Performance Portable Optimizations for Loops Containing Communication Operations

Costin Iancu
Lawrence Berkeley National Laboratory
cciancu@lbl.gov

Wei Chen
University of California at Berkeley
wychen@cs.berkeley.edu

Katherine Yelick
University of California at Berkeley
yelick@cs.berkeley.edu

ABSTRACT

Effective use of communication networks is critical to the performance and scalability of parallel applications. Partitioned Global Address Space languages like UPC bring the promise of performance and programmer productivity. Studies of well-tuned programs have suggested that PGAS languages are effective at utilizing modern networks because their one-sided communication is a good match to the underlying network hardware. An open question is whether the manual optimizations required to achieve good performance can be performed automatically by the compiler in a performance portable manner.

In this paper we present a compiler and runtime optimization framework for loops containing communication operations. Our framework performs compile time message vectorization and strip-mining and defers until runtime the selection of the actual communication operations. At runtime, the communication requirements of the program are analyzed, and communication is instantiated and scheduled based on highly tuned network and application performance models. The runtime analysis takes into account network flow control and quality-of-service restrictions, and it is able to select from a large class of available communication primitives the communication schedule best suited for the dynamic combination of input size and system parameters. The results indicate that our framework produces code that scales and performs better than that of manually optimized implementations. Our approach not only improves performance, but increases programmer productivity as well.

Categories and Subject Descriptors

C.4 [**Performance of Systems**]: [Design studies, Modeling techniques]; D.2.4 [**Software Engineering**]: Metrics—*Performance measures*; D.1.3 [**Programming Techniques**]: Concurrent Programming—*Parallel programming*; D.3 [**Programming Languages**]: [Parallel, Compilers]; I.6.4 [**Computing Methodologies**]: Simulation and Modeling—*Model Validation and Analysis*

ICS'08, June 7–12, 2008, Island of Kos, Aegean Sea, Greece.

General Terms

Performance, Measurement, Languages, Design

Keywords

Parallel Programming, Program Transformations, Performance Portability, Communication Code Generation, Latency Hiding

1. INTRODUCTION

As high end computing systems continue to scale in CPU computational power and overall node count, optimization techniques that can reduce communication overhead have proven important [7, 10, 29]. Communication optimizations have been explored in the context of parallelizing compilers and data parallel languages. Most of these studies have traditionally been performed using MPI as the communication library and at a time when networks had a relatively high latency and low bandwidth. As a result, most techniques [8, 21] concentrate on eliminating redundant messages and reducing message count through aggregation. Research [4, 12] on recent networks has shown that significant performance improvements can be achieved using fine grained communication decomposition and overlap.

Manual application of communication optimizations affects programmer productivity as these transformations are tedious and error-prone. Multiple code generation schemes are available for the communication in a given loop nest, and the best performance depends on a large set [17] of architecture and application parameters that cannot be estimated statically. Optimizations need to account for flow control network restrictions and the quality of service provided at a given system scale. It is thus difficult for programmers to generate code that can achieve good performance under different application input sets or on different cluster systems. An open research question is whether the manual optimizations required to achieve best performance can be performed automatically by the compiler.

In this paper, we present a loop optimization framework designed to achieve both efficient overlap of communication with computation and performance portability. The framework has been implemented in the Berkeley UPC compiler [9] and uses a combination of compile time and runtime analysis. The resulting infrastructure is capable of performing strip-mining optimizations and to implement a multi-protocol approach for programs that use scatter/gather style operations.

Experimental results indicate that our framework pro-

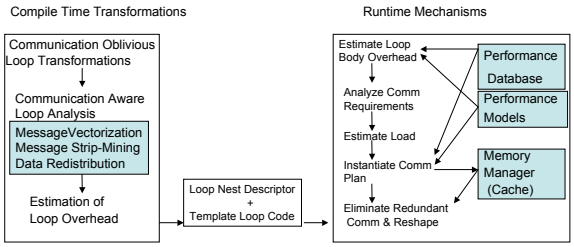

Figure 1: Overall design.

duces code that is faster, more scalable, and exhibits more performance portability than that of manually optimized implementations. We have observed an average speedup of 9.5% over manually optimized implementations for application kernels from the NAS Parallel Benchmarks suite. The average speedup is as high as 17% for some kernels for a large class of configurations evaluated. To our knowledge, this is the first compiler research effort able to exploit non-blocking communication in such a scalable manner. Since our approach is mostly transparent to the user, it not only improves performance and scalability but increases programmer productivity as well.

2. DESIGN

UPC [28] is a parallel extension of the ISO C programming language aimed at supporting high performance scientific applications. The language adopts the Single-Program-Multiple-Data (SPMD) programming model and provides a shared memory space abstraction. Communication in UPC is either explicit through library calls or implicit through shared variable accesses. The Berkeley UPC compiler is implemented using the Open64 [23] infrastructure and generates calls to the GASNet [5] communication layer.

Our goal is to provide performance portable loop optimizations in the presence of communication: for any given loop nest we need to select the best sequence of transformations to provide good serial performance and to hide communication latency. Even for simple cases, communication performance depends on a variety of dynamic factors described in Section 4, and a static compile time approach alone cannot determine the best performing optimizations. This problem is compounded by the fact that for each system, there is a wide set of communication interfaces available for code generation, each with different performance characteristics.

We analyze and transform programs written in a shared memory style, with fine-grained remote array accesses. Figure 1 presents the overall design of our approach. We extend the compiler to perform message vectorization and message strip mining optimizations. At compile time loop nests are analyzed, their communication requirements determined, and the computation overhead estimated. The compiler passes analysis information to the runtime, and performance portability is achieved by decoupling data movement from local computation and using system specific models for communication instantiation. We generate template code that uses the transferred data without making any assumptions about the communication mechanism. At each loop boundary the generated code contains callbacks into a runtime analysis module. Based on actual application and network parameters, the runtime analysis phase selects the most efficient communication operations for a loop nest. For this we use a performance model and heuristics to determine dynamic application characteristics, such as computational and communication load. The communication schedule computed by our analysis takes into account flow control restrictions, network quality of service and application communication topology. For any given scenario, the analysis is able to select dynamically between contiguous communication primitives (*Put, Get*) and scatter/gather primitives implemented using Active Messages.

A major challenge of such approach is designing a light–weight yet efficient code generator for an optimization space with many dimensions. We achieve this by using an expressive program representation and by enforcing as little hardware awareness as possible[1]. Our work makes contributions in the areas of: 1) code generation strategies for loops containing one-sided communication; and 2) runtime mechanisms for performance portability and adaptation at high concurrency across a variety of communication interfaces.

3. PERFORMANCE PORTABLE LOOP OPTIMIZATIONS

Consider the code in Figure 2-(1) that performs a multiplication between a local vector **b** and a remote matrix **a**. In this unoptimized code, the remote accesses in every iteration of the loop add significant overhead. Message vectorization eliminates the overhead of fine-grained transfers by fetching the remote data in a single bulk copy outside the loop nest. The transformed code is presented in Figure 2-(2). The serial code has been blocked for cache and contains a triple-nested loop.

Message vectorization alone usually does not achieve optimal performance because it does not exploit the potential of communication and computation overlap. A more aggressive optimization called *message strip-mining*, shown in Figure 2-(5), can be applied to further reduce the communication overhead. Message strip-mining divides the communication and computation of a loop nest into sub-blocks (strips) and pipelines the communication. In this particular example, strip-mining is performed at the granularity of B elements, imposed by the cache blocking. Compared to the vectorized code, the optimization increases the number of messages and thus the startup overhead, but could hide communication latencies through the overlap of non-blocking transfers with independent computation.

The effectiveness of message strip-mining clearly depends on the strip size; an overly coarse-grained decomposition means insufficient overlap, while an overly fine-grained decomposition may result in excessive message start-up costs. The optimal strip size and communication schedule for an optimized loop are determined by both architectural parameters (e.g., latency and bandwidth) and application characteristics (e.g., data volume, local computation overhead, and communication pattern). For optimal performance, the optimizer needs to determine precisely the communication granularity and schedule for a given setting. For performance portability, the transformed code needs to be able to accommodate different granularities and schedules. The values of most of the important parameters become avail-

[1]We try to minimize the number of hardware related performance parameters incorporated in any model.

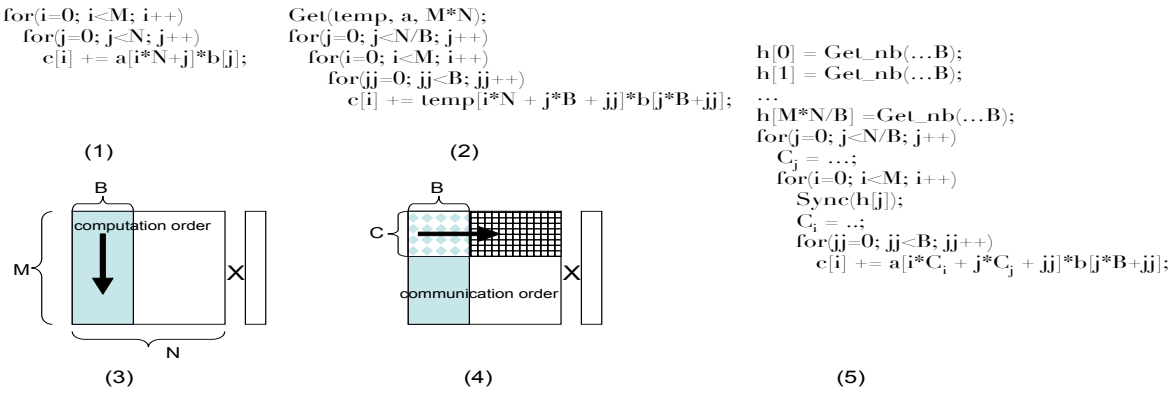

Figure 2: (1) Fine-grained loop; (2) Loop after message vectorization and cache blocking; (3) Memory order for cache blocked loop; (4) Possible memory order for packed communication; (5) Code for strip-mined loop;

able only at runtime and in order to achieve good overlap, dynamic techniques are required.

Another complication is that many applications perform non-contiguous remote sub-array accesses, either due to stri–ded accesses or accesses to a rectangular section of multi-dimen–sional arrays. Examples include transfers of boundary data in finite difference calculations, particle-mesh structures or as a result of tiling optimizations (e.g., Figure 2-(4)). For these disjoint memory regions, the optimizer has to choose between multiple protocols for data communication. For example, one option is to copy each contiguous region individually and use communication pipelining to overlap their latencies, with the alternative to aggregate the non-contiguous regions by performing packing/unpacking inside the runtime. The performance for both approaches again is highly sensitive to the network and application parameters. Several previous studies [14, 19, 27] have examined the performance of multi-protocol scatter/gather operations, but fall short of providing a good methodology for choosing the right implementation for a given application, especially for systems with wide SMP nodes.

4. NETWORK PERFORMANCE

For efficient optimizations, an understanding of the variation of performance parameters across both small and large time scales is required. In [17], we examine the variation of the LogGP [1, 11] network performance parameters for one-sided *Put/Get* primitives on the InfiniBand and Elan networks. The model parameters are o, the send overhead of a message; L, round-trip network latency; and G, the inverse network bandwidth. A brief summary of these findings follows.

On each network, the overhead of initiating non-blocking communication is minimized for a certain number of consecutive operations. The overhead of initiating communication operations is payload dependent. Network resource constraints matter and application level optimizations should work their way around such constraints. Network Interface Cards (NIC) have a limit on the number of outstanding communication operations allowed and implement flow control mechanisms that can affect the amount of effective overlap achieved. The fairness of the bandwidth allocation provided by the network varies with system scale and load. For communication patterns that require full bisection, there is a large difference in the bandwidth observed by communicating processor pairs and end-to-end application performance is directly determined by this bandwidth repartition.

In our implementation, we achieve performance portability for strip-mining optimizations by using some of the models presented in [17], which we extend with runtime techniques for instantaneous system load estimation. Our runtime mechanisms take into account the variation of communication initiation overhead with payload and number of back-to-back messages: $o = o(b, S)$, where S is payload and b is the number of consecutive messages. The fairness of bandwidth allocation at scale is captured by the bandwidth parameters $G^h(P, S)$ and $G^l(P, S)$: the upper and lower bound of the service level achieved at each degree of concurrency (P is the number of nodes). For efficient strip-mining, the models require an estimate of loop overhead.

The networking layer used for the Berkeley UPC compiler also provides scatter-gather style communication primitives [6] which are collectively referred to as Vector-Index-Strided (VIS) functions. VIS calls transfer efficiently multiple non-contiguous memory regions and are implemented by GASNet using Active Messages (AM) and data packing. Our runtime communication analysis is able to combine disjoint *Put/Get* transfers into VIS calls. The GASNet VIS implementation packs any non-contiguous data items into contiguous internal buffers and overlaps the packing with the data transmission. At the remote end, received data is unpacked and copied into its final destination. The length of the internal buffers, referred to as $AMSize$, can be selected at GASNet installation time and has a default value of 1500 bytes. All of our site installations use this default value.

For our runtime mechanisms we develop novel heuristics to choose dynamically between instantiating communication using pipelined *Put/Get* operations and VIS calls, based on the problem settings. A distinguishing characteristic of our approach is the ability to handle systems with wide SMP nodes. Multi-protocol scatter/gather operations have been previously studied [14, 19, 27], but we believe that the ex-

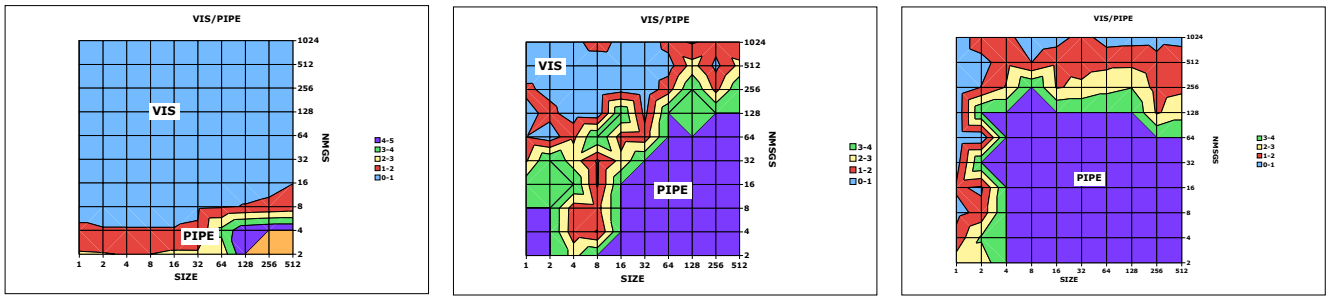

Figure 3: Relative speed of multi-protocol gather on 8 way PowerPC SMP cluster. Color-map represents the ratio T_{VIS}/T_{PIPE}. Values lower than 1 indicate that the VIS implementation is faster. $SIZE$ = length of contiguous region in *doubles*, $NMSGS$ = number of regions. Left: Inter-node, 1 CPU per node active. Center: Inter-node, 8 CPUs active per-node. Right: Intra-node (SMP only), 8CPUs.

isting work falls short on choosing the right implementation on clusters of wide SMPs or systems where noise is present due to limited hardware resources.

Our approach is based on the following assumption. The performance of such implementations depends on a seemingly large parameter space. However, on each parameter axis there are large contiguous regions where the performance of a given implementation behaves in a continuous manner. In those regions, a simple comparison of previously observed performance will indicate the best performing implementation. Whenever noise appears in the system, we assume that the implementation that best avoids it will perform best.

To guide the protocol selection we directly compare the performance of pipelined implementations with the performance of VIS implementations across the parameter space. The parameters we consider are: message length, number of messages, number of active processors and whether communication is intra-node or inter-node. For each setting we record the duration of the slowest operation observed across all endpoints involved. We then build comparative maps across the parameter space.

Figure 3 presents the evolution of the comparative performance of the two implementations for various scenarios on the IBM Power5 cluster described in Table 1. The number of regions is labeled *NMSGS* and the length of a region is labeled *SIZE*. The graph shows the evolution of $\frac{T_{VIS}}{T_{PIPELINE}}$. Similar trends are observed on the InfiniBand and the Cray XT networks.

The sequence of the graphs shows how the effectiveness of the VIS implementations is gradually affected by resource contention within an SMP node. The leftmost graph corresponds to a scenario where only one processor is actively communicating within an eight-way SMP node. The center graph shows the performance when all eight processors within a node are communicating outside the node. The rightmost graph shows the comparative performance when all communication occurs within an SMP node.

We build these performance profiles for an increasing degree of node concurrency and incorporate them directly into the runtime mechanisms described in Section 8. For lack of space we do not show results for higher concurrency. The results indicate that VIS implementations gradually start over-performing pipelined implementations due to less[2] traffic injected into the network.

5. LINEAR MEMORY ACCESS DESCRIPTORS

The runtime system in our framework retrieves the description of the structure of a communication operation from a compile time generated representation: Linear Memory Access Descriptors (LMAD). The LMAD has been introduced by Paek et al [24] as a representation designed to capture precisely the access region of a loop nest and to enable analysis techniques that expose the simple memory footprint of access regions. Their basic observation is that sometimes loop index expressions that cannot be easily analyzed or correlated end up accessing memory in similar patterns. The patterns are characterized by two factors: *stride* and *span*.

The *stride* records the distance traveled in memory when a loop index is incremented. The *span* records the total distance traveled in memory due to the variation of a single index, with the other indices constant. Given a loop nest with induction variables $(i_1, i_2, ..., i_d)$ and an array $\mathcal{A}$, the following representation, called LMAD, captures the entire access pattern for an array reference

$$\mathcal{A}^{\delta_{i_1},\delta_{i_2},...,\delta_{i_d}}_{\sigma_{i_1},\sigma_{i_2},...,\sigma_{i_d}} + \tau$$

where δ_{i_k} and σ_{i_k} are the *stride* and *span* due to index i_k, and τ is the *base offset* of the array and holds the contribution of loop constant terms.

They show how to compute the terms for the LMAD associated with an array reference and define a rich set of operations that can be used to simplify and reason about the memory regions accessed by two LMADs. They distinguish the following types of LMADs: 1) *coalescible* where the number of dimensions within the LMAD can be reduced; 2) *interleaved* which combine address streams with different strides and 3) *contiguous* where the combination of all the

[2]Data packing produces fewer network packets when the length of any contiguous region is lower than the $AMSize$ threshold.

address streams describes a contiguous region. The constraints on the loop induction variables are represented by a *polytope*.

The polytope associated to the nest in Figure 2-1) is the set $\{(0,N),(0,M)\}$. The array reference **a[i*N+j]** is represented by the LMAD $\mathbf{a}^{N,1}_{M*N,N}+0$. For the code in Figure 2-(2) the LMAD describing **temp[i*N+j*B+jj]** is $\mathbf{a}^{B,N,1}_{N,M*N,B}+0$. The two LMADs in this example are equivalent and using the *coalescing* operation they are simplified to $\mathbf{a}^{1}_{M*N}+0$ which captures the total memory footprint of $M*N$. An example of *contiguous* LMADS is encountered in stencil operations, e.g. the references **a[i][j], a[i][j+1]** can be represented by one single LMAD $\mathbf{a}^{1}_{M*(N+1)}$. Interleaved LMADS are generated by references such as **a[i*100+1], a[i*100+2]** which are both covered by $\mathbf{a}^{1,100}_{2,N+2}$. More details about LMAD simplification can be found in [24].

6. COMPILE TIME ANALYSIS AND TRANSFORMATIONS

Open64 contains a rich loop optimization infrastructure, capable of performing unimodular, tiling and cross nest transformations which we have extended to perform communication aware analyses and optimizations. The supported optimizations are message vectorization and message strip-mining. The compile time analysis also extracts information about the serial overhead of loop bodies. We have also written an analysis pass able to recognize data redistribution operations where the target buffers are explicitly provided by the application. In this case data can be transferred directly into its final destination without any intermediate runtime buffering.

During the vectorization process, for each remote reference we extract the LMAD corresponding to the index expression. For each nest that has been vectorized, we generate template code with entries into the runtime analysis layer. The code for each nest contains the loop description, the LMADs that describe communication and place holders for communication synchronization operations. The code resembles the interface for a communication iterator, and Figure 4 shows the code generated for a reduction operation.

The first calls describe the loop bounds and the LMADs involved in communication. Selection of the actual communication primitive is deferred until runtime. The call `analyze_transfers` performs LMAD simplification operations, determining for the communication operations required their granularity and schedule. Communication operations can be instantiated or retired inside any of the calls: `analyze_transfers`, `advance_dim(level)`, `finalize_dim(level)`. Implementation details are presented in Section 7. For multiple loop nests, each nesting level contains calls into the runtime (`advance_dim` and `finalize_dim`) that can be used for communication synchronization. With this abstract description the runtime can dynamically choose the granularity of the communication operations and overlap. Temporary communication buffers are managed by the runtime (`get_local_address`).

To provide potential for communication overlap, we automatically strip-mine single nested loops at compile time, as shown in Figure 4 (`get_strips` call). Since the strip size is not determined until execution time, the runtime analysis is able to determine the cases where this optimization is not

```
ln = start_nest( key);
add_polytope_dim(ln, DEPTH, LB, UB, STRIDE);
br = new_base_ref(ln, ALIAS, element_size;
lmad = new_lmad(ln, br, base_ptr, READ);
add_sos_dim(ln, br, lmad, 0, stride, span);
refvect = analyze_transfers(ln);
if(refvect== 1) {
  lbase = (double *) get_local_address(ln, br, lmad);
  sd = get_strips(ln);
  for(oidx = 0; oidx <= ((N-1 / sd) -1); oidx = oidx + 1) {
     advance_dim(ln, DEPTH);
     for(iidx = 0; iidx <= (sd -1); iidx = iidx + 1) {
      i = iidx + oidx * sd;
     sumv =lbase[i] + sumv;
  }
  }
  //patch-up code
   finalize_dim(ln, 0);
else {
    //fallback code - shared memory version
}
end_nest(ln);
```

Figure 4: Optimized Code

actually profitable and set the number of strips to one. For deeper loop nests, overlap is already present due to the loop structure. However, it might be the case that long innermost loops can benefit from further blocking. Accordingly, we always strip-mine unit strided innermost loops. Loops containing references with non-unit stride are likely to require VIS calls and are not currently blocked. If we find a motivating application, future extensions to this work will consider decomposition and overlap across VIS calls.

Our approach for discovering and exploiting overlap is dynamic. We are aware of only one other compiler effort to exploit overlap for loop nests using one sided communication. This is work performed by Paek and presented in his PhD Thesis. He only considers a static approach where overlap is provided at a compile time static nesting level.

The analysis we have implemented does not handle loops with conditionals and does not perform cross nest optimizations. For the compile time LMAD generation analysis we have not implemented yet the full symbolic simplification described by Paek. Our compile time analysis does not attempt to eliminate redundant communication operations. Such a case is encountered in the CG benchmark described in Section 9 where transferred data is reused twice in subsequent computation. To eliminate redundant communication we provide a runtime data caching module that preserves transferred data according to the UPC language memory consistency model. However, in all applications we have examined, the parts of the code where data is reused are separated by synchronization operations which, according to the memory model, will trigger an invalidation of the runtime caches. This situation is encountered in the CG benchmark. Due to these limitations, application developers using our infrastructure might have to perform manual elimination of redundant communication.

Optimizations are performed only for loop nests where the regions accessed in any complete loop iteration are disjoint. This requirement is captured by LMADS that satisfy the constraint $span(i-1) \leq stride(i) || span(i-1) == 0$, for any dimension i. The first constraint specifies non-overlapping memory regions, the second allows for reuse of a region as encountered in tiled loops. All of the applications examined in this paper exhibit these characteristics.

7. RUNTIME ANALYSIS

The runtime analysis part of our framework is responsible for instantiating the communication operations and determining their granularity and schedule. For each source level array reference, the compile time analysis will generate code to describe the associated LMAD. The goal of the analysis is to determine for each source level reference an equivalent communication LMAD whose memory footprint subsumes the footprint of the original one and it captures all contiguous memory regions. For brevity and clarity, in the rest of this description we use the word *reference* to refer to source level array references. We will use the word LMAD to refer only to the communication descriptors that are internal to the analysis.

At the beginning of the analysis, there exists a one to one correspondence between references and LMADs. Simplification operations will reduce the number of LMADs (if possible) and at the end one communication LMAD might subsume multiple references. In order to illustrate the analysis steps, we will use as example a "stencil" operation.

```
a[100][100], b[100][100];
...
for(j=1; j<99; j++)
  for(i=1; i<99; i++)
     a[i][j] = b[i][j] + b[i+1][j];
```

The first step of the analysis is simplification which starts by ordering the LMADs by their starting address. The analysis starts with the list of references R and the list of LMADs L.

$$\mathrm{R} = \{\ \mathbf{b}_{990,99}^{100,1} + 0, \mathbf{b}_{990,99}^{100,1} + 100\ \},$$

$$\mathrm{L} = \{\ \mathbf{b}_{990,99}^{100,1} + 0, \mathbf{b}_{990,99}^{100,1} + 100\ \}.$$

Afterward, *coalescing* is attempted for each LMAD. During this process, the list of dimension *(stride, span)* descriptors is sorted by stride. In order to keep track of the nesting level at which communication needs to be synchronized, the resulting list has associated with each dimension its position in the original list. The result at the end of this step is

$$\mathrm{R} = \{\ \mathbf{b}_{990,99}^{100,1} + 0, \mathbf{b}_{990,99}^{100,1} + 100\ \},$$

$$\mathrm{L1} = \{\ \mathbf{b}_{99,990}^{1,100} + 0, \mathbf{b}_{99,990}^{1,100} + 100\ \}.$$

In $L1$, the (stride, span) pairs have changed positions when compared to L and will have associated their original nesting level in order ensure proper synchronization. There is still a one to one association between the elements of R and $L1$.

After *coalescing* each LMAD, the analysis combines the LMADs that are contiguous or interleaved. This is done by scanning the LMAD list only once. During this process, whenever two LMADs are combined, the analysis continuously tracks the memory offsets of the participating LMADs and maintains a "memory distance" required for correct communication synchronization. The result after this stage is

$$\mathrm{R} = \{\ \mathbf{b}_{990,99}^{100,1} + 0, \mathbf{b}_{990,99}^{100,1} + 100\ \},$$

$$\mathrm{L2} = \{\ \mathbf{b}_{99,100}^{1,100} + 0\ \}.$$

There is a many-to-one correspondence between references and LMADs and the element of $L2$ has associated the information that for the loop with stride 100 (index i), it needs to fetch 100 elements in advance in case that communication will be generated at the i loop boundary.

At the end of the simplification stage, the runtime has a global view of the loop transfer requirements. Based on the performance models, the runtime determines the instantiation of communication operations, granularity of decomposition and schedule. For LMADs that require the transfer of a single contiguous region, the runtime analysis uses the models presented in [17] to choose the communication granularity and schedule. The initiation and synchronization of communication calls are dynamically associated with the entry points for the loop prologue or epilogue at the appropriate nesting level. For LMADs that require the transfer of multiple disjoint memory regions, the runtime chooses between pipelining Put/Get calls or synthesizes a VIS call using a model based on the performance profiles described in Section 4. The result of this step is a *communication plan*. A communication plan can be viewed as a tuple:

$$\begin{aligned} CP \quad = \quad & (Type, Comm\_Args, N, S, B\_Issue, \\ & Init\_Level, Sync\_Level, B\_Sync) \end{aligned} \tag{1}$$

Inside a communication plan $Type$ can be either a point-to-point operation or a VIS operation, $Comm\_Args$ hold the actual arguments, N is the number of total operations, S is the transfer granularity, B_Issue is the burst length for issuing communication operations. $Init\_Level$ indicates the nest depth where communication should be issued and is associated to the `advance_dim/finalize_dim` calls. $Sync\_Level$ indicates the nest depth where communication is retired and finally B_Sync indicates the number of communication operations retired in one step.

For our example, this step will choose to instantiate all the communication outside the (j) loop and it will apply the heuristics to determine whether it should pipeline 100 messages of 99 doubles each or call directly the VIS library. Our communication generation can be easily extended to accommodate other techniques, given proper guiding heuristics. In our contrived example, one could imagine performing only one "bounding box" transfer for the whole 100x100 domain.

8. PERFORMANCE DATABASE AND HEURISTICS

The performance database to guide the optimization of strip mined loops contains the network specific parameters

$$\{o(b,S), G^l(P,S), G^h(P,S)\}$$

described in Section 4. We use separate models for communication bound loops and computation bound loops as described in [17]. To estimate computation load we use a micro-benchmark to determine the minimum number of unit stride independent memory streams present in a loop that will make it computation bound. Any loop containing less streams than this threshold value is classified as communication bound. Our prototyping shows that this approach works well in practice.

Load estimation is the most important factor in the strip-mining performance models. For each communicating pair

System	Network	CPU type
AMD cluster [18]	InfiniBand 4x	640 x 2.2GHz Opteron
IBM p575 [2]	Federation	888 x 1.9 Ghz POWER5
Cray XT3[3]	Custom	2068 x 2.6 Ghz Opteron

Table 1: Systems Used for Benchmarks

of processors, our estimator computes a communication distance (CD) measure, defined as the distance between the ranks of the endpoints. The heuristic to choose the bandwidth profile based on CD is:

$$
\begin{aligned}
&if(CD < MIN\_THRESH)\\
&\quad choose \quad G^h(P,S)\\
&else \quad if(MIN\_THRESH < CD < \tfrac{P}{2})\\
&\quad choose \quad G^l(\tfrac{P}{2},S)\\
&else\\
&\quad choose \quad G^l(P,S)
\end{aligned}
$$

The value MIN_THRESH for the fat-tree networks we investigated is chosen to be the number of ports of the first level of switches and the heuristic to estimate the communication distance works well in practice for the benchmarks we have examined. For loop nests with a large number of independent remote references a weighted average communication distance might be required but we have not encountered this situation in the applications we have examined.

The performance data used to guide the selection of pipe–lined communication over VIS calls uses only the performance profiles for inter-node and intra-node communication when all the processors within an SMP node are active. In this case we ignore the effects of load (network scale and application communication topology) and use only the profiles determined with two nodes communicating. Our experimental results indicate that this simple strategy performs well in practice.

9. EXPERIMENTS

We validate our compiler and runtime optimization framework using the CG, MG, SP, BT, IS and FT application kernels from the NAS [22] Parallel Benchmarks suite. Table 1 presents the systems used in our experiments. We evaluate three different machines: a mid-size InfiniBand cluster [18], a large IBM p575 POWER5 system [2] and a large scale Cray XT3 system [3]. The InfiniBand and IBM systems are connected in a fat-tree topology, while the Cray has a torus network architecture. The IBM system has 8-way SMP nodes, while the other systems contain 2-way SMP nodes. On the Cray system we report an incomplete set of results and all results on this system have been obtained in runs with only one processor active inside the 2-way SMP nodes. The missing results on the Cray XT3 are due to problems with the native Portals communication library.

We compare the performance of manually optimized versions of the benchmarks with the performance achieved by our compiler and runtime optimizations. All versions are based on the officially released UPC implementation [28] of the NAS benchmarks, which we use as a performance baseline. The selected benchmarks cover a wide range of communication patterns. CG (*get*) and MG (*put*) perform point to point contiguous communication. In CG the granularity of communication for a given call site is fixed by the problem and system size and messages are small to mid-size, ranging from few $Kbytes$ to tens of $Kbytes$. In MG, the communication granularity varies dynamically at each call site and messages range in size from few *bytes* to few $Kbytes$. SP (*put*) and BT (*put* and *get*) issue a large number of messages to a given processor, and the count and granularity of these messages varies with problem and system size. In BT, the contiguous transfers range from few *bytes* to few $Kbytes$. SP exhibits characteristics similar to BT. IS and FT perform all-to-all communication operations with granularity determined by the problem and system size. Both benchmarks perform large message transfers, ranging from hundreds of $Kbytes$ to several $Mbytes$.

Figure 5 presents the performance results obtained on the InfiniBand and the Cray XT3 systems. We report P_{OPT}/P_{BASE}, where P_{BASE} represents the performance of the baseline implementation, which uses manually vectorized blocking communication. We use for the comparison the performance in *operations per second* as reported by the benchmarks. Each benchmark has been run for ten times an each system. Unless noted otherwise, we use the fastest run for the comparison.

For each benchmark we manually modify the implementation to use non-blocking communication for maximal overlap without any other source modifications. In particular, we do not perform manual strip-mining. Where multiple implementations are available, we always report the performance for the best one under the bars labeled HAND. The bars labeled OPT show the performance of the programs compiled within our optimizations framework. For these implementations we replace all bulk communication calls in the original program with shared memory style code. If the target buffer of a communication operation is reused within the program (as in the CG benchmark), we do not perform redundant communication elimination. For the IS and FT benchmarks we had to manually break data dependences and introduce a double-buffering scheme directly in the application. Thus, these two benchmarks execute additional serial work.

The results indicate that the compiler based approach outperforms in most cases the manually optimized implementations and provide performance portability across systems. The performance provided by our approach scales well with system and problem size: best speedups are obtained at high concurrency. The optimization parameters chosen for each instance vary dynamically with the problem setting and system architecture. For example, on the InfiniBand system the average speedup over the manual version is 9.5% across all experiments, the average speedup per benchmark is as high as 17% for some classes, and the maximum[3] speedup observed over all experiments is 27%.

9.1 Strip Mining Performance

The performance of CG, IS, and FT benefits from strip-mining and depends on the estimation of computation and system load. For brevity, we present full only results for the InfiniBand system and discuss the trends observed on the XT3 system.

Across all platforms, the performance improvements for the CG benchmark are modest. There are several reasons for this behavior. The compiler optimized version of CG performs redundant communication operations and the mes-

[3]For IS and FT at very high concurrencies we obtain speedups of 80%. This behavior is caused by un-tuned all-to-all implementations. We have discarded these results in computing the averages.

sages exchanged have short to medium size for all problem instances. The benefits of strip mining are less pronounced for short messages and the benchmark itself does not spend a significant amount of time performing communication operations. The benefits of our optimizations are most pronounced on the InfiniBand system. This system has lower communication initiation overhead and lower relative bandwidth than both the Cray and the IBM systems. When compared to manual optimizations, compiler techniques improve performance by additional 3% across the CG workload, with a maximum improvement of 12%.

The FT and IS benchmarks capture the network response under congestion. On the InfiniBand system, compiler optimizations improve performance by 12% for the FT workload and by 18% for the IS workload. On all systems, the higher the concurrency, the greater the impact of our optimizations. These benchmarks also illustrate the benefits of our load estimation technique for strip-mining optimizations. In Figure 5, the bars labeled FT-NLE show how the benefits of strip-mining are diminished at higher concurrency (32, 64) when load estimation is not performed. In this case we use in the model the bandwidth reported by a two node experiment. This choice leads to a finer grained decomposition and a larger number of independent transfers. At high concurrency, the latency of transfers is adversely affected by congestion and for example, with 64 processors the FT benchmark which is 40% faster than the baseline in the optimal case, becomes 2% slower than the baseline.

The torus network in the XT3 system responds differently to congestion than the fat-tree networks. On this system, the performance trends for FT and IS at lower concurrencies are similar to trends observed on the InfiniBand system. At high concurrencies, the performance of optimized implementations either becomes noisy or degrades drastically. The application of our techniques improves considerably the performance of the non-blocking implementations, albeit the improvements over the baseline blocking implementation are more modest. While our techniques definitely improve performance on this system, further tuning of the load estimation heuristics might help significantly.

9.2 Vector (VIS) Operations

The performance of MG, SP, and BT depends on the dynamic selection of pipelined Put/Get operations instead of proper VIS calls and our techniques clearly improve the performance of these kernels. Figure 6 shows the impact of our techniques. Note that the results in Figure 5 correspond to statically choosing the VIS calls.

The behavior of BT and SP benchmarks at different scales and across systems illustrates well the pitfalls of achieving performance portability when using manual optimization techniques. These benchmarks contain several communication stages that are expressed at source level as multiple independent loop nests containing conditionals. Our compiler optimizations analyze only the innermost loops that do not contain conditionals and the runtime analysis does not attempt cross nest optimizations.

The results in Figure 6 report performance normalized to that of a manual implementation that optimizes one nest with conditionals at a time. Thus, this implementation provides significantly more overlap than the compiler optimized code. The PIPE-GLOB and VIS-GLOB implementations schedule communication across multiple nests and these implementations exploit all of the overlap available in the benchmarks.

On the InfiniBand system, the overall performance of compiler generated code is superior to that of any manually attempted optimization. On this system, global communication scheduling of pipelined implementations does not seem to improve performance. Global scheduling of VIS implementations improves performance to a lesser degree than our automated approach.

On the IBM system. which has wide SMP nodes, the performance results paint a less clear picture. Different manual implementations perform better depending on the benchmark setting. The impact of our optimization techniques is best illustrated by the performance of the MG benchmark which does not provide opportunities for global communication scheduling. For this benchmark our approach clearly outperforms manual optimization. For the SP and BT benchmarks, the overall performance of our approach is better than the performance of any manual optimization that performs local scheduling but less than the performance of global scheduling. Manual optimizations labeled as "local" also exploit more overlap than compiler based optimizations.

Experimental results on the wide-SMP IBM cluster indicate that instantaneous load on the node network interface is a deciding performance factor. We are currently working on refining our intra-node heuristics for dynamic VIS selection. Preliminary results are very promising and indicate that with a good intra-node load estimator the compiler based approach outperforms manually optimized implementations that are not load aware. We believe that given the current trend of increasing number of cores within a multicore processor, such techniques will become important for achieving good communication performance.

9.3 Analysis Overhead

On all systems, the overhead of the dynamic runtime analysis is very small and for all benchmarks it amounts to a small fraction of a percent of the total running time. The benchmarks run for tens to thousands seconds and the analysis accounts for tens to thousands of milliseconds. We split the analysis overhead in the time to describe the problem and the time to analyze the problem. For all systems and problems we observe average overheads of $5\mu s$ for the problem description and $3\mu s$ for the analysis and communication plan generation. For reference, the round trip latency of the InfiniBand system is 14 μs and the round-trip latency of the Elan4 system is 9 μs.

10. RELATED WORK

Communication optimizations for parallel programs have been studied extensively in the context of data parallel languages [15, 16, 20, 25, 26]. Initial efforts focused on array optimizations and performed message vectorization and coalescing on a loop nest basis. More recent efforts focus on global program analysis. Chakrabarti et al. [7] present a global algorithm for communication analysis and placement implemented in the IBM pHPF compiler using control flow graph analysis. Kandemir et al. [20, 21] present data flow techniques for global communication scheduling for HPF programs, assuming either MPI communication or one-sided communication. They perform message vectorization, message coalescing and redundancy elimination across multiple loop nests. One common characteristic of these

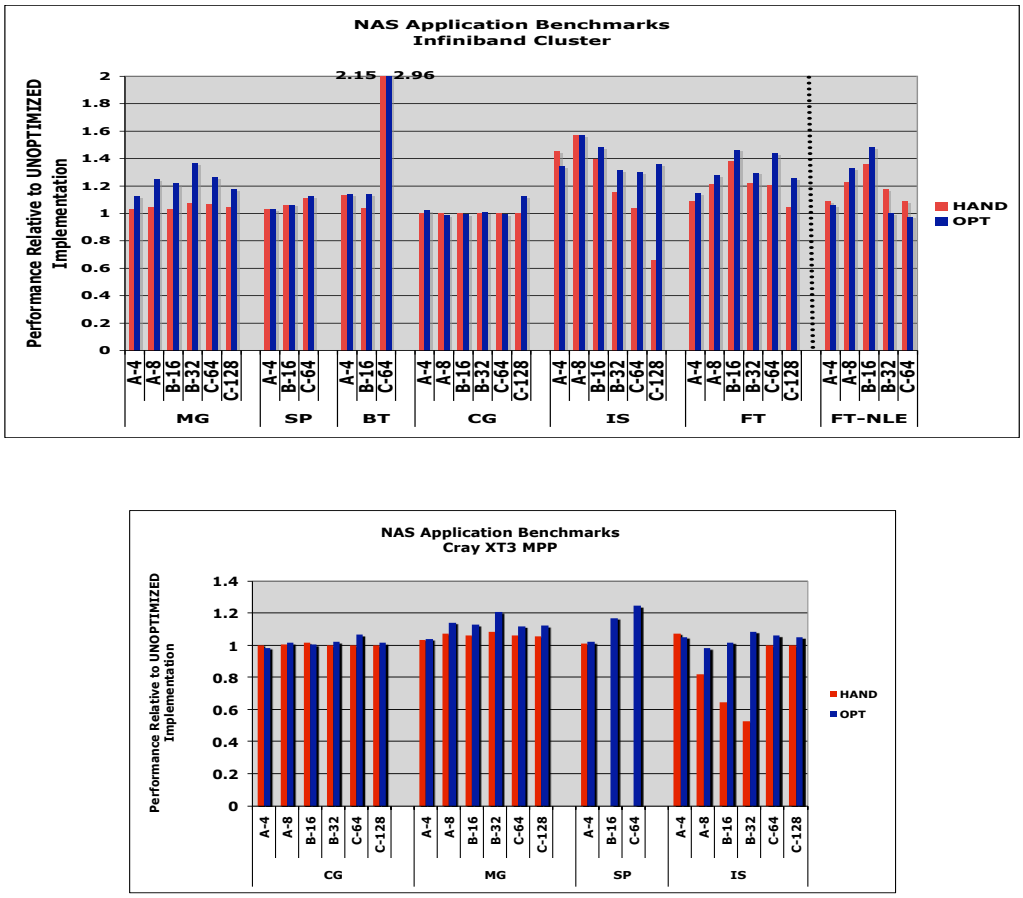

Figure 5: Results for NAS application kernels. Class A on 4 and 8 processors, Class B on 16 and 32 processors and Class C on 64 and 128 processors. Performance (Mops) is reported as relative to that of the officially released version. HAND refers to the best performing manually optimized implementation. OPT refers to the performance of compiler optimized code.

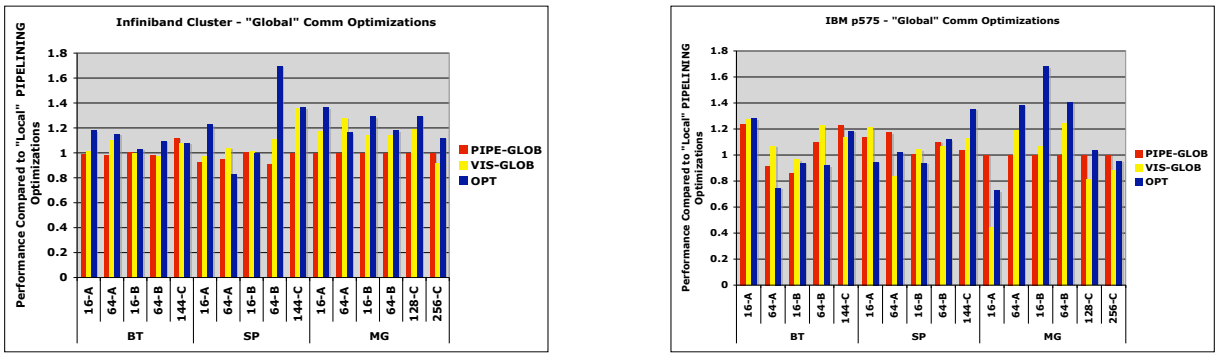

Figure 6: Performance of selected benchmarks using "global" optimization techniques compared to the performance of more "local" optimizations. PIPE-GLOB refers to global communication scheduling using pipelined communication only. VIS-GLOB refers to global scheduling of VIS operations, OPT refers to the performance of compiler generated code.

efforts is that they focus on minimizing the number and volume of communication operations. These techniques worked well at the time due to the high latency and the low bandwidth of the networks. For contemporary networks, studies have shown that a finer-grained interleaving of communication and computation operations is able to provide better overlap. Furthermore, until very recently, the performance of MPI implementations was relatively system agnostic, and previous research did not target performance portability.

Communication optimizations have also been studied in the context of parallelizing compilers. Of these efforts, most relevant to our work is the approach taken in the Polaris compiler. Paek et al. [24] introduce the Linear Memory Access Descriptor (LMAD), which is used to efficiently represent generic array regions. In particular, Paek discusses code generation for loops containing communication operations using a one-sided communication model. He presents simple heuristics for placement of communication operations in order to optimize memory consumption and achieve communication pipelining. His approach is static and does not discuss how to achieve optimal overlap.

The existing compilers for PGAS languages perform various communication optimizations. The Co-Array Fortran compiler [13] supports message vectorization but does not perform code generation using higher level data packing and unpacking primitives. It also does not perform strip-mining or other compile time optimizations to exploit non-blocking communication. The authors report very noisy performance results when manually using VIS primitives in applications. The Titanium compiler and runtime provide array copy libraries that can select either contiguous or data packing calls. This selection is static and the compiler does not perform communication and computation overlap transformations. Su and Yelick [27] describe a compile and runtime approach for inspector-executor based programs. They provide models for data packing latency estimation and selection of communication strategies but they have not validated their approach on wide SMP systems.

11. CONCLUSION

Effective use of communication networks is critical to the performance and scalability of parallel applications. Partitioned Global Address Space languages have proven effective at utilizing modern networks because their one-sided communication is a good match to underlying network hardware. These languages also provide the means to leverage communication overlap for latency hiding, however the use of split-phase communication operations has primarily been applied manually by programmers.

In this paper we have presented a compiler and runtime optimization framework for loops containing communication operations. Our framework performs compile time message vectorization and strip-mining, and defers until runtime the instantiation of the actual communication operations. At runtime, the communication requirements of the program are analyzed, and communication is instantiated and scheduled based on highly tuned network and application performance models. The runtime analysis is able to select from a large class of available communication interfaces the interface and communication schedule best suited for the dynamic combination of input size and system load. The results indicate that our framework produces code that is better performing and more scalable than manually optimized implementations. The dynamic optimization approach used in our system increases both programmer productivity and performance portability.

We believe that our results are of interest to application developers as well as communication library implementors and language designers. Compiler assisted techniques might be able to produce well performing implementations of some classes of collective communication calls such as reductions and ALLTOALLs, thereby either reducing the implementation effort or completely eliminating the need for such primitives in communication libraries. Implementors of autotuning parallel libraries could use our optimization approach and decouple the serial optimizations from the communication optimizations.

12. REFERENCES

[1] A. Alexandrov, M. F. Ionescu, K. E. Schauser, and C. Scheiman. LogGP: Incorporating Long Messages into the LogP Model for Parallel Computation. *Journal of Parallel and Distributed Computing*, 44(1):71–79, 1997.

[2] Bassi IBM p575 POWER5. LBNL National Energy Research Supercomputing Center.

[3] Bigben Cray XT3 MPP. Pittsburgh Supercomputing Center.

[4] C. Bell, D. Bonachea, R. Nishtala, and K. Yelick. Optimizing Bandwidth Limited Problems Using One-Sided Communication and Overlap. *Proceedings of the 20th International Parallel and Distributed Processing Symposium (IPDPS)*, 2006.

[5] D. Bonachea. GASNet Specification, v1.1. Technical Report CSD-02-1207, University of California at Berkeley, October 2002.

[6] D. Bonachea. Proposal for Extending the UPC Memory Copy Library Functions and Supporting Extensions to GASNet, v1.0. Technical Report LBNL-56495, Lawrence Berkeley National Laboratory, 2004.

[7] S. Chakrabarti, M. Gupta, and J.-D. Choi. Global Communication Analysis and Optimization. In *Proceedings of the SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 68–78, 1996.

[8] D. Chavarria-Miranda and J. Mellor-Crummey. Effective Communication Coalescing for Data-Parallel Applications. In *Proceedings of the 10th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, pages 14–25, New York, NY, USA, 2005. ACM Press.

[9] W. Chen, D. Bonachea, J. Duell, P. Husbands, C. Iancu, and K. Yelick. A Performance Analysis of the Berkeley UPC Compiler. In *Proceedings of the 17th International Conference on Supercomputing (ICS)*, June 2003.

[10] S.-E. Choi and L. Snyder. Quantifying the Effects of Communication Optimizations. In *Proceedings of the international Conference on Parallel Processing (ICPP)*, pages 218–222, Washington, DC, USA, 1997. IEEE Computer Society.

[11] D. E. Culler, R. M. Karp, D. A. Patterson, A. Sahay, K. E. Schauser, E. Santos, R. Subramonian, and T. von Eicken. LogP: Towards a Realistic Model of Parallel Computation. In *Proceedings of the ACM Symposium on the Principles and Practice of Parallel Programming (PPoPP)*, pages 1–12, 1993.

[12] A. Danalis, K.-Y. Kim, L. Pollock, and M. Swany. Transformations to Parallel Codes for Communication-Computation Overlap. In *Proceedings of the 2005 ACM/IEEE Conference on Supercomputing (SC05)*, page 58, 2005.

[13] Y. Dotsenko, C. Coarfa, and J. Mellor-Crummey. A Multiplatform Co-array Fortran Compiler. In *Proceedings of the IEEE Parallel Architecture and Compilation Techniques Conference (PACT)*, Antibes Juan-les-Pins, France, 2004.

[14] Gopalakrishnan Santhanaraman and Dhabaleswar Wu and Dhabaleswar K. Panda. Zero-Copy MPI Derived Datatype Communication over InfiniBand. In *Recent Advances in Parallel Virtual Machine and Message Passing Interface, 11th European PVM/MPI Users' Group Meeting*, 2004.

[15] M. Gupta, S. Midkiff, and E. S. et al. A HPF compiler for the IBM SP2. In *Supercomputing 1995*, November 1995.

[16] M. Gupta, E. Schonberg, and H. Srinivasan. A Unified Framework for Optimizing Communication in Data-Parallel Programs. *IEEE Transactions on Parallel and Distributed Systems*, July 1996.

[17] C. Iancu and E. Strohmaier. Optimizing Communication Overlap for High-Speed Networks. *Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, 2007.

[18] Jacquard AMD Opteron cluster. LBNL National Energy Research Supercomputing Center.

[19] Jarek Nieplocha and Vinod Tipparaju and Manoj Krishnan. Optimizing Strided Remote Memory Access Operations on the Quadrics QsNetII Network Interconnect. In *HPCASIA '05: Proceedings of the Eighth International Conference on High-Performance Computing in Asia-Pacific Region*, page 28, Washington, DC, USA, 2005. IEEE Computer Society.

[20] M. Kandemir, P. Banerjee, A. Choudhary, J. Ramanujam, and N. Shenoy. A Global Communication Optimization Technique Based on Data-Flow Analysis and Linear Algebra. *ACM Transactions on Programming Languages and Systems*, 21(6):1251–1297, 1999.

[21] M. T. Kandemir, A. N. Choudhary, P. Banerjee, J. Ramanujam, and N. Shenoy. Minimizing Data and Synchronization Costs in One-Way Communication. *IEEE Transactions on Parallel and Distributed Systems*, 11(12):1232–1251, 2000.

[22] The NAS Parallel Benchmarks. Available at http://www.nas.nasa.gov/Software/NPB.

[23] Open64 compiler tools. http://open64.sourceforge.net.

[24] Y. Paek, J. Hoeflinger, and D. Padua. Efficient and Precise Array Access Analysis. *ACM Trans. Program. Lang. Syst.*, 24(1):65–109, 2002.

[25] Seema Hiranandani and Ken Kennedy and Chau-Wen Tseng. Compiling Fortran D for MIMD Distributed-Memory Machines. *Communications of the ACM*, 35(8):66–80, 1992.

[26] E. Su, A. Lain, S. Ramaswamy, D. J. Palermo, E. W. Hodges, and P. Banerjee. Advanced Compilation Techniques in the PARADIGM Compiler for Distributed-Memory Multicomputers. In *Proceedings of the 9th ACM International Conference on Supercomputing (ICS)*, pages 424–433, July 1995.

[27] J. Su and K. Yelick. Automatic Support for Irregular Computations in a High-Level Language. In *Proceedings of the 19th IEEE International Parallel and Distributed Processing Symposium (IPDPS'05)*, 2005.

[28] UPC Language Specification, Version 1.0. Available at http://upc.gwu.edu.

[29] Y. Zhu and L. J. Hendren. Communication Optimizations for Parallel C Programs. *Journal of Parallel and Distributed Computing*, 58(2):301–332, 1999.