A Data Transfer Library for Communicating Data-Parallel Tasks

B. Avalani*, A. Choudhary*, I. Foster†, R. Krishnaiyer*, and M. Xu†

Abstract

Many computations can be structured as sets of communicating data-parallel tasks. Individual tasks may be coded in HPF, pC++, etc.; periodically, tasks exchange distributed arrays via channel operations, virtual file operations, message passing, etc. The implementation of these operations is complicated by the fact that the processes engaging in the communication may execute on different numbers of processors and may have different distributions for communicated data structures. In addition, they may be connected by different sorts of networks. In this paper, we describe a communicating data-parallel tasks (CDT) library that we are developing for constructing applications of this sort. We outline the techniques used to implement this library, and we describe a range of data transfer strategies and several algorithms based on these strategies. We also present performance results for several algorithms. The CDT library is being used as a compiler target for an HPF compiler augmented with Fortran M extensions.

1 Introduction

We consider the problem of efficient data transfer in computations comprising sets of concurrently executing data-parallel tasks. Programs with this structure occur in numerous domains, including multidisciplinary analysis and image processing. Each task executes a data-parallel program on multiple processors. Periodically, tasks exchange data, for example, using send and receive operations on a shared channel.

Efficient data transfer between data-parallel tasks is a nontrivial problem. Sending and receiving tasks may execute on different numbers of processors and use different data distributions for communicated data structures. Tasks may execute on overlapping or disjoint sets of processors within the same computer, or on different computers connected by various types of network such as Ethernet or ATM wide area network. The data to be transferred may be fully distributed, using block or cyclic distributions in one or more dimensions, or may be replicated. Finally, tasks may perform a series of transfers using the same data distributions, or may change data distributions on each transfer. A flexible data transfer library should allow efficient execution in at least the most common of these situations. We have designed such a library and used it to experiment with several alternative data transfer algorithms.

There are intriguing similarities between some aspects of the data transfer problem and the problems of array redistribution in HPF [15, 12], all-to-all communication [4, 5, 14], and parallel I/O [3]. Some techniques developed for those problems can be applied to the data transfer problem; some techniques described in this paper may be applicable to parallel I/O. In other related work, Hatcher and Quinn [9] describe an implementation of communicating data-parallel C tasks on iWarp; however, communications pass via a single control processor. Subhlok et al. [13] consider communicating HPF tasks, also on iWarp. Foster et al. [7] describe the use of Fortran M constructs to coordinate HPF computations, but do not consider data redistribution.

The principal contributions of this paper are the description of a simple but flexible data transfer library for communicating data-parallel tasks, a description and preliminary analysis of novel data transfer strategies and algorithms, and experimental results allowing evaluation of the efficiency of some of these algorithms.

*Computer Information Science/Computer Science Engineering, Syracuse University, Syracuse, NY 13244.
†Mathematics and Computer Science Division, Argonne National Laboratory, Argonne, IL 60439.

2 The Communication Library

Our communicating data-parallel tasks (CDT) library comprises two components:

1. A set of *interface routines* allows a compiler or library writer to supply data distribution information, to invoke send and receive operations, and to provide other information to the runtime library that will allow it to choose efficient data transfer algorithms.

2. A *runtime library* provides an implementation of the interface routines. It uses parameters provided by the interface routines to select between alternative data transfer algorithms.

The library is designed to support point-to-point communication between pairs of data-parallel tasks. Collective operations such as multicast can use many of the same techniques but also introduce additional issues.

2.1 Interface

The interface supports communication from one task (the sender) to another (the receiver). The sender and receiver tasks are assumed to be HPF-like data-parallel computations executing HPF, Vienna Fortran, Fortran D, or pC++ [2, 8, 10, 11] programs as S and R "processes" (HPF processors), respectively. The interface allows the sender to send an arbitrary array, distributed using any legal HPF distribution over its S processes; the receiver must receive into an array of the same size and shape, which again can be distributed using any legal HPF distribution over its R processes.

The interface comprises five main functions, for which we provide a high-level description in the following. The interface separates the actions of *initializing* and *executing* a communication. In the initialization phase, sending and receiving tasks can exchange distribution information and precompute communication schedules. In the execution phase, the actual data transfer is performed. The advantage of this separation is that if the same communication is to be performed many times, initialization costs can be amortized over many communications.

cdt_sender_init(SENDDIST, COUNT, RECEIVER, HANDLE): This routine performs initialization in the sender task. `SENDDIST` indicates the size, shape, and distribution of the array that is to be communicated, `COUNT` is an estimate of the number of communication operations to be performed with this distribution, and `RECEIVER` names the receiver task. The routine returns a `HANDLE` containing a pointer to a data structure containing communication schedule information. The `COUNT` argument allows the runtime library to determine how much time can be spent precomputing communication schedules. (The interface may eventually expand to allow a compiler or runtime library to provide other information besides estimated execution counts.)

cdt_receiver_init(RECVDIST, COUNT, SENDER, HANDLE): This is the equivalent of the previous routine on the receiver's side.

cdt_sender_exec(HANDLE, DATA): This call performs the actual data transfer; it can be invoked only after a corresponding initialization call.

cdt_receiver_exec(HANDLE, DATA): This call performs the actual data transfer; it can be invoked only after a corresponding initialization call.

cdt_free(HANDLE): This call frees the data structure associated with the supplied `HANDLE`.

The two initialization routines are intended to be executed as a single collective operation by the two tasks participating in a communication operation. The implementation must ensure correct behavior when the same task is involved in several concurrent data transfer operations: for exmaple, if executing in a pipeline.

2.2 Data Transfer Algorithms

The CDT library is designed to incorporate a wide range of data transfer algorithms, each with performance advantages in different situations. We first present a set of basic strategies that can be used in designing these algorithms. These strategies trade off communication volume for number of messages in various ways. For simplicity, we restrict our attention to the general problem of redistributions involving fully distributed arrays, in which all senders must communicate the same amount of data to all receivers. Other redistributions are special cases of this problem. We assume S senders and R receivers, and an array of size D. Without loss of generality, we assume $S \geq R$. Figure 1 illustrates these strategies.

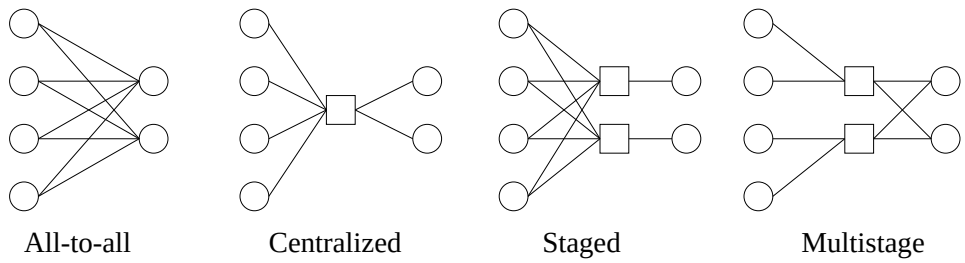

Figure 1: Four different communication strategies when $S = 4$, $R = 2$. In each case, sending processors are on the left and receiving processors on the right. Squares represent staging processors, which would typically be mapped to sending processors.

1. **All-to-All.** Each sender sends data to each receiver. This strategy can be useful if message startups are cheap and the network connecting senders and receivers provides high connectivity.

2. **Multistage.** As in all-to-all communication [5], the number of messages required for a data transfer can be reduced by combining data destined for each receiver using a multistage communication structure, such as a butterfly. Total data volume is increased, however. This strategy can be useful if message startups are expensive and the amount of data to be transferred is small.

3. **Staged.** We can accumulate data on a set of C staging processors, typically a subset of the sending processors. Each staging processor then transfers data to a single receiver, which distributes data among receivers if $C < R$. This strategy increases total data volume (as each data value is communicated more than once) but reduces the number of messages sent from senders to receivers. Hence, it can be useful if message startup costs from senders to receivers are high: for example, if senders and receivers execute on separate MPPs connected by a local area network or wide area network.

While the multistage and staged strategies seek to reduce the number of messages, another strategy that can be effective if message startups are inexpensive is to break up data transfers into smaller pieces so as to pipeline computation on data with its communication. This strategy is not supported directly by our interface, and requires compiler assistance if it is to be performed automatically.

These strategies can be used in various combinations to develop a wide range of algorithms. We have chosen to focus on five in our initial investigations.

1. **All-to-All**. This involves $S \times R$ messages and the communication of D data.

2. **Centralized.** All data is accumulated in a single sender, which then distributes it to receivers. This requires a total of $S + R - 1$ messages ($S - 1$ for accumulation and R for distribution) and the transfer of approximately $2D$ data. This algorithm does not permit concurrent execution but may be useful if a network provides low connectivity.

3. **Staged.** Data is accumulated in R senders using all-to-all communication; these staging processors then send it to the receivers. This requires a total of $S \times R$ messages ($(S - 1) \times R$ for accumulation

and R for transfer) and the transfer of approximately $2D$ data. This algorithm may be useful if sender and receiver are connected by a network with high message startup costs.

4. **Multistage.** A butterfly-like combining network is used to achieve data transfer in $\mathcal{O}(\log_2 S)$ steps. This algorithm requires a total of about $S \log_2 S$ messages and the transfer of approximately $D \log_2 S$ data. This algorithm may be useful in an MPP with high connectivity if message startup costs are high.

5. **All-to-All/Staged Hybrid.** This is a variant of the all-to-all algorithm appropriate when $S > R$. It proceeds in a synchronous fashion, in multiple steps. In each step, R senders transfer data to R receivers. At the same time, the $(S - R)$ idle senders engage in staging. As illustrated in Figure 2, this algorithm can reduce the number of messages from senders to receivers relative to all-to-all, without increasing the amount of data transferred.

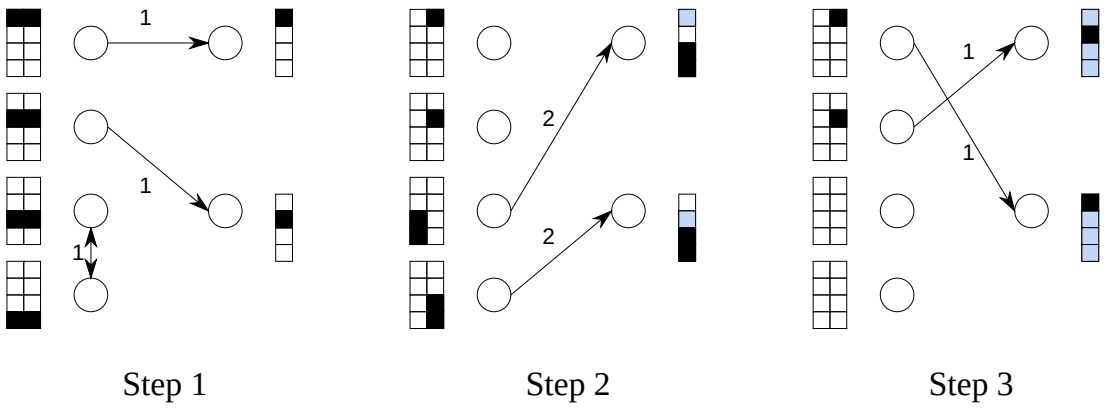

Figure 2: A data transfer from four senders to two receivers using the "all-to-all/staged hybrid" algorithm. The transfer proceeds in three steps. In the arrays of blocks to the left of the sending processors, column I represents data destined for processor I, while row J represents data from processor J. Notice the staging performed in the first stage. This allows the data transfer to proceed in three instead of four stages, as would be required with the all-to-all algorithm.

2.3 Implementation

A prototype of the data transfer library has been constructed using the Fortran M (FM) [6] extensions to Fortran. For the purposes of the experimental studies described in the next section, calls to this library were incorporated by hand into code generated by the Syracuse HPF compiler [1]. We are developing compiler extensions that will allow communications to be specified using FM-like `CHANNEL` statements embedded in HPF programs. The compiler would then generate calls to CDT interface routines to perform the necessary communication. (An alternative approach would be to call the CDT routines as HPF extrinsic procedures.) Figure 3 shows an example of the type of program that might be written using these constructs. (This is one of the programs used for benchmarking purposes.)

In our implementation, each HPF task is implemented as a collection of FM processes, and communication both within and between tasks is implemented using FM channels [7]. Communicating data-parallel tasks may execute on the same or different processors; in the experiments described in this paper, we place them on disjoint sets of processors. Separate channels are established for the transfer of data and control information.

Calls to the sender and receiver initialization routines cause designated master processes on each side to exchange data distribution information. This information is then broadcast to the senders and receivers, which can compute the communication schedule in a distributed fashion. The maximum communication cost incurred by any processor in this exchange should be $1 + \max(\log_2 S, \log_2 R)$ messages. (Our current implementation uses an $\mathcal{O}(N)$ broadcast, and maximum cost is $1 + 2 * \max(S, R)$ messages.)

```
      program example
!hpf$ processors pr(24)
      INPORT  (real x(128,128)) pi
      OUTPORT (real x(128,128)) po
      CHANNEL(in=pi, out=po)
      PROCESSES
         PROCESSCALL sender(po) SUBMACHINE(pr(1:16))
         PROCESSCALL receiver(pi) SUBMACHINE(pr(17:24))
      ENDPROCESSES
      end

      PROCESS sender(po)
!hpf$ processors pr(16)
      outport (real x(128,128)) po
      real a(128,128)
!hpf$ distribute a(block,*)
      do i = 1,10
        call produce(a)
        SEND(po) a
      enddo
      ENDCHANNEL(po)
      end

      PROCESS receiver(pi)
!hpf$ processors pr(8)
      inport (real x(128,128)) pi
      real b(128,128)
!hpf$ distribute b(*,block)
      do i = 1,10
        RECEIVE(pi,end=10) b
        call use(b)
      enddo
   10 continue
      end
```

Figure 3: A simple HPF program augmented with syntax (in bold face) for specifying creation and communication between concurrent tasks. The main program creates a channel and two tasks, `producer` and `consumer`. The two tasks execute on 16 and 8 processors, respectively, and the producer sends a sequence of arrays distributed `(BLOCK,*)` to the consumer, which receives them into an array distributed `(*,BLOCK)`.

Calls to the sender and receiver execution routines cause the actual data transfer to take place. In our current implementation, the communication schedule for each transfer is computed on the fly during the execution phase. This is probably the more efficient strategy if every communication involves a different schedule, as it allows the computation of the schedule to be overlapped with communication. In the future, we will precompute schedules whenever the initialization call indicates that the same schedule can be reused (that is, if `COUNT` > 1 in `cdt_sender_init` and `cdt_receiver_init`).

3 Experimental Studies

Section 2.2 listed five different data transfer algorithms, each apparently suitable for different situations. Our eventual goal is for our library to select optimal algorithms automatically, given information about system and problem characteristics. As a first step, we incorporated the "all-to-all" and "centralized" algorithms into our prototype library. Our implementation supports arbitrary numbers of processors and distributions on the sending and receiving ends. In this section, we present the results of several experimental studies conducted using this library. These provide some insights into the costs of our data transfer algorithms. Experiments were performed on the Argonne IBM SP1 multicomputer, which comprises 128 RS/6000 microprocessors connected by a multistage crossbar providing peak data rates of 6 MB/sec. Processors are also connected via Ethernet.

3.1 All-to-All Implementations

In our first set of experiments, we fix $S = R = 8$ and measure execution time as a function of array size for three implementations of the all-to-all algorithm on 16 processors. The redistribution considered is `(BLOCK,*)` to `(*,BLOCK)`, which requires all-to-all communication.

The first implementation considered uses a generic communication strategy that computes communication schedules on the fly, with each processor computing the destination processor for each of its data elements. Data elements are accumulated in buckets, one per receiver, and a single message is generated for each bucket. This is a general-purpose strategy but performs much unnecessary computation.

The second implementation is specialized for the `(BLOCK,*)` to `(*,BLOCK)` redistribution. This uses block copy operations and performs significantly less computation.

The third implementation is also specialized for `(BLOCK,*)` to `(*,BLOCK)` but does not perform initialization before each communication. This allows us to quantify the cost associated with initialization, and hence the benefits of compiler optimizations that minimize the number of initialization operations performed.

The cost of a single transfer for each of these three implementations (termed generic, optimized, and async, respectively) was determined by averaging over a large number of identical transfers. Results are presented in Figure 4, along with fits from the following simple performance model:

$$T = t_m M + t_w \frac{D}{(S+R)/2}, \tag{1}$$

where M is the number of messages per processor: 25 for the first two algorithms (as noted above, the prototype implementation uses a rather inefficient setup strategy) and 8 for the second algorithm, D is the array size in words, t_m is a per-message cost, and t_w is a per-word cost. The t_s and t_m values obtained are as follows, in μsec; for convenience, the t_w values are also expressed as MB/sec.

Algorithm	M	t_m	t_w	MB/sec
Generic	25	293	19	0.2
Optimized	25	288	2.8	1.4
Async	8	383	1.8	2.2

The fits are good, suggesting that our simple model is a reasonable characterization of algorithm behavior. The per-word costs are considerably higher for the generic algorithm, emphasizing the importance of both incorporating specialized algorithms for common data distributions and precomputing communication schedules. The two "optimized" algorithms achieve data transfer rates of about 1.4 MB/sec and 2.2 MB/sec, respectively. As the peak bandwidth of the SP1 is around 6 MB/sec, there is clearly

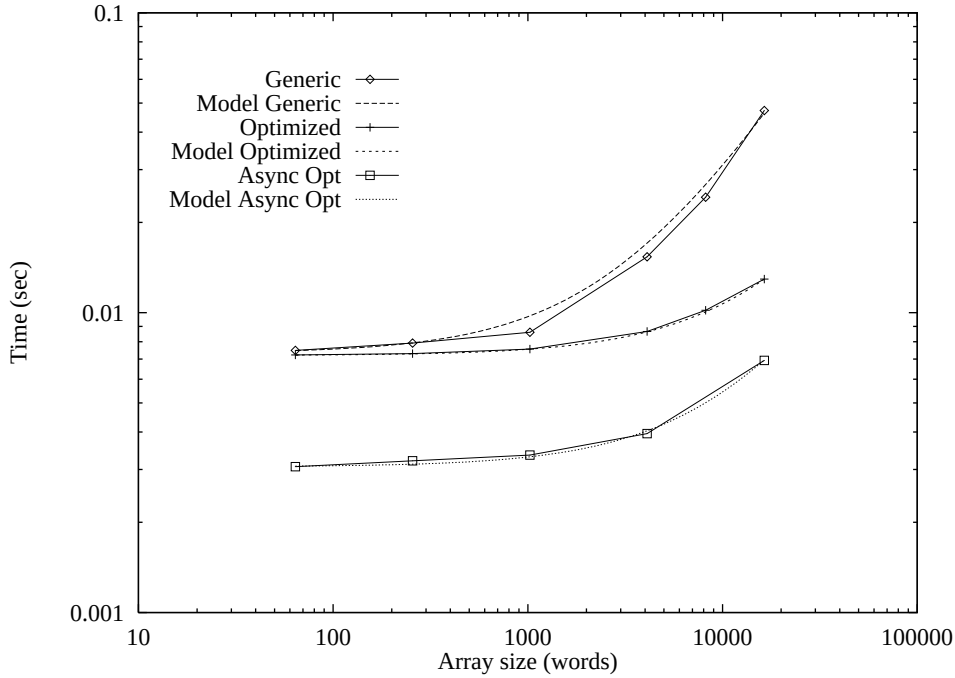

Figure 4: Execution time per data transfer for three different algorithm variants (described in text) with $S = R = 8$, using all-to-all algorithm on 16 IBM SP1 processors

room for additional improvement. The per-message costs are surprisingly high, but are reasonably consistent across algorithmic variants, although somewhat higher in the async algorithm. We are currently investigating the reasons for the high per-message costs.

As would be expected, the async algorithm, which does not perform initialization at each step, is considerably more efficient than the other two algorithms. Note, however, that the cost associated with initialization (17 additional messages) is unrealistically high in our prototype. Using an $\mathcal{O}(\log N)$ broadcast instead of the current $\mathcal{O}(N)$ broadcast would reduce the number of additional messages to $1 + \log_2 N = 4$ in this case, making the cost of initialization negligible for all but small arrays and small numbers of processors.

3.2 All-to-All Vs. Centralized

In our second experiment, we compare the performance of the all-to-all and centralized algorithms. This comparison is performed using the Ethernet interconnect on the SP1, an environment in which we suspect that bandwidth limitations may be significant. Results are presented in Figure 5. We find that the all-to-all algorithm is always faster; it appears that bandwidth limitations are not an issue, at least in this problem size regime.

Figure 5 gives in addition to the observed data, fits to two simple performance models. The all-to-all model is Equation 1, with a fit of $t_m = 1230$ μsec and $t_w = 41$ μsec; the centralized model is the same but with the data volume term scaled by the number of processors, to reflect the fact that data communication is performed sequentially in this algorithm. The latter model gives a reasonable fit but clearly does not account for all aspects of algorithm behavior.

3.3 Other Problem Sizes

Finally, we present results using the generic all-to-all algorithm for two other configurations: $S = 8/R = 4$ and $S = 4/R = 2$. Performance results are in Figure 6, along with results predicted by Equation 1 with $M = 2S + R + 1$ and using the t_m and t_w values obtained previously by the fit to the

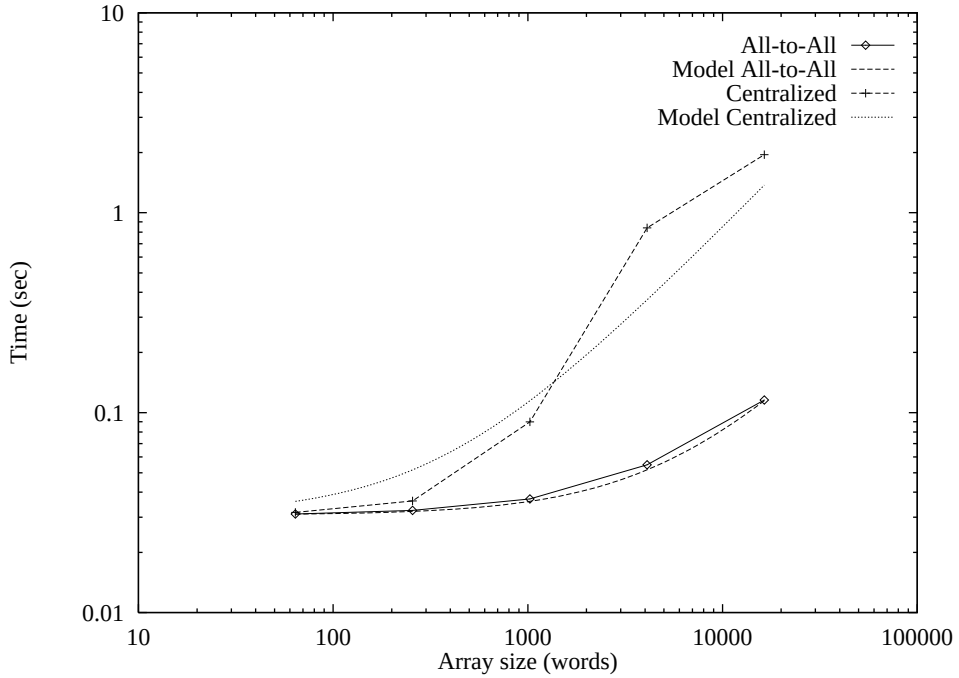

Figure 5: Execution time per data transfer for all-to-all and centralized algorithm on Ethernet-connect RS6000 workstations, with $S = R = 8$.

generic algorithm. We get an good fit to the model and see that, as should be expected, the $S = 4/R = 2$ data transfer is less costly for small arrays (since there are fewer messages) but more costly for larger arrays (since each processor transfers more data). Note that the crossover point in this graph would be at a lower value of D if a more efficient initialization algorithm were used.

3.4 Other Applications

In addition to the synthetic examples described in this section, we have used our library to implement several more substantial programs, some based on a benchmark suite from CMU [13]. For example, a two-dimensional (2-D) fast Fourier transform (FFT) may be implemented as a purely data-parallel program or as two communicating data-parallel tasks, each responsible for performing 1-D FFTs in one direction. The second implementation gives superior performance in many circumstances. Space does not permit the presentation of performance results in this paper.

4 Conclusions

A general-purpose library for data transfer between communicating data-parallel tasks would be of considerable utility in many applications. In this paper, we have presented a design for such a library, outlined the algorithms that it might contain, described an approach to its implementation, and presented performance results from a prototype implementation. In future work, we will extend the range of algorithms incorporated in our framework, tune and characterize the performance of these algorithms on different architectures, and incorporate the framework into an HPF compiler augmented with task-parallel constructs. We are particularly interested in investigating performance on larger numbers of processors and in providing specialized data transfer algorithms for other common redistributions. Another interesting direction for further work is the development of specialized network protocols for the types of data transfer considered in this paper. For example, Turner et al. [16] describe specialized network protocols for the execution of data-parallel tasks on workstation networks.

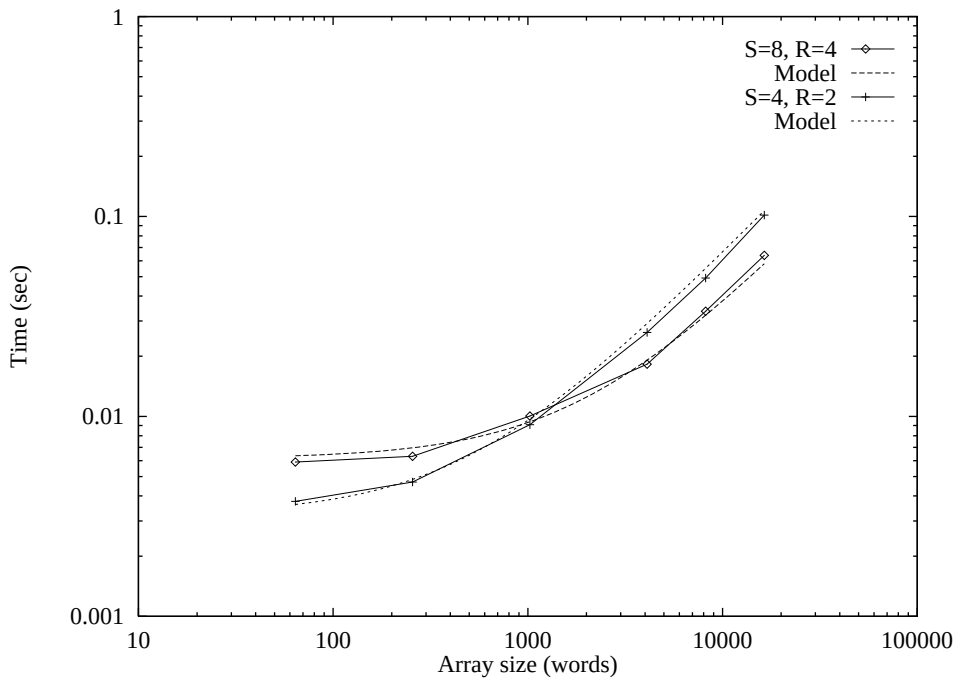

Figure 6: Execution time per data transfer for the all-to-all algorithm on IBM SP1 for two different configurations.

Acknowledgments

This work was supported by the National Science Foundation's Center for Research in Parallel Computation under Contract CCR-8809615; by the Office of Scientific Computing, U.S. Department of Energy, under Contract W-31-109-Eng-38; and by ARPA under Contract DABT63-91-C-0028. Alok Choudhary is supported by NSF Young Investigator Award CCR-9357840. We are grateful to other members of the Center for Research on Parallel Computation, particularly Mani Chandy, Carl Kesselman, Robert Olson, and Steven Tuecke, for their contributions to this work. We also acknowledge helpful discussions with Marc Snir.

References

[1] Z. Bozkus, A. Choudhary, G. Fox, T. Haupt, and S. Ranka, Fortran 90D/HPF compiler for distributed memory MIMD computers: Design, implementation, and performance results, *Proc. Supercomputing '93*, IEEE, 1993.

[2] B. Chapman, P. Mehrotra, and H. Zima, Vienna Fortran — A Fortran language extension for distributed memory systems, *Languages, Compilers, and Run-time Environments for Distributed Memory Machines*, Elsevier Press, 1992.

[3] J. del Rosario and A. Choudhary, High-performance I/O for parallel computers: Problems and prospects, *IEEE Computer*, 27(3), 59–68, 1994.

[4] A. Edelman, Optimal matrix transposition and bit reversal on hypercubes: All-to-all personalized communication, *J. Par. Dist. Comp.*, 11, 328–331, 1991.

[5] J. O. Eklundh, A fast computer method for matrix transposing, *IEEE Trans. Comput.*, C-21, 801–803, 1972.

[6] I. Foster and K. M. Chandy. Fortran M: A language for modular parallel programming. *J. Parallel and Distributed Computing* (to appear), and Preprint MCS-P327-0992, Argonne National Laboratory, 1992.

[7] I. Foster, B. Avalani, A. Choudhary, and M. Xu. A compilation system that integrates High Performance Fortran and Fortran M, *Proc. Scalable High Performance Computing Conf.*, IEEE, 1994.

[8] G. Fox, S. Hiranandani, K. Kennedy, C. Koelbel, U. Kremer, C. Tseng, and M. Wu, Fortran D language specification, Technical Report TR90-141, Department of Computer Science, Rice University, Houston, Texas, 1990.

[9] P. Hatcher and M. Quinn. *Data-parallel Programming on MIMD Computers.* The MIT Press, Cambridge, Mass., 1991.

[10] High Performance Fortran Forum. High Performance Fortran language specification, version 1.0. Technical Report CRPC-TR92225, Center for Research on Parallel Computation, Rice University, Houston, Texas, January 1993.

[11] A. Malony, B. Mohr, P. Beckman, D. Gannon, S. Yang, F. Bodin, and S. Kesavan, Implementing a parallel C++ runtime system for scalable parallel systems, *Proc. Supercomputing '93*, IEEE, 1993.

[12] S. Ramaswamy and P. Banerjee, Automatic generation of efficient array redistribution routines for distributed memory multicomputers, Technical Report CRHC-94-09, University of Illinois, 1994.

[13] J. Subhlok, J. Stichnoth, D. O'Hallaron, and T. Gross, Exploiting task and data parallelism on a multicomputer, *Proc. 4th ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming*, ACM, 1993.

[14] S. S. Takkella and S. R. Seidel, Complete exchange and broadcast algorithms for meshes, *Proc. Scalable High Performance Computing Conf.*, IEEE, 1994.

[15] R. Thakur, A. Choudhary and G. Fox, Runtime array redistributions in HPF Programs, *Proc. Scalable High Performance Computing Conf.*, IEEE, 309–316.

[16] C. J. Turner, D. Mosberger, and L. L. Peterson. Cluster-C*: Understanding the performance limits. In Proceedings of the Scalable High Performance Computing Conference, May 1994.