

In an early implementation of multi-tier KeLP, we assumed a lightweight thread package that would support parallel execution of one parallel thread per physical processor in an SMP node. Unfortunately, on one platform (Digital Al-
phaServer running Digital UNIX 4.0), we found that the POSIX-compliant thread library does not reliably provide parallel execution of \( p \) threads on a \( p \)-processor SMP. In our experience, the thread scheduler often maps several threads onto one physical processor, while leaving another processor idle. Furthermore, the threads migrate often, destroying cache locality. Since the POSIX standard [82] provides no user control over the mapping of lightweight threads to physical processors, we could not realize acceptable parallel performance using the lightweight threads.

From this point on, we will use the word “thread” to refer to a stream of instructions managed by the KeLP 2.0 implementation. Depending on the system environment, KeLP implements threads using either POSIX-style lightweight threads or with heavyweight processes.

For single-tier multicomputers, we do not assume that the system provides any mechanism to support multiple threads on a node. The KeLP implementation will run on single-tier multicomputers using only one thread per physical processor.

### 3.3 Implementation Structure

The KeLP programming abstractions coordinate several aspects of program behavior; most importantly: parallel control, storage, and data motion. We now discuss the design choices in order to implement these facets of program coordination.

#### 3.3.1 Parallel Control

The KeLP control model has three levels of control: the collective, node, and processor levels. We implement the three levels as follows. Since we rely on MPI for inter-node communication, the \texttt{main()} procedure will run under MPI control as one SPMD process per SMP node. Recall that we assume that the operating system provides a mechanism for creating parallel execution streams, with either lightweight threads or heavyweight processes. Since spawning heavyweight processes will likely be quite expensive, we choose to spawn parallel control
threads only once, at program startup. Thus, at program startup, the implementation spawns a number of “threads” on each SMP node. All threads execute the collective level code, analogously to SPMD execution. The nodeIterator and procIterator constructs mask out execution with a conditional, so only the appropriate threads execute the appropriate loop bodies.

An alternative implementation could have the Iterator constructs spawn new threads dynamically. However, this alternative would suffer the overhead to spawn a thread at the beginning of every procIterator loop. This strategy would incur prohibitive overheads when forced to rely on heavyweight processes to implement threads.

3.3.2 Storage Model

As discussed in the previous Section, the KeLP implementation maintains at least $p$ threads running on each SMP node. When running at the processor level, each thread must sometimes dynamically allocate private storage. However, multiple threads at a node will share some KeLP data structures, such as KeLP Grids. To support both types of data structures, the implementation must provide a private stack and heap for each thread, as well as a shared heap at each SMP node.

Lightweight thread packages such as POSIX threads provide this memory model automatically. These packages give each thread a shared stack, with a heap shared by all. Individual threads can reserve private heap storage simply by calling malloc.

As discussed earlier, we cannot assume system support for lightweight threads. When using heavyweight processes for parallel control, we implement a shared heap using a memory-mapped shared file\(^1\). On program startup, before forking processes, KeLP allocates a large memory-mapped file to serve as the shared heap. The KeLP software provides malloc access from the shared heap. This solution limits the implementation to a pre-defined maximum shared heap

\(^1\)Thanks to Abdul Tabbara for his assistance with the implementation
size. However, since the heap is allocated and memory-mapped at program startup, the implementation avoids further system calls when dynamically allocating shared memory.

To efficiently manage the shared heap, the implementation must reclaim storage allocated to shared dynamic objects when possible. The KeLP implementation maintains reference counts to dynamic shared objects, which the system reclaims when the reference count reaches 0. The KeLP reference count implementation follows the design presented by Stroustrup [129]. Reference counting adds a few extra indirect memory references to access shared objects. However, the implementation also allows access to the memory locations directly, to avoid the overhead in performance-critical inner loops.

### 3.3.3 Data Motion Strategy

Recall that we rely on MPI as the inter-node communication substrate, and do not assume the MPI implementation is thread-safe. With this limitation, the implementation must ensure that multiple threads do not invoke multiprocessor-unsafe MPI operations concurrently. Since the KeLP programming model performs inter-node communication at the node level, the KeLP implementation ensures that only one thread on each node invokes MPI message-passing calls. This design bypasses the MPI thread safety issue.

The KeLP 2.0 implementation includes two mechanisms for overlapping communication and computation. The first mechanism devotes an extra thread on each SMP node, devoted solely to inter-node communication. The second mechanism does not use an extra thread, but instead relies on MPI non-blocking messages to overlap communication and computation.

The tradeoffs and lower-level details of these two strategies are discussed in Section 3.4.2.
3.4 Software Layers

We have implemented the multi-tier KeLP abstractions in KeLP 2.0, a set of C++ class libraries. The implementation has several layers, as illustrated in Figure 3.1.

The mp++ layer, inherited from LPARX, provides a portable interface to the message-passing system. The emergence of MPI as a ubiquitous standard mitigates the need for a custom message-passing interface. Nevertheless, KeLP 2.0 retains mp++ for convenience, since it provides simpler calling conventions than the more general MPI standard.

![Figure 3.1: Software layers in the KeLP 2.0 implementation.](image)

3.4.1 Shared Memory Layer

The smp++ layer provides a portable interface to lower-level thread routines and shared memory abstractions.

Smp+ implements limited lightweight thread and shared memory abstractions without relying on POSIX or lightweight thread support. Depending on the target hardware, an smp++ thread is implemented as either a POSIX thread or a
<table>
<thead>
<tr>
<th>Name</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>smpInit</td>
<td>Initialize thread library</td>
</tr>
<tr>
<td>smpExit</td>
<td>Exit and clean up</td>
</tr>
<tr>
<td>smpSpawn</td>
<td>Spawn a thread</td>
</tr>
<tr>
<td>smpJoin</td>
<td>Wait until a thread exits</td>
</tr>
<tr>
<td>smpJoinAll</td>
<td>Wait until all threads exit</td>
</tr>
<tr>
<td>smpReturn</td>
<td>The calling thread exits</td>
</tr>
<tr>
<td>smpBarrier</td>
<td>Synchronize all threads at a node</td>
</tr>
<tr>
<td>smpBeginCriticalSection</td>
<td>Acquire a global mutex</td>
</tr>
<tr>
<td>smpEndCriticalSection</td>
<td>Release a global mutex</td>
</tr>
<tr>
<td>smpMyID</td>
<td>Returns a thread ID, an integer</td>
</tr>
<tr>
<td>smpNumThreads</td>
<td>Returns the number of threads spawned</td>
</tr>
<tr>
<td>smpBindMeToCPU</td>
<td>Bind a thread to a physical processor</td>
</tr>
<tr>
<td>shmalloc</td>
<td>Allocate storage on the shared heap</td>
</tr>
<tr>
<td>shfree</td>
<td>Release storage on the shared heap</td>
</tr>
<tr>
<td>shmallocAcquire</td>
<td>Acquire a mutex for shmalloc routines</td>
</tr>
<tr>
<td>shmallocRelease</td>
<td>Release a mutex for shmalloc routines</td>
</tr>
<tr>
<td>SharedObject</td>
<td>An object that lives on the shared heap</td>
</tr>
<tr>
<td>SharedList</td>
<td>a list of objects on the shared heap</td>
</tr>
<tr>
<td>Semaphore</td>
<td>generic Semaphore class</td>
</tr>
<tr>
<td>Mutex</td>
<td>generic Mutex class</td>
</tr>
</tbody>
</table>

Table 3.1: The smp++ functions.

Heavyweight process. The implementation spawns a fixed number of threads at the beginning of the program, so thread creation performance does not significantly impact performance.

All smp++ threads at a node can access a shared heap. To allocate storage on the shared heap, smp++ provides a shared-memory dynamic memory allocator (shmalloc). POSIX and lightweight thread packages provide a shared heap automatically. Unfortunately, as mentioned earlier, we were forced to resort to heavyweight processes on some platforms. For heavyweight processes, the shmalloc facility allocates the shared heap as a large memory-mapped file using mmap, and manages dynamic memory allocation with a modified version of the GNU malloc routine written by Mike Haertel.
In addition to rudimentary thread spawn and join routines, \texttt{smp++} provides a number of convenient higher-level operations. To simplify synchronization, \texttt{smp++} provides thread-level barriers, semaphores, and mutex objects. Additionally, the \texttt{SharedObject} virtual base class supports dynamically-allocated objects on the shared heap.

### 3.4.2 KeLP Layer

The KeLP 2.0 infrastructure implements the KeLP programming abstractions (Grid, XArray, FloorPlan, etc.) as a layer which calls \texttt{smp++} and \texttt{mp++} facilities. This layer implements each of the KeLP abstractions as a C++ object, as well as routines to initialize global KeLP data structures, spawn threads, and perform reductions and synchronizations.

The Mover implementation deserves special attention, since the Mover must perform inspector/executor analysis of the data motion pattern, issue message-passing calls and memory copies to effect data motion, and overlap communication and computation as needed.

The KeLP 2.0 Mover implementation processes the data motion pattern and issues the appropriate message-passing calls. The run-time communication analysis falls into two phases.

Phase 1 performs the following operations on each node:

1. Allocate message buffers for incoming and outgoing messages.

2. Post for incoming non-blocking MPI messages from other nodes.

3. Pack outgoing Grid data into message buffers, where necessary.

4. Send non-blocking MPI messages to other nodes.

5. Perform local memory-to-memory copies for Grid copies between two local Grids.

Phase 2 performs the remainder:
1. While incoming messages are expected:
   
   (a) Poll until an incoming message arrives.
   (b) Unpack the detected message data into the target Grid.
   (c) Free the message buffer.

2. Wait until all non-blocking outgoing messages have been sent.

3. Free all outgoing message buffers.

By default, the implementation avoids buffer-packing for contiguous data. That is, if a Grid copy’s data happens to lie contiguously in memory, then the Mover will send or receive the data directly from its original storage locations.

An important implementation decision concerns the method used to overlap communication and computation. Some architectures [19, 111, 128] provide specialized hardware support to overlap communication and computation. We considered two possibilities: using MPI non-blocking message-passing primitives, and devoting a single \texttt{smp++} thread solely to communication.

The MPI overlapping design is straightforward. When the program calls the Mover \texttt{start()} member function from collective control flow, one thread on each node performs phase one of the message-passing algorithm described above. Note that phase 1 invokes non-blocking MPI messages for inter-node communication. Later, the program calls \texttt{Mover::wait()} to complete the asynchronous Mover operation. At this call, one thread on each node performs phase 2 of the message-passing algorithm, finishing the non-blocking message-passing. The other threads synchronize with \texttt{smp++} calls.

To implement communication overlap using a dedicated thread, we use a producer-consumer queue in shared-memory. When the program calls the Mover \texttt{start()} function from collective control flow, one thread appends a pointer to the Mover on a queue in shared memory, and signals the communication thread by incrementing a counting semaphore. The communication thread waits on the semaphore. When the communication thread wakes up, it dereferences the Mover
pointer, and invokes phase 1 and phase 2 of the message-passing algorithm on this Mover.

When the program calls `Mover::wait()`, the running threads wait on a semaphore. When the communication thread completes the data motion, it posts to this semaphore, waking up the sleeping threads.

### 3.5 Performance Evaluation

In this Section, we evaluate the performance of the KeLP 2.0 implementation with a series of microbenchmarks. First, we consider some small simple message-passing tests, to evaluate the efficiency of the KeLP Mover implementation. Next, we measure the costs of intra-node inter-processor synchronization. Finally, we examine implementation issues, costs and benefits of overlapping communication and computation.

#### 3.5.1 Experimental Platforms

This dissertation presents empirical performance results from three platforms: the Maryland Digital AlphaServer, the “bronco” SparcStation cluster at Oregon State University, and the Cray T3E at the San Diego Supercomputer Center (SDSC). The first two platforms are SMP clusters. We include results on the Cray T3E to evaluate support for single-tier multicomputers.

The Maryland Digital AlphaServer is a cluster of 10 AlphaServer 2100 nodes [76]. Each AlphaServer 2100 contains four 275 MHz Alpha 21064A processors. Each processor has an on-chip 16kB direct-mapped data cache and a 16kB direct-mapped instruction cache. Additionally, each processor has a 4MB direct-mapped off-chip unified cache. Each node has 256MB of main memory, which the processors access over a 42 MHz shared system bus with a peak theoretical bandwidth of 533 MB/s. The nodes communicate over a Digital Gigaswitch OC-3 ATM switch, with a peak point-to-point bandwidth of 155Mb/s.
The Oregon State SparcStation cluster contains four SparcStation 20 nodes. Each SparcStation contains four 50 MHz SuperSPARC processors [103]. Each processor has an on-chip 16kB four-way set associative data cache and a 20 kB five-way set associative instruction cache. Additionally, each processor has a 1MB direct-mapped off-chip unified cache. Each node has 128 MB of main memory, which the processors access over a shared 40 MHz shared system bus with a peak theoretical bandwidth of 320 MB/s. The nodes communicate over 10Mb/s Ethernet. The cluster also has a Myrinet interconnection network, but during the experimental period (early 1998), the Myrinet was not completely functional and so was not used for our experiments.

The SDSC Cray T3E has 256 nodes, each with one 300 MHz Alpha 21164 processor. Each processor has an on-chip 8kB direct-mapped instruction and data caches, and additionally a 96kB three-way set associative unified second level cache. There is no off-chip cache. Each node has 128MB of main memory. The nodes communicate over the T3E’s proprietary interconnection network, which provides a peak theoretical bandwidth of 600 MB/s for point-to-point messages.

The KeLP 2.0 implementation relies on system-provided C++ and Fortran compilers, UNIX system calls, and an MPI implementation. Table 3.2 shows the software environments used for all experiments in the dissertation.

For all experimental results reported in the dissertation, the timed kernel is repeated so that the total timed running time is at least ten seconds. Timings reported are based on wall-clock times, obtained via the MPI_Wtime() call.

### 3.5.2 Message Passing Overheads

KeLP’s MotionPlan and Mover abstractions provide a high-level interface to the message-passing system. As discussed previously, the Mover must interpret and execute the MotionPlan pattern at run-time. This run-time inspector/executor processing incurs some overhead.

In order to evaluate the overhead, we compare KeLP performance to MPI performance on two simple message-passing microbenchmarks. The first bench-
<table>
<thead>
<tr>
<th>Platform</th>
<th>C++ Compiler</th>
<th>Flags</th>
<th>Fortran Compiler</th>
<th>Flags</th>
</tr>
</thead>
<tbody>
<tr>
<td>AlphaServers</td>
<td>gcc v2.7.2</td>
<td>-O2</td>
<td>Digital Fortran v4.0</td>
<td>-O5 -tune ev4 -automatic -u -assume noaccuracy_sensitive</td>
</tr>
<tr>
<td>SparcStations</td>
<td>gcc v2.7.2</td>
<td>-O2</td>
<td>SunPRO f77 SC4.2</td>
<td>-O4 -u -stackvar</td>
</tr>
<tr>
<td>T3E</td>
<td>Cray C++ v3.0.2.1</td>
<td>-O3</td>
<td>Cray CF90 v3.0.2.1</td>
<td>-O3 -dp</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Platform</th>
<th>Operating System</th>
<th>MPI Implementation</th>
</tr>
</thead>
<tbody>
<tr>
<td>AlphaServers</td>
<td>Digital UNIX 4.0</td>
<td>MPICH 1.1.0 [72]</td>
</tr>
<tr>
<td>SparcStations</td>
<td>SunOS 5.6</td>
<td>MPICH 1.1.0</td>
</tr>
<tr>
<td>T3E</td>
<td>UNICOS 2.0.2.27</td>
<td>Cray Message Passing Toolkit v1.2.0.1</td>
</tr>
</tbody>
</table>

Table 3.2: Software environments for all experiments in the dissertation.

mark, ring, simply passes messages around a ring of processors. The second benchmark, exch, organizes the nodes in a logical 1D ring (toroidal) topology. Each node sends and receives messages from its two immediate neighbors. Figures 3.2 and 3.3 show the KeLP code for these two microbenchmarks. The MPI codes, not shown, are straightforward.

We ran these two benchmarks on the Maryland Digital AlphaServer, the SparcStation cluster, and the Cray T3E. On the two SMP clusters, we use only one processor per node. For the SMP cluster experiments, we configure the KeLP implementation to use two threads: one to run the user code, and an extra thread to handle communication. Thus, invoking a Mover involves fairly expensive communication and synchronization between the threads. The T3E implementation uses only one thread per node.

Figures 3.4 and 3.5 compare the performance of the KeLP and MPI codes on these benchmarks over a range of problem sizes.

The results show that for the ring benchmark, KeLP performs comparably
\[
\begin{align*}
n &= \text{number of nodes} \\
\text{FloorPlan} &= F(n) \\
\text{Region} &= \{1, \text{MESSAGE\_SIZE}\} \\
\text{for each } i &\in F \\
&\quad F.\text{setOwner}(i, i) \\
&\quad F.\text{setRegion}(i, R) \\
\text{end for each } i &\in F \\
\text{XArray} &= X(F) \\
\text{do } i &= 1, \text{TIMES\_AROUND} \\
&\quad \text{do } j &= 0, n \\
&\quad &\quad \text{MotionPlan} &= M \\
&\quad &\quad \text{integer } r &= j + 1 \text{ mode } n \\
&\quad &\quad M.\text{copyOnIntersection}(X, j, X, r, R) \\
&\quad &\quad \text{Mover} &= m(X, X, M) \\
&\quad &\quad m.\text{execute}() \\
&\quad \text{end do} \\
&\text{end do} \\
\end{align*}
\]

Figure 3.2: KeLP code for the ring message-passing microbenchmark.

to MPI for long messages, but incurs significant overhead for short messages. On
the AlphaServers, KeLP performs within 5\% of the MPI code on messages longer
than 256kB. For shorter messages, KeLP degrades performance by up to 50\%. On
the SparcStations, KeLP performs within 11\% of the MPI code for messages longer
than 16kB. On shorter messages, KeLP degrades performance by up to 75\%.

On the T3E, KeLP performs within 5\% of the MPI code for messages
of 1MB. For shorter messages, KeLP degrades performance by up to 78\%. Note
that the T3E performance has some counter-intuitive characteristics. With the
MPI ring program, performance has a local peak at 16kB messages, then degrades
until messages reach 131kB. We suspect the MPI implementation uses two dif-
erent implementation techniques for messages below and above 16kB. The KeLP
implementation does not show the local maximum at 16kB. We suspect Mover
overheads obscure the underlying effect.
\[
n = \text{number of nodes}
\]
\[
\text{FloorPlan } F(n)
\]
\[
\text{Region } R = [1, \text{MESSAGE\_SIZE}]
\]
for each \( i \in F \)
\[
F.\text{setOwner}(i, i)
\]
\[
F.\text{setRegion}(i, R)
\]
end for each \( i \in F \)
\[
\text{XArray } X(F)
\]
\[
\text{MotionPlan } M
\]
do \( j = 0, n - 1 \)
\[
\text{integer } \text{left} = j - 1 \mod n - 1
\]
\[
\text{integer } \text{right} = j + 1 \mod n - 1
\]
\[
M.\text{copyOnIntersection}(X, j, X, \text{left}, R)
\]
\[
M.\text{copyOnIntersection}(X, j, X, \text{right}, R)
\]
end do
do \( i = 1, \text{TIMES\_AROUND} \)
\[
\text{Mover } m(X, X, M)
\]
\[
m.\text{execute}()
\]
end do

Figure 3.3: KeLP code for the exchange message-passing microbenchmark.

The exchange benchmark results show similar trends, although overheads and a few other effects become more pronounced. On the AlphaServers, KeLP performs within 15% of the MPI code for messages longer than 16kB. For shorter messages, KeLP degrades performance by up to 51%. The SparcStations show erratic, unpredictable performance for messages than 16kB; however, in most cases, KeLP performs within 10% of the MPI code. On the T3E, KeLP slightly outperforms the MPI code for 1MB messages. For shorter messages, KeLP introduces overheads of up to 72%.

Note that on four nodes and eight nodes of the AlphaServers, performance on the exch pattern significantly degrades for messages longer than 8kB. Two nodes do not exhibit the degradation. We do not know the cause of the degradation, although we suspect a poor interaction between the TCP/IP protocol and the
Figure 3.4: Performance of the ring microbenchmark on a) the AlphaServer cluster, b) the SparcStation cluster, and c) the T3e.

ATM switch protocols. In any case, we can attempt to alleviate the problem, by breaking long messages into 8kB pieces, and sending the small messages, one at a time. To test this strategy, we modified the Mover implementation to perform this optimization.

Figure 3.6 shows the performance of the \texttt{exch} pattern using this modified flow control algorithm. The results show that on four and eight nodes, the modified algorithm achieves up to 7.1 MB/s on long messages. Without the algorithm, bandwidth peaks at 6.2 MB/sec. Also note that the modified algorithm degrades
Figure 3.5: Performance of the exchange microbenchmark on a) the AlphaServer cluster and b) the SparcStation cluster.

Based on these results, we configure the KeLP implementation to invoke the modified Mover flow control protocol on the AlphaServers algorithm where appropriate. The implementation never uses the modified protocol when running on only two nodes. We attempted a similar optimization for the T3E, but did not improve performance.

In summary, the message-passing microbenchmarks confirm that compared to simple message-passing codes, KeLP introduces significant overhead for
short messages, but suffers small performance degradation for long messages. Based on these results, KeLP should perform well for coarse-grain applications which rely on large messages, but does not suit fine-grain applications with smaller messages.

Also recall that the KeLP Mover performs message aggregation, to concatenate multiple small messages into one longer message. The microbenchmark results validate the need for this optimization.

3.5.3 Overlapping Communication and Computation

As discussed earlier, we consider two possible implementation strategies to overlap communication and computation. We now present some simple microbenchmarks to evaluate these two designs, and evaluate scheduling policies relevant to the communication overlap implementation.

Our experiments will examine communication overlap and its interaction with memory traffic on an SMP node. On an SMP node, bandwidth and latency to the shared memory banks will greatly impact a code’s performance. Codes that demand heavy memory traffic may saturate the shared memory bus or banks [10], degrading parallel speedup. On the other hand, codes that heavily reuse data in cache may not saturate the memory system, and will achieve greater parallel
efficiency.

Additionally, the local kernel's memory traffic will affect the efficacy of communication overlap. When performing message-passing and local computation concurrently, both activities generate traffic on the shared memory system. Potentially, the two activities will compete for shared memory bandwidth, and each activity will slow down the other.

To account for this effect, we present a communication overlap microbenchmark that measures the efficacy of communication overlap under varying memory traffic demands. Figure 3.7 shows the local numerical kernel used in these experiments. This code performs a dot product on two long vectors, \( x \) and \( y \). The length of these vectors, \( n \), is chosen large enough so that the vectors do not fit in any level of cache. Additionally, in each iteration of the \( i \) loop, the kernel performs \( k \) floating-point multiplications, which operate directly from registers.

```fortran
double precision x(n), y(n), result
double precision c
parameter (c = 0.99d0)
result = 0d0
do i=1,n
   result = result + x(i)*y(i)
   do j = 1, k
      y(i) = y(i)*c
   end do
end do
```

Figure 3.7: Fortran 77 code for the local kernel used in the overlap efficiency experiments.

In this code, the \( k \) parameter roughly corresponds to locality of the memory references. As \( k \) increases, the loop performs more operations directly from registers, and memory accesses become less frequent. When \( k \) is large, the loop achieves a fast MFLOP rate and generates little memory traffic. When \( k \) is small, the loop generates memory traffic more frequently, and performance suffers in turn.
Figure 3.8 shows the performance of this kernel on a single SMP node, varying $k$. As expected, results show that on both the AlphaServer and the SparcStation, the performance of the kernel increases with $k$. Additionally, the Figure shows that the parallel speedup increases with $k$. For $k=0$, the kernel saturates the shared memory system with two processors. For $k=100$, the code achieves nearly linear parallel speedup.

![Locality Kernel Performance](image1)

Figure 3.8: Performance results for the kernel for Figure 3.7 on a) the AlphaServers and b) the SparcStation cluster.

Now, we use this kernel in a microbenchmark to assess the efficacy of communication overlap and its interaction with memory traffic generated by the local kernels. The overlap microbenchmark uses two SMP nodes. The microbenchmark uses the KeLP Mover to asynchronously perform the exchange message-passing pattern presented in Section 3.2. Concurrently with the message passing, the microbenchmark calls the local kernel, overlapping the local work with the Mover execution. We chose the message size in the long message regime (at least 1MB), and repeat the microbenchmark repeatedly so the running time exceeds 10 seconds.

We evaluate overlap with an overlap efficiency metric similar to one presented by Sohn and Biswas [126]. Define the following terms:

- $T_{nocomm}$: the wall clock time to run the calculation, with no communication.
• $T_{\text{noolap}}$: the wall clock time to run the calculation, with communication but no communication overlap.

• $T_{\text{olap}}$: the wall clock time to run the calculation, with communication and communication overlap.

• $C_{\text{noolap}} = T_{\text{noolap}} - T_{\text{nocomm}}$: the cost of communication with no overlap

• $C_{\text{olap}} = T_{\text{olap}} - T_{\text{nocomm}}$: the cost of communication with overlap

Then the overlap efficiency is defined as:

$$E_{\text{olap}} = \frac{C_{\text{noolap}} - C_{\text{olap}}}{C_{\text{noolap}}}$$

(3.1)

Note that memory bandwidth tradeoffs will influence the efficacy of communication overlap. For example, with $k=0$, the local kernel saturates the memory system with fewer than four processors. So, naively, we might expect to dedicate the fourth processor to message-passing without losing much computational power. However, in reality, the fourth processor’s communication traffic will also consume memory bandwidth, and will further retard the three computing processors.

Suppose we devote the fourth processor to communication with $k=100$. In this case, we do not expect the communication memory traffic to impact performance, since the kernels access memory infrequently. However, the fourth processor’s cycles might be better used for computation, since the memory bandwidth is not saturated.

To quantify these effects, we present microbenchmark results varying the $k$ parameter. We consider the two implementation strategies for communication overlap described earlier. The first strategy, “MPI”, overlaps communication and computation using MPI non-blocking message passing calls. The second strategy, “extra thread”, spawns an extra thread on each node to handle message-passing.

To evaluate these alternatives, we use the overlap microbenchmark to assess the overlap efficiency resulting from each implementation. We consider two scenarios. First, we assume that one processor on each SMP node serves as a
dedicated Mover coprocessor. In this scenario, we reserve one processor to handle only communication, and it never participates in running the local numeric kernel. When using the “extra thread” implementation, we park the extra thread on the fourth processor.

Recall that each of our SMP cluster platforms has four processors per node. For this coprocessor scenario, we obtain \( T_{\text{no comm}}, T_{\text{no overlap}}, \) and \( T_{\text{overlap}} \) by running the local kernel using only three processors on each node. We compute the overlap efficiency assuming that the fourth processor can only perform Mover activity, and cannot contribute to local computational work.

Figure 3.9 shows the overlap efficiency obtained under this scenario. The Figure shows that the “MPI” overlap strategy fails. In fact, attempting to overlap communication using MPI non-blocking messages slows down the code in all cases. We also observe the expected trend that in this coprocessor scenario, overlap efficiency increases with \( k \). On the AlphaServers, the \( k=0 \) kernel results in \( E_{\text{overlap}} = 54\% \), while \( k=100 \) gives \( E_{\text{overlap}} = 95\% \). On the SparcStations, \( E_{\text{overlap}} \) ranges from 75\% for \( k=0 \) to 90\% for \( k=100 \).

The results from the first scenario clearly demonstrate that the memory traffic from the local kernels and the communication activity interfere. These
results show that dedicating one processor of an SMP as a message coprocessor achieves overlap efficiency of 50-95%, depending on the locality of the local kernel. However, in practice, it may be wasteful to devote one processor entirely to communication. Many sections of a real code will not involve communication, and the first scenario wastes cycles on the fourth processor for these sections.

So now we consider overlap efficiency, accounting for the fact that without communication overlap, all four processors contribute to local work. So now, we compute $T_{\text{no comm}}$, $T_{\text{no overlap}}$, and $T_{\text{overlap}}$ by running the local kernel on all four processors on each node.

When using the “MPI” strategy, we need only four threads on each node, one parked on each physical processor. However, the “extra thread” strategy requires five threads running on a four-processor SMP. This raises the question of how to schedule the threads to processors.

We consider two alternatives. The first, called “OS scheduling”, leaves the thread scheduling decisions to the operating system. In this strategy, we divide the total work evenly among the four computing threads. We let the operating system load balance the five threads. Ideally, the operating system will achieve perfect load balance. Note that the kernel steps through large vectors, effectively flushing the cache in each iteration. Thus, thread migration will not affect cache locality for these experiments.

The second alternative, called “user scheduling”, leaves load balancing decisions to the program. With this strategy, we explicitly bind each thread to a physical CPU with system calls. By default, we bind the extra communication thread to processor 0. So, the threads do not migrate, relieving the OS from having to make load balancing decisions. Instead, the user program dynamically load balances the work among the threads.

Figure 3.10 shows the overlap efficiency results from this scenario. As before, the MPI implementation almost always fails to improve performance with communication overlap. On both platforms, the extra thread strategy achieves positive overlap efficiency, and in all cases user-scheduling of the work surpasses
load balancing by the operating system. On the AlphaServers, the “OS scheduling” policy averages $E_{\text{olap}} = 9.5\%$, while “user scheduling” averages $E_{\text{olap}} = 60\%$. On the SparcStations, “OS scheduling” averages $E_{\text{olap}} = 29\%$, while “user scheduling” averages $E_{\text{olap}} = 53\%$.

Figure 3.10: Overlap efficiency results on a) the AlphaServer cluster and b) the SparcStation cluster.

Unlike the previous scenario with a reserved Mover coprocessor, these results do not show a clear trend between $k$ and $E_{\text{olap}}$. Instead, these results reflect a tradeoff; when $k = 0$, memory bandwidth concerns limit $E_{\text{olap}}$, while for $k > 50$, the extra thread consumes processor cycles that could otherwise contribute to local numerical work. In many cases, the data shows that with an extra thread and user-scheduling of work, these two factors tend to balance out. This result depends on the underlying platform’s hardware and software environment; other platforms might demonstrate other trends.

Recall that the Figures report the average timings for a number of iterations chosen so the total running time exceeds 10 seconds. Even so, the data does not show any clear dependence on $k$. The data shows significant fluctuations between adjacent $k$ values, but we have no satisfactory explanation for these fluctuations. We hypothesize that cache conflicts, context switch timings, bus collisions, or network collisions might contribute to this phenomenon. In general, we lack
adequate analytic techniques to predict overlap efficiency on these real systems.

3.5.4 Synchronization Overheads

KeLP provides a number of collective operations, such as barriers and reductions. The implementation of these operations entails several synchronization points. In this Section, we document the synchronization overhead for these operations.

Based on the results of the previous Section, we only consider the KeLP configuration that dedicates an extra thread on each node to handle inter-node communication.

First, we consider the kelpBarrier() operation. Called from the collective control stream, the kelpBarrier() operation synchronizes all threads on all nodes. The implementation of kelpBarrier() relies on a specialization of Mover, called the NodeBarrier. The NodeBarrier, publicly derived from Mover, is a Mover that instead of interpreting a MotionPlan, instead just calls MPI_Barrier().

The kelpBarrier() implementation proceeds as follows. Processor 0 on each node creates a NodeBarrier object, and calls NodeBarrier::start() followed immediately by NodeBarrier::wait(). Then, all processors at each node perform a node-level barrier using smp++ Semaphores.

So, in addition to the time to perform MPI_Barrier(), kelpBarrier() involves extra overhead to set up and invoke a Mover, synchronize on the Mover’s completion, and to synchronize all processors within a node. Figure 3.11 shows the running time for the KeLP barrier() operation using between one and four processors per node.

The results show that the overhead for the barrier is quite high, and that intra-node synchronization costs rival the inter-node synchronization costs. Furthermore, the cost of a barrier is roughly linear in the logarithm of the number of nodes. On the AlphaServers, using only one thread per node, an MPI barrier on $n$ nodes takes roughly $1040 \log n \mu s$. Synchronizing all four threads on each node adds overhead of roughly $1000 \mu s$. On the SparcStations, an MPI barrier on $n$
nodes takes roughly $1290 \log n \mu s$. Synchronizing all four threads adds additional overhead of roughly $745 \mu s$.

We conclude KeLP programs should avoid the overhead of barrier synchronization wherever possible. The KeLP implementation avoids inter-node barrier synchronizations, but must still incur intra-node synchronizations at the end of each procIterator loop.

![Barrier Sync Performance](image.png)

Figure 3.11: Barrier synchronization performance on a) the AlphaServer cluster and b) the SparcStation cluster.

Inter-node and inter-processor synchronization also determines the performance of collective operations such as reductions. We now present the performance of \texttt{kelpReduce()}, a KeLP collective routine to perform a reduction. We use \texttt{kelpReduce()} to sum integers across an SMP cluster, and compare performance to the MPI \texttt{Allreduce()} library call.

Figure 3.12 shows the results. Once again, we see overhead that is roughly linear in the logarithm of the number of nodes. On the AlphaServers, using only one thread per node, an MPI reduction on $n$ nodes takes roughly $1720 \log n \mu s$. Synchronizing all four threads on each node adds overhead of roughly $4000\mu s$. On the SparcStations, an MPI barrier on $n$ nodes takes roughly $1840 \log n \mu s$. Synchronizing all four threads adds additional overhead of roughly $3600 \mu s$.

We conclude that, as expected, inter-node synchronization costs are quite
Figure 3.12: Reduction performance on a) the AlphaServer cluster and b) the SparcStation cluster.

...high, on the order of 1ms on both platforms. Somewhat unexpectedly, with four threads, intra-node synchronization costs rival the inter-node costs. The KeLP programmer should consider barriers and reductions expensive, and should endeavor to structure an application to call these primitives infrequently. Applications with make frequent calls to collective synchronization points will suffer from the high synchronization overheads.

## 3.6 Related Work

Sohn and Biswas [126] examined the utility of communication overlap for FFT and bitonic sorting on the IBM SP-2 and SGI PowerCHALLENGEArray. This work introduced the overlap efficiency metric, which we adapted to evaluate overlap efficiency in KeLP. Sohn and Biswas report overlap efficiencies of up 40% on the SP-2, but only up to 10% on the PowerCHALLENGEArray. They did not consider dedicating one SMP processor to handle communication, which would have resulted in much higher overlap performance.

The Proteus machine [127, 128] is a custom-built hierarchical SMP cluster designed for image processing applications. Each Proteus cluster a number of
Intel i860 “pixel processors” which run user code, an Intel i960 Cluster Controller, and shared memory modules. The Cluster Controller handles all inter-node communication. Like the KeLP2.0 implementation, pixel processors request inter-node messages via shared-memory mailboxes available to the Cluster Controller. The Proteus programming API presents a uniform message-passing model, which hides the two-level non-uniform memory hierarchy but implements intra-node messaging efficiently in shared memory.

Lim et al. present message proxies [95]. A message proxy is a kernel-level process running on an idle SMP processor that provides protected access to the network interface. User-level processes communicate with the message proxy through queues in shared memory. The message proxy handles all network interface access, so concurrent user processes do not need to rely on system calls, locks, or interrupts to enforce protection and atomicity. With this advantage, message proxies provide low overhead short messages.

Several studies have addressed architectural and implementation tradeoffs on SMP clusters with hardware- or software-supported distributed shared memory. Our work focuses on carefully orchestrated message-passing codes, restructured to avoid idle time on compute processors. In contrast, shared memory systems must hide latency of remote references to reduce compute processor stalls.

Three prominent cluster-based NUMA shared memory machines were the Illinois Cedar architecture[133], the Stanford DASH [94] and the CMU Cm* [69] projects. The Cedar machine consisted of a small number of Alliant FX/8 multiprocessors, each containing up to eight pipelined processors. Cedar connected the multiprocessor nodes to a global shared memory via an omega network. The DASH implementation featured clusters of 4-processor Silicon Graphics 4D/340 multiprocessors, connected by a custom interconnect. The DASH provided consistent shared memory via a full directory cache coherence protocol [42]. The Cm* contained ten clusters, each containing of five “computer modules” (processor-memory pairs). Processors could communicate either through the non-uniform shared memory system or by message-passing.
Karlsson and Stenstrom [87] examined performance of an SMP cluster with ATM interconnection and distributed virtual shared memory. Simulated results indicated that no extra advantage was gained by dedicating one SMP processor to act as a dedicated coherence protocol processor, compared to an implementation that scavenged cycles from compute processors to handle coherence protocol processing. Their work assumes that compute processors suffer much idle time due to stalls for remote read requests over the high-latency ATM switch.

Falsafi and Wood [61] also compared implementation tradeoffs when using a communication coprocessor on a SMP cluster with distributed virtual shared memory. They report that dedicating a single SMP processor to handle protocol processing improves performance for high-overhead protocols and communication-intensive applications, while the strategy does not help for low-overhead lightweight DSM protocols.

Erlichson et al. consider implementation tradeoffs for SoftFLASH, a software-based distributed virtual shared memory system implemented on a cluster of SGI Challenge multiprocessors [60]. SoftFLASH, implemented in the Irix operating system, provides shared memory, mutual exclusion, and synchronization facilities for the application programmer. This study examined the tradeoffs of dedicating up to five processors on each SMP node to handle protocol processing. The study concludes that dedicated coprocessors improve performance, but not in proportion to the processing resources consumed.

In a separate study, Erlichson et al. [59] considered the impact of clustering in a shared memory system for various applications. This study concluded that in simulations with realistic hardware assumptions, using SMP nodes in a shared memory multiprocessor does not significantly improve performance compared to uniprocessor nodes on a variety of applications. We note that the applications Erlichson et al. considered were not restructured to exploit the multi-tier memory hierarchy.

Several distributed memory parallel computers devote specialized hardware on each node to handle communication protocols. Several commercial parallel
computers, such as the IBM SP-2 [2], Intel Paragon [111], and Meiko CS-2 [19], augment each node with a general purpose microprocessor to handle message-passing events. This strategy reduces the software overhead of message-passing on the compute processors, and provides the opportunity to overlap communication and computation.

Several distributed shared memory machines also dedicate hardware for protocol processing. The Wisconsin Typhoon architecture [115] dedicates a general-purpose processor from each SMP node to perform protocol processing. In the Stanford FLASH multiprocessor [77], each node contains a custom programmable protocol processor. The Princeton SHRIMP multicomputer [28] implements a hardwired custom network interface to handle cache coherency protocol between commodity workstations.

A few projects implement high-level programming models specifically for SMP clusters. The SIMPLE [14] implementation implements collective operations similar to those provided by KeLP. SIMPLE also implements several levels of parallel control by masking out loop iterations. On the Maryland Digital AlphaServer, SIMPLE implements a custom message-passing layer, shown to be more efficient than MPI on this platform. It might be possible to re-implement KeLP, using SIMPLE to provide some needed underlying multi-tier primitives.

The Fortran-P [120] language has been ported the SGI PowerChallengeARRAY. Like KeLP, the underlying Fortran P implementation devotes a single thread on each SMP to handle communication. Unlike KeLP, the Fortran P implementation manages parallelism at each node using dynamic load balancing based on self-similar partitionings of the domain. The Fortran-P compiler translates self-similar Fortran code to a form usable by the underlying run-time system. It is possible that KeLP facilities would simplify the implementation of the Fortran P run-time system.
3.7 Discussion

3.7.1 Observations

We have described the design and implementation of the KeLP 2.0 programming system.

The message-passing microbenchmark results provide evidence that KeLP handles long messages efficiently. On the two simple message-passing patterns tested, KeLP performance falls with 15% of hand-coded MPI for long messages. Based on these results, we hypothesize that translating a coarse-grain message-passing code to the single-tier KeLP subset will not sacrifice much performance due to data motion concerns. In other words, the programmer can encode data motion with KeLP’s high level MotionPlan and Mover abstractions, without sacrificing performance relative to MPI point-to-point messages. In the next Chapter, we test this hypothesis for several applications.

For short messages, KeLP’s implementation adds significant overhead, up to 75%. We conclude that the KeLP 2.0 implementation will not perform well for applications with frequent fine-grained communication.

We observe that the message-passing microbenchmarks provide only one perspective on the message-passing efficiency of the KeLP implementation. In the next Chapter, we present single-tier KeLP performance results, which assess the message-passing efficiency in the context of complete applications. We note that all of the applications considered in the next Chapter use predominantly long messages.

On the AlphaServer cluster, we saw that a special flow control message-passing implementation outperforms basic message-passing code with long messages. This result emphasizes the advantages of encapsulating data motion information in a high-level abstract form. To tune the message-passing code for the AlphaServers, we need only modify the Mover implementation. Most importantly, the user-level applications need not be modified to take advantage of the more efficient message-passing algorithm. The Mover optimization will apply to
all codes that link with optimized KeLP library, without any modification of the calling application. Thus, the KeLP abstractions separates the expression of correct programs from optimizations that affect performance. This type of separation of concerns results in easier-to-develop, more maintainable code \cite{90}.

The microbenchmark results also document some important hardware characteristics of these platforms. First, we note that codes with poor cache locality quickly saturate the memory system, limiting speedup with more than two processors. Secondly, note that interprocessor synchronization can be quite expensive. By default, KeLP performs a node-level barrier synchronization at the end of each procIterator loop. Thus, the procIterator work should be coarse-grained to amortize this overhead. Alternatively, we can relax the synchronization constraints and try to avoid unnecessary barriers.

### 3.7.2 Limitations

In retrospect, we can identify some mistakes made when designing the KeLP 2.0 implementation. We believe that future implementations of the KeLP programming model could correct these mistakes.

Perhaps the greatest mistake was a failure to address interoperability in the implementation design. We did not specify interoperability as a design goal, and as a result, the KeLP 2.0 infrastructure does not support interoperability with non-KeLP programs.

Since we chose to support systems with unreliable lightweight thread packages, the implementation includes many complex data structures to support reference-counted data structures in a KeLP-managed shared heap. The KeLP classes manage this storage with the support of the smp++ library. However, to interoperate with other codes, the KeLP classes would need to operate on data allocated externally from KeLP. The current implementation’s complicated memory model precludes this functionality.

Much of the implementation’s complexity deals with problems associated with sharing memory between heavyweight processes. If we restricted the tar-
get systems to those that provide reliable POSIX threads, we could substantially reduce the implementation’s complexity.

The KeLP 2.0 implementation uses only one processor one each node to handle Mover activities. So, the KeLP 2.0 implementation serializes all Mover data motion on a node. Perhaps a better implementation could accelerate communication by devoting more than one processor per node to Mover processing. For example, KeLP 2.0 serializes even intra-node Mover data motion. On some architectures, more than one processor can accelerate intra-node memory-to-memory copies.
Chapter 4

Application Studies

In this Chapter, we study KeLP implementations of six block-structured scientific applications. First, we examine three finite difference codes, the application class originally targeted by KeLP. Next, we consider an FFT code and two algorithms from dense linear algebra.

For each code, we present implementation trade-offs and performance consequences of the KeLP programming model. We evaluate KeLP performance on single-tier multicomputers, on a single SMP, and on SMP clusters. On SMP clusters, we restructure each algorithm to tolerate slow inter-node communication delays. We also discuss KeLP domain-specific libraries, which provide specialized functionality for particular problem domains.

4.1 DOCK Domain-Specific Library

The KeLP abstractions provide flexibility to express a wide range of algorithms and computational structures. While each application uses the KeLP abstractions as needed, similar applications will use KeLP in similar ways. With object-oriented design principles, the application programmer can promote software re-use by encapsulating common usage patterns in a class library.

We define a Domain-Specific Library (DSL) to be a collection of classes
that provide functionality germane to a particular application class. KeLP 2.0 implements all KeLP abstractions as C++ classes. A KeLP DSL will usually specialize the KeLP base classes, adding functionality as needed to support a particular application class.

For example, the KeLP FloorPlan can represent any block-structured domain partitioning, including irregular collections of blocks. However, many sections of code require only HPF-style regular BLOCK partitioning s \cite{78}. To promote code re-use in these sections, we have implemented a KeLP DSL to support regular BLOCK partitionings.

The KeLP DOCK (Decomposition Objects for KeLP) library provides domain-specific abstractions to manage regular HPF-style block data decompositions and partitioning decisions. Following the general strategy introduced in pC++ \cite{30}, DOCK provides first-class C++ classes to represent the Fortran-D \cite{67} three stage mapping process adopted in HPF \cite{78}. Unlike pC++ or HPF, DOCK supports two levels of BLOCK data decomposition: either decompositions over multiple SMP nodes, or over the multiple processors at a single SMP node.

In particular, the DOCK library provides the following abstractions:

- **Processors**: The Processors object represents a virtual Processor array, and a mapping from the virtual processors to integers. By default, the Processors object sets up the mapping to correspond to the nodes of the SMP clusters. Typically, the Processors object is used at the collective program level.

- **smpProcessors**: The smpProcessors object is a specialization of Processors that by default maps virtual processors to correspond to the processors at a single SMP node. Typically, the smpProcessors object is used at the node program level.

- **GhostPlan**: The GhostPlan is a specialization of FloorPlan that carries around information regarding ghost cells.

- **Decomposition**: The Decomposition is a specialization of GhostPlan. Given
a Processors object and a Region, the Decomposition `distribute()` function sets up a Floorplan which represents the BLOCK decomposition of the Region over the virtual processor array, mapped to integers as specified by the Processors object.

- **dSection**: The dSection represents a section of a Decomposition, useful for implementing operations over processor subsets.

Figure 4.1 lists the KeLP and DOCK classes, as well as two other DSLs which will be presented later in the chapter.

![Inheritance relationship between KeLP objects and the DOCK, FD-DSL, and dGrid DSLs](image)

Figure 4.1: Inheritance relationships between KeLP objects and the DOCK, FD-DSL, and dGrid DSLs. Arrows indicate inheritance.

As an example of DOCK usage, Figure 4.2 shows a version of the multi-tier redblack3D relaxation routine, previously presented as Figure 2.8. Now, we have rewritten the routine to use the DOCK facilities to perform the intra-node block partitioning used to control the procIterator loop.

The DOCK abstractions prove useful in a variety of contexts, even for globally irregular calculations that have some local regular structure. All of the
Relax(\texttt{XArray} \(X\), \texttt{XArray} \(rhs\), \texttt{MotionPlan} \(M\), integer \(rb\))
begin
\textbf{Mover} \texttt{mov}(\(X\), \(X\), \(M\))
\texttt{mov.start}()
\texttt{mov.wait}()
for \(\texttt{nodeIterator}\ ni(X); ni; ++ni\) \{ 
    integer \(n = ni()\)
    \texttt{smpProcessors} \(P\)
    \texttt{Decomposition} \(D(X.region(n))\)
    \(D.distribute(BLOCK,P)\)
    for \(\texttt{procIterator}\ pi(D); pi; ++pi\) \{ 
        integer \(p = pi()\)
        \texttt{serialRelax}(\(X(n),rhs(n),D(p),rb\))
    \}
\}
end

Figure 4.2: Multi-tier KeLP code for redblack3D relaxation, using the DOCK DSL to perform the intra-node partitioning.

sample applications presented in this dissertation use DOCK.

4.2 Finite Difference Applications

The KeLP programming model explicitly targets finite difference methods for the solution of partial differential equations. Finite difference codes are KeLP’s “bread and butter”; the KeLP abstractions closely match the computational structures arising in these codes. Since KeLP supports irregular block data sets and dynamic irregular block data motion patterns, the KeLP abstractions especially suit multi-block and structured adaptive mesh refinement codes [20, 21].

In this Section, we evaluate performance of KeLP on three finite difference codes. First, we consider the simple second-order red-black Gauss-Seidel Poisson solver presented earlier in Section 2.3. Next, we examine a KeLP version of the
NAS MG multigrid benchmark. Finally, we address irregular and multilevel codes with a multigrid solver over an irregular block domain.

In previous work, Kohn [92] proposed domain-specific library software design for adaptive finite difference codes. The KeLP finite difference codes build on Kohn’s work. In the next Section, we describe the design and implementation of domain-specific abstractions for finite difference codes. Then, we present performance studies of the three target finite difference applications.

4.2.1 Finite Difference Domain-Specific Library

The finite difference applications in this Section all rely on two KeLP domain-specific libraries (DSLs): the DOCK partitioning library and FD-DSL (Finite Difference Domain Specific Library), a DSL to support finite difference codes. The KeLP FD-DSL provides specialized versions of KeLP classes, which match computational structures of the application class. In particular, FD-DSL implements the following classes:

- **mGrid**: An mGrid is a specialization of the KeLP Grid class. The mGrid carries a layer of ghost cells, and provides a few member functions to set initial conditions and compute norms.

- **IrregularGrid**: The IrregularGrid, a specialization of the KeLP XArray class, represents a distributed finite difference grid. This class is described in more detail below.

- **Adder**: The Adder class, a specialization of the Mover, implements data motion with an addition operation at the destination. This facility is useful in certain multigrid algorithms.

The central class of the finite difference DSL is the **IrregularGrid**, based on a facility Kohn specified in [92]. The FD-DSL IrregularGrid provides much the same interface as Kohn’s, although the underlying implementation differs. The IrregularGrid abstraction represents a finite difference grid, physically partitioned
into blocks which are distributed across multiple nodes of a parallel computer. Each block carries a layer of ghost cells to buffer off-node values needed for stencil update operations. The geometry and partitioning may have an irregular shape.

The KeLP implementation of IrregularGrid is derived from an XArray. The IrregularGrid carries around a MotionPlan to describe the communication needed to fill ghost cells. Thus, the IrregularGrid computes the communication schedule only once. To overlap communication and computation, the Irregular Grid provides asynchronous calls to start and wait for the ghost cell data motion. The IrregularGrid also provides a number of collective operations to transfer data between two IrregularGrids.

Figure 4.3 shows the main procedure for redblack3D, as listed in Figure 2.6, recoded to use DOCK and FD-DSL facilities. This routine demonstrates how the DOCK classes perform the inter-node partitioning. The IrregularGrid class computes and carries around a MotionPlan to fill in ghost cells.

FD-DSL facilitates data motion, data decomposition, and a few extremely simple arithmetic and norm calculations for finite difference methods. Other libraries, such as POOMA [118], OVERTURE [36], and PETSc [16] implement more extensive numerical operations for PDE solvers. KeLP supports the development of more extensive and powerful domain-specific libraries [38]. The design and implementation of such libraries raises many algorithmic and software engineering issues that require expert application-specific knowledge. These issues lie beyond the scope of this dissertation.

### 4.2.2 Red-Black Poisson Solver

The first finite difference application, redblack3D, solves Poisson’s equation over a uniform cube using a second-order stencil relaxation. We use red-black Gauss-Seidel ordering to expose parallelism and Dirichlet boundary conditions.

Section 2.3 describes the multi-tier KeLP implementation of redblack3D in detail.

To assess the efficiency of the KeLP primitives for this application, we
Figure 4.3: Main procedure for redblack3D example, using FD-API and DOCK facilities.

c ompare with a hand-coded version calling MPI directly. This C++ code calls the relaxation kernel in Fortran 77. All communication occurs when filling ghost cells. The MPI code for this routine first posts for non-blocking incoming messages, then sends non-blocking outgoing messages, and finally waits for all non-blocking message operations to finish.

We first examine performance on using the SMP clusters as single-tier multicomputers, using only one processor per node. We present results using the two SMP clusters described in Section 3.5.1, the AlphaServer farm and SparcStation farm. In these single-tier experiments, we use only one processor on each SMP node, and ignore the presence of other processors. We scale the problem size with the number of nodes, assigning each node roughly two million grid points. We
Table 4.1: Problem sizes for the redblack3D experiments.

round the problem size to preserve perfect load balance, and the exact problem sizes used are listed in Table 4.1.

Figure 4.4 compares the performance of the KeLP and MPI codes, using the SMP clusters as single-tier multicomputers.

Figure 4.4: Performance of single-tier redblack3D implementations on a) the AlphaServers, and b) the SparcStations.

Using the Alpha farm as a single-tier multicomputer, the KeLP code performs within 8% of the MPI code. On the SparcStations, the KeLP code performs with 1% of the MPI code. In some cases, the KeLP run-time system avoids message buffer-packing for faces that lie contiguously in memory. The MPI code does not perform this optimization. This fact accounts for the slight KeLP performance advantage on the SMP clusters on some experiments.

The single-tier results establish that the KeLP implementation manages
the inter-node communication about as efficiently as MPI. That is, even though
the KeLP program expresses communication with a high-level, abstract interface,
the overheads compared to lower-level message-passing calls are negligible.

Next, we examine the performance of redblack3D on one SMP node. Figure 4.5 shows the performance on one four-processor AlphaServer node and on
one four-processor SparcStation node. The figure compares the multi-tier KeLP
code, parallelized using the procIterator, to the MPI code run with one process
per physical processor. The MPI code requires the number of processes to be a
power of two, so the Figure reports no MPI results for three processors.

![Redblack3D performance on one SMP node: a) AlphaServer b) SparcStation.](image)

The Figure shows that on the AlphaServer, the KeLP code outperforms
the MPI code, and the performance advantage increases with the number of pro-
cessors. This is to be expected, since the KeLP code exploits shared memory
on each node, and does not use ghost cells. The MPI code allocates ghost cells
for each processor, incurring extra data motion. On the SparcStation show less
performance difference, and the two codes perform within 1% of each other.

As the Figure shows, on both platforms, per-processor performance de-
grades as we add more processors. On the AlphaServer, one processor obtains
14.08 MFLOPS, while four processors obtain only 6.44 MFLOPS per processor. One SparcStation processor obtains 5.92 MFLOPS, while four processors obtain only 3.04 MFLOPS per processor. These results show that the shared memory systems of these SMPs cannot support the memory bandwidth demands of four processors on this numeric kernel. The MPI code obtains similar performance, slightly degraded by extra message-passing code to fill internal ghost cells. However, the results indicate that the cost of the extra message-passing is small, as shared memory contention is the major factor limiting performance.

The single-node SMP results expose memory contention problems for this code, even without considering message-passing effects. Now, we examine the performance of the code on the full SMP clusters, using multiple nodes and multiple processors on each node.

Figure 4.6 shows the performance of redblack3D on the two SMP clusters, the AlphaServer cluster and the SparcStation cluster. As before, the problem sizes are listed in Table 4.1. The Figure compares three versions of the code.

The results labeled “MPI” report results from running the MPI code, with one MPI process per physical processor in the system. Thus, on $n$ nodes, these results use $4n$ MPI processes. The MPI code uses an $r \times q$ virtual processor array,
with one virtual processor for each of the \( p \) MPI processes, such that \( rq = p = 4n \). The MPI code maps the virtual processors to physical processors in column-major order. So, for example, on 4 nodes, the MPI code uses a \( 4 \times 4 \) virtual processor array, and each column of the virtual processor array maps to the four processors on each SMP node. In other words, the MPI code maps the processes into a 1D BLOCK node-level decomposition.

Note that this process mapping incurs unnecessary inter-node communication, since it does consider the two-level locality structure of the SMP cluster memory hierarchy. To alleviate this problem, we modified the MPI code to perform an intelligent mapping of the virtual processor array to the physical processor set, in order to effect 2D BLOCK node-level decomposition. The results labeled “MPI: optimized” report the performance using the MPI code with this multi-tier process mapping.

The “multi-tier” KeLP results use the two-level KeLP program described in Section 2.3. This code decomposes the program first across \( n \) nodes, and then parallelizes the local numeric kernels across the four processors at each node using a second level of parallel KeLP constructs.

The “OVERLAP” results use the multi-tier code with explicit communication overlap, also described earlier. In this case, we use only three processors per node for parallelizing the local kernels, dedicating the fourth processor solely to communication.

This strategy potentially wastes cycles on the fourth processor. When not communicating, the fourth processor sits idle. However, the results from Figure 4.5 show that on both platforms, four processors execute at roughly the same speed as three processors. That is, since the SMP nodes’ memory systems cannot support four processors on this kernel, the fourth processor’s idle cycles will not significantly affect aggregate performance.

The results show that the multi-tier programming style outperforms the message-passing codes on the SMP clusters. On eight AlphaServer nodes, the multi-tier KeLP code outperforms the naive MPI code by 105% and the optimized
MPI code by 18%. High message-passing software overheads account for this difference. The MPICH implementation used passes all messages with TCP/IP, even between processors on a single SMP node \(^1\). A more efficient MPI implementation might achieve performance close to the basic multi-tier KeLP code. On four SparcStations, the basic multi-tier code outperforms the naive MPI code by 9%, and performs equivalently to the optimized MPI code.

The Figure also shows that the restructured overlap algorithm significantly improves performance. On eight AlphaServers, the overlap algorithm improves performance by 22% over the basic multi-tier KeLP implementation. On four SparcStations, overlap improves performance by 76%. The trends in the Figure show that the benefits of communication overlap increase with the number of nodes. On these platforms, as we add more nodes, communication demands increase and performance of the underlying interconnection network degrades. In other words, communication costs more when using more nodes, so latency-hiding techniques become more important.

Table 4.2 breaks down the cost of local computation and communication for the multi-tier code on the AlphaServers. The columns labeled ‘Local’ report the time the code spends perform local stencil updates. The columns labeled ‘Comm’ report the time spent waiting for communication to complete. If communication is completely overlapped, then the ‘Comm’ column reports no time waiting for communication to complete. The times reported are the maximum reported from all nodes; thus, the local computation and communication times do not add up exactly to the total time.

Without communication overlap, communication accounts for up to 41% of the total running time. The overlap strategy improves performance by keeping other processors busy during the message-passing. As the table shows, the code spends very little or no time waiting for communication to complete. That is, for all problem sizes considered, the local work dominates the the communication

\(^1\)We tried to employ an MPICH implementation which utilizes shared memory transfers where possible. However, this implementation was unstable and all codes crashed.
Table 4.2: Execution time break-down, in ms, for one iteration of redblack3D on the AlphaServer cluster. The column labeled Comm gives the time the application spends waiting for communication to complete.

<table>
<thead>
<tr>
<th>Nodes x Procs</th>
<th>No Overlap</th>
<th>Overlap</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Comm</td>
<td>Local</td>
</tr>
<tr>
<td>1x4</td>
<td>5.57</td>
<td>609</td>
</tr>
<tr>
<td>2x4</td>
<td>186</td>
<td>564</td>
</tr>
<tr>
<td>4x4</td>
<td>258</td>
<td>539</td>
</tr>
<tr>
<td>8x4</td>
<td>411</td>
<td>680</td>
</tr>
</tbody>
</table>

Table 4.3: Execution time break-down, in ms, for one iteration of redblack3D on the SparcStation cluster. The column labeled Comm gives the time the application spends waiting for communication to complete.

<table>
<thead>
<tr>
<th>Nodes x Procs</th>
<th>No Overlap</th>
<th>Overlap</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Comm</td>
<td>Local</td>
</tr>
<tr>
<td>1x4</td>
<td>7.20</td>
<td>1335</td>
</tr>
<tr>
<td>2x4</td>
<td>806</td>
<td>1420</td>
</tr>
<tr>
<td>4x4</td>
<td>1258</td>
<td>1518</td>
</tr>
</tbody>
</table>

Note that memory contention plays a large role in the efficacy of the overlap strategy. Naively, one might think that dedicating the fourth processor to handle message-passing would not significantly slow down the computational rate. The results demonstrate that this is inaccurate. For example, on one AlphaServer node, three processors execute the local kernel at 8.25 MFLOPS per processor. However, on eight AlphaServers, when using three processors for local computation and the fourth to overlap communication, each processor obtains only 6.79 MFLOPS. The memory traffic generated by the dedicated message co-processor slows down the computation at each processor by 18%.

Table 4.3 shows the execution-time breakdown on the SparcStation clus-
Slow message-passing plays an even greater role in these experiments, accounting for up to 45% of the total running time. On two nodes, memory system contention from communication slows down local computation by 13%. On four nodes, local computation slows down by only 4.5%. That is, the communication co-processor memory traffic does not affect local computation as heavily as on the AlphaServers. Possibly, the message co-processor on each node spends more time idling, waiting for service from the shared Ethernet bus, as opposed to the crossbar network implemented by the AlphaServer cluster ATM switch.

Memory contention works two ways; it also slows down the message-passing performance. Table 4.3 illustrates this phenomenon on four SparcStations. Without communication overlap, message-passing takes 1258 ms. With communication overlap, we perform local computation for 1587 ms, and then wait for communication to complete for an additional 208 ms. In total, communication takes 1795 ms. In other words, overlapping communication and computation at a SparcStation node retarded inter-node communication by 43%.

### 4.2.3 NAS Multigrid Benchmark

The NAS-MG multigrid benchmark [15] solves Poisson’s equation in 3D using a multigrid V-cycle [35]. Each level of the multigrid algorithm involves a relaxation operation, and a residual computation. These two operations involve stencil calculations, and we can parallelize them with techniques similar to those for redblack3D. Each level of the multigrid hierarchy defines a grid at a different resolution. Between single-level relaxations and residual computations, the algorithm transfers values between levels with prolongation and restriction operations.

The NAS-MG code requires ghost cells to be transferred across periodic boundary conditions, and the 27-point stencil requires ghost cells on the corners and edges of each processor’s 3D grid. The NPB 2.1 code, available in the public domain, uses a three-stage dimensional exchange algorithm to satisfy boundary conditions.

Figure 4.7 illustrates the dimensional exchange in two dimensions. The
2D algorithm has two steps. In the first step, each processor exchanges ghost cells along just one dimension of the domain. So, in Figure 4.7a, node 1 sends a strip of ghost cells to node 2. Similarly, although not shown, node 2 sends ghost cells to node 1, and nodes 3 and 4 exchange ghost cells values. In the second step of the algorithm, each processor exchanges ghost cells along the other dimension. As shown in Figure 4.7b, node 2 sends ghost cells to node 4. However, note that node 2 includes in this message the corner value previously obtained from node 1, shown in grey. Thus, after the second step, node 4 has obtained a corner ghost cell value from node 1 by way of node 2. Similarly, all other corner ghost cells are filled by this two-step process.

Figure 4.7: A two-stage dimensional exchange algorithm used for ghost cell updates in 2D. a) Stage 1 b) Stage 2. The Figure only shows messages passed which affect the final state of the left edge of node 4. Other messages are omitted, but symmetric to those shown.

The dimensional exchange algorithm reduces the number of message starts, compared to direct transmission of all ghost cell values. In 3D, each node has 26 neighbors. However, the three-step dimensional exchange algorithm communicates with only six neighbors. Additionally, using MPI, the dimensional exchange al-
Algorithm requires less bookkeeping than sending 26 messages, keeping track of all edges, corners, and faces.

When implementing the NAS-MG code in KeLP, we will overlap the ghost cell data motion with local computation, as demonstrated for redblack3D. In redblack3D, one Mover handles all the ghost cell data motion. For NAS-MG, three Movers are required, which complicates the situation. The first Mover exchanges ghost cells along the 1st dimension (x axis), the second Mover exchanges ghost cells along the 2nd dimension (y axis), and the third Mover communicates along the 3rd dimension (z axis). We wish to asynchronously start the data motion, which must invoke three Movers in succession. We cannot naively start all three Movers at once, since we must enforce the completion of Mover 1 before invoking Mover 2, and similarly between Movers 2 and 3. This problem would also crop up using MPI non-blocking messages.

To solve this problem, the NAS-MG KeLP implementation uses an object called the MultiMover. The MultiMover, derived from Mover, serially invokes a sequence of Movers. The MultiMover implementation stores a linked list of pointers to Movers. When called to perform data motion, the MultiMover executes each Mover in succession. To the calling application, the three-stage MultiMover execution appears to be an atomic operation.

Recall that when implementing KeLP, we considered two methods to overlap communication with computation. The first method uses MPI non-blocking message calls, and the second devotes an extra thread to communication. This communication example highlights the superior expressive power of the thread solution over the MPI non-blocking calls. By using an extra thread, the programmer can asynchronously execute an arbitrary sequence of message-passing and synchronization operations. In contrast, the MPI calls asynchronously start only one message-passing operation at a time. To overlap communication using a sequence of operations, the MPI program must periodically poll for the completion of non-blocking message calls, in order to start the next sequence of calls in a timely manner. The programmer may have to interrupt highly-tuned numeric kernels to
poll, degrading performance and tangling program structure. The more powerful thread design lends itself to better structured and more efficient code.

We now evaluate the single-tier, single-node, and multi-tier performance of the KeLP NAS-MG implementation. For reference, we consider the NAS Parallel Benchmarks 2.1 MPI implementation.

We scale the problem size with the number of nodes. However, since the NAS-MG code restricts the problem size to powers of two, we have only limited flexibility. On the AlphaServers, we use a fine grid of 128x128x128 on one and two nodes, and a fine grid of 256x256x256 on four and eight nodes. On the slower SparcStations, we hold the problem size constant at 128x128x128 on one, two, and four nodes. For all experiments, we report times for 10 multigrid V-cycles.

Figure 4.8 shows the single-tier performance on three platforms, using only one processor per node. Using one processor per node on the AlphaServers, the KeLP code outperforms the MPI code by up to 17%. The KeLP Mover implementation avoids extraneous message buffer-packing for some faces, which accounts for the KeLP performance advantage. On the SparcStations, the single-tier KeLP code performs within 4% of the MPI code.

We conclude from these results that the KeLP implementation efficiently handles message-passing for the NAS-MG code, incurring only slight performance penalties for utilizing KeLP’s higher-level programming abstractions.

Figure 4.9 shows the NAS-MG performance on one SMP node. The Figure compares the KeLP multi-tier code to the MPI code, using one MPI process per physical processor. Since the MPI code requires the number of processes to be a power of two, the Figure does not report MPI results on three processors.

The Figure shows that the KeLP implementation, using shared memory, outperforms the MPI code on one SMP node. The MPI code uses ghost cells to communicate between processors, while the KeLP code uses shared memory to avoid unnecessary ghost cell data motion. On one AlphaServer, the KeLP code outperforms the MPI code by up to 26%. On one SparcStation, the KeLP code outperforms the MPI code by up to 11%.
The results also show that memory system contention plays a large role in this application’s performance. On both platforms, the shared memory system cannot support the memory bandwidth requirements of four processors executing the multigrid computation. One AlphaServer processor achieves 27.63 MFLOPS, but four processors achieve only 12.78 MFLOPS per processor. One SparcStation processor achieves 7.78 MFLOPS, but four processors achieve only 5.67 MFLOPS per processor.

Figure 4.10 shows the performance of the multi-tier code on multiple SMP nodes, using multiple processors per node. The Figure compares three versions of the code. The MPI code is the NPB2.1 reference implementation, run with one MPI process on each physical processor. The NPB2.1 code uses a BLOCK virtual processor array similar to that described earlier for redblack3D. However, the NPB2.1 code assigns virtual processors to physical processors with a recursive strategy that happens to preserve locality between communicating processes on an SMP node. That is, the NPB2.1 process decomposition matches the optimized process mapping considered earlier for redblack3D. For the “multi-tier” data reports results from basic two-level KeLP code. This code first decomposes the problem across SMP nodes, and then parallelizes each local kernel independently using the
procIterator.

The “OVERLAP” code modifies the “multi-tier” code to overlap communication and computation. The multigrid code performs several numeric operations on each grid, including relaxation, computation of the residual, prolongation, and restriction. The relaxation and residual computations operate on a single level of the multigrid hierarchy. We overlap communication and computation in these stencil operations using a similar strategy to that presented earlier for the redblack3D code. However, in the NAS-MG code, we update ghost cells after updating local values. Thus, the overlap algorithm proceeds as follows:

1. Update the annulus of each node’s grid.

2. Begin ghost cell communication.

3. Update the interior of each node’s grid.

4. Wait for ghost cell communication to complete.

We distribute grids on each level of the multigrid hierarchy so that prolongation and restriction incur no inter-node communication. We do not attempt
to overlap communication and computation for these activities.

The results show that the basic multi-tier KeLP code outperforms the MPI code by 4% on eight AlphaServers. The MPI code outperforms the multi-tier KeLP code by 7% on four SparcStations. The communication overlap algorithm improves performance by up to 12% on the Alphas and 20% on the SparcStations.

![Graph A](image1.png)  
![Graph B](image2.png)  

Figure 4.10: NAS-MG performance on a) the AlphaServer cluster and b) the SparcStation cluster.

Table 4.4 shows the breakdown of costs, by level, for the two multi-tier KeLP codes on eight AlphaServers. The multigrid hierarchy has 8 levels, with 7 as the finest level and 0 as the coarsest. The Table breaks the code into five activities: local stencil smoothing, residual calculation, filling ghost cells, coarsen, and restrict. Only the ghost cell activity incurs message-passing communication. The Table reports the maximum time any one node spends in each activity, so the sum of the reported times may not add up to the total time.

The Table shows that activity at the finest level of the multigrid hierarchy dominates the running time, and that communication to fill ghost cells accounts for 49% of the total running time.

The overlap of communication and computation effectively hides the communication at the finest level, reducing the time waiting for communication on level 7 from 12.56 seconds to 2.49 seconds. However, at coarser levels of the hierarchy,
<table>
<thead>
<tr>
<th>Level</th>
<th>Smooth</th>
<th>Residual</th>
<th>FillGhost</th>
<th>Coarsen</th>
<th>Refine</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Non-Overlapped Algorithm</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>3.61</td>
<td>11.09</td>
<td>12.56</td>
<td>1.78</td>
<td>1.80</td>
</tr>
<tr>
<td>6</td>
<td>0.367</td>
<td>0.363</td>
<td>3.39</td>
<td>0.207</td>
<td>0.312</td>
</tr>
<tr>
<td>5</td>
<td>0.070</td>
<td>0.074</td>
<td>1.11</td>
<td>0.039</td>
<td>0.070</td>
</tr>
<tr>
<td>4</td>
<td>0.023</td>
<td>0.023</td>
<td>0.578</td>
<td>0.020</td>
<td>0.027</td>
</tr>
<tr>
<td>3</td>
<td>0.019</td>
<td>0.012</td>
<td>0.516</td>
<td>0.016</td>
<td>0.020</td>
</tr>
<tr>
<td>2</td>
<td>0.016</td>
<td>0.019</td>
<td>0.503</td>
<td>0.008</td>
<td>0.012</td>
</tr>
<tr>
<td>1</td>
<td>0.016</td>
<td>0.012</td>
<td>0.387</td>
<td>0.000</td>
<td>0.016</td>
</tr>
<tr>
<td>0</td>
<td>0.000</td>
<td>0.000</td>
<td>0.164</td>
<td>0.000</td>
<td>0.000</td>
</tr>
<tr>
<td>All</td>
<td>4.12</td>
<td>11.60</td>
<td>19.22</td>
<td>2.07</td>
<td>2.26</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Overlapped Algorithm</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>5.25</td>
<td>13.90</td>
<td>2.49</td>
<td>1.83</td>
<td>1.96</td>
</tr>
<tr>
<td>6</td>
<td>0.793</td>
<td>0.832</td>
<td>2.88</td>
<td>0.242</td>
<td>0.223</td>
</tr>
<tr>
<td>5</td>
<td>0.215</td>
<td>0.230</td>
<td>0.816</td>
<td>0.043</td>
<td>0.063</td>
</tr>
<tr>
<td>4</td>
<td>0.152</td>
<td>0.148</td>
<td>0.328</td>
<td>0.023</td>
<td>0.027</td>
</tr>
<tr>
<td>3</td>
<td>0.133</td>
<td>0.129</td>
<td>0.316</td>
<td>0.020</td>
<td>0.020</td>
</tr>
<tr>
<td>2</td>
<td>0.121</td>
<td>0.121</td>
<td>0.328</td>
<td>0.023</td>
<td>0.012</td>
</tr>
<tr>
<td>1</td>
<td>0.115</td>
<td>0.109</td>
<td>0.297</td>
<td>0.012</td>
<td>0.012</td>
</tr>
<tr>
<td>0</td>
<td>0.078</td>
<td>0.000</td>
<td>0.113</td>
<td>0.000</td>
<td>0.000</td>
</tr>
<tr>
<td>All</td>
<td>6.86</td>
<td>15.47</td>
<td>7.57</td>
<td>2.19</td>
<td>2.31</td>
</tr>
</tbody>
</table>

Table 4.4: Execution-time breakdown, in seconds, for 10 V-cycles of the multi-tier KeLP NAS-MG codes on eight AlphaServers.

The overlap strategy is not effective. Examining the breakdown, we see that coarser levels simply do not perform enough local computation to hide the communication load.

Furthermore, the Table show that memory contention degrades performance of the local computation when overlapping communication and computation. For example, the total time spent in smoothing increases from 4.12 seconds without communication overlap to 6.86 seconds in the overlapped code. Several factors contribute to this slowdown. Most importantly, as discussed for the red-black3D code, message-passing activity generates traffic on the shared memory subsystem that slows down performance on other processors. Additionally, recall
that for the overlap algorithm, we process the annulus of each node’s grid independently. For the NAS-MG code with periodic boundary conditions, these annulus computations inefficiently use cache, compared to sweeping over the entire grid as a whole.

Table 4.5 shows the execution time breakdown for four SparcStation nodes. With a slower communication network, the SparcStation cluster is more communication-bound than the AlphaServers, with ghost cell communication accounting for 71% of the total running time. Even on the finest multigrid level, the code does not perform enough local computational work to hide all the communication. On the finest level, the local computation takes roughly half the time of the communication, so the overlap algorithm hides only half the communication time. Once again, memory contention slows down local computation; in this case by 55%.

Although we have documented the effects of memory contention on the local computational rate, note that for these communication-bound codes, the local computational rate slowdown does not impact the application’s performance. Since there is not enough computational work to hide communication, processors sit idle waiting for communication to complete. Speeding up the local computational rate would only increase the idle time. In other words, the high communication costs leave slack for memory contention effects.

4.2.4 Multi-block Multigrid

The previous two examples evaluated KeLP support for explicit finite difference codes with regular geometries. However, KeLP targets irregular geometries as well. To examine KeLP support for finite difference codes on irregular domains, we have implemented a 2D multigrid solver for an irregular domain.

With the previous two Sections, we presented evidence that the KeLP infrastructure handles regular finite difference methods efficiently by comparing the KeLP implementation with an MPI code. An MPI implementation of the multiblock multigrid code would be significantly more complex, due to the irregular grid
<table>
<thead>
<tr>
<th>Level</th>
<th>Smooth</th>
<th>Residual</th>
<th>FillGhost</th>
<th>Coarsen</th>
<th>Refine</th>
<th>Non-Overlapped Algorithm</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>2.52</td>
<td>6.02</td>
<td>21.68</td>
<td>1.24</td>
<td>1.47</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>0.328</td>
<td>0.325</td>
<td>5.80</td>
<td>0.199</td>
<td>0.261</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>0.054</td>
<td>0.053</td>
<td>1.69</td>
<td>0.039</td>
<td>0.061</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>0.017</td>
<td>0.017</td>
<td>0.750</td>
<td>0.012</td>
<td>0.026</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>0.011</td>
<td>0.011</td>
<td>0.515</td>
<td>0.010</td>
<td>0.017</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>0.010</td>
<td>0.009</td>
<td>0.467</td>
<td>0.009</td>
<td>0.015</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0.009</td>
<td>0.014</td>
<td>0.390</td>
<td>0.009</td>
<td>0.013</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0.001</td>
<td>0.000</td>
<td>0.180</td>
<td>0.000</td>
<td>0.000</td>
<td></td>
</tr>
<tr>
<td>All</td>
<td>2.97</td>
<td>6.45</td>
<td>31.48</td>
<td>1.52</td>
<td>1.86</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Level</th>
<th>Smooth</th>
<th>Residual</th>
<th>FillGhost</th>
<th>Coarsen</th>
<th>Refine</th>
<th>Overlapped Algorithm</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>3.68</td>
<td>8.56</td>
<td>11.69</td>
<td>0.928</td>
<td>1.53</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>0.555</td>
<td>0.575</td>
<td>4.57</td>
<td>0.135</td>
<td>0.236</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>0.212</td>
<td>0.198</td>
<td>1.43</td>
<td>0.028</td>
<td>0.061</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>0.109</td>
<td>0.102</td>
<td>0.617</td>
<td>0.014</td>
<td>0.025</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>0.088</td>
<td>0.090</td>
<td>0.382</td>
<td>0.009</td>
<td>0.016</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>0.086</td>
<td>0.090</td>
<td>0.338</td>
<td>0.010</td>
<td>0.015</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0.085</td>
<td>0.081</td>
<td>0.246</td>
<td>0.010</td>
<td>0.010</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0.070</td>
<td>0.000</td>
<td>0.104</td>
<td>0.000</td>
<td>0.000</td>
<td></td>
</tr>
<tr>
<td>All</td>
<td>4.88</td>
<td>9.70</td>
<td>19.38</td>
<td>1.13</td>
<td>1.90</td>
<td></td>
</tr>
</tbody>
</table>

Table 4.5: Execution-time breakdown, in seconds, for 10 V-cycles with the multi-tier KeLP NAS-MG codes on four SparcStations.

geometries and communication patterns. Instead, we compare the performance of the KeLP multi-block code against the same code on a regular geometry.

Like the previous two examples, the multi-block code solves Poisson’s equation. However, several factors distinguish the multi-block code from the previous examples. Most importantly, the multi-block code accepts any grid geometry as run-time input. The multi-block code solve the 2D Poisson equation, unlike the previous 3D codes. The 2D stencil relaxation involves a 5-point block Gauss-Seidel relaxation stencil, which changes the communication-computation ratio compared to the NAS-MG 27-point Stencil. The multi-block code performs four smoothing operations per multigrid level; the NAS-MG code performs only one. Finally, the
multi-block code implements Dirichlet boundary conditions; unlike the NAS-MG code's periodic boundaries.

In the following experiments, we solve Poisson's equation over a grid structure covering the geometrical shape of Lake Superior. We obtained grid geometries generated and partitioned across nodes by Yingxin Pang (Department of Computer Science, UCSD) [108], using a heuristic based on work described in [113]. The grid geometries result from laying a quad-tree over the domain, and assigning grids to processors using the recursive spectral bisection partitioning algorithm [112]. At several phases of the grid generation process, Pang combines adjacent small grids in order to reduce the total number of grids, increase patch granularity, and better differentiate small eigenvalues during the spectral partitioning algorithm. Figure 4.11 shows the resultant grids and partitioning assignments generated by Pang's process.

We now assess the efficiency of the KeLP programming model for the multi-block multigrid solver. We compare performance on the irregular multi-block domain to solving multigrid over a square domain, block partitioned across the nodes. In each case, we use five levels of multigrid. We scale the problem size with the number of nodes. On the AlphaServers, we use roughly four million grid points per node. On the slower SparcStations, we use roughly one million grid points per node.

Figure 4.12 presents the single-tier performance on these codes, using only one processor per node. On eight AlphaServers, the square domain outperforms the irregular domain by 28%. Results on four SparcStations show a performance differential of 19%.

Several factors account for this performance differential. First, while the regular domain is perfectly load balanced, the irregular domain suffers from some load imbalance. Table 4.6 shows the number of grid points assigned per node. The Table reports the maximum, minimum, and mean load assigned to a node, measured in grid points on the finest level. The load imbalance is the difference between the maximum load and the mean load, divided by the mean load.
Figure 4.11: Grid geometries used for the multi-block multigrid code. The colors denote the assignment of grids to nodes on a) two nodes, b) four nodes, and c) eight nodes.
Secondly, the irregular partitioning incurs more communication. Table 4.7 shows the number of ghost cells transferred over the interconnection network. The Table reports the maximum number of ghost cells handled by any one node, the minimum number, and the average number. As the Table shows, the irregular domain incurs more inter-node communication than the square domain, due to the irregular cuts of the global domain. Furthermore, the communication load exhibits imbalance, further exacerbating the load balance problems mentioned previously.

In contrast to the previous finite difference code examples, the multi-block code assigns multiple grids to each node. To parallelize the computation at each node, we chose to parallelize the computation over each block individually. That is, we parallelize the computation in each block, rather than parallelizing across blocks. This intra-node partitioning gives perfect load balance; each processor receives the same amount of work.

Figure 4.13 shows the performance on one AlphaServer node and on one SparcStation. The Figure shows that the square geometry outperforms the irregular geometry, even on one node. On four Alpha processors, the square geometry outperforms the irregular shape by 39%. On four SuperSparcs, the square geome-
try outperforms the irregular one by 111%.

Note that load imbalance does not affect performance here, since we partition each block evenly among the processors. Two factors account for the performance differential. First, the irregular domain entails ghost cells to be filled between grids, while the square domain uses only one block per node, so needs no ghost cells. The KeLP 2.0 implementation does not parallelize local data motion, so the intra-node ghost cell communication is a serial bottleneck.

Secondly, the lake geometry entails more interprocessor synchronization. Figure 4.14a shows the structure of the main procedure to perform relaxation. On one node, the Lake Superior geometry consists of 47 grids, so the nodeIterator loops is executed 47 times. For each iteration, we parallelize the relaxation of a grid among the four processors. The procIterator loop accomplishes this.

Recall that KeLP semantics mandate a logical synchronization point at the end of a procIterator loop. The KeLP 2.0 implementation enforces a node-level barrier synchronization at the end of the barrier loop. Thus, with the Lake Superior geometry, each call to Relax incurs 47 node-level barriers.

However, in this application, each nodeIterator iteration may proceed

<table>
<thead>
<tr>
<th>AlphaServers</th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Nodes</td>
<td>Max</td>
<td>Min</td>
<td>Mean</td>
<td>Imb</td>
</tr>
<tr>
<td>1</td>
<td>4117025</td>
<td>4117025</td>
<td>4117025</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>4079152</td>
<td>3990217</td>
<td>4034684</td>
<td>1.1</td>
</tr>
<tr>
<td>4</td>
<td>4609829</td>
<td>3564300</td>
<td>4042510</td>
<td>14.0</td>
</tr>
<tr>
<td>8</td>
<td>5217520</td>
<td>2798499</td>
<td>4034686</td>
<td>29.3</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>SparcStations</th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Nodes</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1067024</td>
<td>1067024</td>
<td>1067024</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>1046448</td>
<td>1023633</td>
<td>1035040</td>
<td>1.1</td>
</tr>
<tr>
<td>4</td>
<td>1173700</td>
<td>907500</td>
<td>1029256</td>
<td>14.0</td>
</tr>
</tbody>
</table>

Table 4.6: The number of grid points assigned to nodes with the Lake Superior multi-block geometry.
Table 4.7: The number of ghost cells, per node, which require inter-node communication in the 2D multi-block multigrid experiments.

<table>
<thead>
<tr>
<th>Nodes</th>
<th>AlphaServers</th>
<th>Lake Superior Geometry</th>
<th>Square Geometry</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Max</td>
<td>Min</td>
</tr>
<tr>
<td>1</td>
<td></td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>3784</td>
<td>3783</td>
<td>3783</td>
</tr>
<tr>
<td>4</td>
<td>8292</td>
<td>2513</td>
<td>5786</td>
</tr>
<tr>
<td>8</td>
<td>10018</td>
<td>3238</td>
<td>6937</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Nodes</th>
<th>SparcStations</th>
<th>Lake Superior Geometry</th>
<th>Square Geometry</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Max</td>
<td>Min</td>
</tr>
<tr>
<td>1</td>
<td></td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>1922</td>
<td>1921</td>
<td>1921</td>
</tr>
</tbody>
</table>

independently. Since there are no data dependencies between iterations, there is no need for a synchronization point. So, to improve performance, we turn off the synchronization of the procIterator. Figure 4.14b shows the revised KeLP 2.0 code.

Figure 4.13 shows the performance of the relaxed synchronization version. On the AlphaServer, the relaxed synchronization loop improves performance by up to 6%. On the SparcStation, the technique improves performance by up to 25%. For the remainder of the Section, all results use the relaxed synchronization strategy.

Figure 4.15 shows multi-tier performance on multiple nodes, using multiple processors per node. As before, we consider both a basic version and a version that explicitly overlaps communication and computation. Once again, we overlap communication and computation by deferring the relaxation on each grid’s annulus, as described for the previous finite difference applications.

The Figure illustrates the performance consequences of the irregular geometry. On eight AlphaServer nodes, the irregular geometry incurs a performance hit of 22% as compared to the square domain. Four SparcStations incur a performance hit of 23%.
Figure 4.13: Multi-block multigrid performance on a) one AlphaServer node and b) one SparcStation node.

The Figure also shows that the overlap optimization is not effective, for either the square or the Lake Superior domain. Tables 4.8 and 4.9 show the breakdown of execution time for the codes. The Tables report the maximum time spent by any node in each Section of the code. Thus, in the presence of load imbalance, the sum of the reported times will not add up to the total time for the program.

For the square domain on eight AlphaServers, filling in ghost cells accounts for 19% of the total running time. On the finest level of the multigrid cycle, the overlap algorithm reduces this time by 15%. However, on coarser levels of the hierarchy, communication time dominates the computation time. For these levels, the overlap strategy backfires, and actually degrades communication performance. Table 4.9 shows similar behavior on the four SparcStations. The performance degradation at the coarser levels nullifies the performance gain at the finest level.

Additionally, the overlap algorithm did not improve performance on the Lake Superior geometry. Note that for this irregular geometry, on eight AlphaServers, the sum of the reported times significantly exceeds the total time reported. As mentioned previously, this pattern indicates load imbalance among the nodes. In particular, at least one node spends most of its time (82%) waiting
XArray $X$

/* loop over each block of $X$/

for (nodeIterator $ni(X); ni; ++ni) {
    /* intra-node parallelization of each block */
    for (procIterator $pi(...); pi; ++pi) {
        serialRelax(..)
    }
}

Figure 4.14: a) Structure of the basic Multi-tier Multiblock relaxation loop. b) A modified version to reduce intra-node synchronization.
Table 4.8: Execution-time breakdown, in seconds, for 10 V-cycles of the multi-tier KeLP multiblock multigrid code on eight AlphaServers. For each code section, the Table reports the maximum time spent in that section by any node; therefore, the sum of the individual sections does not add up to the total running time.
<table>
<thead>
<tr>
<th>Level</th>
<th>Smooth</th>
<th>Residual</th>
<th>FillGhost</th>
<th>Coarsen</th>
<th>Refine</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Square Domain: Non-Overlapped Algorithm</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>10.85</td>
<td>3.79</td>
<td>2.21</td>
<td>2.22</td>
<td>4.04</td>
</tr>
<tr>
<td>3</td>
<td>2.59</td>
<td>0.93</td>
<td>1.19</td>
<td>0.52</td>
<td>0.78</td>
</tr>
<tr>
<td>2</td>
<td>0.55</td>
<td>0.16</td>
<td>0.72</td>
<td>0.11</td>
<td>0.13</td>
</tr>
<tr>
<td>1</td>
<td>0.17</td>
<td>0.05</td>
<td>0.48</td>
<td>0.04</td>
<td>0.04</td>
</tr>
<tr>
<td>0</td>
<td>0.08</td>
<td>0.00</td>
<td>0.34</td>
<td>0.00</td>
<td>0.00</td>
</tr>
<tr>
<td>All</td>
<td>14.23</td>
<td>4.93</td>
<td>4.94</td>
<td>2.88</td>
<td>4.99</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Entire Program: 32.21</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Square Domain: Overlapped Algorithm</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>11.07</td>
<td>3.63</td>
<td>1.98</td>
<td>3.74</td>
<td>4.01</td>
</tr>
<tr>
<td>3</td>
<td>2.51</td>
<td>0.91</td>
<td>3.91</td>
<td>0.79</td>
<td>0.80</td>
</tr>
<tr>
<td>2</td>
<td>0.36</td>
<td>0.14</td>
<td>1.75</td>
<td>0.10</td>
<td>0.15</td>
</tr>
<tr>
<td>1</td>
<td>0.12</td>
<td>0.05</td>
<td>0.51</td>
<td>0.03</td>
<td>0.04</td>
</tr>
<tr>
<td>0</td>
<td>0.14</td>
<td>0.00</td>
<td>0.33</td>
<td>0.00</td>
<td>0.00</td>
</tr>
<tr>
<td>All</td>
<td>14.19</td>
<td>4.72</td>
<td>8.48</td>
<td>4.66</td>
<td>5.01</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Entire Program: 34.30</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Lake Superior Domain: Non-Overlapped Algorithm</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>13.48</td>
<td>4.52</td>
<td>5.17</td>
<td>2.74</td>
<td>3.62</td>
</tr>
<tr>
<td>3</td>
<td>3.16</td>
<td>1.09</td>
<td>3.37</td>
<td>0.77</td>
<td>0.85</td>
</tr>
<tr>
<td>2</td>
<td>0.86</td>
<td>0.25</td>
<td>1.29</td>
<td>0.33</td>
<td>0.30</td>
</tr>
<tr>
<td>1</td>
<td>0.43</td>
<td>0.12</td>
<td>0.86</td>
<td>0.28</td>
<td>0.20</td>
</tr>
<tr>
<td>0</td>
<td>0.40</td>
<td>0.00</td>
<td>0.65</td>
<td>0.00</td>
<td>0.00</td>
</tr>
<tr>
<td>All</td>
<td>18.32</td>
<td>5.98</td>
<td>11.34</td>
<td>4.12</td>
<td>4.96</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Entire Program: 40.72</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Lake Superior Domain: Overlapped Algorithm</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>14.31</td>
<td>4.86</td>
<td>12.49</td>
<td>4.11</td>
<td>3.62</td>
</tr>
<tr>
<td>3</td>
<td>2.99</td>
<td>0.99</td>
<td>6.85</td>
<td>0.97</td>
<td>0.86</td>
</tr>
<tr>
<td>2</td>
<td>1.44</td>
<td>0.37</td>
<td>1.21</td>
<td>0.26</td>
<td>0.32</td>
</tr>
<tr>
<td>1</td>
<td>1.44</td>
<td>0.45</td>
<td>0.63</td>
<td>0.19</td>
<td>0.22</td>
</tr>
<tr>
<td>0</td>
<td>1.46</td>
<td>0.00</td>
<td>0.33</td>
<td>0.00</td>
<td>0.00</td>
</tr>
<tr>
<td>All</td>
<td>21.64</td>
<td>6.67</td>
<td>21.50</td>
<td>5.53</td>
<td>5.02</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Entire Program: 44.07</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 4.9: Execution-time breakdown, in seconds, for 10 V-cycles of the multi-tier KeLP multiblock multigrid code on four SparcStations. For each code section, The Table reports the maximum time spent in that section by any node; therefore, the sum of the individual sections does not add up to the total running time.
Figure 4.15: Multi-tier performance of the multi-block multigrid code on a) the AlphaServer cluster and b) the SparcStation cluster.

for communication to complete. We suspect that due to load imbalance, the most lightly loaded node finishes each iteration early, and spends most of its time waiting for the most heavily loaded node to catch up. Perhaps this pattern defeats the overlap strategy, since the most heavily loaded node must forever attempt to catch up with the others. If so, the most heavily loaded node does not overlap communication and computation. Since this node is the bottleneck, it defeats the overlap strategy.

Table 4.9 reports similar patterns on the SparcStation cluster.

We conclude that load imbalance in the grid geometries causes the performance differential between the square and Lake Superior geometries in all cases. The overlap algorithm fails to improve performance for both geometries. While load imbalance could account for the failure of overlap on the lake geometry, other system interactions defeat the strategy on the square geometry.

4.3 Other Applications

In the last Section, we examined KeLP support for finite difference codes. While KeLP abstractions best suit finite difference codes, many real codes include
several algorithms in different phases of the code. In this Section, we examine whether KeLP supports the expression of other block-structured algorithms, including the FFT and some efficient blocked algorithms for dense linear algebra.

In this Chapter, we examine several applications with these structures. In particular, we evaluate KeLP implementations of the NAS Fourier Transform benchmark, the SUMMA matrix multiplication algorithm, and dense Gaussian Elimination with Partial Pivoting.

4.3.1 Distributed Grid Domain-Specific Library

When discussing the finite difference applications, we presented domain-specific abstractions to support irregular grid collections and ghost cells. The applications in this Section give rise to different computational structures. In many cases, these codes rely on regular HPF-style data decompositions, replicated data structures, and processor subsets.

To simplify implementation of these types of codes, we have developed the \texttt{dgrid} DSL. This DSL contains the following classes:

- \texttt{arithGrid}: The \texttt{arithGrid} class, derived from Grid, implements a Grid of double. The \texttt{arithGrid} provides a few BLAS-like operations, to simplify coding of some global matrix operations and linear algebra.

- \texttt{dGrid}: The \texttt{dGrid} class, derived from XArray, represents an XArray of \texttt{arithGrid} that happens to have a regular BLOCK decomposition, represented by the DOCK Decomposition class. The \texttt{dGrid} provides a number of built in operations to support global matrix operations and linear algebra.

- \texttt{replicatedGrid}: The \texttt{replicatedGrid} class, derived from XArray, provides an abstraction to represent a Grid whose values are replicated over a collection of SMP nodes. The key member function, \texttt{replicate}, broadcasts the values of the Grid from a source to all instances of the replicated Grid, thus enforcing a consistent view of the object.
- **CollectiveGroup**: The CollectiveGroup class provides a KeLP analog of the MPI Communicator; a subset of the global node set. The CollectiveGroup is useful for operations that should be performed only on a subset of the nodes, including reductions and broadcasts.

In order to implement the dGrid library efficiently, in a few cases the code drops below the KeLP abstractions, and utilizes the smp++ and mp++ abstraction layers directly. For example, lightweight dGrid operations drop down to smp++ to avoid unnecessary barrier synchronizations. The CollectiveGroup class deals directly with MPI Communicator objects, which lie below the KeLP level.

### 4.3.2 NAS-FT Benchmark

The NAS Fourier Transform (FT) benchmark solves a 3D diffusion equation for an unknown $u(x, y, z, t)$ [15]:

$$\frac{du}{dt} = \alpha \nabla^2 u$$

(4.1)

After applying a Fourier transform to each side of this equation, the solution is given by the function

$$f(x, y, z, t) = e^{-4\alpha t} \|x\|^2 v(x, y, z, t = 0)$$

(4.2)

where $v(t = 0)$ is the Discrete Fourier Transform of $u(t = 0)$.

The FT benchmark uses this formula to compute the solution to the diffusion equation for a number of time steps.

The structure of the application is as follows, with $U$, $W$, and $V$ are 3D double-precision complex arrays:

The NAS Parallel Benchmark suite version 2.1, like many parallel 3D FFT codes, uses a 1D block data decomposition. The 3D array $U$ is distributed block-wise along the 3rd(Z) axis. Each node owns a slab of data, spanning the entire 1st(X) and 2nd(Y) dimensions. To perform a forward 3D FFT, we first perform 1D FFTs along the X axis, then along the Y axis, then along the Z axis. The X
Set initial conditions of $U$
$V = \text{Forward 3D-FFT of } U$

\begin{verbatim}
for $t = 0, T - 1$
    $W_t = f(V, t)$
    $U_t = \text{Inverse 3D-FFT of } W_t$
    Normalize $U_t$
    Compute Check Sum on $U_t$
\end{verbatim}

Figure 4.16: Structure of the NAS-FT benchmark.

and Y-axis FFTs may be performed without any inter-node communication, given the decomposition of $U$ as stated. Then, we perform a global matrix transpose, permuting the axes so the Z axis interchanges with the X axis. On the transposed array, the original Z axis is not distributed, so the final set of 1D FFTs incurs no inter-node communication.

The inverse transform follows a similar algorithm. To perform an inverse 3D FFT on array $W$:

1. Perform a set of inverse 1D FFTs along the first dimension of $W$ (corresponding to the third dimension of the original matrix $U$).
2. Perform a global matrix transpose of $W$.
3. Perform a set of inverse 1D FFTs along the first dimension of $W$.
4. Perform a set of inverse 1D FFTs along the second dimension of $W$.

The global matrix transpose step can be very expensive, especially for SMP clusters. To tolerate this overhead, Agarwal et al. [1] presented a restructured version of the NAS-FT benchmark to explicitly overlap communication and computation. Their algorithm pipelines the inverse 3D FFT with the other computation in the time-stepping loop. Agarwal et al.’s presentation of the algorithm
involves three timesteps in the main loop. When implementing the algorithm, we found it conceptually easier to re-order pipeline stages so the main loop manages data structures from only two timesteps. Our formulation of the algorithm appears in Figure 4.17.

Set initial conditions on $U$
$V = $ Forward 3D-FFT of $U$
$W_0 = f(V, 0)$
**Start** $U_0 = $ Inverse FFT of $W_0$
for $t = 1, T - 1$
  $W_t = f(V, t)$
  **Start** $U_t = $ Inverse FFT of $W_t$
  **Finish** $U_{t-1} = $ Inverse FFT of $W_{t-1}$
  Normalize $U_{t-1}$
  Compute Check Sum for $U_{t-1}$
end for
**Finish** $U_{T-1} = $ Inverse FFT of $W_{T-1}$
Normalize $U_{T-1}$
Compute Check Sum for $U_{T-1}$

Figure 4.17: Structure of Agarwal et al.'s overlap algorithm for NAS-FT.

In Figure 4.17, we have informally defined inverse FFT **Start** and **Finish** operations. **Start** performs the in-core 2D FFTs on the first two dimensions of $W_t$, performs the local in-core transpose of $W_t$, and asynchronously invokes the Mover to start the inter-node message passing activity. **Finish** waits on the Mover, and performs the final 1D in-core FFT to complete the inverse 3D FFT.

This algorithm pipelines communication and computations between multiple 3D FFTs. Agarwal et al. also presented an algorithm to pipeline communication and computation within a single 3D FFT. We present performance numbers only from the algorithm presented above, since in a real PDE solver, we expect
the computation to cover many time steps.

Agarwal et al. did not implement this algorithm, since their target hardware, the IBM SP-1, did not support overlap of communication and computation.

We implemented the NAS-FT benchmark in KeLP 2.0. The KeLP code calls the serial Fortran 77 kernels from the NPB2.1 code to perform local FFTs, local matrix transposes, compute local functions, normalization, and local checksum computations.

This Section presents the KeLP performance results for the NAS-FT benchmark. In most cases, we could not compare the KeLP performance to the NPB2.1 code. The NPB2.1 code core dumps under all scenarios on the SparcStations, so we report only KeLP results for the SparcStations. On the AlphaServers, the code crashes in most configurations, including most experiments for the single-tier and single-node results. The problem appears to be related to memory usage for message buffers, but it was beyond the scope of this study to debug the NPB2.1 code. However, the NPB2.1 code runs in the multi-node multi-processor Alpha-farm experiments; we present NPB2.1 numbers for this case.

Figure 4.18 presents performance using only one processor per node on the AlphaServers and SparcStations. On both platforms, we scale the problem size with the number of nodes, with $2^{20}$ unknown variables per node. On eight AlphaServers, the code obtains a speedup of 4.25. On the SparcStations, the performance is dismal, with a speedup of only 1.24 on four processors. The global matrix transpose accounts for up to 70% of the total running time for these experiments, so the speed of the interconnection networks limit application performance.

Figure 4.19 shows the performance on a single AlphaServer node and on a single SparcStation node, using a problem size with $2^{20}$ unknowns. The results show that the code obtains speedups of 1.73, 2.38, and 2.57 on two, three, and four Alpha processors, respectively. Memory contention limits the speedup on the shared-memory node. Some sections of the code, such as the normalization and function computation, place high bandwidth requirements on the shared memory system. Figure 4.19b shows similar results on a single SparcStation node. The
Figure 4.18: Single-tier FT performance on a) the AlphaServer cluster, and b) the SparcStation cluster.

SparcStation obtains speedups of 1.69, 2.13, and 2.18 on two, three, and four processors, respectively.

Figure 4.19: a) FT performance on one AlphaServer. b) FT performance on one SparcStation.

Figure 4.20 shows the performance on multiple nodes, using multiple processors per node. We use the same problem sizes as in the single-tier experiments. For the AlphaServers, the Figure compares the NPB v2.1 code, the basic multi-tier KeLP code, and multi-tier KeLP code that overlaps communication. The
basic multi-tier code parallelizes each local numerical subroutine independently. The overlap multi-tier code does so in the context of Agarwal et al.’s overlapped algorithm. The SparcStation results omit the NPB2.1 results, as mentioned earlier.

Figure 4.20: a) FT performance on the AlphaServer cluster. b) FT performance on the SparcStation cluster.

On the AlphaServers, the results show that the MPI code performs poorly, and cannot utilize more than two SMP nodes. The global matrix transpose limits the performance of the the MPI code. This all-to-all communication pattern forces every MPI process to communicate with every other process. The current MPI implementation does not implement this pattern efficiently.

The Figure shows that although the multi-tier KeLP code performs much better, scaled speedup on eight nodes reaches only 2.59. The overlap algorithm improves performance by up to 13% compared to the basic multi-tier version.

On the SparcStations, performance is disappointing with or without the overlap strategy. The extremely slow global matrix transpose dominates the running time.

Tables 4.10 and 4.11 break down performance of the code by activity. All of the activities listed involve only local computation, save for “Communication”, which accounts for the message-passing to carry out the global matrix transpose. The Tables report the maximum time spent in each code section by any node, so
the sum of the reported times may not add up to the total time. The Tables show that extremely slow inter-node communication limits the performance on both platforms. With these slow interconnection networks, inter-node communication is the dominant cost of the algorithm. The FT benchmark does not perform enough local work to overlap the message-passing costs, so processors must idle waiting for communication to complete.

<table>
<thead>
<tr>
<th>Task</th>
<th>1 node No Overlap</th>
<th>Overlap</th>
<th>2 nodes No Overlap</th>
<th>Overlap</th>
<th>4 nodes No Overlap</th>
<th>Overlap</th>
<th>8 nodes No Overlap</th>
<th>Overlap</th>
</tr>
</thead>
<tbody>
<tr>
<td>Random Numbers</td>
<td>1.39</td>
<td>1.40</td>
<td>1.36</td>
<td>1.37</td>
<td>1.38</td>
<td>1.41</td>
<td>1.47</td>
<td>1.49</td>
</tr>
<tr>
<td>Initialize</td>
<td>0.82</td>
<td>0.79</td>
<td>0.72</td>
<td>0.73</td>
<td>1.51</td>
<td>1.59</td>
<td>0.97</td>
<td>1.53</td>
</tr>
<tr>
<td>Multiply by exponentials</td>
<td>3.16</td>
<td>3.84</td>
<td>3.15</td>
<td>4.33</td>
<td>4.36</td>
<td>4.67</td>
<td>3.19</td>
<td>4.33</td>
</tr>
<tr>
<td>Normalize</td>
<td>1.55</td>
<td>1.6</td>
<td>1.58</td>
<td>1.16</td>
<td>1.24</td>
<td>1.59</td>
<td>1.59</td>
<td>1.33</td>
</tr>
<tr>
<td>Check Sum</td>
<td>0.13</td>
<td>0.13</td>
<td>1.63</td>
<td>0.08</td>
<td>0.11</td>
<td>1.03</td>
<td>1.03</td>
<td>0.17</td>
</tr>
<tr>
<td>Local Transpose</td>
<td>4.08</td>
<td>4.04</td>
<td>2.70</td>
<td>3.39</td>
<td>3.45</td>
<td>3.72</td>
<td>2.72</td>
<td>3.57</td>
</tr>
<tr>
<td>Communication</td>
<td>0.00</td>
<td>1.38</td>
<td>33.09</td>
<td>21.80</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Local FFT</td>
<td>9.16</td>
<td>12.64</td>
<td>9.69</td>
<td>15.33</td>
<td>16.95</td>
<td>21.40</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Total</td>
<td>20.73</td>
<td>26.29</td>
<td>52.80</td>
<td>47.40</td>
<td>63.14</td>
<td>55.25</td>
<td>74.35</td>
<td>63.91</td>
</tr>
</tbody>
</table>

Table 4.10: Execution-time breakdown for 10 timesteps of the basic and overlapped multi-tier KeLP FT codes on the AlphaServer cluster. The Table reports the maximum time, in seconds, spent by any node in each section of the application.

In summary, on both platforms, parallel speedup is disappointing or non-existent due to the extremely high communication costs of the all-to-all matrix
### Table 4.11: Execution-time breakdown for 10 timesteps of the basic and overlapped multi-tier KeLP FT codes on the SparcStation cluster. The Table reports the maximum time, in seconds, spent by any node in each section of the application.

<table>
<thead>
<tr>
<th>Task</th>
<th>1 node No Overlap</th>
<th>1 node Overlap</th>
<th>2 nodes No Overlap</th>
<th>2 nodes Overlap</th>
</tr>
</thead>
<tbody>
<tr>
<td>Random Numbers</td>
<td>1.92</td>
<td>2.07</td>
<td>1.94</td>
<td>1.92</td>
</tr>
<tr>
<td>Initialize</td>
<td>1.45</td>
<td>1.58</td>
<td>1.47</td>
<td>1.50</td>
</tr>
<tr>
<td>Multiply by exponentials</td>
<td>6.62</td>
<td>6.43</td>
<td>3.72</td>
<td>4.39</td>
</tr>
<tr>
<td>Normalize</td>
<td>1.76</td>
<td>1.76</td>
<td>1.76</td>
<td>1.95</td>
</tr>
<tr>
<td>Check Sum</td>
<td>0.12</td>
<td>0.12</td>
<td>0.76</td>
<td>0.18</td>
</tr>
<tr>
<td>Local Transpose</td>
<td>7.13</td>
<td>6.72</td>
<td>6.91</td>
<td>5.54</td>
</tr>
<tr>
<td>Communication</td>
<td>0.00</td>
<td>4.73</td>
<td>92.00</td>
<td>67.12</td>
</tr>
<tr>
<td>Local FFT</td>
<td>14.17</td>
<td>16.14</td>
<td>15.03</td>
<td>20.90</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td><strong>33.35</strong></td>
<td><strong>38.25</strong></td>
<td><strong>121.69</strong></td>
<td><strong>101.64</strong></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Task</th>
<th>4 nodes No Overlap</th>
<th>4 nodes Overlap</th>
</tr>
</thead>
<tbody>
<tr>
<td>Random Numbers</td>
<td>1.92</td>
<td>1.94</td>
</tr>
<tr>
<td>Initialize</td>
<td>1.46</td>
<td>1.54</td>
</tr>
<tr>
<td>Multiply by exponentials</td>
<td>3.74</td>
<td>4.59</td>
</tr>
<tr>
<td>Normalize</td>
<td>1.77</td>
<td>1.94</td>
</tr>
<tr>
<td>Check Sum</td>
<td>43.95</td>
<td>6.18</td>
</tr>
<tr>
<td>Local Transpose</td>
<td>4.39</td>
<td>5.73</td>
</tr>
<tr>
<td>Communication</td>
<td>184.54</td>
<td>165.07</td>
</tr>
<tr>
<td>Local FFT</td>
<td>15.93</td>
<td>24.39</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td><strong>220.23</strong></td>
<td><strong>200.73</strong></td>
</tr>
</tbody>
</table>
transpose. Although multi-tier programming and communication overlap improve performance, the platforms’ interconnection networks cannot support the bandwidth demands of this application.

As a final aside, note that on the AlphaServers, single-tier scaled speedup nearly doubles the best multi-tier scaled speedup. In effect, by utilizing more processors at each node, the multi-tier code exacerbates the architectural imbalance between the interconnection network and the node’s computational power. As processors improve and nodes utilize more processors, we expect this problem to persist unless interconnection networks improve dramatically.

### 4.3.3 SUMMA Matrix Multiplication

All of the previous application examples directly addressed issues in PDE solvers. We now turn our attention to a different problem class: dense linear algebra. Efficient dense linear algorithms operate in block structures in order to make good use of the memory hierarchy [56]. Since KeLP provides facilities to manage distributed block structures, we will evaluate whether KeLP abstractions aid in efficient implementations for blocked dense matrix algorithms.

Perhaps the best-known blocked dense matrix algorithm is dense matrix multiplication. The parallel computing literature presents a number of algorithms for computing the product of two distributed matrices on single-tier multicomputers [50, 81, 39, 68, 96]. We consider the SUMMA (Scalable Universal Matrix Multiply Algorithm), presented by van de Geijin and Watts [135]. SUMMA manipulates large block array sections, and maps naturally into the KeLP and dGrid abstractions. We will consider a basic multi-tier adaptation of SUMMA, and present a new SUMMA variant that explicitly overlaps communication and computation.

First, we review the single-tier SUMMA algorithm. SUMMA considers dense matrix multiplication as a series of outer products. Figure 4.21a shows the usual “ijk” loop ordering. Figure 4.21b reorders the loops, giving a “kij” ordering. Note that with this loop ordering, the inner two loops perform an outer product operation.
for i = 1, N1
  for j = 1, N2
    for k = 1, N3
      C[i,j] = C[i,j] + A[i,k]*B[k,j]
  for i = 1, N1
  for j = 1, N2

Figure 4.21: a) The standard “ijk” matrix-multiply loop. b) With the “kij” ordering, the inner two loops perform an outer product operation.

SUMMA implements matrix multiplication as a series of outer products over distributed matrices. Assume A, B, and C have block data decompositions on a distributed memory parallel computer. We distribute the work based on the decomposition of C; each node will compute the subsection of C that it owns. The SUMMA algorithm proceeds shown in Figure 4.22.

```
for k = 1, N3
  multicast A[;,:k] along rows of virtual processor array
  multicast B[k,:;] along columns of virtual processor array
  local dgemm: C = C + A[;,:k] B[k,:;]
end for
```

Figure 4.22: Basic SUMMA algorithm.

With this algorithm, each node performs a BLAS-1 vector outer product. Since BLAS-1 operations do not run efficiently on cache-based computers, Van de Geijn and Watts present a blocked version of SUMMA. The blocked version groups b outer product calculations into one dense matrix operation, as shown in Figure 4.23.

Figure 4.24 illustrates the data motion performed by one step of the blocked SUMMA algorithm.

Van de Geijn and Watts also suggested a pipelined version of SUMMA, to improve performance. In the pipelined algorithm, each multicast is pipelined using
for $k = 1, N3$ step $b$

multicast $A[:, k : k + b - 1]$ along rows of virtual processor array
multicast $B[k : k + b - 1,:]$ along columns of virtual processor array
local \texttt{dgemm}: $C = C + A[:, k : k + b - 1] B[k : k + b - 1,:]$  

end for

Figure 4.23: Blocked basic SUMMA algorithm

![Multicast A Multicast B Local DGemm](image)

Figure 4.24: Graphical depiction of one stage of the blocked SUMMA algorithm.

a ring topology over each row and column the virtual processor array. That is, during a multicast, each node receives and sends only one message. This protocol overlaps communication and computation between nodes, as some processors will lag behind others waiting for multicast messages.

To adapt the SUMMA algorithm for SMP clusters, we first consider a basic multi-tier implementation. This implementation uses the pipelined blocked SUMMA algorithm between nodes. So, the node-level program uses the single-tier SUMMA algorithm, ignoring the presence of multiple processors per node. At each node, we parallelize the local \texttt{dgemm} kernels using a simple domain decomposition over $C$. Each each processor computes a section of the local $C$ block, parallelizing the computation with simple shared memory domain decomposition.

Note that with the basic multi-tier implementation, processors at each node sit idle during the communication steps. In an attempt to improve per-
formance, we present a restructured version of SUMMA that explicitly overlaps communication and computation at each SMP node. Figure 4.25 shows a version of the SUMMA algorithm that explicitly overlaps communication and computation at each node, by asynchronously starting and stopping each multicast. In this algorithm, we perform the multicasts for iteration $i + 1$ concurrently with the local $\text{dgemm}$ calculation for iteration $i$. In effect, this algorithm sets up a pipeline between iteration of the outer loop.

```
start multicast $A[1:1:b]$ along rows of virtual processor array
start multicast $B[1:b;]$ along columns of virtual processor array

for $k = b + 1, N3$ step $b$
    start multicast $A[;k:k + b - 1]$ along rows of virtual processor array
    start multicast $B[k:k + b - 1;]$ along columns of virtual processor array
    wait for multicasts from previous iteration complete
    local $\text{dgemm}$: $C = C + A[;k - b:k - 1]B[k - b:k - 1;]$

end for

wait for multicasts from last iteration to complete
perform local $\text{dgemm}$: $C = C + A[;N3 - b:N3]B[N3 - b:N3;]$
```

Figure 4.25: Overlapped blocked SUMMA algorithm.

We now examine the performance of the MPI and KeLP SUMMA implementations on single-tier multicomputers, a single SMP, and on multiple SMPs. We present performance results from the Maryland Digital AlphaServer. In all cases, we use local $\text{dgemm}$ kernels from the Digital Extended Math Library (DXML) [86]. On this platform, DXML obtains 160 MFLOPS per processor for matrix sizes that do not fit in the L2 cache. We always select problem sizes that do not fit in the L2 cache. We did not have access to tuned BLAS on the SparcStation cluster, so we do not present SUMMA results for the SparcStations.

The "MPP" results reported in this Section use the C+MPI SUMMA
version made publicly available by van de Geijn and Watts and described in [135]. We modified this code to call the DXML \texttt{dgemm} kernel.

Figure 4.26 shows the performance of SUMMA on the the Digital AlphaServers using only one processor per node. We scale the problem size with the number of nodes, assigning each node roughly one million matrix elements. All experiments use a panel size (blocking factor) of 100. On the AlphaServers, the MPI code outperforms the KeLP code by up to 20%. Unlike the previous codes, this code implements a pipelined message-passing pattern. In order to implement a pipelined pattern in KeLP, we must use a sequence of Movers, one for each pipeline stage. The overhead of creating and invoking, and destroying multiple Movers accounts for the performance differential. This result indicates that for pipelined message-passing algorithms, the KeLP implementation incurs significant overhead. Efficient KeLP support for pipelined communication patterns remains a target for future work.

![SUMMA Single-Tier Performance: AlphaServers](image)

Figure 4.26: Single-tier SUMMA performance on the AlphaServer cluster.

Figure 4.27a shows the performance of matrix multiplication on one SMP node, for matrices of size 1000 x 1000. The MPI results run the MPI SUMMA code, with one MPI process on each physical processor. As discussed previously, the KeLP code uses SUMMA between nodes, and a simple shared-memory parallelization within each node. In this scenario, with only one SMP node, only the shared-memory parallelization is relevant. So, these results compare the use
of the distributed-memory SUMMA algorithm to a much simpler, shared-memory algorithm.

The results show that on one AlphaServer node, the KeLP code outperforms the MPI code by up to 25%. The MPI code suffers from unnecessary data motion and interprocessor synchronization, while the KeLP code takes advantage of the shared-memory subsystem to achieve better performance. As a result, the message-passing algorithm cannot compete with the shared memory algorithm on a single SMP node. Note that even the simple KeLP code achieves only a speedup of 3.29 on 4 processors. Apparently, memory contention plays a role even for matrix multiplication, which makes very good use of the local caches. Also note that the vendor’s local `dgemm` kernels achieve less than 60% of the theoretical peak performance of 275 MFLOPS.

Figure 4.27b shows the performance on the SMP cluster, using multiple processors per node. We compare the MPI version to the basic multi-tier KeLP version, and the explicitly overlapped algorithm presented earlier. These experiments use the same problem sizes as the single-tier experiments.

The results show that the MPI code slightly outperforms the basic multi-tier code on this algorithm. The pipelined nature of the message-passing algorithm
accounts for this result. Consider a simple pipeline, as illustrated in Figure 4.28. The Figure shows the time-line for the first four processors executing several iterations of an algorithm with a pipelined message passing structure. On a single-tier multicomputer, the pipeline pattern overlaps communication and computation between nodes. For example, at the end of the Figure’s time-line, processors 1 and 2 are computing while processors 3 and 4 communicate.

![Figure 4.28: Representation of a pipelined computational pattern.](image)

Now, suppose we map this message-passing code onto an SMP cluster with four processors per node. In the Figure, suppose processors 1-4 all belong to a single node of the cluster. Then, the pipelined message pattern actually overlaps communication and computation within the SMP node, as some processors at the node compute while others communicate. Also note that a basic multi-tier adaptation of the algorithm would overlap communication between nodes, but not within nodes.

The SUMMA algorithm entails a two-dimensional message-passing pipeline, which results in a more complex staggered pattern than just described. However, the basic idea remains the same, and explains how the MPI SUMMA code performs well compared to the basic multi-tier SUMMA implementation. This
phenomenon demonstrates an unintended, but fortunate, consequence of using a pipelined message-passing algorithm on an SMP cluster. In general, this result indicates that pipelined message-passing codes may perform unexpectedly well on SMP clusters. Conversely, basic multi-tier adaptations of pipelined message-passing algorithms may not improve performance.

While the pipelined single-tier algorithm performs unexpectedly well, the results show that careful orchestration of the overlap pattern with the restructured multi-tier algorithm improves performance further still. Figure 4.27b shows that the explicitly restructured multi-tier overlap algorithm outperforms the MPI code by up to 33%.

Table 4.12 shows the breakdown of execution times for the basic multi-tier KeLP code, and the explicitly overlapped multi-tier KeLP code. The column labeled "Comm" gives the time spent by the message co-processor executing the Movers. The column labeled "dgemm" shows the time spent by computing processors executing local computation.

<table>
<thead>
<tr>
<th>Nodes x Procs</th>
<th>Matrix Size</th>
<th>No-Overlap</th>
<th>Overlap</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Comm</td>
<td>dgemm</td>
</tr>
<tr>
<td>1x4</td>
<td>2000</td>
<td>0.49</td>
<td>3.57</td>
</tr>
<tr>
<td>2x4</td>
<td>2828</td>
<td>3.30</td>
<td>5.24</td>
</tr>
<tr>
<td>4x4</td>
<td>4000</td>
<td>8.99</td>
<td>7.46</td>
</tr>
<tr>
<td>8x4</td>
<td>5656</td>
<td>22.05</td>
<td>10.58</td>
</tr>
</tbody>
</table>

Table 4.12: Execution-time breakdown, in seconds, for one iteration of the basic and overlapped multi-tier KeLP SUMMA codes on the AlphaServer cluster. The column labeled Comm gives the time the application code spends waiting for communication to complete.

The table shows that for the problem size considered, on eight nodes, communication and computation take roughly the same time using the overlap algorithm. Thus, we are keeping all processors busy for almost the entire execution. Note that even for dgemm, the message-passing activity slows down local computa-
tion on other processors. For example, on eight nodes, the “no overlap” algorithm devotes all four processors to \texttt{dgemm} and completes the local computation in 10.58 seconds, corresponding to 133.6 MFLOPS per processor. The overlap algorithm devotes three processors per node to \texttt{dgemm} and completes the local computation in 21.27 seconds, corresponding to 88.6 MFLOPS per processor. The effect of dedicating the fourth processor as a message co-processor and restructuring the algorithm slows down local computation by 66%.

In summary, we conclude that while the KeLP implementation adds significant overheads to the SUMMA communication pattern, the KeLP implementation of the explicitly restructured overlap algorithm significantly improves performance compared to the MPI code. Overall, for the problem size considered, the overlapped KeLP code obtains roughly 50% of the peak \texttt{dgemm} speed on eight AlphaServer nodes.

### 4.3.4 LU Decomposition

Finally, we consider LU decomposition with partial pivoting. This application factors a dense matrix $A$ as $A=LU$, where $L$ is lower triagonal and $U$ is upper triagonal.

For distributed memory machines, we consider the blocked right-looking distributed LU factorization algorithm of ScaLAPACK [49]. This algorithm decomposes LU factorization so that local computation invokes BLAS [56]. Most of the computational work falls to BLAS-3 routines, in order to use cache memory efficiently. We present a brief sketch of the algorithm here; see [49] for a full description.

Figure 4.30 shows a high-level description of the algorithm. The algorithm proceeds in a blocked fashion. In each iteration of the outer loop, the algorithm factors the trailing submatrix $A$, indicated by the union of Regions $A_{00}, A_{01}, A_{10}$ and $A_{11}$ in Figure 4.29.

We decompose the algorithm into four high-level steps. First, the algo-
Figure 4.29: In each iteration of the outer loop, the ScaLAPACK LUD algorithm operates on the trailing submatrix $A$, consisting of the union of $A_{00}$, $A_{01}$, $A_{10}$, and $A_{11}$.

The algorithm performs an LU factorization of block column $(A_{00} \ 0)$, using relatively inefficient BLAS 2 operations. Next, the algorithm interchanges rows in the entire matrix according to the pivoting demands. The next step updates the block row $A_{01}$ again with BLAS 2 operations. Finally, the algorithm updates the trailing sub-matrix $A_{11}$ with BLAS 3 matrix multiplication. Figure 4.30 describes the process in slightly more detail.

Counting floating-point operations, most of the work occurs during the BLAS 3 matrix multiplication. For extremely large problem sizes, this work will dominate the running time, so the algorithm should scale. In practice, on attainable problem sizes, the BLAS 2 work and inter-node communication severely impact performance.

When using distributed arrays on distributed-memory computers, we must choose a data layout for the array. For the ScaLAPACK LU algorithm, it has been observed that the 2D block-cyclic array layout is usually a good choice [48]. This layout has several good properties. First of all, the block-cyclic layout ensures good load balance during the BLAS 3 operations. Secondly, the 2D distribution ensures that no single stage, such as finding a pivot row, becomes a serial bottleneck.
The nested loop of the algorithm performs a level-2 BLAS LU factorization of a block column of A. ScaLAPACK encompasses these steps in the `pdgetf2` routine. `pdgetf2` repeatedly finds the pivot row, swaps pivot rows in a block column, and updates the trailing block column sub-matrix. Using a 2D block-cyclic layout with a \( p_r \times p_c \), \( p_r \) nodes participate in these activities. However, the \( p_r \) nodes must communicate to perform all `pdgetf2` activities.

In contrast, using a 1D block-cyclic layout (setting \( p_r = 1 \)), all `pdgetf2` activities involve only one node. Thus, the 1D block-cyclic layout will not theoretically scale to thousands of nodes, since each `pdgetf2` creates a one-node bottleneck. However, the 1D block-cyclic layout eliminates all inter-node communication during `pdgetf2`. Thus, the 1D block-cyclic layout may outperform the 2D block-cyclic layout on machines with few nodes and high communication costs [48]. Many SMP clusters fall into this category, and on these machines, we should consider the 1D block-cyclic layout.

For either 2D or 1D block-cyclic layouts, we must choose a reasonable block size \( b \). Note that the block size \( b \) used in the LU algorithm does not necessarily correspond to the block size used in the serial BLAS kernels employed at each node.

We now use the ScaLAPACK LU factorization routine to consider trade-offs of layouts and block sizes of the target hardware. We ran the ScaLAPACK code on the AlphaServers, using one MPI process per physical processor, considering both the 1D and 2D block-cyclic layouts for various blocking factors. All LU experiments reported in this Section scale the problem size with the number of nodes, with roughly 4 million matrix elements per node. Table 4.13 shows the results. In the Table, the 2D block-cyclic virtual processor arrays have dimensions \( 2 \times 2 \), \( 2 \times 4 \), \( 4 \times 4 \), and \( 4 \times 8 \) on 1, 2, 4, and 8 nodes, respectively. The 1D block-cyclic virtual processor arrays have dimensions \( 1 \times 4 \), \( 1 \times 8 \), \( 1 \times 16 \), and \( 1 \times 32 \) on 1, 2, 4, and 8 nodes respectively.

In all cases considered, the best 1D block-cyclic decomposition outperforms the best 2D decomposition. Furthermore, in all cases, the blocking factor of 50 outperforms the other choices. As we increase the number of processors,
Table 4.13: Performance, in MFLOPS, of the ScaLAPACK LU code on the AlphaServers, using one MPI process per physical processor. $N$ is the number of nodes used, and $b$ is the block size used in the block-cyclic matrix decompositions.

<table>
<thead>
<tr>
<th>N</th>
<th>$b = 25$</th>
<th>$b = 50$</th>
<th>$b = 75$</th>
<th>$b = 100$</th>
<th>$b = 25$</th>
<th>$b = 50$</th>
<th>$b = 75$</th>
<th>$b = 100$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>157</td>
<td>170</td>
<td>165</td>
<td>170</td>
<td>217</td>
<td>237</td>
<td>222</td>
<td>225</td>
</tr>
<tr>
<td>2</td>
<td>256</td>
<td>278</td>
<td>271</td>
<td>276</td>
<td>361</td>
<td>396</td>
<td>359</td>
<td>349</td>
</tr>
<tr>
<td>4</td>
<td>388</td>
<td>411</td>
<td>412</td>
<td>412</td>
<td>555</td>
<td>607</td>
<td>563</td>
<td>501</td>
</tr>
<tr>
<td>8</td>
<td>784</td>
<td>830</td>
<td>817</td>
<td>810</td>
<td>882</td>
<td>921</td>
<td>842</td>
<td>798</td>
</tr>
</tbody>
</table>

the performance differential between the two strategies increases. Analysis of the algorithm predicts that for a large enough cluster, the 2D layout will eventually overtake the 1D layout. However, for the problem sizes presented, the Maryland Digital AlphaServer is not large enough to reach the crossover point.

Based on these results, for the remainder of this Section, we restrict our attention to 1D block-cyclic layouts using a blocking factor of 50. We present results from the Maryland Digital AlphaServer. All local BLAS routines call the Digital Extended Math Library [86]. We did not have access to tuned BLAS for the SparcStations, so we do not report LU results on the SparcStations.

We implemented the ScaLAPACK algorithm in KeLP. First, we consider performance on single-tier multicomputers, using only one processor per AlphaServer. Figure 4.31 shows the performance on the AlphaServers, using only one processor per node. The Figure compares the KeLP code performance to the ScaLAPACK code. Both codes use the same 1D block-cyclic layout and blocking factor of 50. The Figure shows that in all cases, the KeLP code performs within 15% of the ScaLAPACK code. As we increase the number of nodes, the ScaLAPACK code begins to outperform the KeLP code. The LU algorithm includes a number of pipelined message passing-patterns. As we discussed earlier for SUMMA, the KeLP 2.0 implementation does not handle pipelined patterns efficiently, accounting for the performance differential.
Next, we consider performance of on a single SMP. We compare the ScaLAPACK code to a multi-tier version of the KeLP code. The multi-tier KeLP code parallelizes local kernels with the procIterator loop. However, we still run the blocked algorithm for efficient cache utilization.

Figure 4.31b shows the performance on a single AlphaServer SMP. The Figure shows on up to three processors, the performance of the ScaLAPACK code and KeLP code are comparable. On four processors, the ScaLAPACK performance degrades. The ScaLAPACK code uses MPI to transfer data between processors, while the KeLP code exploits shared memory in the node for more efficient data motion. Using all four processors, the KeLP code outperforms the MPI code by 54%. The KeLP code achieves a speedup of 2.37 on four processors. Memory contention, data motion, and synchronization costs limit the speedup.

Finally, we consider performance on the full SMP cluster, using multiple processors per node. We will compare three versions of the code: the ScaLAPACK code, a basic multi-tier KeLP version, and a restructured multi-tier version that explicitly overlaps communication and computation.

In previous work, Desprez et al. [55] present a restructured version of the ScaLAPACK LU algorithm to overlap communication and computation on single-tier multicomputers. They propose two optimizations. First, their algorithm overlaps communication and computation between nodes by starting the broadcast of a column panel before solving on the block row. This optimization exploits the task-parallel flavor of the algorithm, dividing the processor set into two groups, each accomplishing different tasks. Secondly, the algorithm uses asynchronous message passing to swap rows in the block column while scaling the block column. Reported experimental results on an Intel Paragon show that these optimizations improve performance by up to 15%, but that the improvement decreases as the matrix size grows.

Desprez et al. only overlap parallel activities within one iteration of the outer loop. In contrast, we propose a new restructured LUD algorithm tailored for multi-tier architectures that pipelines activities between multiple iterations.
of the inner loop. Our algorithm overlaps BLAS 3 computation with BLAS 2 computation and communication at each SMP node, conserving the scarce resource of shared memory bandwidth.

Our algorithm delays the level-3 BLAS trailing sub-matrix update in each iteration, and overlap this activity with the `pdgetf2` and communication activities in the next iteration. Number the processors at each SMP node from 0 to \( p \). Processor 0 at each SMP node performs the block column tasks, including the pivot row calculation, swapping the rows, scaling the block column, and broadcasting the block column data. Concurrently, the other processors at each SMP node perform the `dgemm` calculation from the previous iteration. Figure 4.32 gives the high-level form of the restructured algorithm.

Figure 4.33 shows the performance on the AlphaServer cluster using multiple processors per node. The Figure compares the ScaLAPACK MPI code to the basic multi-tier KeLP code and to the restructured overlap KeLP code.

The results show that the ScaLAPACK MPI code outperforms the basic multi-tier code. On up to four nodes, the performance of the basic multi-tier code and ScaLAPACK differ by at most 11%. On eight nodes, the ScaLAPACK code outperforms the basic multi-tier KeLP code by 65%. Several factors account for the differential. First, we have noted that the KeLP 2.0 implementation does not handle the pipelined message-passing efficiently. In fact, the KeLP 2.0 overhead grows with the number of nodes. Secondly, as in SUMMA, the ScaLAPACK code benefits communication overlap from pipelining between processors. The ScaLAPACK implementation pipelines some parallel BLAS routines, which results in good performance on the SMP hardware. The basic multi-tier KeLP code breaks this pipeline, reducing performance.

The explicitly overlapped multi-tier version improves performance over the basic multi-tier version by up to 15%. The explicitly overlapped KeLP code performs as well or better than the ScaLAPACK code on fewer than four nodes. On eight nodes, other KeLP overheads dominate the performance, and the KeLP code cannot compete with ScaLAPACK.
Table 4.14 shows the execution time breakdown for the basic and overlapped multi-tier KeLP codes. The Table reports the time spent by \textit{processor 0} on each SMP node, in each stage of the algorithm. Note that processor 0 participates in all code sections, while the other processors do not. If the overlap algorithm is effective, then processors 1-3 perform \texttt{dgemm} while processor 0 performs the rest of the algorithm. So, with the overlap algorithm, the time spent by processor 0 in \texttt{dgemm} should decrease.

The Table shows that on eight nodes, with no communication overlap, \texttt{dgemm} accounts for 16\% of the total running time. The overlap algorithm completely hides this cost, and reduces the total running time by 16\%. Thus, the overlap algorithm effectively hides the entire cost of the \texttt{dgemm}.

Theoretically, as the problem size scales, \texttt{dgemm} costs should dominate the running time. Unfortunately, the experimental results reflect otherwise, since as we increase the number of nodes, the time spent communicating grows faster then \texttt{dgemm} costs. If KeLP could manage the pipelined communication more efficiently, the overlap optimization should result in larger performance gains.

In summary, we conclude that the KeLP 2.0 overheads for pipelined communication patterns limit performance for the ScaLAPACK LUD algorithm. The LU algorithm strays too far from KeLP’s original intended problem domain, and KeLP system overheads for this algorithmic pattern limit performance. We note that on the experimental platform, the 1D block-cyclic decomposition performs best for all problem sizes considered. We presented a new algorithm for overlapping communication and computation in LUD, which hides the cost of the \texttt{dgemm} in our experiments, improving KeLP performance by up 15\%.

We also note that the best results from ScaLAPACK achieve 10.5\% of the theoretical peak performance on eight nodes.
<table>
<thead>
<tr>
<th>Section</th>
<th>1 node matrix size 2000</th>
<th>2 nodes matrix size 2800</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>No Olap</td>
<td>Olap</td>
</tr>
<tr>
<td>Find Pivot</td>
<td>0.80</td>
<td>0.91</td>
</tr>
<tr>
<td>Swap Rows in</td>
<td>0.17</td>
<td>0.27</td>
</tr>
<tr>
<td>in Block Column</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Scale Column</td>
<td>0.41</td>
<td>0.38</td>
</tr>
<tr>
<td>Update Trailing Sub-matrix</td>
<td>2.82</td>
<td>3.43</td>
</tr>
<tr>
<td>in Block Column</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Broadcast Swap information</td>
<td>0.11</td>
<td>0.01</td>
</tr>
<tr>
<td>Delayed Row Swap</td>
<td>3.05</td>
<td>3.11</td>
</tr>
<tr>
<td>Multicast Lower Triangle</td>
<td>0.24</td>
<td>0.23</td>
</tr>
<tr>
<td>Backsolve in Block Row</td>
<td>0.5</td>
<td>0.5</td>
</tr>
<tr>
<td>Multicasts</td>
<td>0.46</td>
<td>0.32</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td>19.81</td>
<td>18.77</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Section</th>
<th>4 nodes matrix size 4000</th>
<th>8 nodes matrix size 5600</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>No Olap</td>
<td>Olap</td>
</tr>
<tr>
<td>Find Pivot</td>
<td>0.79</td>
<td>0.88</td>
</tr>
<tr>
<td>Swap Rows in</td>
<td>0.13</td>
<td>0.13</td>
</tr>
<tr>
<td>in Block Column</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Scale Column</td>
<td>0.50</td>
<td>0.54</td>
</tr>
<tr>
<td>Update Trailing Sub-matrix</td>
<td>2.89</td>
<td>3.51</td>
</tr>
<tr>
<td>in Block Column</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Broadcast Swap information</td>
<td>16.43</td>
<td>18.66</td>
</tr>
<tr>
<td>Delayed Row Swap</td>
<td>3.37</td>
<td>3.46</td>
</tr>
<tr>
<td>Multicast Lower Triangle</td>
<td>4.56</td>
<td>5.66</td>
</tr>
<tr>
<td>Backsolve in Block Row</td>
<td>0.67</td>
<td>0.67</td>
</tr>
<tr>
<td>Multicasts</td>
<td>25.21</td>
<td>30.09</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td>79.02</td>
<td>61.65</td>
</tr>
</tbody>
</table>

Table 4.14: Execution-time breakdown, in seconds, reported by processor 0 on each node for one iteration of the KeLP LU code on the AlphaServer cluster.
for \(ib = 1, n - 1\) step \(b\)

\[
end = \min(ib + b - 1, n)
\]

// LU factorize block column \((A_{00})_A\)

for \(i = ib, end\)

- Find pivot row \(k\) in \((A_{00})_{A_{10}}\)
- Swap rows \(i\) and \(k\) in \((A_{00})_{A_{10}}\)
- Scale column \(i\) in \((A_{00})_{A_{10}}\)
- Update trailing sub-matrix of \((A_{00})_{A_{10}}\)

end for

// Perform delayed row swaps
Broadcast swap information to all nodes
Perform delayed row swaps on all other columns

// Update block row \(A_{01}\)
Multicast lower triangular piece of \(A_{00}\) right
Backsolve on block row \(A_{01}\)

// Update trailing submatrix \(A_{11}\)
Multicast \(A_{10}\) and \(A_{01}\)
Update \(A_{11}\) with \texttt{dgemm}

end for

Figure 4.30: High-level description of the ScaLAPACK LU algorithm.
Figure 4.31: a) Single-tier LU Performance on the AlphaServers. b) Single-node LU performance on one AlphaServer.
\begin{algorithm}
\begin{algorithmic}
  \For{$i b = 1 , n - 1$ \textbf{step} $b$} \\
  $\text{end} = \min(i b + b - 1, n)$ \\
  \textbf{On one Processor per SMP Node do:}
  \For{$i = i b , \text{end}$} \\
  Find pivot row $k$ in $\begin{pmatrix} A_{00} \\ A_{10} \end{pmatrix}$
  Swap rows $i$ and $k$ in $\begin{pmatrix} A_{00} \\ A_{10} \end{pmatrix}$
  Scale column $i$ in $\begin{pmatrix} A_{00} \\ A_{10} \end{pmatrix}$
  Update trailing sub-matrix of $\begin{pmatrix} A_{00} \\ A_{10} \end{pmatrix}$
  \EndFor
  Broadcast swap information left and right
  Perform delayed row swaps on all other columns
  Multicast $A_{10}$
  Help update $A_{11}$ with \texttt{dgemm}

  \textbf{Synchronize all processor at each SMP node}

  \textbf{All processors do:}
  Multicast lower triangular piece of $A_{00}$ right
  Backsolve on block row $A_{01}$
  Multicast $A_{01}$
  Update first block column of $A_{11}$ with \texttt{dgemm}

  \textbf{All processors but one on each SMP node:}
  \textbf{Begin to update $A_{11}$ with \texttt{dgemm}}
\EndFor
\end{algorithmic}
\end{algorithm}

Figure 4.32: High level description of the multi-tier LU algorithm that explicitly overlaps several activities.
Figure 4.33: Performance of the multi-tier LU codes on the AlphaServer cluster.
4.4 Single-Tier Performance Evaluation

The previous Sections have described KeLP performance on two SMP clusters. Single-tier parallel computers have only one processor per node. A previous version of KeLP (KeLP 1.0) supported only single-tier parallel computers. Previous results have shown that on a single-tier multicomputer, KeLP1.0 performed comparably with hand-coded MPI codes [63].

We previously stated as a design goal that KeLP support for multi-tier architectures should not significantly impact performance on single-tier multicomputers. We now test this assumption by comparing hand-coded MPI implementations to KeLP 2.0 implementations on the T3E. We present results from the applications described in the previous Sections. We use the basic multi-tier codes described previously, treating the T3E as an $n \times p$ SMP cluster with $p = 1$. Section 1.2 describes the T3E hardware and software environment for these experiments.

Figure 4.34 compares performance of the redblack3D MPI and KeLP codes on the T3E. These runs scale the problem size with the number of nodes, keeping the number of grid points constant at roughly 2 million grid points per node. Figure shows that in all cases computational rate of the KeLP code is within 1.5% of the MPI code.

![Figure 4.34: Performance of single-tier redblack3D implementations on the T3E.](image)

Figure 4.34 compares performance of the NAS-MG MPI and KeLP codes
on the T3E. As before, the MPI code is from the NAS Parallel Benchmarks v2.1. We use problem sizes of $128^3$, $256^3$, and $51263$ on 1-4, 8-32, and 64 processors, respectively. The Figure shows that the KeLP code performs within 3% of the MPI code in all cases.

Figure 4.35: Single-tier performance for the NAS-MG benchmark on the T3E.

Based on the previous two results, we conclude that KeLP manages regular finite difference codes about as efficiently as hand-coded MPI. For comparison, we now examine performance on the multi-block multigrid code. Figure 4.34 compares performance on the multi-block code on the square and Lake Superior domains, using roughly 1 million grid points per node. As before, grid generation and partitioning were done off-line. We restrict our attentions to the grid geometries considered in previous Sections; thus, we present results on up to eight nodes of the T3E.

Figure 4.36 shows the results. On eight nodes, the irregular geometry incurs a performance degradation of 27% compared to the square geometry. This matches our expectation, based on the load imbalance presented in Table 4.6.

Figure 4.37 compares results of the NPB2.1 FT code and the KeLP NAS-FT code. The problem size scales with the number of nodes, using $2^{18}$ unknown variables per node. The results shows that the KeLP code always performs within 12% of the NPB 2.1 code. In most cases the performance difference is 3% or less.
Figure 4.36: Single-tier performance for the multi-block multigrid benchmark on the T3E.

The main communication activity in this benchmark is the global matrix transpose. The MPI code performs the inter-node communication using the MPI `all_to_all` call. The KeLP code encodes the matrix transpose using the MotionPlan. The KeLP 2.0 Mover interprets the MotionPlan using point-to-point MPI messages. So, the difference in running time between the MPI and KeLP codes on single-tier multicomputers reflects the efficiency of MPI `all_to_all` compared to point-to-point messages. These results indicate, somewhat surprisingly, that KeLP implements the global matrix transpose about as efficiently as the MPI collective call.

Figure 4.37: Single-tier FT performance the T3E.
Figure 4.38 compares the hand-coded SUMMA and KeLP SUMMA implementations. These experiments scale the matrix size with the number of nodes so that each node owns roughly 250,000 matrix elements. These experiments call the T3E’s vendor-provided dgemm kernel for local computation. The Figure shows that on the T3E, the performance of the single-tier KeLP code and MPI code lie within 18%. The scaled speedups are high, with the KeLP code achieving a scaled speedup of 56 on 64 processors. As on the SMP clusters, KeLP’s inefficient handling of the pipelined message-passing pattern accounts for the performance differential.

![SUMMA performance](image)

Figure 4.38: Single-tier SUMMA performance on the T3E.

The T3E Fortran compiler would not accept the ScaLAPACK LUD implementation, so we do not present LUD results on the T3E.

### 4.5 Discussion

Experience with these sample applications has shed light on principles and practice for programming block-structured scientific calculations on dedicated SMP clusters. We now draw conclusions from our experience. First, we evaluate the software infrastructure with respect to programming and engineering issues. Secondly, we discuss general performance consequences and issues illuminated in this Chapter.
4.5.1 Software Engineering Issues

We believe that the application studies demonstrate that the KeLP programming model provides a good level of abstraction for block-structured scientific calculations. We considered three crucial application classes: finite difference codes, FFTs, and blocked dense linear algebra algorithms. Each of these application classes demanded quite different data layout strategies, and communication patterns. Nevertheless, we found that the KeLP abstractions allowed concise expression of all applications. The Region calculus was particularly useful, allowing the programmer to concisely describe block application structures ranging from irregular data dependencies in applications as diverse as multi-block multigrid and LUD.

The multi-tier KeLP programming model provides structured parallel loops through the two iterators. These structured loops restrict the programming model, disallowing the more general unstructured control flow when can be expressed through fork-join structures. However, by disallowing unstructured parallel control flow, the structured loops implicitly handle inter-processor synchronization, simplifying the programming model. The application studies demonstrate that with asynchronous data motion, the structured parallel control model allows sufficient generality to express the overlap algorithms for all six applications.

We have demonstrated that explicit overlap of communication and computation improves performance on the FFT and dense linear algebra codes. Typically, the programmer will rely on libraries such as ScaLAPACK [48] or FFT-PacK [131] to implement these operations. In order to explicitly overlap communication and computation, we propose that standard libraries provide asynchronous entry points for numerical routines. For example, an FFT library should provide startFFT() and finishFFT() calls, which the programmer can use to structure the calling application as needed. While ScaLAPACK does not provide asynchronous operations, the NetSolve interface provides asynchronous access to numerical library routines for distributed systems [41]. The results from this dissertation further reinforce the benefits of asynchronous entry points to numerical libraries,
even in more tightly coupled cluster environments.

We note that the current KeLP model does not explicitly support block-cyclic data layouts. The LU application simulated a block-cyclic layout as a multi-block layout, assigning many blocks per SMP node. This approach has several disadvantages. First, libraries like ScaLAPACK that understand block-cyclic can support much finer block sizes, without sacrificing granularity or overhead invoking serial numeric kernels. Secondly, the KeLP approach will not scale. Based on these limitations, we conclude that KeLP would much better support dense linear algebra with built-in efficient support for block-cyclic layouts. Consequences of this extension to the overall KeLP model remains a subject for future research.

All the sample applications provided benefited from Domain-Specific Library support. The DOCK library, which provides a run-time HPF-like data decomposition model, contributed to every application. Domain-specific application libraries provide an ever higher-level interface for the scientific programmer, and show great promise for making parallel computing more accessible to the community at large [71].

This Chapter presented results from small, manageable application kernels. The next Chapter surveys some of the real-world applications coded with KeLP. Every KeLP application we know of builds on one or more of the sample applications or Domain-Specific libraries presented in this Chapter. This experience validates that the dissertation’s choice of sample applications has relevance in other scenarios.

### 4.5.2 Performance

We have examined KeLP performance on six block-structured applications.

For all finite difference codes, the single-tier KeLP code performed comparably to hand-coded MPI. These results show that the implementation of the KeLP Mover handles message-passing for these applications about as efficiently as a hand-coded MPI application. Note, however, that the KeLP communication
primitives provide a higher level of abstraction than MPI message-passing code, resulting in shorter, cleaner, easier-to-read code. We conclude that the KeLP primitives for expressing data motion in finite difference codes incur no practical performance penalty compared to more tedious, lower-level MPI code. The NAS-FT code showed similar results.

The SUMMA and LU single-tier results expose a limitation in the KeLP 2.0 implementation for pipelined message-passing patterns. KeLP must use a separate Mover for each stage in a pipeline, resulting in significant overheads compared to hand-coded MPI. Perhaps KeLP would better suit these applications with a Mover abstractions that represents pipelined communication patterns and handles them efficiently. Results also show that pipelined message-passing algorithms may perform quite well on SMP clusters. In effect, the pipelined algorithmic structure mediates access to a SMP node’s network interface, effectively overlapping communication and computation as a side effect.

The results from single-node SMP experiments illustrate bottlenecks when using a bulk-synchronous programming model on a shared memory systems. Both the AlphaServers and SparcStations exhibit drastic performance losses due to contention for the shared memory system. As expected, the slowdown varies directly with the frequency of memory access of the numeric kernel. Kernels with poor locality, such as multigrid sweeps, saturate the memory systems of the SMPs and do not scale well. Kernels with good locality, such as matrix multiply, scale better. Additionally, the results show that the high cost of intra-node synchronization can have a noticeable impact on performance.

The multi-tier results show that an implementation strategy that exploits both shared memory and message-passing mechanisms usually outperforms a single-tier message-passing code. The MPI results reflect the performance achieved by naively porting a single-tier code to the multi-tier hardware. This does not represent the maximum possible MPI performance. Of course, the programmer can optimize an MPI code to exploit shared memory facilities and implement custom data decompositions, as done in underlying KeLP implementation. The results from
Chapter 3 show that such programming may be difficult. The KeLP programmer does not need to worry about many such implementation details.

The multi-tier KeLP results stress the need to focus on locality-enhancing optimizations for the single-processor memory hierarchy. Reducing memory traffic is critical both for single-processor performance, and for performance involving multiple processors. In all applications, we have seen that memory bandwidth limits the efficiency of both local computation and message-passing with communication overlap. The KeLP abstractions do not directly address optimizations such as tiling for the single-processor memory hierarchy [104]. The application studies indicate that this area deserves more attention in future research.
Chapter 5

Conclusion

5.1 Research Summary

This dissertation has presented a programming model to facilitate high-performance implementations of block-structured scientific calculations on SMP clusters. The programming model provides a small set of geometric abstractions to manage data layout, data motion, and parallel control flow on multi-tier architectures.

The KeLP programming model introduces new mechanisms to manage two levels of structured parallel control flow for multi-tier architectures. The model cleanly separates meta-data descriptions of application structure from objects that implement the structural decisions. In effect, this philosophy separates correctness and performance issues in a parallel code. KeLP provides a new collective communication structure that combines ideas from structural abstraction and inspector/executor paradigm. Additionally, the data motion model handles overlap of communication and computation at a high level, representing potentially complex collective data motion patterns as atomic collective operations.

Most importantly, the programming abstractions contribute a novel division between mechanism and policy for parallel applications. The system provides high-level intuitive geometric mechanisms that expose two levels of parallelism and
locality for multi-tier architectures. With these mechanisms, the programmer can implement a variety of algorithmic policies, without drowning in low-level implementation details.

We have demonstrated that the programming abstractions provide enough flexibility to express a variety of block-structured algorithms, including regular and irregular structured grid methods, the FFT, dense matrix multiplication, and dense Gaussian elimination with partial pivoting. For each application, we exposed a two-level parallel structure appropriate for multi-tier architectures, and presented a restructured algorithm to tolerate slow inter-node communication costs. Application studies show that the KeLP abstractions help express all algorithmic versions considered.

Performance results in a variety of contexts show that for coarse-grain applications, KeLP handles message-passing activity about as efficiently as hand-coded MPI. Results also show that the efficacy of overlapping message-passing and local computation depends strongly on the locality characteristics of the local kernels. For some applications, the results show that overlap of communication can effectively improve performance, which suggests that libraries for SMP clusters might provide asynchronous entry points for numerical routines. For overlap to be effective, the implementation must carefully manage SMP node processor and memory resources, and manage scheduling and synchronization issues intelligently.

We have demonstrated KeLP’s utility for several representative applications. Section 5.3 discuss other KeLP projects, which provide evidence that the KeLP abstractions prove useful for a variety of users and application domains.

5.2 Outstanding Research Issues

The problem of implementation scientific codes on SMP clusters raises many unresolved issues.

Predicting performance on SMP clusters remains a difficult task. Traditional parallel performance models [53, 134] focus on message-passing costs.
Based on these models, researchers have developed analytic models to predict performance for many applications, including those considered in this dissertation [135, 65, 48, 55]. However, these analytic techniques do not suffice to accurately predict performance on SMP clusters.

In particular, message-passing parallel performance models neglect the effects of shared memory contention and synchronization on application performance. For several decades, many papers have presented analytic or queuing models for single-tier shared memory system performance under a variety of assumptions (eg. [105, 17, 79, 122, 139, 46, 84, 31, 99, 24, 114, 26]). Most of this work relies on assumptions regarding typical application behavior, and it is not clear how to apply these models to obtain performance predictions for a given application. Additionally, this dissertation has demonstrated that interactions between shared memory activity and message-passing activity significantly affect performance. We are not aware of any performance models that address this interaction.

The KeLP programming model presented here manages locality and parallelism for a specific hardware model. However, the KeLP model offers promise for adaptation to more general architectural structures, including out-of-core applications and clusters of clusters of machines. Extending KeLP to these scenarios appears to be a promising direction, raising many open issues regarding the evolution of the programming model, implementation techniques, and algorithms.

This dissertation has proposed a programming methodology for only block-structured applications. The KeLP model does not admit many classes of codes, including sparse matrix methods, tree-based data structures, or unstructured meshes. These application classes will demand new implementation techniques and programming abstractions.

Since each application class likely demands different software support, we believe it is unlikely that one software tool will suit all applications. Additionally, we note that many applications combine multiple numerical methods and data structures. For these applications, various types of software infrastructure must interoperate. We have not yet demonstrated interoperability of KeLP with other
systems, and a framework for interoperability between different classes scientific codes remains a topic for future investigation.

5.3 KeLPification Survey

The dissertation has presented a programming model and software implementation alleviate the difficulties of parallel programming for block-structured scientific calculations. In addition to the studies considered in this dissertation, several external research projects have employed the software infrastructure in the course of computational science and computer science research. These studies support our claim that the KeLP abstractions are useful for a variety of application domains. Some external efforts include:

- Scott Kohn, Elizabeth Ong (Center for Applied Scientific Computing, Lawrence Livermore National Laboratory) John Weare (Department of Chemistry, UCSD), and Scott Baden (Department of Computer Science and Engineering, UCSD) implemented a structured adaptive mesh refinement algorithm to first principles simulations of electronic structures using KeLP [91]. This application uses adaptive mesh refinement to solve the Local Density Approximation to the Schrödinger equation for materials.

- In collaboration with Keiko Nomura and Tamara Grimmett (Department of Applied and Mechanical Engineering, UCSD), Scott Baden and Jeffrey Howe (Department of Computer Science and Engineering, UCSD) used KeLP to produce a parallel implementation of a legacy Navier-Stokes solver for computational fluid dynamics [80]. This KDISTUF code performs direct numerical simulations of incompressible homogeneous sheared and unsheared turbulent flows.

- Mary Wheeler (University of Texas at Austin) and collaborators are using KeLP for petroleum reservoir simulations. This project investigates multi-
phase flow models with general boundary conditions, multiblock geometry, and adaptive refinements.

- The National Partnership for Advanced Computational Infrastructure (NPACI) supports ongoing KeLP infrastructure development as part of the Common Adaptive Runtime Environment research project. This project will incorporate KeLP along with other NPACI software tools to realize a comprehensive software infrastructure for parallel adaptive methods.

- John Weare (Department of Chemistry, UCSD), collaborating with Scott Baden, is using KeLP to parallelize a Fourier-based planewave basis code for materials science applications.

- John Merlin (Vienna Center for Parallel Computation) has implemented an interface between KeLP and the SHPF data-parallel language [101]. This project combines the independent strengths of KeLP and HPF to provide a powerful programming model for multi-block applications.

- Michelle Ketcham (Department of Applied and Mechanical Engineering, UCSD) has developed a parallel version of the MIXLAYER code using KeLP [89]. This program performs direct numerical simulation of a 2D mixing layer using the Euler equations.

- Silvia Figueira (Department of CSE, UCSD) used KeLP applications to investigate performance in multi-user parallel environments [62].

- Fran Berman, Rich Wolski, and collaborators (Department of CSE, UCSD) have used KeLP to investigate dynamic load balancing policies on heterogeneous workstation clusters [22, 23, 137]. KeLP’s support for irregular block data decompositions simplifies implementation of load balancing algorithms for this environment.

- KeLP has been used for instruction in graduate and undergraduate parallel computing classes at UCSD.
• Zia Ansari used KeLP to implement PILOT, a visualization system for irregular block-structured data sets [9].

• Scott Baden, Larry Carter, and Jeanne Ferrante (Department of CSE, UCSD) are defining KeLP$^N$, an extension to the KeLP programming model based on the Parallel Memory Hierarchy model [6]. The KeLP$^N$ model will support programs that run on more general architectural structures, including out-of-core programs and clusters of clusters of machines.
References


Proceedings of the First International Workshop on High-Level Programming Models and Supportive Environments, Honolulu, HI, April 1996.


