To appear, IEEE Trans. on Software Engineering.

A Programming Methodology for Dual-tier Multicomputers

Scott B. Baden
University of California, San Diego,
Department of Computer Science and Engineering
9500 Gilman Drive, La Jolla, CA 92093-0114 USA
Tel. +1 (858) 534-8861
Fax. +1 (858) 534-7029
email: baden@cs.ucsd.edu

Stephen J. Fink
IBM Thomas J. Watson Research Center
P.O. Box 704
Yorktown Heights, NY 10598 USA
Tel. +1 (914)784-7776
Fax. +1 (914)784-6201
email:sjfink@us.ibm.com

A Programming Methodology for Dual-tier Multicomputers

Abstract

Hierarchically-organized ensembles of shared memory multiprocessors possess a richer and more complex model of locality than previous generation multicomputers with single processor nodes. These *dual-tier computers* introduce many new factors into the programmer's performance model. We present a methodology for implementing block-structured numerical applications on dual-tier computers, and a run-time infrastructure, called KeLP2, that implements the methodology. KeLP2 supports two levels of locality and parallelism via hierarchical SPMD control flow, run-time geometric meta-data, and asynchronous collective communication. KeLP applications can effectively overlap communication with computation under conditions where non-blocking point-to-point message passing fails to do so. KeLP's abstractions hide considerable detail without sacrificing performance, and dual-tier applications written in KeLP consistently outperform equivalent single-tier implementations written in MPI. We describe the KeLP2 model and show how it facilitates the implementation of five block-structured applications specially formulated to hide communication latency on dual-tiered architectures. We support our arguments with empirical data from running the applications on various single- and dual-tier multicomputers. KeLP2 supports a migration path from single-tier to dual-tier platforms, and we illustrate this capability with a detailed programming example.

Index words: dual-tier parallel computers, hierarchical parallelism, KeLP, block-structured scientific applications, scientific application requirements, C++ framework, SMP clusters.

1 Introduction

Memory locality models on parallel computers are becoming increasingly complex, in order to bridge the widening gap in processor and memory speeds. Another contributing factor has been the emergence of the *dual-tier multicomputer:* a hierarchically-organized parallel computer with two levels of locality and parallelism. Dual-tier clusters of SMP workstations have taken a role for solving diverse computationally-intensive problems [1], and tighter coupled dual-tier parallel computers with faster interconnect have appeared as well [2].

Compared with *single-tier* multicomputers–which have a single compute processor at each node–dual-tier computers have a multi-processor at each node, which is typically a symmetric multiprocessor (SMP). This is shown in Fig. 1. Communication in dual-tier multicomputers exhibits a two-level cost function. Processors within a single node may communicate relatively quickly through shared memory, whereas processors on different nodes communicate relatively slowly via the inter-node interconnect.

Although dual-tier architectures can potentially deliver unprecedented performance for computationally intensive scientific calculations, realizing the hardware's potential remains a formidable task. The principal difficulty is that increased node performance due to multiprocessing amplifies the cost of inter-node communication. Relative to the computational rate, the available inter-node bandwidth on dual-tier systems tends to be lower than for single-tier systems with the same number of processors. Thus, the need to tolerate communication latency is extremely important, as any failure of the message passing layer to meet the needs of the application compounds the high cost of communication.

At present, a general purpose programming methodology appropriate for implementing scientific applications on dual-tier computers remains elusive. The programmer must carefully orchestrate parallelism and locality in the application, managing the interaction of processes, threads, shared memory, message-passing, synchronization, scheduling, and load balancing [3, 4, 5]. Such software techniques are beyond the reach of many application programmers, and the lack of effective software tools hinder efficient implementations of scientific calculations on dual-tier architectures by the scientific community.

This paper presents a domain-specific programming methodology for dual-tier multicomputers running bulk-synchronous numerical algorithms, that carry out relatively long periods of computation interspersed with coarse grain communication. We have implemented our methodology as a C++ framework called KeLP2 [4, 6, 7]. (From now on we will refer to the system simply as KeLP, dropping the 2.)

KeLP supports hierarchical control flow and data decompositions, as well as a hierarchical model of collective asynchronous communication. These mechanisms expose opportunities for improving performance by expressing latency-tolerant, dual-tier parallel algorithms. While the KeLP programmer must consciously attend to high-level algorithmic decisions, KeLP provides intuitive, concise abstractions to help the programmer implement efficient algorithmic decisions. In a variety of applications, KeLP's dual-tier formulations consistently outperform equivalent single-tier implementations hand-coded in MPI [8]. This observation is consistent with previous experience with a single-tier variant of KeLP, which has been in use for the past three years [9, 10, 11, 12, 13, 14].

This paper focuses on software engineering issues; detailed performance studies are reported elsewhere [4, 6, 7]. We describe how the KeLP dual-tier model meets the requirements of five different block structured applications: single-mesh and multi-level finite difference methods, the Fast Fourier Transform, and two blocked algorithms for dense numerical linear algebra–matrix multiply and blocked LU decomposition with partial pivoting. We discuss the current limitations of KeLP and suggest future research directions.

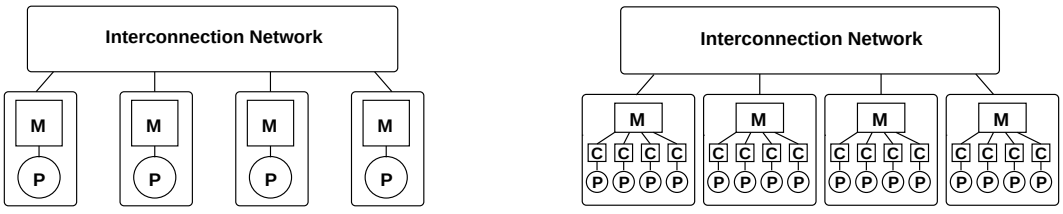


Figure 1: Block diagram of a single-tier (left) and dual-tier (right) computer. In this figure, P=processor, M=memory, C=external cache.

2 Assumptions

2.1 System Assumptions

We employ the following definition of a dual-tier parallel multicomputer as illustrated on the right side of Fig. 1. A dual-tier parallel multicomputer is hierarchical collection of n compute nodes each comprising p processors. The nodes execute n separate system images, and the p processors on each node share the private address space managed by the node. Parallelism exists at two levels: across the n nodes of the machine, and among the p processors within each node. Computations on a node are multi-threaded, and we assume that the thread scheduler does a fair job of assigning threads to processors, or that we can achieve a good schedule by binding threads to processors. We consider dedicated hardware running a single application at a time, without interference from other users.[1]

We assume blocking and non-blocking forms of point-to-point communication are supported, and that communication between nodes is costly relative to the collective floating point performance delivered by each node. In practice, communication may be far more costly than on a *single-tier* multi-computer, which is a special case of a dual-tier system with $p = 1$.

Although some designs may include a communication co-processor at each node to assist in managing communication, we do *not* assume that the co-processor can completely and effectively overlap non-blocking communication with computation as expressed by the messaging layer, e.g. MPI. This condition arises for a two principal reasons. First, the co-processor may be able to realize overlap only under limited operating conditions [15]. Second, the messaging layer implementation may not take advantage of the co-processor hardware's overlapping capabilities. In the interest of conserving development costs, a developer may choose to defer treatment of "advanced" capabilities in the messaging layer. For example, linearization of non-contiguous messages is particularly troublesome in MPI. The programmer is often better off packing their own data instead of relying on the MPI data typing mechanism to handle the activity[2]. This behavior reduces the effectiveness of overlap, tying up the "compute" processor with activities that the programmer might assume were

[1] Though operating system activities may occasionally interfere, their affect is assumed to be benign.

[2] Rusty Lusk, *private communications, 1998.*

handled by the co-processor.

2.2 Hardware Platforms

The results reported here were obtained from three platforms: a cluster of Digital AlphaServer 2100's with four Alpha 21064A processors per node interconnected by 155 Mbit/sec ATM; a cluster of 50MHz quad-processor SparcStation 20's interconnected by 10 Mbit/sec Ethernet; and a single-tier platform, the Cray T3E with 300MHz Alpha 21164 processors. All platforms use MPI [8] for message passing. The Alpha and Sun Clusters ran MPICH 1.0.12 [**?**], while the Cray T3E ran with the manufacturer-supplied version of MPI.

2.3 Application Domain

KeLP implements a domain-specific programming model targeted to block-structured applications. These applications carry out highly repetitive computations on coupled collections of uniform blocks of data, represented by multi-dimensional arrays. (Figs. 2 and 10.) Applications execute in bulk-synchronous SPMD fashion with long periods of computation interspersed by shorter periods of communication. Nodes transmit regular sections of data, attaining near-peak communication delivery rates.[3] Message lengths in KeLP applications tend to be tens to hundreds of thousands of bytes or longer.

Block-structured applications typically exhibit highly correlated patterns of collective communication involving sets of atomic regular section moves of multidimensional slices of data [9, 17, 18]. These patterns may not be known at compile time. They can depend on the input to the problem, to conditions evolving at run time, or both. We may describe these block-structured communication patterns using a table of meta-data, containing descriptions of the regular sections to be moved, i.e. a communication schedule [19]. This model is sufficiently general to treat a wide range of applications, including uniform finite difference methods (Fig. 2), blocked numerical linear algebra (Fig. 10), and irregular adaptive and multilevel methods [6].

We have just seen how a collective model captures the communication patterns inherent in a variety of block structured problems. For various reasons, non-blocking *point-to-point*

[3]Though the underlying blocks of data are structured, their sizes may be non-uniform, giving rise to an irregular communication structure.

communication may be inappropriate to express communication overlap in such applications. We have previously identified some common technological causes in §2.1. In addition, the non-blocking point-to-point communication model may also be inappropriate from software and algorithmic viewpoints. For example, multi-phase communication algorithms, such as dimension exchange or hypercube broadcast, impose a strict ordering on message transmissions. As a consequence, overlap strategies based on non-blocking communication must poll several times to ensure correct synchronization of the communication sequence. The inclusion of multiple synchronization points within application software tangles program structure, especially in cases where the number of synchronization points depends on quantities which cannot be known at compile time, such as the number of processors. Profligate synchronization can disrupt the execution of tightly optimized loop nests, lowering CPU performance. The techniques required to work around this difficulty, e.g. mixed mode programming, are beyond the means of many programmers. The interaction between message passing and threads is difficult to understand [3, 4, 5].[4]

2.4 Summary

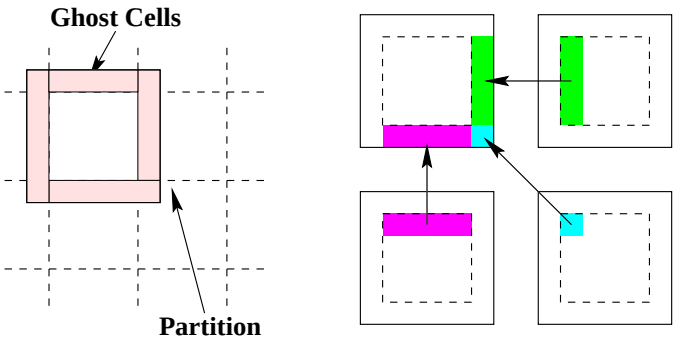
We have now identified a set of system and application requirements. In sum, we require run-time data decomposition and collective communication models that may be customized to the needs of the application and even to the specific input. Communication is assumed to be expensive and we require a means of overlapping it with computation. However, due to variations among different computing systems, we must be able to express such overlap without knowing the details of how the system will support the activity.

3 A Motivating Example

3.1 A Finite Difference Method

Consider a typical iterative finite difference application: solve Poisson's equation in three dimensions with a 7-point stencil using the Gauss-Seidel method with red-black ordering. We will refer to this application as *RedBlack3D*. We begin with a single-tier implementation

[4]The emerging OpenMP standard can simplify shared memory parallelization in mixed mode programming, but doesn't eliminate the underlying performance interactions between message passing and threads.




```
XArray X, MotionPlan M
for each i ∈ X
  Region RX = X(i).region()
  Region RI = trim(RX)
  for each j ∈ X
    if (i ≠ j) then
      Region R = RI ∩ X(j)
      M.Copy(X, i, R, X, j, R)
    end if
  end for
end for
```

Figure 2: A close up view of a block-partitioned two-dimensional grid, showing the ghost region for a typical partition (left). The middle figure shows some of the dependencies that must be satisfied to refresh the ghost cells on one processor's local grid. The right figure shows the KeLP code to express the communication, and will be described in a later section.

written with explicit message passing, e.g. MPI. The customary approach to parallelizing this iterative method is to split the subdomain into subregions using a regular blocked decomposition, and then surround each subdomain with a buffer called a *ghost region,* holding off-processor data used to update the boundary points of the subdomain. (Fig. 2.) The calculation consists of successive steps that compute and then communicate to fill the ghost cells.

3.2 Single-Tier Implementation

If we run our program on the Digital SMP cluster described in the last section, with one MPI process per processor, we treat the $n \times 4$ machine as a flattened structure with $4 \cdot n$ nodes. This seems reasonable since MPI is portable. However, performance may not be portable. When we run a 128^3 problem on 1 node, we observe that performance is 23 megaflops. If we run on 8 nodes, scaling the number of unknowns in proportion to the number of nodes, we find that performance is only 65 megaflops on a 256^3 problem. Hardware utilization is low– about 35%. We may improve performance significantly if we reorganize the mapping of data to processors. In our original code we let MPI decide how to assign the work. MPI configured the 32 processors into an 8×4 array, with each node occupying a single 4-processor column of the array. A hierarchical decomposition can improve locality, by configuring the nodes into

a 4×2 array, and the processors on each node into a 2×2 array. This optimization increases the ratio of on-node to off-node communication, improving performance to 114 megaflops.

3.3 Multi-Tier Implementation

The MPI implementation we used on the Alpha Cluster incurs the full TCP/IP overhead even when passing messages between processors on a single node, rather than using a multi-protocol messaging layer [5] to handle on-node messages via shared memory. We will try an alternative dual-tier strategy. We run with one MPI process per *node* and parallelize numerical computation on the node using shared memory techniques. We store ghosts calls only for data coming from outside the node. This optimization increases performance to 134 megaflops, again on 32 CPUs.

Still, there is room for improvement. The hardware sits idle 40% of the time waiting on communication. Our solution is to employ pre-fetching to mask the latency of communication [4, 20]. To implement this optimization we separate the points laying adjacent to the ghost region from the remaining interior points, which do not depend directly on the state of the ghost cells. (The left of Fig. 3). This partitioning enables communication to execute concurrently with the bulk of the computation. Once communication completes, we may then update the remaining work that borders the ghost cell region. The pre-fetching strategy works well and improves performance on 32 CPUs by an additional 22%–to 163 megaflops.[5]

We have improved the performance of the naive MPI implementation by a factor of 3.5 and are content with a scaled speedup of 7. However, the programming effort required to implement the performance optimizations is substantial. The principal difficulty is that the implementation of MPICH that we used on the AlphaServer is incapable of overlapping communication via non-blocking point-to-point communication. (Another problem is that we must we must manage irregular decompositions.) We must therefore resort to multi-threading to provide the overlap we require. Thus, we must employ a hybrid programming model managing threads, processes, synchronization, messages, and shared memory.

[5] We might be able to increase performance slightly by decreasing the granularity of our pre-fetch strategy, e.g. a wavefront method. This technique is admitted by KeLP, though we did not implement it.

Figure 3: (a) Collective-level partitioning over 4 nodes showing the halo region, and (b) the node-level partitioning on dual-processor nodes, detailing the irregular partitioning used to implement a pre-fetching strategy. The regions marked 0 and 1 correspond to the FloorPlan F_i in the code of Fig. 8, and the regions marked 2 through 5 to FloorPlan F_a. This decomposition is duplicated on each node.

Managing the interactions between these mechanisms adds considerable complexity to the programming effort, and is beyond the reach of many application programmers. This large programming overhead motivates the design of higher level abstractions to hide the low-level interactions.

3.4 Software Engineering Issues

A major requirement of our abstractions is to separate the expression of correct programs from optimizations that affect performance. This type of separation of concerns results in easier-to-develop, more maintainable code [21], with three three benefits. First, we may optimize the single processor performance of the numerical inner loops independently of how we parallelize the application. This permits us to build upon existing serial numerical code [11], which has often been carefully optimized. Second, we obtain a convenient *migration path*: a systematic approach to converting a single-tier program to an equivalent dual-tier version. Finally, we obtain *backward compatibility*: our dual-tier program may run efficiently on a single-tier computer, which is just a special case of a dual-tier computer with one processor per node. In fact, our dual-tier application outperforms the MPI implementation running on two single-tier configurations of the AlphaServer: a single SMP, or the entire machine running just one processor per node. On the Cray T3E, the dual-tier KeLP code runs within 1.5% of the performance of the hand-coded MPI variant.

Our simple programming example reveals a number of important requirements which generalize to a diversity of block-structured applications. We next describe our programming model more formally and then discuss its application to four other numerical problems.

4 The KeLP Programming Model

4.1 Overview

KeLP supplies mechanisms to help the programmer coordinate data decomposition, data motion, and parallel control flow. While KeLP hides the low-level details of managing resources, e.g. message-passing, processes, threads, synchronization, and memory allocation, it does not analyze program source code to make high-level restructuring and algorithmic design decisions as in a compiled language like HPF [22]. Rather, the KeLP philosophy is to empower the user to make such decisions–possibly at run time–as necessary to meet the requirements of the application [17]. Similarly, KeLP does not support automated data partitioning. The user is presumed to know how best to accomplish this task. Instead, KeLP provides a framework that facilitates the construction of partitioning libraries. Indeed, we have developed a variety of such libraries, and all the applications described in this paper have used at least one [6, 23].

4.2 Hierarchical Control Flow and Communication

Most existing SPMD parallel programming models reflect the single-tier design of previous generation multicomputers. They support two levels of control flow: *collective* level and *node* level. This two-level approach is clearly articulated in the Phase Abstractions programming model [24], which is embodied in MPI [8]. A program performs collective operations, such as reductions, barriers and broadcasts, interspersed within a node level program. The node-level instructions form separate threads of control and execute independently. In most cases, the node level will invoke highly tuned serial numeric kernels.

In contrast to the two levels of single-tier SPMD programming, KeLP supports three levels of control: a *collective* level, a *node* level, and a *processor* level, as shown in Fig.4. KeLP programs express parallelism at the node and processor levels and they express communication at collective and node levels. We note, however, that processors on different

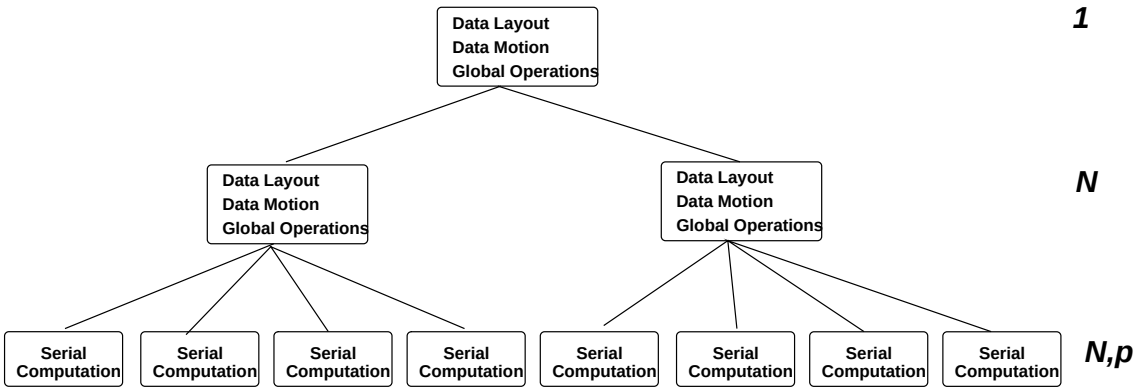


Figure 4: The KeLP 3-level control hierarchy.

nodes must communicate via their respective nodes and may not communicate directly by passing messages. Under KeLP, communication is always ascribed to the node rather than to the individual processors. This three-level model is similar to that employed in PMH[25]. The collective and node levels each manage their own data layouts and data motion. The processor-level control stream executes a serial instruction stream on a single physical processor. KeLP abstractions help manage each level independently where desired, and also help manage interactions between the levels where necessary.

4.3 Programming Abstractions

The KeLP abstractions fall into two categories: *meta-data* and *instantiation*. These abstractions are listed in Table 1. KeLP meta-data objects represent the abstract structure of some facet of the calculation, such as a decomposition or a communication pattern. Instantiation objects carry out program behavior based on information contained in meta-data objects.

Meta-data abstractions. There are four meta-data abstractions: the *Region, MotionPlan*, *Map*, and *FloorPlan*. The `Region` represents a rectangular subset of Z^n; i.e., a regular section with stride one. KeLP provides the *Region calculus*, a set of high-level geometric operations to help the programmer manipulate Regions. Typical operations include *shift*, *intersection*, *pad*, and *trim*.

The `Map` class implements a function $Map : \{0, \ldots, k-1\} \rightarrow Z$, for some integer k. That is, for $0 \leq i < k$, $Map(i)$ returns some integer. The Map forms the basis for node and

Meta-Data Abstractions		
Name	Definition	Interpretation
Point	$<$ int i_0, int i_1, ... int $i_{D-1}>$	A point in Z^D
Map	$f : \{0, \ldots k-1\} \rightarrow Z$	an integer mapping
Region	$<$ Point l, Point $h>$	A rectangular subset of Z^D
FloorPlan	$<$ Array of Region, Map $>$	A set of Regions, each with an integer *owner*
MotionPlan	List of $<$ Region R_s, int i, Region R_d, int $j>$	Block-structured communication pattern between two FloorPlans
Instantiators		
Name	Description	
Grid	A multidimensional array whose index space is a Region	
XArray	An array of Grids; structure represented by a FloorPlan	
Mover	Object that atomically performs the data motion pattern described by a MotionPlan	

Table 1: A brief synopsis of the KeLP data types.

processor assignments in KeLP partitioning. The `FloorPlan` consists of a Map along with an array of Regions, and it is a table of meta-data that represents a potentially irregular block data decomposition. Alternatively, a FloorPlan can represent distribution of work among processors within a single node, as it is derived from Map.

The `MotionPlan` implements a dependence descriptor, which is also known as a communication schedule [19]. The programmer builds and manipulates MotionPlans using geometric Region calculus operations, a process which will be described shortly.

4.4 Storage Model

The Point, Region, Map, FloorPlan, and MotionPlan meta-data may live at any of the three levels of control flow. 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; e.g. the contents of a FloorPlan written at the processor level are undefined at the node and collective levels.

KeLP meta-data objects describe only the structure of data and communication within a program. The KeLP `Grid` and `XArray` objects hold the actual data of an application. A Grid is an array of objects all of the same type, whose index space is a Region. For example,

the Fortran 90 array `real A(3:7)` corresponds to a one-dimensional Grid A of `real` and has a region `A.region() == [3:7]`.

The Grid storage class may live only at the node level. This built-in assumption was made in the interest of efficiency. A KeLP program may access the Grid data from the node-level instruction stream at that node, or from processor-level instruction streams nested at that node, but not from the collective level (or from other nodes). A collective Grid, in effect, would define a shared-memory address space across all nodes. However, it cannot be assumed that the hardware supports global shared memory, since code that made this assumption could not offer portability with performance. KeLP's minimalist philosophy is to avoid abstractions that cannot be known to deliver portable performance. KeLP provides constructs such as the XArray and MotionPlan for building abstractions like grid partitioners. The performance of such libraries can be readily understood, since all data motion is explicit.

An `XArray` is an array of Grids, whose structure is represented by a FloorPlan. It is derived from Map. All elements must have the same number of spatial dimensions, but the index set of each Grid component (KeLP Region) can be different. Grid components can have overlapping Regions, but they do not share memory. The application is responsible for managing any such sharing explicitly. An XArray is a collective object, and must be created from the collective program level. Owing to built-in assumptions concerning the Grid storage class, we also make the built-in assumption that an XArray lives only at the collective level. An interesting variation would be to allow node level XArrays, and is the subject of future research.

For an XArray X, $X(i)$ denotes the ith Grid component of X. Processor assignments are determined by the Map used to construct the XArray's FloorPlan. The indices of an XArray are virtual. The number of XArray elements may be greater than the number of physical nodes or processors, and the mapping of XArray elements to processors may be many-to-one. This capability is useful in handling load balancing.

KeLP supports a collective communication operation which performs block transfers of regular array sections between two XArrays. We will discuss data motion in the next section.

5 Programming example

To illustrate KeLP's programming constructs, we describe the implementation of RedBlack3D, which was introduced in §3. We begin with a single-tier implementation for a distributed memory computer with uniprocessor nodes. Then, we migrate the code to a dual-tier architecture, adding optimizations that improve performance by overlapping communication with computation. For illustrative purposes, our code examples use an abstract syntax and in a few cases we substitute English descriptions for simple operations.

5.1 Single-tier code

The single-tier version of RedBlack3D manages a single level of parallelism and locality. As described in §3 we follow the usual SPMD implementation strategy that employs HPF-style `BLOCK` data decomposition [22] and carries additional ghost cells to buffer off-processor data. Each relaxation sweep 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. Fig. 5 shows the main routine for RedBlack3D. The program begins execution at the collective program level, and there is a single logical thread of control. All nodes execute the same statements in the main procedure.

Data distribution. We distribute the $N \times N \times N$ computational domain (at line 1, the Region object called *domain*) with the help of a partitioning library written in KeLP.[6] We pad each node's local subdomain with a layer of ghost cells at line (3); the `pad()` operation adds space for ghost cells to each FloorPlan element $T(i)$.

With the FloorPlan T initialized, we instantiate storage by creating two XArrays: u and rhs (lines 4-5) holding, respectively, the computed solution and right-hand side for the Poisson equation. The XArray constructor creates an XArray with one Grid for each element of T. The ith Grid of u, $u(i)$, has the domain $T(i).region()$ and exists on node $T.owner(i)$.

Data motion. We are now ready to handle data motion. As mentioned previously, KeLP supports a collective model of communication. Two KeLP classes are used to express communication: the *MotionPlan* and *Mover*. To understand the operation of the MotionPlan

[6] For example, KeLP2 includes a library called DOCK that implements run-time `BLOCK` decompositions [23].

```
( 1)    Region domain = [1 : N, 1 : N, 1 : N]
( 2)    Floorplan T = blockPartition(domain)
( 3)    for each i ∈ T, T.setRegion(i,pad(T(i)))
( 4)    XArray u(T)
( 5)    XArray rhs(T)
( 6)    MotionPlan M
( 7)    BuildFillGhostPattern(u, M)
( 8)    Mover mov(u, u, M)
( 9)    InitialConditions(u, rhs, mov)
(10)    integer rb = 0
(11)    while (not converged) do
(12)        Relax(u, rhs, mvr, rb)
(13)        rb = (rb + 1) mod 2
        end while
```

Figure 5: Main procedure for redblack3D example.

and Mover, we use following notation. Let D and S be two XArrays, $D(i)$ be element i of D, and $S(j)$ element j of S. Let R_s and R_d be two Regions. Then, the notation

$$(D(i) \ \ on \ \ R_d) \Leftarrow (S(i) \ \ on \ \ R_s) \tag{1}$$

denotes a block copy operation: copy the values from $S(j)$ over all the indices in R_s into $D(i)$ over the indices in R_d. We assume that R_d and R_s contain the same number of points, though their shapes may be different. The copy operation visits the points in R_d and R_s in a defined systematic order, i.e. column major.

Now, we encode a communication pattern as records of a MotionPlan ρ, a list containing entries $\rho(k)$. These entries are 4-tuples of the following form:

$$\langle Region\ R_s, \ \ int\ i, \ \ Region\ R_d, \ \ int\ j \rangle. \tag{2}$$

The components of the 4-tuple describe communication in a manner consistent with the discussion of the previous paragraph. We use the `add()` member function to add new MotionPlan entries.

We instantiate a *Mover* object μ by binding a MotionPlan ρ to two XArrays: `Mover` μ `= Mover(`ρ`, S, D)`. The result is an object with two operations, *start()* and *wait()*. The Mover

executes collective communication as an atomic operation, initiating the data transfers via the *start*() member function. This call is asynchronous and a return does not indicate that communication has completed. The *wait*() member function is used to detect completion.[7]

When the Mover returns from *wait()*, all transfers will have completed. In particular, the Mover will have executed the following communication operation for each $\rho(k)$: ($D(\rho(k).i)$ on $\rho(k).R_d$) $\Leftarrow$ ($S(\rho(k).i)$ on $\rho(k).R_s$), where we designate the entries of the $\rho(k)$ with the C-style `struct` qualifiers $\rho(k).R_s$, $\rho(k).R_d$, $\rho(k).i$, $\rho(k).j$. Since the specific order of transfers is not defined, correctness must not depend on that ordering.

When the Mover is invoked from the collective level, each node will synchronize its own processors. There is no collective synchronization across nodes, since such synchronization is implicit in the execution of the Mover, which satisfies inter-node dependencies.

Returning to our code of Fig. 5, we build a MotionPlan M (lines 6 and 7) to describe the ghost cell update pattern, and then instantiate a Mover object (line 8) which will carry out the communication. The algorithm to build the MotionPlan is simple and is shown in Fig. 2.

We build a Mover for each data motion pattern to be executed. Since *rhs* is a static data structure, we do not carry out any data motion on it, and so there is no need to build a Mover for it. We note that the constructor arguments to the Mover *mov* (line 8) specify that the source and destination XArray are the same.

The data motion is actually carried out in the `Relax` routine, which is shown in Fig. 6. This code is written in single-tier form and will run on at most one processor per node. `Relax` begins execution from collective control flow. At lines 1 and 2 the nodes invoke the Mover to update the ghost cells.

The Mover `start` member function begins moving the data asynchronously (line 1). Since we are not going to overlap communication and computation, we immediately call the Mover `wait` member function to block until the data motion completes (line 2). Once `wait` returns, the program performs the serial relaxation kernel on each node: it drops from the collective

[7]Individual nodes or processors may block on `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, providing a more localized and less costly form of synchronization than a barrier.

```
Relax(XArray u, XArray rhs, Mover mov, integer rb)
begin
(1)     mov.start()
(2)     mov.wait()
(3)     for nodeIterator ni ∈ u do
(4)         serialRelax( u(ni), rhs(ni), u.region(ni), rb )
        end do
end
```

Figure 6: Single-tier KeLP code for redblack3D relaxation.

control level to node-level control using the KeLP nodeIterator loop (line 3).

The nodeIterator constructor takes a `Map`, which specifies the number of loop iterations and the mapping of these iterations to nodes. Recall that XArray is derived from Map. Thus, we may use the XArray u as the Map for constructing this iterator. NodeIterator ni uses the owner-computes rule to create one loop iteration for each element of u. Each loop iteration ni executes on node u.owner(ni) only, and in this case entails calling `serialRelax`, a serial numerical kernel which performs the relaxation (line 6). Inside the loop, the program runs in node-level control, with one stream per node.

5.2 Dual-tier code

We next modify our single-tier program to take advantage of the hierarchical organization of a dual-tier machine. Fig. 7 shows dual-tier KeLP code to implement the `Relax` subroutine with two levels of parallelism. As before, `Relax` starts in collective control flow, updates ghost cells by invoking the Mover (lines 2 and 3), and drops to node-level control via the nodeIterator, with one iteration per block of XArray u (line 3).

From the node-level loop, we now parallelize the relaxation on each block across the processors on the node. Consider iteration ni of the loop, executing on node $u.owner(ni)$, which relaxes Grid $u(ni)$, with Region $u.region(ni)$. We parallelize the numerical computation across the p processors by partitioning $u.region(ni)$ into p blocks and then assigning each block to a single processor. As before, we rely on a partitioner utility (line 4) [23]. This partitioner maps each grid $u(ni)$ with a private FloorPlan F, such that $F.owner(i) =$

```
Relax(XArray u, XArray rhs, Mover mov, integer rb)
begin
(1)       mov.start()
(2)       mov.wait()
(3)       for nodeIterator ni ∈ u
(4)            Floorplan F = IntranodeBlockPartition(u.region(ni))
(5)            for procIterator pi ∈ F
(6)                 serialRelax( u(ni), rhs(ni), F(pi), rb )
           end for
      end for
end
```

Figure 7: Dual-tier KeLP code for redblack3D relaxation.

processor i.

We now drop to processor-level control via the procIterator (line 5). Similar to the nodeIterator, the procIterator executes one iteration for each element of F, and executes iteration i on processor $F.owner(i)$. (Recall that FloorPlan is derived from Map). From the processor-level control, each iteration invokes the serial kernel `serialRelax` independently (line 6). The `serialRelax` routine accepts a region-valued argument $F(pi)$ specifying the subset of Grid $u(ni)$ assigned to the iteration.

In looking at the above code we notice the similarity in how we build a FloorPlan at each level of control flow, and the use of a single iterator construct to handle that control flow. This approach should be contrasted with one that employs threads and message passing explicitly. In the latter case the programmer must manage processes and threads with separate sets of primitives. Threads communicate via shared memory and use locks, barriers, and events to synchronize. Processes communicate and typically synchronize by passing messages. They may also use barriers. The mechanisms are not symmetric, and their interaction is extremely difficult to manage.

There is one aspect of KeLP programming model, however, which is not symmetric. In KeLP's memory model, a Grid G lives in a single address space corresponding to one node. The program can access the data of G from node-level control or from processor-level

control. Since these semantics introduce the potential for race conditions, the programmer is responsible for avoiding them. This was a conscious decision on our part; a more restrictive model that guaranteed safety would require compiler analysis or otherwise limit KeLP's ability to deliver high performance.

By default, KeLP enforces a logical synchronization point at the end of each procIterator loop, but not at the nodeIterator loop. No action is required to enforce a logical nodeIterator barrier, since the Mover is assumed to carry out data motion correctly. In effect the Mover implements an unsafe form of local barrier synchronization. This design decision was made intentionally, reflecting the belief that the programmer should have the flexibility to avoid costly global synchronization at the risk of introducing program errors. In the RedBlack3D code, the procIterator barrier 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 node-level synchronization if desired, but is responsible for ensuring correctness.

5.3 Dual-tier code with communication overlap

Using the techniques described in §3.3 we restructure the dual-tier implementation of the `Relax()` routine to introduce pre-fetching. The restructured code appears in Fig. 8. In order to overlap communication and computation we will use the following strategy. (1) asynchronously begin communication; (2) perform local computation on the interior of each Grid (as shown in Fig. 3); (3) wait for communication to complete; (4) perform local computation on the annulus of each Grid. As before, the collective level invokes the Mover to asynchronously initiate communication. Unlike the previous examples, where the program immediately blocked on the Mover, we defer the wait on communication, and begin local computation immediately.

To manage this overlapped communication algorithm in KeLP, we generate partitioning information at node-level control, storing the information in two FloorPlans, FloorPlans F_a and F_i (lines 3 and 4), which are shown in Fig. 3. With the FloorPlans set up, two procIterator loops carry out the computation as follows. The first procIterator loop (lines 5-6) sweeps over the interior points. Next, the Mover waits for communication to complete (line 7). Once the ghost cells have arrived, the procIterator loop at lines 9-10 sweeps over

```
Relax(XArray u, XArray rhs, Mover mov, integer rb)
begin
( 1)      mov.start()
( 2)      for nodeIterator ni ∈ u
( 3)           Floorplan F_i, F_a
( 4)           IntranodeDepPartition(u.region(n), F_i, F_a)
( 5)           for procIterator pi ∈ F_i
( 6)                serialRelax(u(ni), rhs(ni), F_i(pi), rb)
               end for
          end for
( 7)      mov.wait()
( 8)      for nodeIterator ni ∈ u
( 9)           for procIterator pi ∈ F_a
(10)                serialRelax(u(ni),rhs(ni),F_a(pi),rb)
               end for
          end for
end
```

Figure 8: Dual-tier KeLP code with communication overlap for redblack3D relaxation.

the points in the annulus.

5.4 Discussion

Beginning with a single-tier implementation we have incrementally constructed a optimized dual-tier parallel code that overlaps communication and computation. Two mechanisms in KeLP worked synergistically to improve performance: hierarchical collective communication and hierarchical flow control. The KeLP Mover encapsulates composed communication patterns, executing communication on each node as a separate task in parallel with computation. Thus, the implementation policies for managing parallelism within the communication task may be handled separately from computation. Since the dual-tier systems we used did not support communication overlap via a co-processor, we chose to implement the Mover to run on a reserved SMP processor. The effect was to reduce the length of the critical path of application, even though the numerical computation time actually increased. This was accomplished without entangling the application program with the details. The

KeLP abstractions are sufficiently general to express optimized algorithms for a variety of block-structured scientific calculations, which we will examine in the next section.

An important design goal of KeLP is to hide unnecessary detail from the user without sacrificing performance. Another goal was to enable the re-use of existing numerical kernels without entailing massive reprogramming. To understand how well KeLP meets these goals we built a hand-coded MPI version of RedBlack3D and compared it against the KeLP code. We ignored the code shared in common by the two implementations, that handles command line argument processing and the C++-to-Fortran interface. Excluding comments, the relevant MPI code consists of 354 lines of C++, plus 54 lines of Fortran 77 to handle the relaxation. The dual-tier KeLP code comprises 236 lines of C++ and uses the *identical* Fortran 77 module. The KeLP code provides overlap capabilities, and it runs efficiently on both single-tier as well as dual-tier architectures as described in §3. The module which handles the irregular decomposition accounts for 71 of those 236 lines, and indeed the single-tier KeLP code weighs in at just 157 lines, about half the size of the MPI code. But code size doesn't reveal the full picture. The most challenging part of the MPI code–that handles communication–is about 180 lines long. The equivalent KeLP encoding is about only 15 lines long. This striking reduction in code complexity comes without a performance penalty [17, 6].

6 Application Study

In this section, we evaluate the KeLP programming model against four additional applications that raise distinct programming issues. First, we examine a multilevel finite difference method, the application class originally targeted by KeLP. We next consider Fast Fourier Transform, blocked matrix multiplication, and blocked dense LU factorization with partial pivoting–which is well outside of KeLP's intended problem domain due to the use of block cyclic data decompositions. The matrix multiplication and LU algorithms implement new pipelined overlap formulations due to Fink[6]. For each application, we describe how KeLP facilitated the design of the software and comment on the appropriateness of the KeLP model. With a few noted exceptions, all five codes were run on the three platforms described in §2.2.

6.1 NAS Multigrid Benchmark

The NAS-MG multigrid benchmark [26] solves Poisson's equation in 3D using a multigrid V-cycle [27]. In this stencil-based computation, a series of meshes are organized into levels and we parallelize each level with techniques similar to those for RedBlack3D.

The NPB 2.1 code specifies a three-stage dimensional exchange algorithm to satisfy boundary conditions. The dimension exchange algorithm introduces an interesting software design issue. Owing to a synchronization constraint that exists between phases of dimension exchange, we cannot naively build three separate Movers and start all three at once. Instead, we derive a new class from Mover, called `MultiMover`, which serially invokes a sequence of Movers, as shown in Fig. 9. To the calling application, the three-stage execution sequence of MultiMover appears to be an atomic operation.

This communication example highlights the expressive power of KeLP. By representing the multi-phase communication as an atomic object, the programmer can asynchronously execute an arbitrary sequence of message-passing and synchronization operations. In contrast, non-blocking MPI calls asynchronously start only one message-passing operation at a time. To overlap communication using a sequence of operations, an 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. This polling activity is highly disruptive, since it may interrupt highly-tuned numeric kernels, degrading performance. It also tangles the program structure. The more powerful KeLP design lends itself to better structured and more efficient code.

Computational results show that the dual-tier KeLP code outperforms the MPI code by 12% on eight AlphaServers and by 13% on four SparcStations. On the single-tier Cray T3E, performance of the KeLP version of the application is nearly indistinguishable from that of MPI, coming within 3% on 64 processors. The KeLP code outperforms the MPI code on a single node of the AlphaServer or the SparcStation cluster. Thus, KeLP's higher level of abstraction does not come at the expense of performance–and it actually boosts performance on dual-tier computers.

6.2 NAS-FT Benchmark

The NAS FT benchmark solves a 3D diffusion equation using a Fourier method. The bottleneck of this computation is a 3D transpose (total exchange), which is particularly costly on an SMP cluster [5, 4]. To mask some of the latency of communication, we employ Fink's restructured variant[6] of Agarwal *et al.*'s [28] pipelined algorithm.

The publicly available NAS FT benchmark (NBP 2.1) is written in Fortran 77 and uses explicit message passing with MPI. This code performs the total exchange using the MPI `all_to_all` call. The KeLP code encodes the matrix transpose using a MotionPlan and KeLP Mover. On the Cray T3E the KeLP code outperforms the MPI code slightly–by a few percent, on up to 64 processors. These results indicate, somewhat surprisingly, that KeLP implements the global matrix transpose messages about as efficiently as the MPI collective call [4, 9, 17]. On the Alpha cluster, we were able to improve performance by about 13% by introducing software pipelining to overlap communication.

6.3 SUMMA Matrix Multiplication

We now turn our attention to a different and important problem class: dense numerical linear algebra. Since KeLP provides facilities to manage distributed block structures we next consider how well the KeLP abstractions aid in efficient implementations of blocked dense matrix algorithms.

Perhaps the best-known blocked dense matrix algorithm is dense matrix multiplication. We consider the SUMMA (Scalable Universal Matrix Multiply Algorithm), due to van de Geijn and Watts [29]. Using KeLP, we devised a new variant of SUMMA variant that uses software pipelining to overlap communication and computation [4, 6].

SUMMA implements matrix multiplication as a series of blocked outer products over distributed matrices. Assume A, B, and C have `BLOCK` data decompositions over the nodes. We distribute the work based on the decomposition of C; each node will compute the subsection of C that it owns, as shown in Fig. 10. This pipelined algorithm carries out a series of broadcasts as shown in the Figure. The broadcasts transmit a *panel* of data rather than the entire block of data assigned to the processor, that is, a vertical or horizontal slice. This strategy decreases the granularity of the pipeline, increasing the benefits of overlap.

To implement SUMMA we used a domain specific library, call `dGrid`, which we built on top of KeLP [6, 23]. The `dGrid` library handles the details of managing the progression of panel broadcasts across the global matrix. In particular, it implements a replicated grid abstraction, a convenient way to express column and row broadcasts among processors mapped onto virtual grids.

The dual-tier code with communication overlap outperforms the single-tier MPI code by as much as 33% on the Alpha Cluster. On the Cray T3E, the KeLP code runs slightly faster than MPI, though it is ultimately non-scalable due to the way we handled communication in the Mover. In particular, we used a ring-broadcast algorithm, which has a linear running time, in lieu of a logarithmic time broadcast algorithm. On 64 T3E processors, the MPI implementation overtakes KeLP, outperforming it by 18%. We are currently investigating a solution to the non-scalable performance of our broadcast algorithm, as discussed in the next section on LU decomposition.

6.3.1 LU Decomposition

Finally, we consider the blocked right-looking distributed LU factorization algorithm of SCaLAPACK [30]. This application was selected because it is well outside the intended problem domain of KeLP. In particular, due to use of a `BLOCK_CYCLIC` decomposition and a pivot selection step, synchronization requirements of the algorithm are finer-grained than the other applications. A detailed description of the algorithm is described by Fink [6].

As in SUMMA and FT, LU carries out a series of broadcast operations. However, LU is distributed in 1-dimensional `BLOCK_CYCLIC` fashion rather than in `BLOCK` fashion in order to load balance the computation. This complicates the implementation since KeLP does not support `BLOCK_CYCLIC` decompositions primitively. Instead, we emulate them. As a result, each node will carry several (tens) of XArray elements.

The decomposition is by column block and there are two broadcast operations across columns. The first transmits a vector of integers, which refer to the pivot selections. The second transmits a vertical slice of data, a long and thin sub-column of the matrix. Instead of using the MPI broadcast capability, we built a replicated array abstraction using KeLP. Class *ReplicatedArray* is an XArray whose FloorPlan is the replicated instance of a single

KeLP Region. Thus, if we copy data from source to all overlapping elements of a replicated array, we achieve the effect of a broadcast. This abstraction is appealing because it reflects the logical structure of the algorithm: we are broadcasting to block cyclic distributed block columns of data, not to processors.

The LU application is interesting for other reasons. First, although we used pipelining to express parallelism across nodes, we employed task parallelism within each node to express the various computational steps in the LU algorithm [6]. In fact, we used two Movers, one each at the collective and node program levels. The collective Mover handled data motion arising in the pipelining strategy, while the node-level Mover handled data motion within shared memory.

On the Cray T3E, the single-tier implementation ran within 1.5% of the speed of the MPI (SCaLAPACK) code. The dual-tier KeLP implementation of LU factorization ran slightly faster (up to about 10%) than the MPI implementation on up to 4 nodes of the AlphaServer.[8] However, the KeLP implementation does not scale beyond 4 nodes, and is overtaken by the SCaLAPACK code on 8 nodes. This is true because we used a naive ring broadcast algorithm in the implementation of the ReplicatedArray class. This algorithm has a running time that is linear in the number of nodes. We believe that performance could be improved significantly with a logarithmic time hypercube broadcast algorithm. Like the dimension exchange algorithm used in the multigrid application, this is a multi-phase algorithm. However, a more significant challenge is that the KeLP implementation of `BLOCK_CYCLIC` decomposition doesn't scale to larger numbers of nodes.

7 Discussion

Our experiences have shown that KeLP is effective in improving performance by explicitly managing hierarchical locality and by masking communication costs through overlap. KeLP's programming model provides an appropriate level of abstraction for a variety of block-structured scientific calculations, subordinating incidental detail without sacrificing performance. The model is also robust, as KeLP applications achieve portable performance

[8] We did not have access to tuned BLAS on the SparcStation cluster, so we did not run on that machine.

across dual-tier and single-tier architectures.

KeLP introduces a new three-level control flow model, which supports structured parallelism through iterators. As in CC++ structured parallel loops [31], the KeLP iterators simplify the expression of parallelism, but restrict the forms of parallel control flow available to the programmer. However, by disallowing unstructured parallel control flow, we are able to rely on the structured loops to implicitly handle inter-processor synchronization or to permit the programmer to relax synchronization requirements in a controlled way. This approach simplifies the programming model.

The KeLP communication model enables the programmer to express and compose elaborate data motion patterns as collective operations. The ability to encapsulate complex collective communication patterns in turn enables the KeLP programmer to express such communication as a concurrent task, which may then be overlapped using traditional mechanisms, e.g. threads. Thus, on platforms that do not support communication overlap via a co-processor–such as the ones used to produce the results in this paper–KeLP may realize overlap using a spare processor on each node. Even in cases where the co-processor does support overlap, the ability to express communication as a concurrent task may still realize benefits. For example, linearization of non-contiguous data structures may not be supported by a local MPI implementation. Therefore, this activity would not be overlapped in a non-blocking message passing call. Due to the high cost of packing non-contiguous data on some architectures [17], the KeLP Mover could overlap a significant amount of overhead that would otherwise remain in the critical path of the application.

One way to reduce communication costs in a dual-tier multicomputer is to provide a multi-protocol message passing layer that intercepts on-node messages through fast shared memory avoiding the overhead of communication protocols such as TCP/IP [5]. While this implementation strategy would likely improve the KeLP Mover's performance on the dual-tier platforms used in this study[9], it does not deal with the problem of how to conveniently overlap multi-phase communication. The issue here is not related to the *implementation* of non-blocking point-to-point communication, but the *inappropriateness* of the mechanism.

[9]The AlphaServer port of MPI-CH did not support multi-protocol communication, and the Sun port was not robust.

Thus, a major contribution of this paper is to advocate a framework for writing block-structured asynchronous collective communication algorithms. This framework would serve as middleware sitting atop communication APIs like MPI, but could use other APIs as well [32].

The Data Mover resembles an MPI persistent communication object. However, KeLP provides inspector-executor analysis, which is particularly useful in irregular problems, and it also provides first-class support for multidimensional arrays, via user-defined metadata and a geometric region calculus. KeLP avoids MPI's awkward data type mechanism to handle strides of non-contiguous faces that would entail registering a separate data type for each stride appearing in the data. More generally, the KeLP meta-data abstractions, e.g. the Region calculus, provide a more intuitive and concise notation for expressing and managing customized communication domains.

In designing the KeLP model, we made some built in assumptions to meet demanding performance requirements. These assumptions related to the parallel iterator and Grid implementations. We chose to limit parallel iterator loop nesting to two levels of control flow due to implementation concerns. 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, we would raise many open questions regarding how to map the control flow onto the hardware. Answering these questions remains an open research issue.

Our results suggest future directions for numerical library design. Typically, a programmer will rely on a library such as SCaLAPACK [33] to implement a core of common numerical algorithms. 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 [34]. Our results reinforce the benefits of asynchronous entry points to numerical libraries.

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 node. 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.

8 Related Work

Several workers have incorporated hierarchical abstractions into programming languages. The Cedar Fortran language [35] included storage classes and looping constructs to express multiple levels of parallelism and locality for the Cedar machine. The pSather language is based on a cluster machine model for specifying locality [36], and implements a two-level shared address space in the framework of a concurrent object-oriented model. Bader and JáJá have developed SIMPLE [37], a set of collective communication operations for SMP clusters. SIMPLE provides more general, lower-level primitives than KeLP, such as reductions and gather/scatter. But, it does not help with data decomposition nor does it overlap communication with computation. Merlin *et al.* have successfully incorporated KeLP with SHPF, a data-parallel HPF-like Fortran dialect [38].

The NESL language [39] 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. Several task-oriented parallel languages [40, 41, 42, 43] support fork-join parallelism suitable for divide-and-conquer.

Crandall *et. al* [44] report experiences with dual-level parallel programs on an SMP cluster and motivate further research into dual-tier programming models and environments. Proteus [45] is a custom-built hierarchical SMP cluster designed for image processing. 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. Lumetta *et al.* have implemented low overhead message passing on an SMP cluster, that intercepts on-node communication efficiently through shared memory [5].

Erlichson *et al.* consider implementation tradeoffs for SoftFLASH, a software-based distributed virtual shared memory system implemented on a cluster of SGI Challenge multiprocessors [46]. The study concludes that dedicated co-processors improve performance, but not in proportion to the processing resources consumed.

9 Conclusions

We have presented a programming model to facilitate high-performance implementation of block-structured scientific calculations on dual-tier computers. The KeLP programming model introduces new mechanisms to manage two levels of locality and parallel control flow that cleanly separate 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 model that encapsulates complex collective data motion patterns as an atomic operation, and supports their overlap with computation.

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 dual-tier architectures. With these mechanisms, the programmer can implement a variety of algorithmic policies, without drowning in low-level implementation details.

KeLP implements a domain specific programming model and does not admit highly irregular problems such as general sparse matrix methods, tree-based data structures, or unstructured meshes, which exhibit fine-grained communication. These application classes demand new implementation techniques and programming abstractions. However, the notion of a KeLP-style collective Mover is relevant to unstructured problems that manage halo regions, provided they are amenable to bulk-synchronous execution.

The design of dual-tier KeLP has evolved from a single-tier variant in use for three years at the time of this writing [9], and inherits the Point, Region, Grid, and XArray abstractions from the LPARX programming system [47]. A variety of applications [48, 10, 11] and computer science projects [12, 49, 13, 14, 50] have used or are using the single-tier KeLP

system. We are currently studying KeLP2 and its applications on the Department of Energy's ASCI Blue-Pacific TR machine.

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: clusters of clusters, separate computers interconnected over a local or wide area network, and parallel input/output. 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. We are currently investigating a general n-level parameterized model, which relaxes architectural assumptions built into KeLP.

Acknowledgments

This paper written in part while the first author was on sabbatical leave at the University of Karlskrona/Ronneby, Department of Computer Science and Business Administration, Soft Center, S-372 25 Ronneby, Sweden. The authors would like to thank Paul Kelly and the anonymous referees for helpful suggestions on how to improve this paper. The first author dedicates his portion of the work performed on this paper to the memory of Thorsten Becker (1968-1998).

This work was supported in part by NSF contract ASC-9520372 and in part by NSF contract ACI-9619020, "National Partnership for Advanced Computational Infrastructure." Stephen J. Fink was supported by the DOE Computational Science Graduate Fellowship Program. Computer time on the Cray T3E was provided by a UCSD Jacobs School of Engineering Block Grant and the San Diego Supercomputer Center. Computer time on the Maryland Digital AlphaServer was provided by NSF CISE Institutional Infrastructure Award CDA9401151 and a grant from Digital Equipment Corp.

References

[1] P.R. Woodward, "Perspectives on Supercomputing: Three Decades of Change," *IEEE Computer*, Vol. 29, Oct. 1996, pp. 99–111.

[2] "Accelerated Strategic Computing Initiative (ASCI)," Tech. Report UCRL-MI-125923, Lawrence Livermore Nat'l. Laboratory, 1998.

[3] W.W. Gropp and E.L. Lusk, "A Taxonomy of Programming Models for Symmetric Multiprocessors and SMP Clusters," *Programming Models for Massively Parallel Computers,* W.K. Giloi, S.Jahnichen, and B.D. Shriver, eds, IEEE Computer Society Press, 1995, pp. 2–7.

[4] S.J. Fink and S.B. Baden, "Runtime Support for Multi-Tier Programming of Block-Structured Applications on SMP Clusters," *Scientific Computing in Object-Oriented Parallel Environments,* Y. Ishikawa, R. Oldehoeft, J.V.W. Reynders, and M. Tholburn, eds., Lecture Notes in Computer Science, Vol. 1343, Springer-Verlag, Berlin Heidelberg New York, 1997, pp. 1–8.

[5] S.S. Lumetta, A.M. Mainwaring, and D.E. Culler, "Multi-Protocol Active Messages on a Cluster of SMPs," *Proc. SC97*, IEEE Computer Soc. Press, 1997, also available at http://www.sc98.org/sc97/proceedings/TECH/LUMETTA/INDEX.HTM (Sept. 1999).

[6] S.J. Fink, *Hierarchical Programming for Block–Structured Scientific Calculations*, doctoral dissertation, Univ. of California, San Diego, Dept. Computer Science and Eng., 1998.

[7] S.B. Baden and S.J. Fink, "Communication Overlap in Multi-Tier Parallel algorithms," *Proc. SC '98*, IEEE Computer Soc. Press, 1998, also available at http://www.supercomp.org/sc98/TechPapers/sc98_FullAbstracts/Baden708/INDEX.HTM (Sept. 1999).

[8] Message Passing Interface Forum, "MPI: A Message-Passing Interface Standard," http://www-unix.mcs.anl.gov/mpi/ (available Sept 1999), Jun. 1995.

[9] S.J. Fink, S.B. Baden, and S.R. Kohn, "Flexible Communication Mechanisms for Dynamic Structured Applications," *Parallel Algorithms for Irregularly Structured Problems, Third Intl. Workshop, Irregular '96,* A.Ferreira, J.Rolim, Y.Saad, T.Yang, eds., Lecture Notes in Computer Science, Vol. 1117, Springer-Verlag, Berlin, Heidelberg New York, 1996, pp. 203–215.

[10] S.R. Kohn, J.H. Weare, E.M.G. Ong, and S.B. Baden, "Software Abstractions and Computational Issues in Parallel Structured Adaptive Mesh Methods for Electronic Structure Calculations," *Structured Adaptive Mesh Refinement Grid Methods,* S.B. Baden, N.Chrisochoides, M.Norman, and D.Gannon, eds., Lecture Notes in Mathematics, Springer-Verlag, Berlin Heidelberg New York, 1999. (In press.).

[11] J.Howe, S.B. Baden, T.Grimmett, and K.Nomura, "Modernization of Legacy Application Software," *Applied Parallel Computing: Large Scale Scientific and Industrial Problem: 4th International Workshop*, B. Kågström, J. Dongarra, E. Elmroth, and J. Wasniewski, eds., Lecture Notes in Computer Science, Vol. 1541, Springer-Verlag, Berlin, Heidelberg New York, 1997, pp. 255–262.

[12] S.Figueira, *Modeling the Effects of Contention on Application Performance in Multi-User Environments*, doctoral dissertation, Univ. of California, San Diego, Dept. Computer Science and Eng., 1997.

[13] F.Berman, R.Wolski, S.Figueira, J.Schopf, and G.Shao, "Application-Level Scheduling on Distributed Heterogeneous Networks," *Proc. Supercomputing '96*, IEEE Computer Soc. Press, 1996, also available

at http://www.supercomp.org/sc96/proceedings/SC96PROC/BERMAN/INDEX.HTM (Sept 1999).

[14] R.Wolski, G.Shao, and F.Berman, "Predicting the Cost of Redistribution in Scheduling," *Proc. Eighth SIAM Conf. Parallel Processing Sci. Computing,* SIAM, 1997. (Proceedings on CD ROM.)

[15] A.Sohn and R.Biswas, "Communication Studies of DMP and SMP Machines," Tech. Report NAS-97-004, NASA Ames Res. Ctr., Mar. 1997.

[16] W.Gropp, E.Lusk, N.Doss, and A.Skjellum, "A High-Performance, Portable Implementation of the MPI Message Passing Interface Standard," *Parallel Computing,* Vol. 22, No. 6, Sept. 1996, pp. 789–828.

[17] S.J. Fink, S.B. Baden, and S.R. Kohn, "Efficient Run-time Support for Irregular Block-Structured Applications," *J. Parallel Distrib. Comput.*, Vol. 50, Apr.-May 1998, pp. 61–62.

[18] S.J. Fink and S.B. Baden, "Run-time Data Distribution for Block-Structured Applications on Distributed Memory Computers," *Proc. Seventh SIAM Conf. on Parallel Processing for Scientific Computing*, SIAM, Feb. 1995, pp. 762–767.

[19] G.Agrawal, A.Sussman, and J.Saltz, "An Integrated Runtime and Compile-Time Approach for Parallelizing Structured and Block Structured Applications," *IEEE Transactions on Parallel and Distributed Systems*, Vol. 6, No. 7, Jul. 1995, pp. 747–754.

[20] A.C. Sawdey, M.T. O'Keefe, and W.B. Jones, "A General Programming Model for Developing Scalable Ocean Circulation Applications," *Proc. ECMWF Workshop on the Use of Parallel Processors in Meteorology*, Jan. 1997.

[21] G.Kiczales, J.Lamping, A.Mendhekar, C.Maeda, C.Lopes, J.-M. Longtier, and J.Irwin, *Aspect-Oriented Programming,* Tech. Report SPL97-008 P9710042, Xerox PARC, Palo Alto, CA, Feb. 1997.

[22] High Performance Fortran Forum, *High Performance Fortran Language Specification, Version 2.0*, Jan. 1997, http://dacnet.rice.edu/Depts/CRPC/HPFF/versions/index.cfm (available September 1999).

[23] S.J. Fink, *KeLP Reference Manual v2.0,* Dept. Computer Science and Eng., Univ. of California, San Diego, Jun. 1998.

[24] L.Snyder, "Foundations of Practical Parallel Programming Languages," *Portability and Performance of Parallel Processing*, T.Hey and J.Ferrante, eds., John Wiley and Sons, 1993.

[25] B.Alpern, L.Carter, and J.Ferrante, "Modeling Parallel Computers as Memory Hierarchies," *Programming Models for Massively Parallel Computers,* W.K. Giloi, S.Jahnichen, and B.D. Shriver, eds., IEEE Computer Soc. Press, Sept. 1993, pp. 116–123.

[26] D.Bailey, E.Barszcz, J.Barton, D.Browning, R.Carter, L.Dagum, R.Fatoohi, S.Fineberg, P.Frederickson, T.Lasinski, R.Schreiber, H.Simon, V.Venkatakrishnan, and S.Weeratunga, *The NAS Parallel Benchmarks,* Tech. Report RNR-94-007, NASA Ames Res. Ctr., Mar. 1994.

[27] W.L. Briggs, *A Multigrid Tutorial.* SIAM, 1987.

[28] R.C. Agarwal, F.G. Gustavson, and M.Zubair, "An Efficient Parallel Algorithm for the 3-D FFT NAS Parallel Benchmark," *Proc. SHPCC '94*, IEEE Computer Soc., May 1994, pp. 129–133.

[29] R.van de Geign and J.Watts, "SUMMA: Scalable Universal Matrix Multiplication Algorithm," *Concurrency: Practice and Experience*, Vol. 9, Apr. 1997, pp. 255–274.

[30] J.Choi, J.J. Dongarra, L.S. Ostrouchov, A.P. Petitet, D.W. Walker, and R.C. Whaley, "Design and Implementation of the ScaLAPACK LU, QR, and Cholesky Factorization Routines," *Scientific Programming*, Vol.5, Fall 1996, pp. 173–184.

[31] K.M.Chandy and C. Kesselman. "Compositional C++: Compositional Parallel Programming," *Languages and Compilers for Parallel Computing, Fifth Intl. Workshop Proc.*, U.Banerjee, D.Gelernter, A.Nicolau, and D.Padua eds., Lecture Notes in Computer Science, Vol. 757, Springer-Verlag, Berlin Heidelberg New York, 1992, pp. 124–144.

[32] S.Pakin, V.Karamcheti, and A.A. Chien, "Fast Messages: Efficient Portable Communication for Workstation Clusters and MPPs," *IEEE Concurrency*, Vol. 5, No. 2, 1997, pp. 60–72.

[33] J.Choi, A.Cleary , J.Demmel, I.Dhillon , J.Dongarra, S.Hammarling, G.Henry, S.Ostrouchov, A.Petitet, K.Stanley, D. Walker, and R.C. Whaley, "ScaLAPACK: A Portable Linear Algebra Library for Distributed Memory Computers - Design Issues and Performance." *Proc. Supercomputing '96,* IEEE Computer Soc. Press, 1996, also available at http://www.sc98.org/sc96/proceedings/SC96PROC/DONGARRA/INDEX.HTM (Sept. 1999.)

[34] H.Casanova and J.Dongarra, "NetSolve: a Network Enabled Server for Solving Computational Science Problems," *Int. Journal of Supercomputer Applications and High Performance Computing*, Vol. 11, Fall 1997, pp. 212–223.

[35] R.Eigenmann, J.Hoeflinger, G.Jaxson, and D.Padua, "Cedar Fortran and its Compiler," *CONPAR 90-VAPP IV, Joint Int. Conf. on Vector and Parallel Processing*, 1990, pp. 288–299.

[36] S.Murer, J.Feldman, C.-C. Lim, and M.-M. Seidel, *pSather: Layered Extensions to an Object-Oriented Language for Efficient Parallel Computation,* Tech. Report TR-93-028, Computer Science Division, U.C. Berkeley, Dec. 1993.

[37] D.A. Bader and J.JáJá, *SIMPLE: A Methodology for Programming High Performance Algorithms on Clusters of Symmetric Multiprocessors,* Tech. Report CS-TR-3798, UMIACS-TR-97-48, Inst. for Adv. Computer Studies, Univ. of Md., College Park, 1997.

[38] J.H. Merlin, S.B. Baden, S.J. Fink, and B.M. Chapman, "Multiple data parallelism with HPF and KeLP," *Proc. HPCN '98*, Apr. 1998.

[39] G.E. Blelloch, S.Chatterjee, J.C. Hardwick, J.Sipelstein, and M.Zagha, "Implementation of a Portable Nested Data-Parallel Language," *Fourth ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming*, ACM, Jul. 1993, pp. 102–111.

[40] A.S. Grimshaw, J.B. Weissman, and T.Strayer, *Portable Run-Time Support for Dynamic Object-Oriented Parallel Processing,* Tech. Report CS-93-40, Univ. Virginia, Dept. Computer Science, Jul. 1993.

[41] R.D. Blumofe, C.F. Joerg, B.C. Kuszmaul, C.E. Leiserson, K.H. Randall, and Y.Zhou, "Cilk: An Efficient Multithreaded Runtime System," *Proc. Fifth ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming*, ACM SIGPLAN, Jul. 1995, pp. 207–216.

[42] I.T. Foster and K.M. Chandy, "Fortran M: A Language for Modular Parallel Programming." *J. Parallel and Dist. Computing*, Vol. 5, No. 1, 1995.

[43] D.Grunwald and S.Vajracharya, *The DUDE Runtime System: An Object-Oriented Macro-Dataflow Approach to Integrated Task and Object Parallelism,* Tech. Report CU-CS-779-95, Dept. Computer Science, Univ. of Colorado, 1994.

[44] P.E. Crandall, E.V. Sumithasri, J.Leichtl, and M.A. Clement, *A Taxonomy for Dual-Level Parallelism in Cluster Computing*, Technical Report, Univ. Connecticut, Mansfield, Dept. Computer Science and Engineering, 1998.

[45] A.K. Somani and A.M. Sansano, *Minimizing Overhead in Parallel Algorithms through Overlapping Communication/Computation,* Tech. Report 97-8, NASA ICASE, Langley, VA., Feb. 1997.

[46] A.Erlichson, N.Nuckolls, G.Chasson, and J.Hennessy, "SoftFLASH: Analyzing the performance of Clustered Distributed Virtual Shared Memory," *Proc. Seventh Int. Conf. Architectural Support for Programming Languages and Operating Systems*, ACM, 1996, pp. 210–220.

[47] S.R. Kohn, *A Parallel Software Infrastructure for Dynamic Block-Irregular Scientific Calculations*, doctoral dissertation, Dept. Computer Science and Eng., Univ. of California at San Diego, La Jolla, CA, 1995.

[48] S.Kohn, J.Weare, M.E. Ong, and S.B. Baden, "Parallel Adaptive Mesh Refinement for Electronic Structure Calculations", *Proc. SIAM Conf. Parallel Processing for Scientific Computing,* SIAM, Mar. 1997. (Proceedings on CD ROM).

[49] F.Berman and R.Wolski, "Scheduling from the Perspective of the Application," *Proc. Fifth IEEE Int. Symp. on High Performance Distributed Computing*, IEEE Computer Soc., Aug. 1996, pp. 100–111.

[50] J.Saltz, A.Sussman, S.Graham, J.Demmel, S.Baden, and J.Dongarra, "Programming Tools and Environments," *Comm. ACM*, Vol. 41, Nov. 1998, pp. 64–73.

Affiliation of Authors

Scott B. Baden is an Associate Professor in the Department of Computer Science and Engineering, University of California, San Diego, in La Jolla, California. **Stephen J. Fink** is currently a Research Staff Member at the IBM T. J. Watson Research Center in Hawthorne, New York.

Biographies of Authors

Scott B. Baden is an Associate Professor of Computer Science and Engineering at the University of California, San Diego and is also a Senior Fellow at the San Diego Supercomputer Center. He received the B.S. degree (*magna cum laude*) in electrical engineering from Duke University in 1978, and the M.S. and Ph.D. degrees in computer science from the University of California, Berkeley, in 1982 and 1987, respectively. He was a post-doc in the Mathematics Group at the University of California's Lawrence Berkeley Laboratory between 1987 and 1990, taking time off to travel. Dr. Baden's current research interests are in the areas of parallel and scientific computation: programming methodology, irregular problems, load balancing, and performance.

Stephen J. Fink is currently a Research Staff Member at the IBM T. J. Watson Research Center in Hawthorne, NY. He received the B.S. degree in Computer Science from Duke University in 1992 with an additional major in Mathematics, and the M.S and Ph.D. degree in Computer Science from the University of California, San Diego in 1994 and 1998. His research interests include dynamic compilation and programming language design and implementation for high performance and scientific computation.

```
class MultiMover : public Mover
{
   void add( ... ) { .. }
   void start() { ... }
   void wait() {  ... }
}

FloorPlan F = SetUpFloorPlan(...) ;
XArray2 U(F);

MotionPlan3 dirX, dirY, dirZ;
SetUpMPlans(dirX, dirY, dirZ);

Mover Ex1(U,U,dirX);
Mover Ex2(U,U,dirY);
Mover Ex3(U,U,dirZ);

M = new MultiMover;

// Add the three Movers
M->add(Ex1);     // Ex1 completes before
M->add(Ex2);     // Ex2 which completes
M->add(Ex3);     // before Ex3

M->start();      // Initiate communication

// overlapped computation

// Wait for communication to complete
M->wait();
```

Figure 9: A MultiMover that implements the asynchronous dimension exchange communication algorithm.

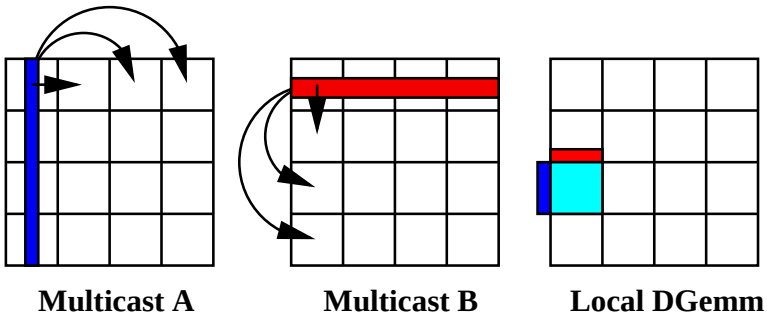


Figure 10: Graphical depiction of one stage of the blocked SUMMA algorithm.