Performance Portable Optimizations for Loops Containing Communication Operations

Costin Iancu

Lawrence Berkeley National Laboratory
cciancu@lbl.gov

Wei Chen, Katherine Yelick

University of California at Berkeley
{wychen, 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 control flow 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.

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, 36]. 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, 22] concentrate on eliminating redundant messages and reducing message count through aggregation. Research [3, 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 communication code generation schemes are available for a given loop nest, and the best performance depends on a large set [19] of architecture and application parameters that cannot be estimated statically. Optimizations need to account for control flow 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.

Partitioned Global Address Space (PGAS) languages, such as UPC [33], Titanium [35] and Co-Array Fortran [26], have recently emerged as a promising alternative to increasing programmer productivity. They integrate communication directly into the language and provide direct access to lightweight one-sided communication. 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 communication/computation overlap and performance portability. The framework has been implemented in the Berkeley UPC compiler [9] and uses a combination of compile time analysis and runtime mechanisms. 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 control flow 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 gather/scatter primitives implemented using Active Messages. We explore several mechanisms to further increase performance: caching of the communication schedule, caching of transfered data, and data reshaping mechanisms.

Experimental results indicate that our framework produces 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 per-

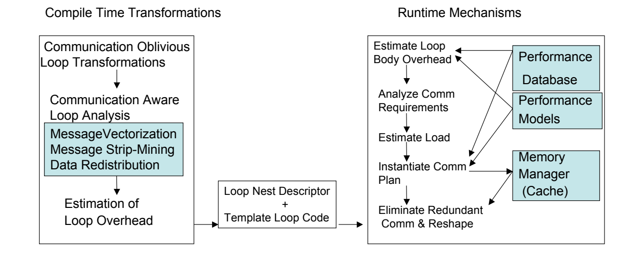

Figure 1. Overall design.

formance and scalability but increases programmer productivity as well.

2. Design

UPC [33] 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 language provides a relaxed memory consistency model [34] that allows for aggressive reordering of communication operations and caching of remote data.

The Berkeley UPC compiler performs source-to-source translation, by converting UPC source code to C code augmented with runtime calls for data management and communication. The compiler is implemented using the Open64 [27] infrastructure. The Berkeley UPC runtime delegates communication operations to the GASNet [5] communication layer, which provides a uniform interface for low-level communication primitives on a large variety of networks. GASNet provides efficient point-to-point *Put/Get* primitives that operate on contiguous data as well as higher level operations that perform aggregation of operations targeting disjoint memory regions.

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 decouple the static serial optimizations from the communication optimizations. After serial optimizations, the communication requirements of a loop nest are analyzed, and code that describes them to the runtime is generated. A runtime analysis module instantiates communication based on performance models and carefully tuned heuristics. A major challenge of such approach is designing a lightweight 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 large variety of communication interfaces.

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

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 offers significant speedup over the fine-grained version by saving on the per-message overhead. It alone usually does not achieve optimal performance because it does not exploit the potentials 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. For this particular example, it might be the case that the optimal strip size contains multiple rows of N elements and does not naturally match the cache blocking.

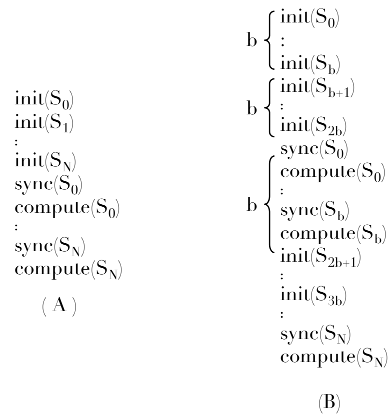

Figure 3. (**A**) Communication schedule for strip-mining; (**B**) Communication schedule with pipelining and strip-mining.

After choosing the decomposition granularity, the optimizer needs to choose a communication schedule. This schedule has to satisfy all the data dependences of the original program, and it indicates the locations in the program where communication operations are initiated and retired. Figure 3 shows two interleavings of communication operations that can be used for the strip-mined transformed code. Depending on the problem settings and the system, any of the two schedules can perform better in practice.

The optimal strip size and communication schedule for a given loop nest are determined by both architectural parameters (e.g., latency and bandwidth) and application characteristics (e.g., data volume, local computation overhead, and communication pattern), and the optimizer could very easily choose an under-performing transformation.

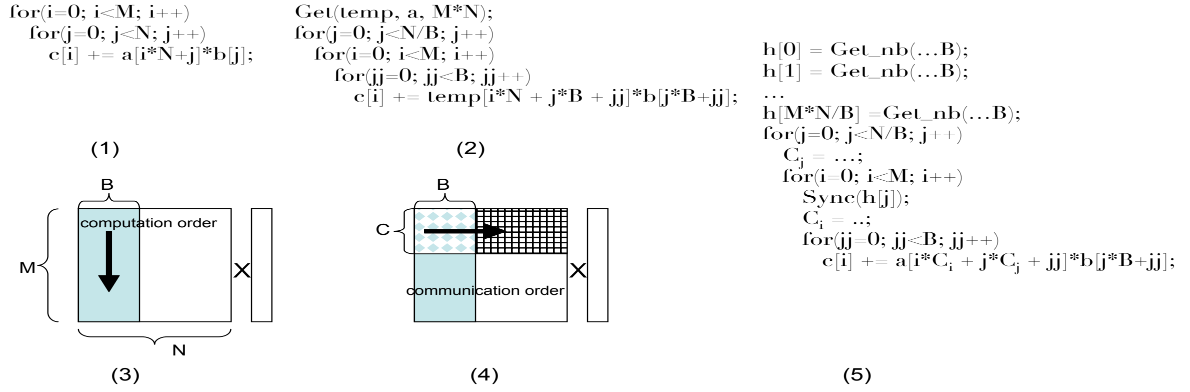

Figure 2. (**1**) Fine-grained loop; (**2**) Loop after message vectorization and cache blocking; (**3**) Memory order for cache blocked loop; (**4**) Possible memory order for packed communication; (**5**) Code for strip-mined loop;

Another complication is that many applications perform non-contiguous remote sub-array accesses, either due to strided accesses or accesses to a rectangular section of multi-dimensional arrays as a result of tiling optimizations (e.g., Figure 2-(4)). For these disjoint memory regions, the optimizer has two choices for data communication. One option is to copy each contiguous region individually and use communication pipelining to overlap their latencies, and the alternative is to perform packing and unpacking in the runtime to aggregate the non-contiguous regions. The performance for both approaches again is highly sensitive to the network and application parameters.

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. For our example, the transformed nest in Figure 2-(5) needs to be able to initiate and retire an arbitrary number of communication operations in the prologue of any of the outer two loops.

4. Network Performance

For efficient optimizations, an understanding of the variation of performance parameters across both small and large time scales is required. In [19], 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.

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. Examples are network control flow and memory footprint of communication operations. Network Interface Cards (NIC) have a limit on the number of outstanding communication operations allowed. Issuing additional operations over this threshold will block the CPU execution until some previous operation has completed. Some network cards (Elan) have an on-board TLB and whenever the footprint of the outstanding communication operations exceeds the TLB footprint, the latency of all operations is adversely affected.

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 by using some of the models presented in [19], 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 the purposes of this work we extend the network performance study with an exploration of overlap when using higher level communication primitives and incorporate these primitives in performance models.

The networking layer used for the Berkeley UPC compiler provides efficient scatter-gather style communication primitives [6] which are collectively referred to as Vector-Index-Strided (VIS) functions. VIS calls transfer efficiently non-contiguous memory regions and are implemented using Active Messages (AM) and data packing. Communication layers that support asynchronous one-sided VIS style operations use different mechanisms to ensure remote end cooperation. The default operation of the GASNet layer is in polling mode. In this setting, asynchronous events, such as AMs, that occur inside the networking layer, are not serviced until the application explicitly enters the communication library. Other communication layers such as ARMCI [25] offload data packing to the NIC, provide progress threads, or support an interrupt mode inside the networking layer.

Efficient implementations of data packing primitives pipeline and overlap data packing with data transmission operations and usually provide a static value for the unit[2] of packing and communication. Depending on the application communication requirements this static decision is not necessarily the most efficient.

The polling mode used by GASNet also raises the concern of application attentiveness to the network: if a particular end-point does not enter the network layer for a period of time, its incoming AMs are unpredictably delayed. Attentiveness to the network directly impacts overlap performance across all networks. This is a problem common to all communication layers that use asynchronous events; incorporating progress is still an open research area [3] and is beyond the scope of this work.

[2] In the GASNet this is referred to as *AMSize*.

[3] X10, one of the proposed DARPA HPCS languages provides asynchronous remote execution.

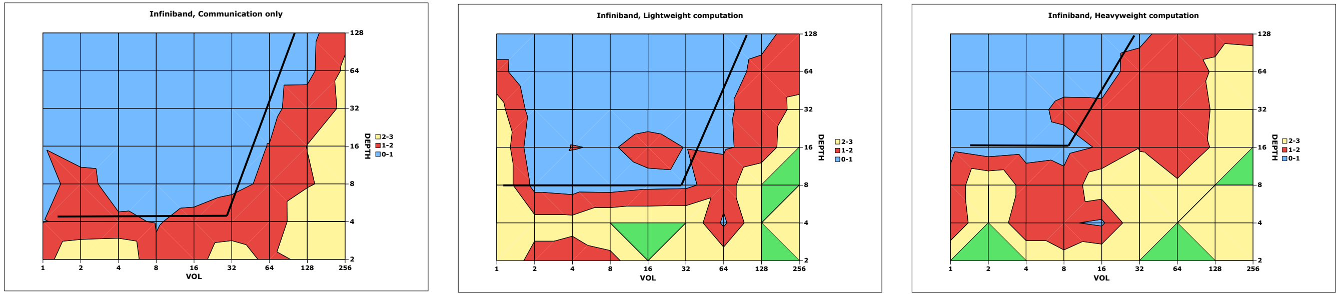

Figure 4. Overlap potential for VIS operations. Color-map represents the speed-up of the VIS implementation when compared with an optimized point-to-point pipelined implementation. Values lower than 1 indicate that the VIS implementation is faster. VOL = length of contiguous region in *doubles*, $DEPTH$ = number of regions. **Left**: communication only. **Center**: overlap with "lightweight" computation. **Right**: overlap with heavyweight computation.

For our runtime mechanisms we develop heuristics to choose dynamically between instantiating communication using pipelined *Put/Get* operations and VIS calls, based on the problem settings. To understand the overlap behavior of VIS operations we compare the performance of a VIS call followed by computation on the transfered data with the performance of the same problem implemented and *optimized* using point to point operations. Our micro-benchmark performs communication and computation on a number of disjoint memory regions. We examine different computation loads since they directly determine network attentiveness. We also consider multiple memory footprints to capture possible TLB effects. In Figure 4 the number of regions is labeled *DEPTH* and the length of a region is labeled *VOL*. The graph shows the evolution of $\frac{T_{VIS}}{T_{PIPELINE}}$.

Figure 4 presents the trends observed for the Infiniband network. We have also examined the Elan4 and Cray XT3 networks. The sequence of the graphs shows how the effectiveness of the VIS implementations is gradually affected by lack of attentiveness. The leftmost graph corresponds directly to a data redistribution operation. The results for Elan networks are not shown for lack of space, but they indicate that the VIS implementation is also very sensitive to the total memory footprint, i.e. the length of a contiguous memory region and the distance between two regions. The default unit for VIS packing and communication in GASNet is *AMSize=1500* bytes. On all networks, the results show that a pipelined implementation will outperform a packing implementation for many cases where the length of the contiguous region is well below this threshold. These cases are network dependent.

We incorporate the performance profiles obtained from our micro-benchmarks directly into the runtime mechanisms described in Section 6.3. Experiences with Co-Array Fortran [14] and ARMCI in application settings, indicate the potential performance advantages of VIS style operations when applied to redistributions and "bulk synchronous" operations. They also report noisy performance results on some targets when these primitives are used.

5. 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. These cases need to be handled separately since the target communication buffers are already provided in the application.

Most of the Open64 loop transformations use convex region representations, and we convert these to Linear Memory Access Descriptors (LMAD) during the vectorization process. Paek [29] introduced the LMAD 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 an access region. The access 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. 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 address streams describes a contiguous region. The constraints on the loop induction variables are represented by a *polytope*.

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 generated code resembles the interface for a communication iterator, and Figure 5 shows the generated code for a reduction operation.

The first calls describe the polytope and the LMADs involved in communication operations. 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)`. 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 choose to instantiate either pattern A or B described in Figure 3. Temporary communication buffers are managed by the runtime (`get_local_address`). The calls `start_nest` and `end_nest` are hooks into a runtime layer that caches the loop descriptions and analysis results.

To provide potential for communication overlap, we automatically strip-mine singly nested loops at compile time, as shown in Figure 5 (`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 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 innermost loops with unit stride. Loops containing references with non-unit stride are likely to require VIS

```
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);
   end_nest(ln);
} else {
   //fallback code - shared memory version
}
```

Figure 5. Optimized Code

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 data indexing sequence into the communication buffers is determined either statically by compile time analysis or dynamically by runtime analysis. For most loops encountered in practice, compile time analysis can determine statically that the initial data memory layout and the application access order will match the layout inside the communication buffers. This corresponds to loops where the LMADs synthesized for communication operations have the general property that $span(i-1) \leq stride(i) || span(i-1) == 0$, for each dimension i. The first clause captures the case where where the regions accessed in any loop iteration are disjoint. This has been the case in all of our benchmark applications. The second clause captures the case where a loop index does not appear in the original index and occurs in practice in tiled loops.

There are cases where assuming a certain layout might prohibit the optimal communication plans or the compile time analysis cannot determine statically the communication buffer indexing sequence. Examples are: 1) tiling of multidimensional arrays, where the runtime might choose to use either VIS or point-to-point calls, and the data layout into the communication buffers cannot be statically predicted; and 2) some *contiguous*[4] LMADs. In these cases we use runtime computed coefficients instead of compile time coefficients. The coefficients C_i and C_j in Figure 2-(5) illustrate this situation. In order to minimize the register pressure we use the same coefficients for all the references within an Uniformly Generated Set[5]. We have not encountered this situation in the applications we have examined. Also, note that this case can be handled by compile time analysis and redundant data transfers.

The runtime analysis is allowed to fail. In this case the code reverts to executing the slow but correct fine-grained version. Failure occurs due to the limitations of the dependency analysis in Open64 and limitations in our runtime analysis and index generation. Paek [29] shows that dependency analysis using the LMAD representation is more powerful than the analysis using convex regions. Failures occur when a loop has read and write sets that are remote, and the compile time analysis cannot determine their intersection and has to be conservative. In these cases, our LMAD based runtime analysis might be able to prove that the two regions are disjoint or simplify their union/intersection into a more manageable form. However there may be cases where we cannot compute the index expression for both references to capture exactly the intersection of the two regions. We do not expect to encounter this case in practice.

One question we had to answer during our implementation was the level of integration between the serial loop optimization framework and the communication optimization framework. High level[6] serial loop transformations are guided by a uniprocessor machine model and oblivious of a UPC loop's communication requirement. These transformations are mostly designed to increase spatial and temporal locality and are guided by machine specific parameters, such as cache size, cache miss latency. Increasing spatial and temporal locality is also beneficial for communication transformations, however, the latter are guided by network specific parameters. Decoupling the two optimization stages might result in non-optimal code.

However, our empirical experience indicates that the runtime communication mechanisms we provide are good enough to overcome any possible performance penalties. For cache blocking transformations, even for L1 caches[7], the block granularities chosen match well enough the optimal communication granularity. It is either the case that the cache blocks are large enough or the latency of the loop body is high and the loop is computation bound. Our dynamic scheduling strategy is also able to mitigate between the differences in granularities. For example, for the code in Figure 2-(5) blocking for the L1 cache with block size B, will result in a runtime choice of a good communication strategy. Some serial transformations (e.g., fission) have the potential of introducing redundant communication operations. For these cases we provide runtime mechanisms such as caching of transferred data.

Therefore, in our framework we enable all the existing Open64 serial loop transformations in a communication agnostic manner and have a separate pass for the communication optimizations. So far, based on our empirical observations we believe that this separation of optimization stages does not lead to a significant decrease in optimization efficiency.

6. Runtime Analysis

The runtime analysis part of our framework is responsible for instantiating the communication operations and determining their granularity and schedule. The first step of the runtime analysis is an estimation of the computation present in a loop. The model used for this task is presented in Section 6.1 and this estimation is an input for the communication performance models.

The second step of the runtime analysis is detection of the communication requirements for the loop nest. The compile time analysis groups together all application level LMADs with the same base reference. For each group, the runtime analysis applies *coalescing* to all the LMADs, followed by tests to determine whether pairs of LMADs are *contiguous* or *interleaved*. At each recombination operation the analysis keeps track of the dependence distance which is needed when strip-mining communication. At the end of this stage, the runtime has a list of communication LMADs. The original computational LMADs are associated to the communication LMADs that subsumes them.

The theoretical complexity of each LMAD simplification operation is in most cases $\mathcal{O}(d\log d)$, where d represents the number of dimensions. Computing the "optimal" solution for the simplification of arbitrary LMAD sets is an NP-hard [28] problem. How-

[4] See examples in [29].

[5] Same index expression in each dimension except for constants

[6] No code scheduling transformations such as software pipelining or register blocking.

[7] On processors used in the HPC systems we studied.

ever, the study in [29] for the SPEC [13] and Perfect [4] benchmark suites indicates that a significant amount of simplification is easily achieved in practice. This study reports statistics for compile time symbolic manipulation only. At runtime the information about the structure and the total span of LMADs is precise. We sort the LMADs within one base reference group by increasing spans and attempt simplifications only for overlapping pairs. The additional complexity of this operation is $\mathcal{O}(N \log N)$, where N is the base reference set cardinality. Furthermore, the number of remote references inside a loop is expected to be small and our experimental results indicate that the analysis is not a significant source of overhead in this case.

Paek [28] explores communication generation only for very simple cases and restricted communication interfaces. Our available communication interfaces are a lot more generic and during this work we have developed new operations for multi-dimensional/multi-index reshaping used for generating communication for complex data redistributions and for generating induction variable coefficients.

At the end of the communication analysis 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. The result of this step is a *communication plan*. A communication plan can be viewed as a tuple:

$$\begin{aligned} CP &= (Type, Comm_Args, N, S, B_Issue, \\ &\quad Init_Level, Sync_Level, B_Sync) \end{aligned} \quad (1)$$

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.

6.1 Estimating Loop Overhead

In order to solve the communication optimization problem, our runtime mechanisms require an estimation of the computation performed inside a loop nest. Loop execution overhead is mostly determined by the number of cache misses, and several compile [16, 23] time approaches to estimate the memory traffic of a loop nest have been described. However, it is not clear how well symbolic estimation correlates with actual performance, since execution time is also determined by the initial cache contents.

We designed our estimation to require minimal hardware awareness and be very computationally lightweight. This design approach is based on the properties of the performance models used in the runtime: 1) regardless of the type of erroneous decision, underestimation of computation time always produces better solutions that overestimation; 2) for communication/computation balanced loops or computationally lightweight loops, the models predict the same solution with an upper bound in the underestimation error of roughly 50% and 3) there is a large range of good decompositions for computationally intensive loops, and the models are able to make good predictions with very coarse computation time estimates.

We estimate computation overhead based on similarity with common operations which we measure directly on the target processors. These operations are vector reductions (`sum+=a[i]`), vector-to-vector operations (`a[i]+=b[i]`), and the raw performance of ALU operations. We record the latency of these type of operations for different data types, operators (+ or /), and memory footprints with both cold and hot caches.

The compile time analysis gathers the information about remote references and counts the number of operations inside a loop nest. Loop optimizations happen before the global scalar optimization phase which performs common subexpression elimination. The operation counting is conservative, e.g., indexing expressions are counted only once. Local references are also analyzed at compile time and separated into locality groups. The information passed to the runtime consists of the number of local references and the number of operators and can be viewed as a tuple:

$$LC = (N_{Local_Ref}, N_{Int_ALU}, N_{FP_ALU}, N_{Div}) \quad (2)$$

After a communication operation, computing on the transferred data is guaranteed to miss in the cache, and we use the remote references as indicators of the loop body overhead. Based on the communication descriptor LMADs and the additional computation information LC, the estimator combines ALU operations with references and incrementally synthesizes vector reduction and vector-to-vector operations. Reuse information is inferred from the polytope description and the communication descriptors. For example, for the matrix-vector multiply operation in Figure 2, assuming that the vector is remote, the synthesized formula is

$$\begin{aligned} T(M,N) &= T^{out}_{red}(M) + (N-1) * T^{in}_{red}(M) + N * T^{in}_{vec}(M) + \\ &\quad 0 * T_{alu}(M * N), \end{aligned} \quad (3)$$

where $T^{out}_{red}(M)$ is the time for an out of cache reduction of length M and $T^{in}_{vec}(N)$ is the time for a in cache vector-to-vector operation.

This approach is computationally lightweight and satisfies all model requirements. It is accurate for communication dominated loops, it underestimates for computation dominated loops and the error range is lower than the precision required by the network models. We have obtained good results for the estimator on various loops including some of the Livermore [15] loops. At this time, we have not experimented with estimators for loops with non-unit inner stride. For these cases the runtime will automatically choose VIS calls which we do not decompose.

6.2 Estimating Communication Load

Load estimation is perhaps the most important source of error in the performance models. For each communicating pair of processors the 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 the XT3 network we choose this value based on visual inspection of the bandwidth profiles at different concurrencies. 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.

6.3 Instantiating Communication

For LMADs that require the transfer of a single contiguous region, the runtime analysis uses the models presented in [19] 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. We consider the three types of scenarios presented in Figure 4: redistributions, lightweight computation and heavyweight computation. Problems are classified based on the computational load as belonging to one of these classes and the choice is made by positioning the problem into the optimization space. On Infiniband and Elan networks, the two subspaces are well separated and we describe the boundary in the optimization space with a simple polygonal shape. This shape, presented in Figure 4, is derived using manual inspection of the performance data and delineates the two implementations. On the Cray XT3 network, where the two subspaces are sparse, we use the whole performance profiles extracted from the tuning micro-benchmarks.

6.4 Memory Management

The runtime is responsible for managing the required communication buffer space and for eliminating redundant communication. Compilers [8, 14] that perform message vectorization usually manage communication buffers at loop nest granularity, or array statement in Fortran language derivatives, and rely on complicated compile time analysis for redundancy elimination. The presence of the UPC memory model, combined with the powerful runtime LMAD representation, allows us to implement coarser grained management and caching techniques.

We use dynamic management from a memory pool that is live between two synchronization operations (barriers, locks, etc.). This scope is also the caching lifetime. Communication LMADs corresponding to previous transfers are associated with the contents of the memory pool, and redundant communication operations are served from the cache whenever possible. Reusing the same communication pool across synchronization points is beneficial for the networks whose performance is affected by the communication memory footprint.

7. Training the Models

The performance database contains the network specific parameters $\{o(b, S), G^l(P, S), G^h(P, S)\}$, the processor specific parameters $\{T^{in}_{red}(S), T^{out}_{red}(S), T^{in}_{vec}(S), T^{out}_{vec}(S), T_{alu}(S)\}$, and the description of the boundary in the problem space where VIS implementations are chosen: $\{Poly_{comm} = \{(V_1, N_1)...\}, Poly_{light} = \{...\}, Poly_{heavy} = \{...\}\}$.

We provide a set of micro-benchmarks to determine all these parameters. The benchmarks have been carefully chosen to minimize the number of experiments required and the total running time. The serial experiments run in minutes, and the parallel experiments run in a matter of hours.

Currently, the data analysis step is performed manually, and it is the most time consuming step. Even with partial "automation" and after performing the analysis for several systems, this step requires weeks of effort.

We have considered providing profiles for new systems based on processor and network families. This approach works well for problems with very large communication volumes but, it fails sometimes for the medium sized transfers which are often encountered in applications. For any new hardware system, our framework will perform only vectorization and automatic synthesis of VIS operations.

The heuristics involved in the choice of the VIS implementations are coarse grained. The effects of computational intensity are captured reasonably well by our classification, refinements are possible in the description of the boundary shape. One important factor for TLB based networks is the total memory footprint of the operation which we currently handle in a very coarse manner. A future area of research is improving this classification with machine learning. Techniques used to support vector machines are applicable in this case, but they will have to be adapted to minimize the complexity of the description of the proposed solution.

System	Network	CPU type
AMD cluster [20]	Infiniband 4x	640 x 2.2GHz Opteron
AMD cluster	Elan4	16 x 2.2 Ghz Opteron
Cray XT3	Custom	2068 x 2.6 Ghz Opteron

Table 1. Systems Used for Benchmarks

8. Experiments

We validate our compiler and runtime optimization framework using the CG, MG, SP, BT, IS and FT application kernels from the NAS [24] Parallel Benchmarks suite. Table 1 presents the systems used in our experiments. We evaluate three different networks: a small departmental cluster with an Elan4 network, a mid-size Infiniband cluster and a large scale Cray XT3 system [2]. The Infiniband and Elan networks are connected in a fat-tree topology, while the Cray system has a torus network architecture. On the Cray XT3 system we report an incomplete set of results which we will extend for the final version of the paper. Some missing results are due to slow queue turnaround on the system, some to implementation problems: the GASNet implementation is still in the testing stage; and we are still validating the models and the runtime implementation.

We compare the performance of a manually optimized version of the benchmarks with the performance achieved by our compiler and runtime optimizations. All versions are based on the officially released UPC implementation [33] 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. In MG, the communication granularity varies dynamically at each call site. SP (*put*) and BT (*put* and *get*) issue a very large number of messages to a given processor, and the count and granularity of these messages varies with problem and system size. IS and FT perform all-to-all communication operations with granularity determined by the problem and system size.

Figure 6 presents the performance results obtained on the Infiniband and the Cray XT3 systems. We report P_{BASE}/P_{OPT}, where P_{BASE} represents the performance of the baseline implementation, which uses manually vectorized blocking communication. The bars labeled HAND show the performance of the manual optimizations. 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. 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 used at multiple sites, 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 cumulative effect of the techniques we employ is directly observable in the benchmark behavior. The performance of CG, IS,

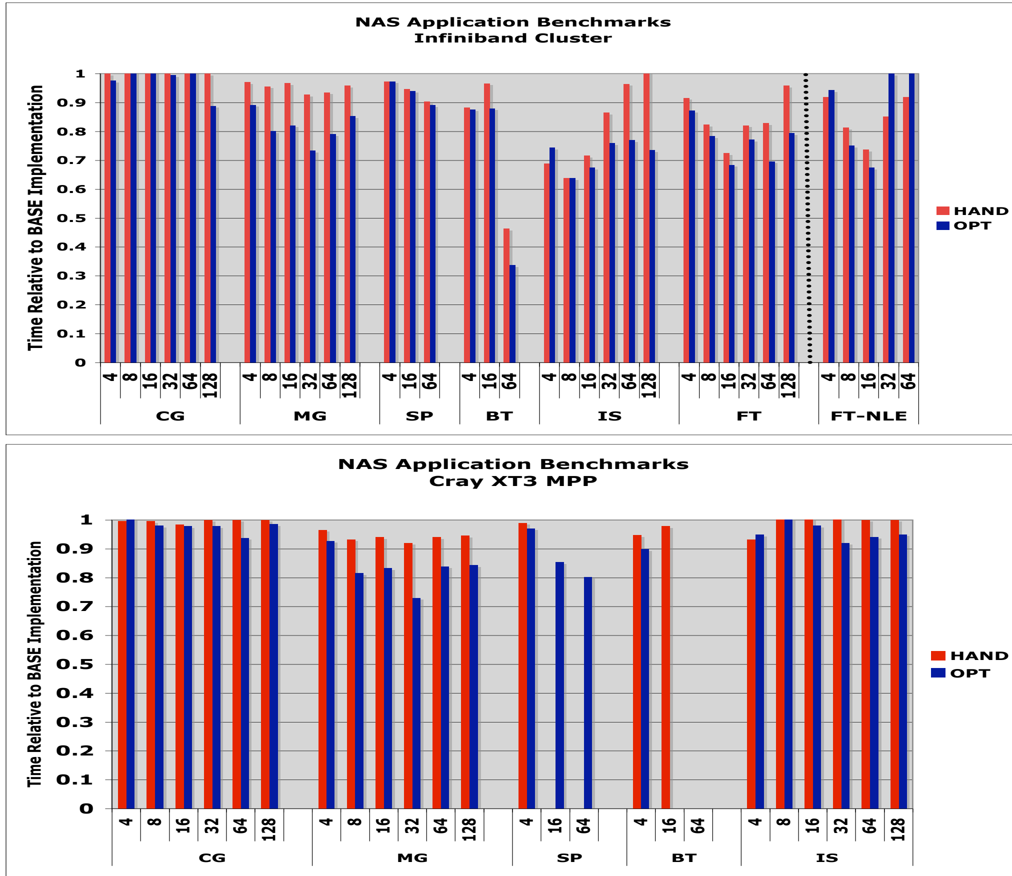

Figure 6. Results for NAS application kernels. Class A on 4 and 8 processors, Class B on 16 and 32 processors and Class C on 64 and 128 processors.

and FT benefits from strip-mining and depends on the estimation of computation and system load. The messages exchanged in CG have short to medium size and the benefits of our optimizations are more pronounced on the Infiniband and Elan systems. These systems have a lower overhead for communication initiation combined with a lower effective bandwidth than on the Cray XT3 system. 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. The choice of an uncongested bandwidth profile in the performance models 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.

The torus network in the XT3 system responds differently to congestion than the fat-tree networks. On this system, the performance of the manually optimized implementations for the benchmarks that perform all-to-all communication (FT,IS) is very uneven and degrades at high concurrencies. For example, the ratio P_{BASE}/P_{HAND} is equal to 2.05 when running the IS benchmark on 128 processors. The application of our techniques improves considerably the performance of the non-blocking implementations, albeit the improvements over the baseline blocking implementation are modest. We are still examining the heuristics involved in the choice of bandwidth profiles for this system, but we believe that topology aware communication scheduling might be required in order to exploit the full potential of non-blocking communication.

The performance of MG, SP, and BT depends on the dynamic selection of pipelining communication over VIS calls. The SP and MG benchmarks communicate a large number of contiguous regions whose length exceeds the unit of data packing (1500 bytes) in the VIS implementation and the implementation automatically chooses pipelining over packing. The BT benchmark transfers regions whose length is in the tens of doubles range and the VIS implementation will automatically choose data packing. For the SP and MG benchmarks the dynamic selection of pipelining communication leads to very modest observable performance improvements in the 1% range. On the Infiniband network, for the BT benchmark the dynamic selection of pipelining leads to additional performance improvements over data packing in the 7% range (7.3% for 32 processors, 6.7% for 64 processors). Note that the results in Figure 6 correspond to statically choosing the VIS calls.

The results on the Elan network are qualitatively similar to the results on the Infiniband network for all benchmarks except CG. This network is more efficient at fine-grained overlap and the CG benchmark exhibits modest ($2\% - 3\%$) speedups even at the low concurrencies examined (4,8,16).

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. 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. Caching of analysis results will eliminate the latter overhead. 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.

The results indicate that the compiler based approach outperforms in most cases the manually optimized implementations. 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. 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[8] speedup observed over all experiments is 27%.

9. Future Work

There are several research and implementation directions that we can pursue to improve the effectiveness and the usability of our framework. The serial performance of our runtime analysis can be further improved and we expect some additional performance benefits from exposing control over analysis at the application level. We also need to finish the experimentation and modeling work on the XT3 system.

We are examining techniques to optimize highly unstructured applications. The applications we have available are a computational fluid dynamics code and a parallel Delaunay triangulation code. In these applications communication is performed inside unstructured `while` loops, and a framework for compile time analysis and cross loop communication scheduling is required. The analysis also needs to be able to handle loops containing conditionals. Our initial results indicate that in order to exploit the full benefit of non-blocking communication on the Cray XT systems, application level mechanisms for global communication scheduling are required. We will explore extending the loop optimization framework of our UPC compiler with transformations and runtime mechanisms for global scheduling.

We believe that our compiler technology can be used for the implementation of high performance collective operations such as reductions, ALLTOALL and ALLTOALLV, eliminating some the need for significant implementation and tuning efforts. The heuristics involved in dynamically choosing between data packing or communication pipelining can be also moved within the communication layer.

Another interesting future research area is to understand better the impact of decoupling serial loop optimizations from communication optimizations. Our initial evaluation indicates that this can be done without overall loss of performance. Auto-tuning parallel libraries can benefit from this approach, as they can employ separate serial performance and communication performance tuning cycles.

10. Related Work

Communication optimizations for parallel programs have been studied extensively in the context of data parallel languages [17, 18, 21, 30, 31]. 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. [21, 22] 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 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 as a goal.

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. [29] 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 [14] 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 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 [32] describe a compile and runtime approach for inspector-executor based programs. They provide models for data packing latency estimation and selection of communication strategies.

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 auto-tuning parallel libraries could use our optimization approach and decouple the serial optimizations from the communication optimizations.

[8] 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.

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] Bigben Cray XT3 MPP. Pittsburgh Supercomputing Center.

[3] 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.

[4] M. Berry, D. Chen, P. Koss, D. Kuck, S. Lo, Y. Pang, L. Pointer, R. Roloff, A. Sameh, E. Clementi, S. Chin, D. Scheider, G. Fox, P. Messina, D. Walker, C. Hsiung, J. Schwarzmeier, K. Lue, S. Orszag, F. Seidl, O. Johnson, R. Goodrum, and J. Martin. The PERFECT Club Benchmarks: Effective Performance Evaluation of Supercomputers. *The International Journal of Supercomputer Applications*, 3(3):5–40, 1989.

[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] K. Dixit and J. Reilly. SPEC Developing New Component Benchmark Suites. SPEC Newsletter, 1991.

[14] 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.

[15] F.H.McMahon. The Livermore Fortran Kernels: A Computer Test Of The Numerical Performance Range. Lawrence Livermore National Laboratory, UCRL-53745, 1986.

[16] S. Ghosh, M. Martonosi, and S. Malik. Cache Miss Equations: An Analytical Representation of Cache Misses. In *Proceedings of the International Conference on Supercomputing (ICS)*, pages 317–324, 1997.

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

[18] 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.

[19] 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.

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

[21] 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.

[22] 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.

[23] T. Mowry. *Tolerating Latency Through Software-Controlled Data Prefetching*. PhD thesis, Stanford University, 1994.

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

[25] J. Nieplocha and B. Carpenter. ARMCI: A Portable Remote Memory Copy Library for Distributed Array Libraries and Compiler Run-Time Systems. *Lecture Notes in Computer Science*, 1586:533–??, 1999.

[26] R. Numrich and J. Reid. Co-Array Fortran for Parallel Programming. Technical Report RAL-TR-1998-060, Rutherford Appleton Laboratory, 1998.

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

[28] Y. Paek. *Compiling for Distributed Memory Multiprocessors Based on Access Region Analysis*. PhD thesis, Univ. of Illinois at Urbana-Champaign, 1997.

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

[30] 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.

[31] 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.

[32] 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.

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

[34] K. Yelick, D. Bonachea, and C. Wallace. A Proposal for a UPC Memory Consistency Model, v1.1. Available at http:upc.lbl.gov/publications/, 2004.

[35] K. Yelick, L. Semenzato, G. Pike, C. Miyamoto, B. Liblit, A. Krishnamurthy, P. Hilfinger, S. Graham, D. Gay, P. Colella, and A. Aiken. Titanium: A High-Performance Java Dialect. In *Proceedings of the ACM 1998 Workshop on Java for High-Performance Network Computing*. ACM Press, 1998.

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