Advanced Collective Communication in Aspen [*]

Qasim Ali
School of Electrical and Computer Engineering,
Purdue University
West Lafayette, IN 47906
qali@purdue.edu

Vijay S. Pai
School of Electrical and Computer Engineering,
Purdue University
West Lafayette, IN 47906
vpai@purdue.edu

Samuel P. Midkiff
School of Electrical and Computer Engineering,
Purdue University
West Lafayette, IN 47906
smidkiff@purdue.edu

ABSTRACT

Aspen is a programming language that relies on high-level messaging to support communication among different program tasks executing in parallel. Unlike MPI, the computational logic of Aspen tasks is specified and developed independently of the global communication structure of the program. A root module specifies the communication structure of the program. The semantics and generality of these specifications enable novel forms of collective communication, including asynchronous and concurrent collective operations and reduction type operations with subsets of the participants being receivers of the reduced data, and with receivers that do not provide data to the reduction. This paper describes efficient implementations of these and other collective communication operations in Aspen. We demonstrate the ease-of-use of these features using several code examples and quantify their performance impact through both microbenchmarks and a quantum chemistry code used in rubber chemistry. Aspen's performance is competitive with, or slightly better than, the performance of MPI implementations for both the chemistry application and the microbenchmarks.

Categories and Subject Descriptors

D.1.3 [**Programming Techniques**]: Parallel programming

General Terms

Algorithms, Performance, Design, Languages

Keywords

Parallel programming, programming languages, algorithms, reductions

1. INTRODUCTION

The dominance of multicore machines has made parallelism, and the need to exploit parallelism in as many applications as possible, more widespread than at any previous time. Applications with high-performance requirements can now achieve those goals in a cost-efficient fashion using a cluster of multicore-based computers. Achieving good performance in such environments requires careful application structuring to minimize communication whenever possible, and efficient implementations of communication to minimize its cost when necessary. Collective communication forms a particularly important class of operations. For example, a collective reduction operation takes input data from multiple participants, performs some associative and commutative operation on the input (e.g., addition), and then provides the results to one participant. Reduction is commonly used, for example, to compute a global sum from per-node local sums. Previous studies have shown that such collective operations consume most of the execution time in parallel applications [21].

The Aspen programming language has been developed to allow programmers skilled in sequential programming to be productive in a parallel environment [23]. Programs in Aspen are expressed as concurrently executing subtasks (called *modules*) that communicate via explicit communication channels. Aspen's programming model, based on high-level message passing and distributed memory semantics, solves many of the problems that plague parallel programming. Aspen can take advantage of hardware shared memory for better performance, but it eliminates the race conditions that make shared memory programming difficult. Modules communicate using *send* and *dequeue* language primitives that are understood by the Aspen compiler and runtime system. Unlike shared memory functions in which any value in shared storage can change without warning, updates to Aspen variables from external sources are always tied to a dequeue. Aspen's communication structure is specified in a *root* module, allowing programmers of individual tasks (or *action* modules) not to be concerned about the details of communication, and to program their computational logic as they would in a sequential setting.

Aspen's global communication is specified as a flow graph in which the nodes correspond to modules and the edges correspond to communication channels. Depending on how the flow graph is constructed, the send and dequeue operations may correspond to point-to-point communication or collective operations. Because the layout of a flow graph is relatively unrestricted, the collective operations can be more general than MPI operations. Thus it is possible for a module to be involved in multiple concurrently executing collective operations, or for the receivers of the result of a collective reduction to be modules that did not provide data to the reduction. Moreover, because *all* communication is via the same general operations, it is essential that they be efficient.

In this paper, we describe Aspen's support for, and use of, col-

[*]This work is supported in part by the National Science Foundation under Grant Nos. CCF-0532448, CCF-0429535, CCF-0313033, and the Purdue Research Foundation.

lective communication in a cluster environment. The contributions include:

1. Support for collective communication operations that target shared, distributed, and mixed environments. These include a new reduce implementation, where the processor(s) receiving the reduced data may not be the same as the processors providing the data, and an efficient reduce algorithm for multiple receivers.
2. Support for collective operations initiated by a single thread to be overlapped with computation and *other collective operations*;
3. Use in a real-world parallel chemistry simulation running on clusters [11];
4. Performance competitive with, or better, than MPI on point-to-point and collective communication in both shared memory and distributed environments using the chemistry application and micro-benchmarks.

The remainder of the paper is organized as follows. Section 2 describes collective communication and the ways in which collective operations are expressed in Aspen. Section 3 describes Aspen's efficient implementation of collective communication. Section 4 gives experimental results. Section 5 describes related work, and Section 6 gives our conclusions.

2. COLLECTIVE COMMUNICATION AND ITS USE IN ASPEN

This section first presents an overview of collective communication and the current state-of-the-art. Next, this section provides an overview of the Aspen programming language and how Aspen specifies communication, including collective communication.

2.1 Collective Communication

Message-passing systems such as MPI provide a variety of collective communication operations. The most commonly used are *reduce*, *broadcast*, and *all-reduce* [21]. The reduce operation collects data from all participants, performs some associative and commutative operation on the data (e.g., addition), and then provides the result to one participant. The broadcast operation sends a piece

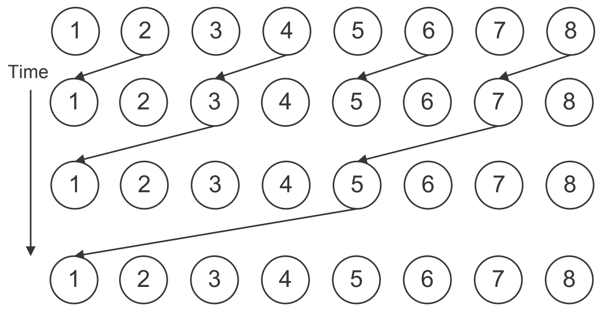


Figure 1: Communication graph of tree-structured reduce.

of data from one node to multiple participants. Both of these are widely implemented using binary tree structures [18]. Figure 1 depicts the communication flow in an 8-node tree-structured reduce; broadcast reverses these stages. The numbers in circles represent nodes, and the arrows represent communication. At each stage in the reduce, the node that receives data performs a reduction of its current data and the data it has received; by the last stage, all data has been processed.

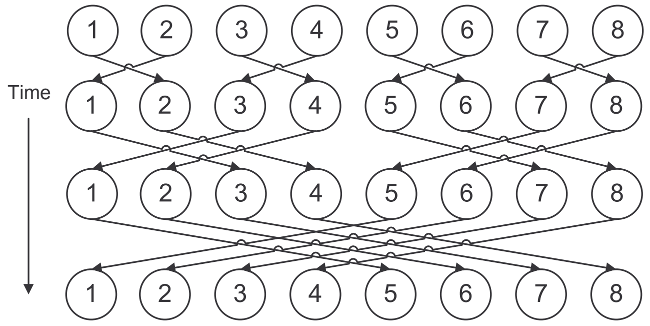


Figure 2: Communication graph of all-reduce using Rabenseifner's algorithm.

The all-reduce operation is semantically a combination of reduce and broadcast: the result of the reduction operation is provided to all participants. Although all-reduce can actually be implemented as reduce followed by broadcast, Rabenseifner presented a more efficient algorithm known as recursive doubling or the butterfly algorithm [21]. Figure 2 shows the butterfly algorithm in action. As in the tree-structured reduce, the even-numbered nodes pass information to the odd-numbered nodes in the first stage. However, the odd-numbered nodes also pass information to their even partners simultaneously. Both sides compute the reduction of their current data and the received data in parallel, and this continues through each successive stage until all nodes have processed all data by the last stage. This algorithm specifically exploits full-duplex communication links to collapse a reduce and broadcast into a single operation.

If the number of nodes involved is not a power of 2, the first stage communicates from the higher-numbered processors to the lower-numbered processors within the next lower power of 2 [20]. The processors within the next lower power of 2 proceed using the butterfly algorithm. After that, the lower-numbered processors feed information back to the higher-numbered processors, for a total of $2 + \lfloor \log_2 p \rfloor$ steps for p cluster nodes in the all-reduce. Note that there are anomalies: for example, a 9-node all-reduce needs 5 steps while a 16-node all-reduce only needs 4. This problem arises because there is not enough aggregate send bandwidth using only 9 nodes to process all input and deliver results within 4 steps. Such anomalies could be avoided if there were extra nodes that could be added into the graph as auxiliaries, or if nodes could send to multiple other target nodes [9], although the resulting network traffic would be higher.

2.2 The Aspen Language, Runtime and Collective Communication

Aspen allows programmers to specify concurrency among tasks and data flows by representing information processing and communication in the style of task flowcharts [23]. The nodes in the graph represent instances of computational modules, while the edges are explicit communication queues between modules. The graph is

```
S1:  Module Main is Root requires Module Chemical, Module Reduce {
S2:      void initialize() {
S3:          Chemical c[16];
S4:          Reduce r;
S5:          flow:
S6:              c ||| r(sum);
         }
     }
S7:  Module Chemical {
S8:      void run() {
S9:          optimization_fcn(...);
         }
S10:     void optimization_fcn (int error_vector_size, int parameter_size,...) {
S11:         QueueElement qs, qr;
S12:         double *error_temp;
             ...
S13:         for (i= 0; i < parameter_size+1; i++) {
S14:             qs = QueueElement(new ChemicalPayload(
                         error_vectors[i], error_vector_size));
S15:             send qs;
S16:             qr = dequeue();
S17:             error_temp = getPayload(qr);
                 ...
             }
             ...
         }
     }
```

(a) The Aspen version of the chemistry code

```
int main(int argc, char *argv[]) {
    int id;
    int p;
    MPI_Init (&argc, &argv);
    MPI_Comm_rank (MPI_COMM_WORLD, &id);
    MPI_Comm_size (MPI_COMM_WORLD, &p);
    optimization_fcn(...);
}
void optimization_fcn (int error_vector_size, int parameter_size,...) {
    double *error_temp;
    ...
    for (i= 0; i < parameter_size+1; i++) {
        if (p > 1) {
            MPI_Allreduce(error_vectors[i], error_temp, error_vector_size,
                MPI_DOUBLE, MPI_SUM, MPI_COMM_WORLD);
            ...
        }
    }
}
```

(b) The MPI version of the chemistry code

Figure 3: Aspen and MPI code for the chemistry application.

specified in the *root* module, while the actual computations are specified in *action* modules. To support hierarchical design, root modules themselves can be composed into graphs by other root modules.

Aspen module instances are similar to objects in C++: they consist of the code and data necessary to implement a data structure and the operations on that structure. Unlike C++ objects, module instances do not include fields (e.g. "class variables") shared among multiple instances of the module. The module definition contains the procedures that make up the functionality of the module including the `init` procedure that initializes the module, the `run` procedure that contains the computation to be executed by this module as part of the application's work flow, and the `data` section, which specifies the variables associated with an instance of the module. All of these components are coded in C++ with a few Aspen-specific extensions.

Aspen also includes built-in modules for functionality that commonly appears across many applications, such as network I/O modules. Unlike standard libraries for traditional languages like Java and C/C++, the semantics of the Aspen built-in modules are understood by the compiler. This understanding can be exploited by the compiler to aid in analysis, optimization and code generation.

Aspen supports collective communication using the built-in modules `Broadcast` and `Reduce`. In addition to performing a reduction on data from various module instances, `Reduce` can send the result to one or more module instances that may or may not contribute data to the reduction, or to all participants if no destination module instances are specified. Thus the Aspen `Reduce` generalizes and extends the common reduce and all-reduce collective operations that appear in MPI and other systems. Module instances contribute to and receive data from collective modules using ordinary send and dequeue operations, respectively. Thus, the action modules actually do not need any knowledge of the collective nature of the communication; the collective behavior is specified entirely in the root module.

We now illustrate Aspen's features using the example of Figure 3, which shows the partial (for brevity) communication structure of the Aspen and MPI implementations of a chemistry code. The chemistry application uses a non-linear Levenberg-Marquardt optimizer to fit the solution of a system of ODEs to a collection of experimental data by repeatedly adjusting parameters to the system. Efficiently collecting global error and load balancing data requires collective communication.

The root module (lines S1 - S6) connects multiple action modules using *flow* statements that describe communication between module instances. The Aspen compiler implements flows as shared-memory queues on a shared-memory platform and TCP sockets on a cluster. Nodes in the cluster are specified using a configuration file that specifies node names and the number of cores in each. The compiler maps module instances to cluster nodes using its own heuristics, but also allows the user to explicitly specify such mappings.

Statement $S3$ declares a vector of module instances of type `Chemical`. The vector notation allows large numbers of module instances of the same type to be easily declared and manipulated. Statement *S4* instantiates a single built-in `Reduce` module.

The actual computation of the Aspen program is done in the `Chemical` module instances. Every module instance contains a copy of the `run` function which is invoked by the Aspen runtime. The `run` function can perform the bulk of the computation, or it can call other functions. In this case `run` calls the function `optimization_fcn` to perform the actual computation.

Most of the statements will be recognized as typical C++ statements, however the `QueueElement` declaration, and the `send`, `getPayload` and `dequeue` statements are of interest. In *S11*, `qs` and `qr` are declared to be of type `QueueElement`, which is the basic unit of communication in Aspen. In *S14*, a `QueueElement` constructor wraps the `ChemicalPayload` object (a user defined object) which contains the data and its size to be enqueued.

In *S15*, the `send` operation places the object on the module instance's default output queue, at which point the Aspen runtime delivers it to the default input queue of the reduction module instance `r`, as described in the `flow` section of the root module. Statement *S16* shows an object being `dequeued`. Note that in both cases the developer of the action module containing the `send` and `dequeue` does not need to know what the external routine will do with the object, how and where it is transmitted, or that the object will be involved in collective communication. Instead, the programmer only needs to follow the contract of the specification for the output queues of the module instance being coded. Thus the effects of the communication are encapsulated. Finally in *S17*, the actual data is extracted from the Aspen's envelope by using `getPayload`.

Although not used in the above example, Aspen also supports modules with multiple input and output queues. These must be declared separately by name. Any output queue may be specified as the source of a flow, and any input queue may serve as the sink.

3. ASPEN'S IMPLEMENTATION OF COLLECTIVE COMMUNICATION

This section describes the algorithms and implementations used by Aspen for its collective communication operations. Because of its flowchart-style of programming, Aspen naturally targets both traditional and non-traditional collective communication uses and algorithms. For example, Aspen naturally allows asynchronous collective communication, as well as collective communication operations where the data must be sent to nodes other than the participants.

3.1 All-reduce Implementation

The program in Figure 3(a) is compiled with a hostfile containing names of 8 nodes, binding two `Chemical` module instances to each node. After parsing the flow statement *S6* and the `hostfile`, the compiler inserts a `Reduce` module on each node and connects that to the `Chemical` module instances and the network I/O modules. Within a cluster node, Aspen performs an all-reduce using either a serial reduction or a local reduction using the algorithm of section 2.1, depending on whether the number of cores or processors in a node makes the binary tree algorithm worthwhile[1]. Across nodes, Aspen uses the butterfly algorithm of Section 2.1.

3.2 Reduce Algorithm

Because collective reductions in Aspen can take many forms, Aspen requires an efficient general algorithm to deal with various cases. The algorithm specified here deals only with reductions across the cluster nodes; operations within a node are processed locally on that node. Figure 4 illustrates the general algorithm below using a specific example.

Call P the set of nodes that are participants in the reduction and R the set of receivers. Further split R into R_p and R_{np} to represent the receivers that are participants and receivers that are not participants. Note that for a traditional collective reduction, R consists of just one element of P, while traditional all-reduce has $R = P$. Aspen allows R to include not only arbitrary subsets of P, but also members that are not in P. The participants should be assigned numbers from 1 to $|P|$ so that participants in R_p have lower numbers than those that are not receivers. Any receivers that are not in P should be assigned successively higher numbers. In Figure 4, the participants are the nodes numbered 1 to 8, while the receivers are nodes 1, 2, 9, 10, and 11.

If $|P|$ is not a power of 2, calculate the value ρ which is the greatest power of 2 less than $|P|$ ($\rho = 2^{\lfloor \log_2 |P| \rfloor}$). Have the elements of P with a number greater than ρ send to the lowest-numbered nodes in the system, as in the all-reduce algorithm. That is, each node with number X greater than ρ should send its data to the node numbered $X - \rho$ so that the lower-numbered node can perform a reduction of its own data and the higher-numbered node's data. Remove the higher-numbered participants from P. If any members of R_p are removed from P, remove them from R_p as well and add them to R_{np}.

At this point, $|P|$ is a power of 2. The following steps will perform the reduction algorithm, providing the results first to receivers in R_p and then to receivers in R_{np}. While $|P| \geq 2 \times \max(|R_p|, |R_{np}|)$, the higher-numbered half of the participants should communicate to the lower-numbered ones. The higher-numbered participants should then be removed from P, and this step should repeat until $|P|$ is less than the desired threshold. Round 1 of Figure 4 shows this stage of the algorithm.

[1]Our current implementation uses a serial algorithm.

At this point, the remaining nodes in P should perform the butterfly algorithm among themselves. In the last step of the butterfly algorithm, keep only the links leading to the first $\max(|R_p|, |R_{np}|)$ number of nodes (or $|P|$, whichever is less). The rest are unnecessary, since they lead to non-receivers and will not be used to provide data to non-participant receivers. Rounds 2 and 3 of Figure 4 show this stage. Note that the butterfly algorithm would normally include a link from node 2 to node 4 in the last stage, but this is eliminated here.

At this point, all receivers in R_p have their needed results; additional nodes may also have the values as needed to propagate them efficiently to R_{np}. The nodes in R_{np} should receive their values from the nodes that have the result values, either directly or through concurrent tree-structured broadcasts starting from each node that has the value. Round 4 of Figure 4 shows R_{np} directly receiving values from participant nodes; additional sends in a tree-structured broadcast fashion would be needed if R_{np} had more members than P.

The algorithm specified here generalizes to any arbitrary participants and receivers, forming a simple tree-structured reduction if there is only one receiver or a butterfly if $R = P$. In general, the maximum number of stages is $2 + \log_2 \lfloor |P| \rfloor$ if $|R_{np}| \leq |P|$ (1 for the initial reduction to a power of 2, $\log_2 \lfloor |P| \rfloor$ for the reduction, and 1 for sending to non-participants), or $2 + \log_2 \lfloor |R_{np}| \rfloor$ when $|R_{np}| > |P|$ (to account for additional steps in propagating results to members of R_{np}).

3.3 Concurrent Reduction.

The above examples show the syntax for specifying collective communication operations in Aspen, but do not show the full power of Aspen collective communication. We now illustrate by means of an example some of the advanced functionality of Aspen collective communication.

Figure 5 shows how Aspen allows two reduces to be performed concurrently, and on overlapping sets of module instances, without needing to use different communicators or groups, as would be the case in MPI. Statements *S6* and *S7* specify input and target modules for the reduction, unlike the all-reduce `r` shown in Figure 3. Statement *S6* specifies that all the module instances of the `Parallel` module are inputs to the reduce `r1`, with the result of the reduce sent to the default input queues of module instances `p[0]` and `p[2]`. Statement *S7* specifies that module instances `p[1]` and `p[2]` are participants in reduce `r2`, with the results of `r2` being sent to the default input queue of module instance `p[3]`. Specifying a reduction target that is not a participant in the reduction, as we have done here, would be very cumbersome in MPI. Statement *S12* declares an output queue named `outQueue1`, and each instance of `Parallel` has its own private `outQueue1`.

The receivers of the value produced by the reduction can differ in both the number of instances and in the type of the instances from the modules that feed into the reduce operation. Because Aspen communication is specified as a unified global operation rather than being constructed piece-wise from many specifications distributed throughout the program, and because the compiler understands the semantics of the communication operations, it is not necessary to specify something analogous to an MPI Communicator. Instead of needing a global object to specify this, the Aspen compiler can compute the module instances involved in the communication on an operation-by-operation basis. The programmer does not need to know any concepts of group communication or membership; the programmer simply needs to send the appropriate data on an output queue.

The reductions in this example are initiated in the `run` procedure

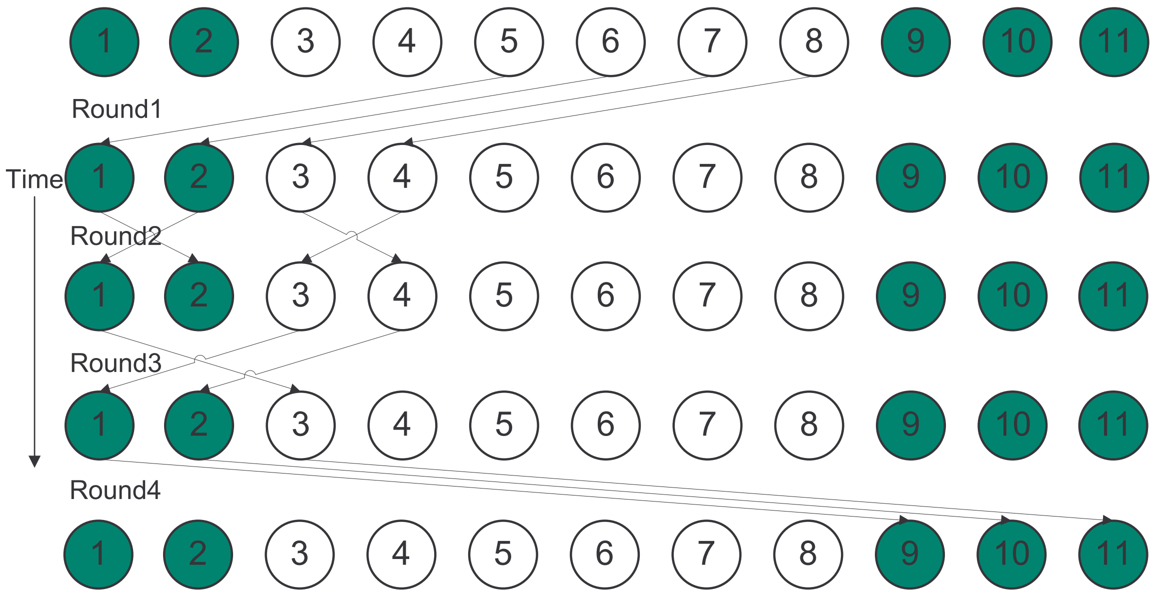


Figure 4: Aspen's Reduce Algorithm.

```
S1:   Module Main is Root requires Module Parallel,Module Reduce {
S2:       void initialize() {
S3:           Parallel p[4];
S4:           Reduce r1, r2;
S5:           flow :
S6:               p ||| r1(sum) ||| p[0], p[2];
S7:               p[1:2].outQueue1 ||| r2(multiply) ||| p[3];
          }
      }

S8:   Declare Module Parallel {
S9:       data_section {
S10:      double *send_buffer1, *send_buffer2;
          }

S11:      Output:
S12:          Output Queue is outQueue1;
      }
S13:  Module Parallel {
S14:      void run() {
S15:          QueueElement qs1, qs2, qr1, qr2;
              ...
S16:          qs1 = QueueElement(new ChemicalPayload(send_buffer1, count));
S17:          qs2 = QueueElement(new ChemicalPayload(send_buffer2, count));
S18:          send qs1;
              ...
S19:          send qs2 on outQueue1;
              ...
S20:          qr1 = dequeue();
              ...
          }
      }
```

Figure 5: Example of concurrent reduces in Aspen

of the `Parallel` function. An instance of `Parallel` will join, or initiate, the reduction `r1` by executing statement *S18*, whose syntax is similar to that of statements *S14* and *S16* in Figure 3. The instance of `Parallel` will join, or initiate, the reduce `r2` by executing statement *S19*, which sends data on the user defined `outQueue1`. Aspen knows what reduce is being initiated because it knows, from the `flow` statement in the *root* module, which

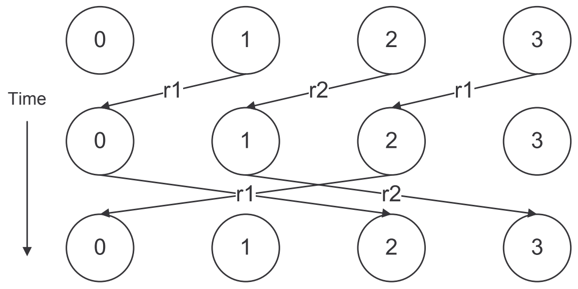
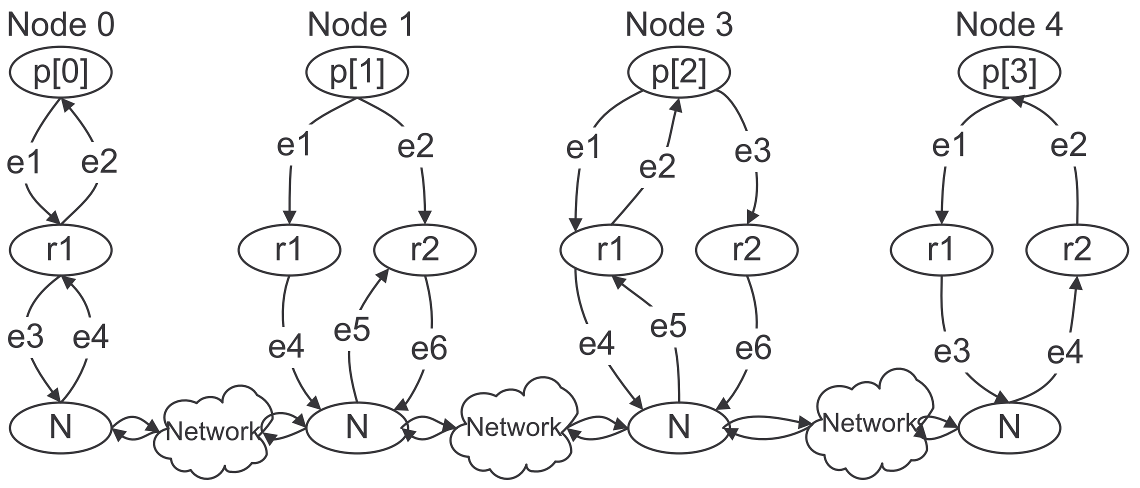


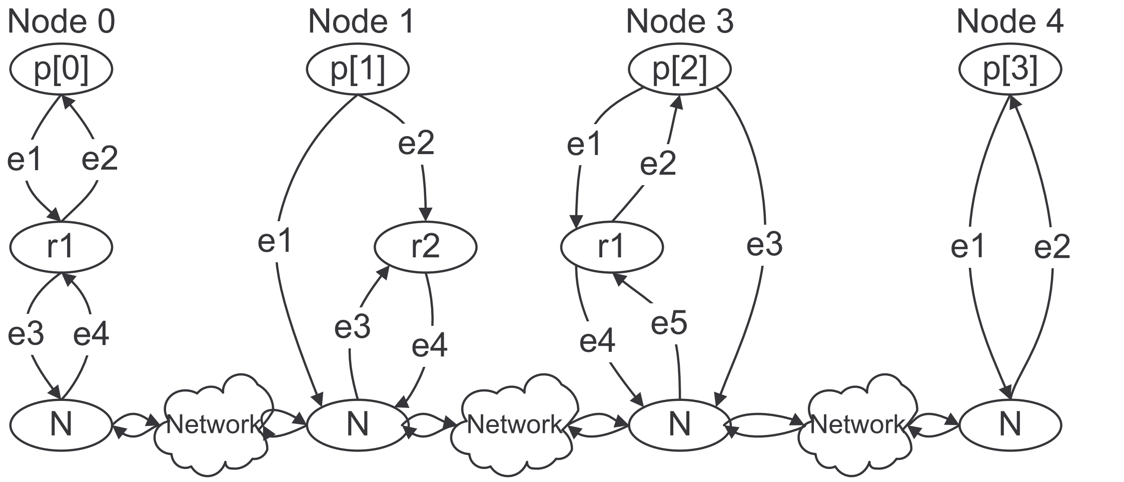
Figure 6: Communication graph of Concurrent Reduce.

queues are bound to which reduce operation. Statement *S20* obtains the result of the reduction `r1` for `p[0]` and `p[2]` and result of the reduction `r2` for `p[3]`. Note that the syntax of the `send` and `dequeue` statements are the same regardless of the type of collective or point-to-point communication being performed. This is part of Aspen's encapsulation of parallelism.

Because sending data to a reduce and receiving data from a reduce are performed as two different operations, the operation is inherently asynchronous. This allows a programmer to insert independent computation between the enqueue and dequeue operations, thus overlapping collective communication with computation in a natural, syntactically clean way. Although some proprietary and research MPI implementations support this feature (for example, IBM's `iReduceAll` operation, or Open Systems Laboratory's LibNBC [14]), we know of no widely-used implementation that does.



(a) After insertion of reduce modules



(b) After optimization (elimination of unnecessary reduce modules)

Figure 7: Aspen Module-Dependence Graph.

Aspen also tries to load balance the communication graph using a greedy heuristic that keeps track of the number of edges coming in and out of a node. For example in Figure 6, after building the communication graph for reduction *r1*, either node 1 can send data to node 2 or node 2 can send data to node 1. If node 2 sends data to node 1, communication at both nodes is serialized and node 2 has to do more computation. If node 2 sends data to node 1, then we have an efficient graph because node 1 can send and receive data at the same time using the duplex communication channel. This load-balance heuristic is evaluated in the experimental section.

If each node runs one instance of the `Parallel` module, the compiler uses the optimized communication graph plus the flow statement to generate a module-dependence graph as shown of Figure 7 for the parallel reduce example in Figure 5. Aspen also optimizes this graph by eliminating reduce modules that simply pass information from one input to one output; for example, node 1 simply has module *p[1]* directly connected to *N's* input queue. The messages are tagged with the destination reduction (i.e. *r1*) so that the message can be routed appropriately. Eliminating modules reduces the number of threads in a program and can greatly reduce the context-switch overhead.

Applications whose MPI implementations use inter-communicators would be natural fits for the advanced collectives in Aspen; an example is 3-D FFT. A 3-D FFT can be split into three one-dimensional FFT's performed along all data points. The data can be distributed block-wise (blocks of xy-planes). A pipeline scheme can be used for the communication. In Aspen, as soon as the first data elements, i.e. planes, are ready, their communication is started in a non-blocking way. Also, as shown in [32], non-blocking collective communication can give up to 34% better performance for conjugate gradient.

4. EXPERIMENTAL EVALUATION

This section reports the performance achieved by the distributed Aspen implementation. The first set of results demonstrates application-level performance, while the second set allows the analysis of the performance of various point-to-point and collective communication operations in Aspen. Both sets of results compare Aspen with MPI; the MPI implementation tested is MPICH 2-1.0.5, and both Aspen and MPI communicate across cluster nodes using TCP sockets. MPI was configured to use the shared memory channel for the shared memory tests (using the `-with-device=ch3:shm` option) and to use the default channel (`-with-device=ch3:sock`) for distributed tests.

The application-level and microbenchmark tests all consider performance on both shared-memory systems with up to 4 cores and distributed-memory cluster systems with up to 16 cores (2 cores per node). The shared-memory tests use a system with 2 AMD Opteron dual-core processors (4 cores total) running at 2.2 GHz and with 4 GB of DRAM. The cluster tests use several identically-configured machines with a dual-core 2.8 Ghz Intel Xeon processor with 4 GB of DRAM. The cluster is interconnected using Gigabit Ethernet, the most commonly-deployed cluster interconnect (including 40% of the Top500 list [1]). All machines use Linux 2.6 and the x86-64 architecture. All tests use the Aspen compiler, and the resulting C++ code is compiled by `g++` compiler.

4.1 Application-level Performance

This section discusses the results of testing the distributed Aspen system using the rubber chemistry modeling application. This system is used to model the vulcanization process of rubber [33, 11]. The total execution time of the runtime phase is recorded to evaluate the performance. The models used were developed as part of an ongoing project in predicting the properties of rubber compounds.

We implemented the chemical application using the distributed Aspen system described in Section 3 and tested it both on shared memory and distributed memory systems. This application spends almost 80 percent of its time in a routine that solves ordinary differential equations (ODE), and has a high computation-to-communication ratio. The performance results seen by Aspen are comparable to those achieved with MPI, showing that Aspen's increased level of abstraction and generality of allowed communication patterns does not exact a performance penalty.

On a single core of the shared-memory platform described above, the ODE solver routine took 53 microseconds per call in the MPI implementation at the highest optimization level that improved performance (-01), and 128 microseconds without optimization. Architecture specific optimization flags did not help performance. The Aspen implementation took 54 microseconds per call to this routine when compiled using `gcc` without optimization, and no optimization level for `gcc` reduced this time.

Figure 8 (a) shows how the application scales with an increasing number of threads. The graph shows three configurations: MPI-O0, MPI-O1, and Aspen; the MPI suffixes are the `gcc` optimization levels used to compile the application. Aspen was always compiled at -O0, as stated above. This graph shows speedup relative to a single thread running MPI-O0 as a function of the number of threads. The Aspen implementation has performance indistinguishable from that of the MPI-O1 version. Both of these versions have linear speedup with up to 4 threads.

When tested on a single core of the distributed memory platform, the ODE solver routine took 177 microseconds per call in the MPI implementation with no optimization, 118 microseconds with either of the "`-O1`" or "`-march=nocona`" optimization flags, and 62 microseconds with both general and architecture-specific tuning. The Aspen implementation took 58 microseconds and was unaffected by general purpose and architecture-specific optimizations.

Figure 8(b) shows the performance speedup of the application running on up to 8 cluster nodes; each cluster node has two threads running on it (one for each core). There are five configurations shown on this chart, one for Aspen and four for MPI (based on whether or not each of general or architecture-specific optimizations were turned on). The Y axis is normalized to the performance of MPI with no optimizations running on two cores (one node). For Aspen and the fully-optimized MPI, this application sees performance scalability linear in the number of threads, with Aspen performing slightly better than the fully-optimized MPI because of the smaller time spent in the ODE solver routine.

4.2 Microbenchmark Performance

Microbenchmarks were implemented to test basic `send`/`recv` type communication, and collective communication. `AllReduce` was used to test collective communication because it is widely used and is part of our chemistry application. All parameters of the hardware and software (including the operating system) were identical for the MPI and Aspen tests. The Microbenchmarks are:

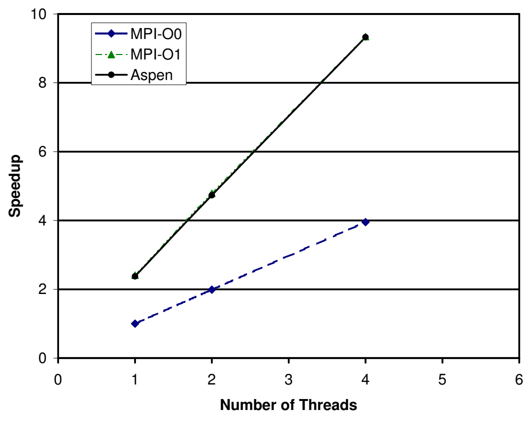
- Ping-pong latency test: measures the round-trip latency by timing how long it takes for a message to arrive at node A after being sent from node A to node B, and then back to node A;
- AllReduce latency test: measures the latency of the `AllReduce` collective operation;
- Bandwidth test: measures the effective bandwidth of a communication channel by having one node send and one node receive as fast as possible.
- Concurrent Reduce and Load-Balancing test: measures the efficiency of the concurrent reduce and the load balancing heuristic proposed in Section 3.3

Figures 9 (a) and (b) show the ping-pong latency test on shared memory and cluster environments, respectively. The latency reported is the round-trip time as a function of the message size in bytes. The latency of both MPI and Aspen are indistinguishable in the cluster environment. However, the latency seen with Aspen is substantially lower than that seen with MPI in the shared-memory environment since Aspen is able to communicate using a more efficient threaded implementation with shared-memory queues, while MPI uses a process model and inter-process communication, since that is its only way to enforce distributed-memory semantics.

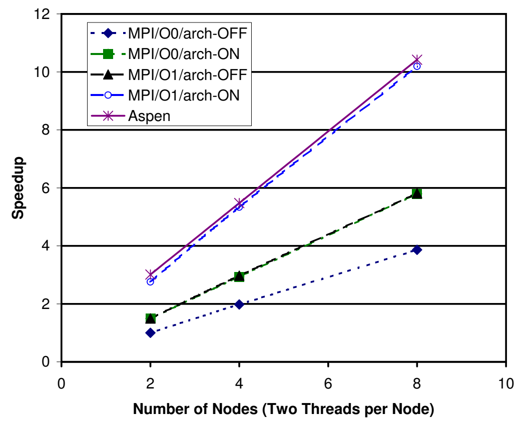
Figures 10 (a) and (b) show the AllReduce latency as a function of the data size in bytes for both shared-memory and cluster environments, respectively. In the case of shared memory, Aspen sees slightly lower latencies because of its faster inter-thread communication. For distributed memory platforms, Aspen performs competitively with MPI.

The bandwidth test is performed only in the cluster environment, with Gigabit Ethernet connections. Figure 11 shows the bandwidth achieved by MPI and Aspen as a function of the message sizes in bytes. Aspen and MPI have almost the same performance. MPI sees a small dip in bandwidth for 256 KB messages because of the protocol switch from eager (short message protocol) to rendezvous (long message protocol) where data is sent to the receiver only when the receiver asks for it.

Figure 12 shows the performance achieved by Aspen's ability

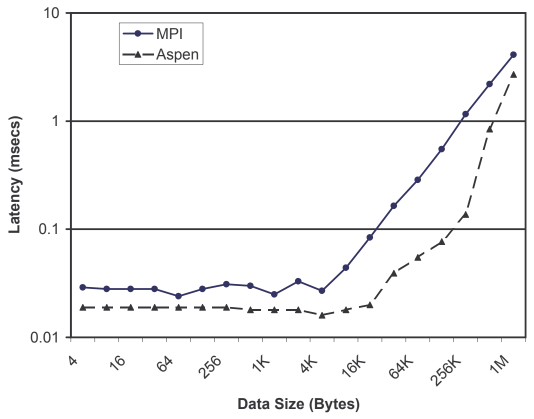


(a) Shared Memory

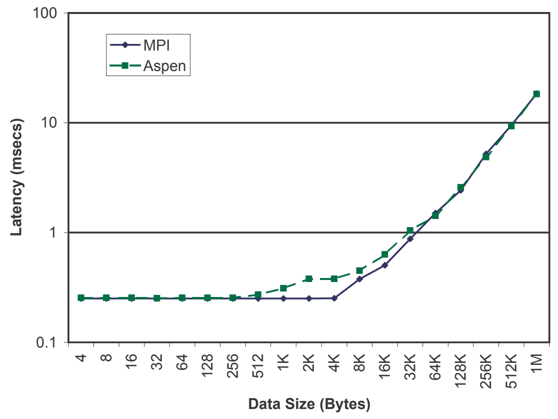


(b) Distributed Memory

Figure 8: Chemistry Application Performance



(a) Shared Memory



(b) Distributed Memory

Figure 9: Ping Pong Latency

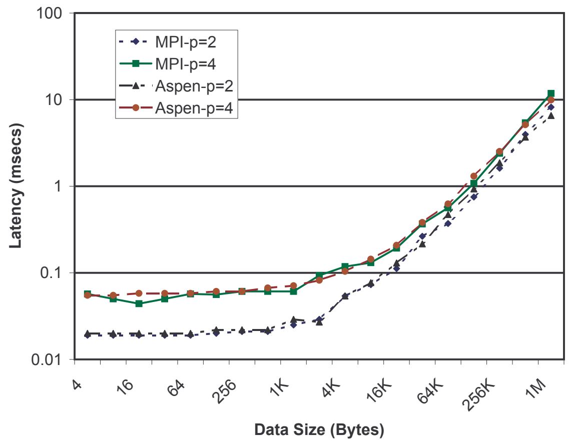
to specify concurrent reductions and the load balancing technique. There are four plots: *MPI* is the example in Figure 5 coded in MPI. *MPI-style Aspen* is an Aspen program coded in MPI-style. Both of those versions use blocking tree-based Reduce algorithms followed by a broadcast if there are multiple receivers. *Aspen-parallel* is the program in Figure 5 with load balancing disabled and *Aspen-parallel-load-balanced* is the same program with load-balancing enabled. The *Aspen-parallel* program is more than twice as fast as the MPI or MPI-Style Aspen programs. The load-balancing optimization in particular provides a maximum performance improvement of 25%

5. RELATED WORK

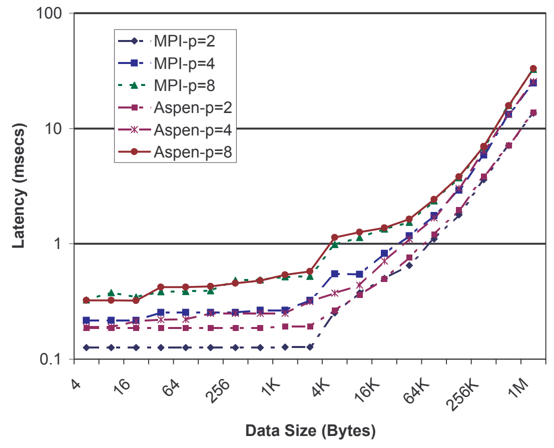
Section 2 discusses the works most closely related to this paper.

5.1 A sampling of related collective communication research.

The development of efficient collective communication operations has a long history. Much of the work has been guided by the MPI and MPI-like libraries. Collective operations are critical for high performance computing [20, 22, 8, 21]. The references in [20] covers much of the major work done in collective communication optimization. However none of those works deal with asynchronous collective communications. Brightwell et al. provide useful data to show the importance of non-blocking collective communication but do not describe an implementation [4]. Hoefler et al. describe LibNBC, which implements non-blocking collective operations for MPI as a library [14]. An earlier work by the same



(a) Shared Memory



(b) Distributed Memory

Figure 10: AllReduce Latency

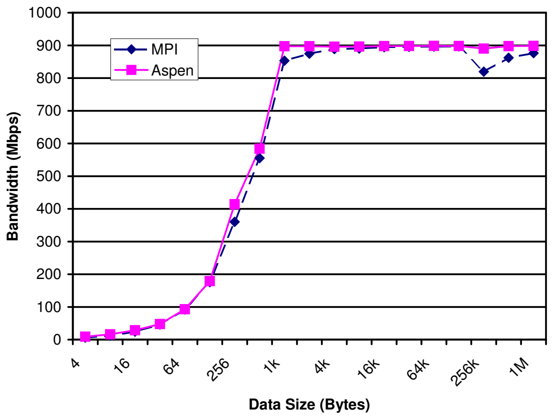


Figure 11: Aspen vs. MPI Bandwidth

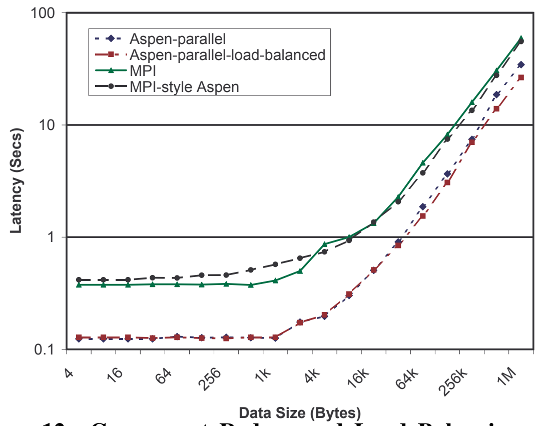


Figure 12: Concurrent Reduce and Load Balancing performance

authors presents some transformations to MPI programs to achieve the computation-communication overlap that can be naturally expressed with Aspen [13].

5.2 Languages for parallel programming

Program composition languages and models.

There have been a variety of languages and models that compose programs out of concurrently executing components. Languages include StreamIt [28, 29] which targets streaming media applications. Like Aspen it communicates via channels and has been ported to a cluster environment. Unlike Aspen, which deals with unpredictable latencies and work arrival rates, StreamIt is synchronous. StreamIt also does not directly include collective operations.

Aspen was originally developed for network servers, as were Flux and SEDA [6, 31]. Aspen allows back-edges in the flow graph, unlike Flux. Aspen is programmed in a sequential fashion, and only Aspen has a cluster implementation or collective operations.

Dataflow languages and languages for distributed computing target the expression of parallelism (e.g. [2, 12, 25, 26] or communication across processes (e.g. [26, 27, 30]. However, they do not address the issues of changing communication primitives to accommodate and optimize for different execution environments, nor do they abstract the communication structure from the logic of the program.

Languages for data parallelism.

Brook targets graphics applications and hardware, allowing programmers to explicitly specify vector data types and arithmetic operations, as well as explicit communication primitives [5]. Programs include parallel functions (called kernels), primitives, and user-specified reductions, but use function calls rather than a flow graph to specify parallelism. Brook has not been ported to a cluster.

HPF [16, 17] allows a programmer to specify a sequential program with distribution primitives, but the compiler was responsible for detecting parallelism. Consequently, HPF does not expose collective operations to the user.

5.3 Shared-address space models

Cluster OpenMP [15] allows specification of parallelism with a shared memory, relaxed consistency model on a cluster. Like OpenMP, Cilk [3, 10] targets shared memory machines, communication is implicit, and through shared memory. Parallelism is created by spawning new threads, and Cilk has support for algorithm analysis. Linda [7] communicates through a tuple space, with tuple keys potentially accessible to all threads. UPC and Co-Array Fortran represent PGAS (partitioned global-address space) languages, with inter-thread communication enabled by explicitly identified shared-memory operations and affinity of data to threads [19, 24]. Like OpenMP, they facilitate the incremental parallelization of large-scale codes and suffer from the problems of shared memory. Aspen differs from all of these in having a shared-nothing programming model and parallelism expressed as a flow graph.

6. CONCLUSIONS

Aspen provides high-level communication abstractions that allow concrete descriptions of communication patterns to be made available to the compiler, runtime, and developer doing code maintenance. Additionally, programmers of the sequential logic of the program can be oblivious to these concrete patterns, only needing to know that a desired data object can be found on a named queue.

Aspen's communication semantics allow for direct and efficient implementation of asynchronous, concurrent, and arbitrary collective communication operations. The computational logic of any given subtask of the program need not know whether it is connected to a collective communication module or any other, so code may be reused in different contexts as long as it obeys the basic software contract associated with the sends and receives on the module's explicit communication channels. The underlying collective operations may be implemented using any arbitrary algorithm without affecting user code. In particular, this paper presents a new and efficient algorithm for collective reduction operations with arbitrary participants and arbitrary receivers, as well as optimizations that target concurrent reduction operations.

This paper evaluates Aspen's support for collective communication using a chemical simulation code and microbenchmarks. The results show performance competitive with, or better than MPI on point-to-point and collective communication in both shared memory and distributed environments using the chemistry application and micro-benchmarks. Additionally, the results show that a load-balancing optimization targeting concurrent reductions can improve performance by up to 25%.

Aspen's collective communication semantics allow operations that are (i) more flexible than the MPI communication primitives in allowing computation and collective communication to be easily overlapped, (ii) are simpler than MPI 2 in allowing different sets of processes to be producers and consumers of data in collective operations, and (iii) provide a higher level of encapsulation of the sequential logic within a procedure from the overall structure of a program than does MPI. All of these benefits come without loss of portability since the same code can be used in both shared-memory nodes and distributed-memory clusters, with the compiler deciding how to allocate module instances to different nodes and how communication channels should actually be implemented. Further, this additional flexibility, portability, and maintainability comes without any degradation in performance in our tested benchmarks.

7. REFERENCES

[1] 29th TOP500 List, June 2007.

[2] K. Arvind and R. S. Nikhil. Executing a program on the mit tagged-token dataflow architecture. *IEEE Trans. Comput.*, 39(3):300–318, 1990.

[3] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou. Cilk: An efficient multithreaded runtime system. *In Proceedings of the Fifth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, pages 207–216, Santa Barbara, California, July 1995.

[4] R. Brightwell, S. P. Goudy, A. Rodrigues, and K. D.Underwood. Implications of application usage characteristics for collective communication offload. *International Journal of High Performance Computing and Networking*, 4:104–116, 2006.

[5] I. Buck, T. Foley, D. Horn, J. Sugerman, K. Fatahalian, M. Houston, and P. Hanrahan. Brook for GPUs: Stream Computing on Graphics Hardware. *In Proceedings of ACM SIGGRAPH 2004*, August 2004

[6] B. Burns, K. Grimaldi, A. Kostadinov, E. D. Berger, and M. D. Corner. Flux: A Language for Programming High-Performance Servers. In *Proceedings of the USENIX 2006 Annual Technical Conference*, pages 129–142, June 2006.

[7] N. Carriero and D. Gelernter. Linda in context. *Commun. ACM*, 32(4):444–458, 1989.

[8] Chan, E.W., Heimlich, M.F., Purkayastha, van de Geijn, and R.A. On optimizing collective communication. *In Proceedings of the 2004 IEEE International Conference on Cluster Computing*, 2004.

[9] E. Chan, R. van de Geijn, W. Gropp, and R. Thakur. Collective communication on architectures that support simultaneous communication over multiple links. *In PPoPP 06: Proceedings of the eleventh ACM SIGPLAN symposium on Principles and practice of parallel programming* , pages 2–11, 2006

[10] M. S. DeBergalis. A parallel file I/O API for Cilk. *Master's thesis, Massachusetts Institute of Technology Department of Electrical Engineering and Computer Science*, May 2000.

[11] A. Goyal, J. Cao, P. Patkar, G. Medvedev, S. P. Midkiff, V. Venkatasubramanian, and J. M. Caruthers. Population balance kinetic model for interaction of 2-bisbenzothiazole-2-2 disulfide (mbts) with sulfur. *Rubber Chemistry and Technology* , 2007. In press.

[12] J. Gurd and W. Bohm. Implicit parallel processing: SISAL on the Manchester dataflow computer. In *Proceedings of the IBM-Europe Institute on Parallel Programming*, Aug. 1987.

[13] T. Hoefler, A. Lumsdaine, and W. Rehm. Transformations to parallel codes for communicationcomputation overlap. *In SC 05: Proceedings of the 2005 ACM/IEEE conference on Supercomputing. IEEE Computer Society/ACM* , 11 2005.

[14] T. Hoefler, A. Lumsdaine, and W. Rehm. Implementation and Performance Analysis of Non-Blocking Collective Operations for MPI. *In proceedings of the 2007 International Conference on High Performance Computing, Networking, Storage and Analysis, SC07. IEEE Computer Society/ACM* , 11 2007

[15] J. Hoeflinger. Extending OpenMP to clusters, 2006. *http://cachewww.intel.com/cd/00/00/28/58/285865 285865.pdf* , last checked Oct. 8, 2007.

[16] K. Kennedy, C. Koelbel, and H. Zima. The rise and fall of High Performance Fortran: an historical object lesson. *In HOPL III: Proceedings of the third ACM SIGPLAN conference on History of programming languages*, pages 7-1-7-22, 2007.

[17] C. Koelbel. An overview of high performance fortran. *SIGPLAN Fortran Forum*, 11(4):9–16, 1992.

[18] D. Kuck. Structure of Computers and Computations. *John Wiley* , 1979.

[19] R. W. Numrich and J. Reid. Co-array Fortran for parallel programming. *SIGPLAN Fortran Forum*, 17(2):1–31, 1998.

[20] J. Pjesivac-Grbovic, T. Angskun, G. Bosilca, G. E. Fagg, E. Gabriel, and J. J. Dongarra. Performance analysis of mpi collective operations. *Cluster Computing Journal*, 10:127–143, 2007.

[21] R. Rabenseifner. Optimization of Collective Reduction Operations. *In Proceedings of the International Conference on Computational Science* , June 2004.

[22] R. Thakur and W. Gropp. Improving the performance of ollective operations in mpich. *In 10th European PVM/MPI Users Group Conference (Euro PVN/MPI 2003)* , September 2003.

[23] G. Upadhyaya, V. S. Pai, and S. P. Midkiff. Expressing and Exploiting Concurrency in Networked Applications with Aspen. *In Proceedings of the 2007 ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* , pages 13–23, March 2007.

[24] UPC Consortium. UPC Language Specification (Version 1.2), June 2005.

[25] N. Harvey and J. Morris. NL: A general purpose visual dataflow language. *Australian Computer Journal*, 12(1):2–12, 1996.

[26] C. Hoare. Communicating sequential processes. *Communications of the ACM*, 21(8), Aug. 1978.

[27] R. Salama, W. Liu, and R. S. Gyurcsik. Software experience with concurrent c and lisp in a distributed system. In *CSC '88: Proceedings of the 1988 ACM sixteenth annual conference on Computer science*, pages 329–334, New York, NY, USA, 1988. ACM Press.

[28] W. Thies, M. Karczmarek, and S. Amarasinghe. Streamit: A language for streaming applications. In *International Conference on Compiler Construction*, Grenoble, France, April 2002.

[29] W. Thies, M. Karczmarek, J. Sermulins, R. Rabbah, and S. Amarasinghe. Teleport messaging for distributed stream programs. *In Proceedings of the Symposium on Principles and Practice of Parallel Programming* , Chicago, Illinois, June 2005.

[30] P. H. Welch. An occam approach to transputer engineering. In *Proceedings of the third conference on Hypercube concurrent computers and applications*, pages 138–147, New York, NY, USA, 1988. ACM Press.

[31] M. Welsh, D. Culler, and E. Brewer. Seda: An architecture for well-conditioned, scalable internet services. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles*, October 2001.

[32] Torsten Hoefler and Peter Gottschling and Andrew Lumsdaine and Wolfgang Rehm. Optimizing a conjugate gradient solver with non-blocking collective operations. In *Parallel Computing Journal*, 9:624–633, 2007.

[33] Jun Cao and Ayush Goyal and Samuel P. Midkiff and James M. Caruthers An Optimizing Compiler for Parallel Chemistry Simulations In *21th International Parallel and Distributed Processing Symposium (IPDPS 2007).*