Toward Supporting Data Parallel Programming on Clusters of Symmetric Multiprocessors

Chia-Lien Chiang
Dept. Computer Science
National Chung Cheng Univ.
Chia-Yi, Taiwan

Jan-Jan Wu
Institute of Information Science
Academia Sinica
Taipei, Taiwan

Nai-Wei Lin
Dept. Computer Science
National Chung Cheng Univ.
Chia-Yi, Taiwan

Abstract

This paper reports the design of a runtime library for data-parallel programming on clusters of symmetric multiprocessors (SMP clusters). Our design algorithms exploit a hybrid methodology which maps directly to the underlying hierarical memory system in SMP clusters, by combining two styles of programming methodologies – threads (shared memory programming) within a SMP node and message passing between SMP nodes. This hybrid approach has been used in the implementation of a library for collective communications. The prototype library is implemented based on standard interfaces for threads (pthread) and message passing (MPI). Experimental results on a cluster of Sun UltraSparc-II workstations are reported.

1. Introduction

High performance parallel computers incorporate proprietary interconnection networks allowing low-latency, high bandwidth interprocessor communications. A large number of scientific computing applications have benefit from such parallel computers. Nowadays, the current level of hardware performance that is found in high-performance workstations, together with advances in distributed systems architecture, makes clusters of workstations (so called NOW systems) one of the highest performance and most cost effective approaches to computing. High-performance workstation clusters can been realistically used for a variety of applications to replace vector supercomputers and massively parallel computers.

In recent years, we have seen a number of such workstations (or PCs) introduced into the market (e.g. the SGI PowerChallenge, Sun Ultra Enterprise Series, DEC AlphaServers, IBM RS6000 with multiple CPUs, and Intel Pentium with two or four CPUs.) The largest commercially available SMPs today (e.g. SGI PowerChallenge with 32 processors) can deliver peak rates of over one billion floating point operations per second; this is projected to increase by over an order of magnitude within the next few years. Such computing power will make this class of machines an attractive choice for solving large problems. It is predicted that SMPs will be the basis of future workstation clusters [12].

Processors within a SMP share a common memory, while each SMP has its own memory which is not directly accessible from processors resided on other SMPs. The performance of a distributed memory computer depends to a large extent on how fast data movement can be performed. Despite significant improvement in hardware technology, the improvements in communication cost are still behind those in the computation power of each processing node. As a result, optimization for communication remains an important issue for SMP clusters. In existing literatures, a large number of specialized, efficient communication algorithms have been devised to facilitate direct programming of massively parallel, distributed memory computers (e.g. [2, 3, 9, 8, 11, 14]). However, none of these algorithms take multiple levels of memory hierarchy , which is a critical factor for performance on SMP clusters, into consideration.

With a number of implementations of MPI and PVM becoming available, parallel programming on workstation clusters has become an easier task. Although MPI and PVM can also be used to manage communication in SMP clusters, they are not necessarily the most efficient methodologies. Our approach exploits a hybrid methodology which maps directly to the underlying hierarical memory system in SMP clusters, by combining two styles of programming methodologies – threads (shared memory programming) within a SMP node and message passing between SMP nodes.

This paper reports the algorithmic design of a communication library based on this hybrid methodology. The library collects a set of frequently occurring data motions in data-parallel algorithms. Among them are those supporting intrinsic operations defined in HPF-like languages, such as *broadcast*, *reduction*, *gather/scatter*, *matrix transpose*, and *uniform shift*. Some others may not be explicitly defined in the language, but are useful for converting data layouts between program modules, such as conversion between column distribution and row distribution, and conversion between block distribution and cyclic distribution.

Our design goals center on high portability, based on existing standards for lightweight threads and message-passing systems, and high efficiency, based on the utility of threads to improve efficiency of data movement. The library is intended to be used to support direct programming or as a runtime support library for high-level data-parallel languages (e.g. HPF) targeting for SMP clusters. We have implemented a lightweight prototype communication library that supports the above data movement operations. Our experimental results on a cluster of Sun UltraSparc-II dual-CPU workstations demonstrate performance improvement against non-threaded implementation.

The rest of the paper is organized as follows. Section 2 describes the execution model and the hybrid programming methodology for SMP clusters. Section 3 presents some of the collective communication primitives and the set of computation primitives we have designed to facilitate multithreaded data-parallel programming. Section 4 reports our preliminary results on the Sun SMP cluster. Section 5 reviews related work and Section 6 concludes.

2. Overview

An SMP cluster consists of a number of SMP nodes interconnected by a communication network. Each SMP node contains a number of identical processors, each equiped with its own on-chip cache (L1) and an off-chip cache (L2). Processors within a SMP node have uniform access to a shared memory (Figure 1).

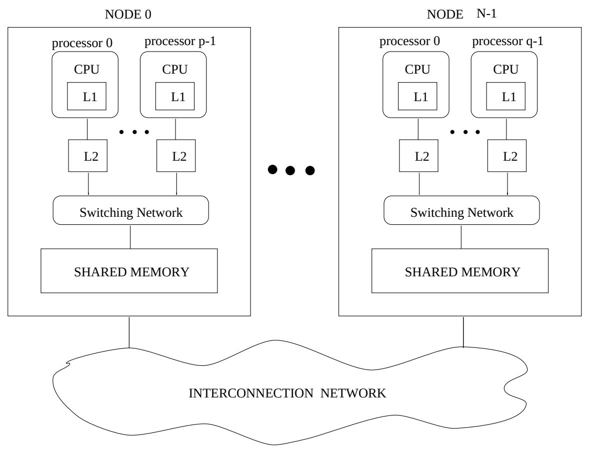


Figure 1. Architecture of a SMP cluster

Lightweight Threads

Threads are commonly used in support of parallelism and asynchronous control in applications and language implementations. In an attempt to provide a standard interface, the POSIX committee has defined a standard threads interface.

Threads are often classified by their "weight", which corresponds to the amount of information that needs be saved/restored when they are context switched. User-level threads are of lighter weight than operating system threads (at least an order of magnitude better). For data parallel programming, threads are most commonly used in two ways:

1. For speeding up computation. For data parallel programming, parallelism is achieved by distributing the program data over a set of processors, where each processor computes its own portion of the data independent of the others. Threads can be used to implement this style of programming. This requires that each thread maintain its own address space (that is, the global data structure needs be separated into separate regions for each thread), something typically not supported by underlying thread package. We tackle this problem by designing a library of computation primitives for multithreaded data parallel programming.

2. For hiding communication latency. Remote memory access in a parallel computer is typically a long-latency operation in comparision with local memory access. Therefore, it is highly desirable to hide (if it cannot be avoided) the latency of remote memory reference. This is accomplished by overlapping useful computation with communication.

   In a multithreaded system, a thread is allowed to execute until a remote memory reference has been initiated, at which time the system switches to another ready thread for execution rather than wait idle for the remote memory request to complete. We extend this idea further in the implementation of a library for collective communications. The library utilizes threads to improve the performance of collective communication on a SMP cluster.

Message Passing Systems

Communication systems for distributed-memory architectures have traditionally been provided by the hardware vendors. Over the past few years, several communication libraries that provide portable interfaces have been established (e.g. PVM and P4). Then, in an effort to encourage vendors to support a single message-passing interface, the

Message Passing Interface Forumn was established to standardize interface for message passing libraries. The result is the Message Passing Interface (MPI).

Model of Execution

Our model of loosely synchronous execution for SMP clusters is depicted in Figure 2. When assigned a task, each SMP node creates a initial thread for housekeeping node execution. The initial thread then launches a number of work threads, each assigned to a processor within that node, that execute in parallel and coordinate with each other via the shared memory (computation phase). When remote accesses to other SMP nodes are required, the work threads may synchronize and enter the communication phase. When the communication is completed, all the threads proceed to next computation phase, and so on.

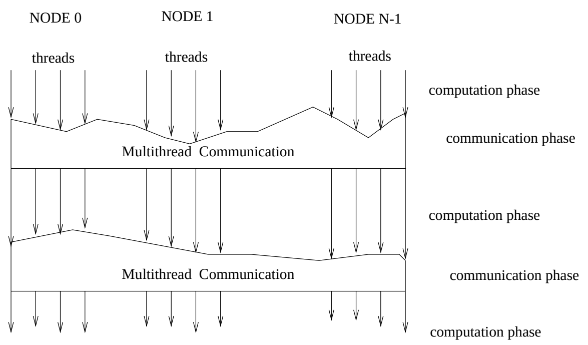


Figure 2. Loosely synchronous execution model for SMP cluster

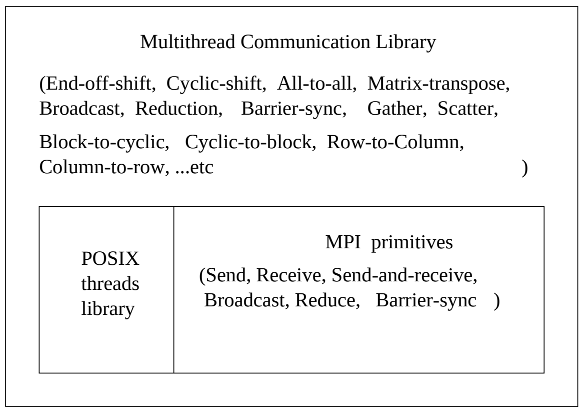

Multithread Communication Library
(End-off-shift, Cyclic-shift, All-to-all, Matrix-transpose, Broadcast, Reduction, Barrier-sync, Gather, Scatter,
Block-to-cyclic, Cyclic-to-block, Row-to-Column, Column-to-row, ...etc)
POSIX threads library
MPI primitives
(Send, Receive, Send-and-receive, Broadcast, Reduce, Barrier-sync)

Figure 3. Organization of the multithread communication library

Multithread Communication Library

Based on the loosely synchronous execution model, we design a multithread communication library. The same set of threads that participate in parallel computation within a SMP node during the computation phase also work together to optimize the communication phase. We have designed several optimizations for collective communications.

- Use multiple threads to speed up local data movement,
- Hide message latency by overlapping local memory copy and sending/receiving messages, by using different threads for these two tasks,
- Pipeline multiple messages to the network interface by using multiple threads for sending/receiving multiple messages.

Figure 3 shows the organization of the multithread communication library. POSIX threads primitives are used to parallelize local memory copy and message buffering within a SMP node. MPI primitives are used for message passing between SMP nodes.

3. Runtime Primitives

This section presents the communication and computation primitives in our runtime library.

3.1. Communication Primitives

Currently, the library supports a set of global operations (barrier synchronization, broadcast, reduction), collective communication (shift, transpose, gather, scatter), and data redistribution (block-cyclic conversion, distribution-dimension permutation), for multithread environment. This section describes the algorithms for some of the primitives.

3.1.1 Broadcast and Reduction

A reduction operation on a SMP cluster is carried out in two steps: shared-memory reduction among multiple threads within a SMP node, and then distributed-memory reduction between SMP nodes.

```
Algorithm reduce(op, source, target, ndata)
op is a binary associative operator,
source is the data array on which the reduction is
to be performed, target stores the reduction result,

{
  Assuming  k threads will be used for the reduction
  on a SMP node, and the data elements are evenly
  partitioned to these k threads.
```

```
 1. For each thread do
    /* computes the reduction on the section of the
      data array that the particular thread is
      responsible for, and store the result to a
      local buffer for this thread.
    */

    local-memory-reduce(op,
        starting_addr_in_source_for_my_thread,
        local-buffer-for-my-thread, ndata/k);

2. Combine the partial results from these threads.
   If k is small, the partial results can be combined
   using a simple locking mechanism shown below:

   Two shared variables are used:
   number-of-threads, initialized with k
   temp: to hold the combined partial result from
         the k threads

   For each thread do
      mutual_exclusion_lock

      accumulate local-buffer-for-my-thread to temp
      decrease number-of-threads by 1
      if number-of-threads=0 (i.e. the last thread)
      then call MPI_Reduce(temp,target,...) to combine
         final results from all these SMP nodes, and
         wake up all the other threads
      else wait for other threads to finish

      mutual_exclusion_unlock
}
```

3.1.2 All-to-All Communication

Alltoall communication is one of the most important collective communication primitives. In all-to-all communication, each SMP node transmit a distinct block of data to every other nodes. The following algorithm performs all-to-all communication with single thread per SMP node.

```
/* single-thread all-to-all communication */
Algorithm alltoall(source,dest,nbytes)

/* each node sends nbytes of distinct data in the
   source buffer to every other node and store the
   received data into corresponding section of the
   dest buffer
*/
{
  /* local memory copy:
    copy nbytes of data from source buffer to dest
    buffer in the section corresponding to my_node */

  1. copy(source+nbytes*my_node, dest+nbytes*my_node);

  /* communicate with other SMP nodes */
  2. for (i=1; i<num_nodes; i++) {
      exchange_node = my_node^i;
      recv_addr = dest + nbytes*exchange_node;
      send_addr = source + nbytes*exchange_node;
      send_and_receive(exchange_node,recv_addr,nbytes,
                     exchange_node,send_addr,nbytes);
  }
}
```

Multithread All-to-all. Suppose there are k CPUs on each SMP node, and k is smaller than the number of SMP nodes (num_nodes), then the all-to-all communication can be further parallelized by using k threads – one thread for the copy statement and $(k-1)$ threads for the communication loop, with each thread handles the communication with $num\_nodes/(k-1)$ SMP nodes. If source and destination are different arrays, then each thread accesses different memory location and thus no memory locking is required. Depending on the characteristics of the target machine, number of threads involved in the all-to-all communication may be tuned to achieve optimal performance.

3.1.3 Matrix Transpose

Matrix transpose is a commonly used operation in data-parallel programming. Assume that the source and the target arrays are two-dimensional n by n arrays, distributed over p nodes. Each node is assigned n/p columns if column partition is used, or n/p rows if row partition is used. A single-thread algorithm is given below.

```
Algorithm transpose(source,dest)
{

   1. locally transpose each n/p by n/p block of data
      in source array,

   2. perform all-to-all communication to reshuffle
      the data, and the received data is placed in
      the destination array,

   3. locally restore memory ordering for the data
      in the destination array.
}
```

Multithread transpose. Use multiple threads for step 1, and use the same number of threads for step 3. Step 2 can be parallelized as described in all-to-all communication. Figure 4 illustrates the three-step procedure for transpose. For clarity of illustration, we only show the data movement for node 0.

3.1.4 Gather/Scatter

We consider Allgather first, in which the data on all the nodes are concatenated together and duplicated to every node. This is very similar to all-to-all personalized communication except that in Allgather each node sends the same data to other nodes.

```
Algorithm Allgather(source,dest,nbytes)
void *source, void *dest;
int nbytes;

/* each node sends nbytes of the same data in
 the source buffer to every other node and store
 the received data into corresponding section
```

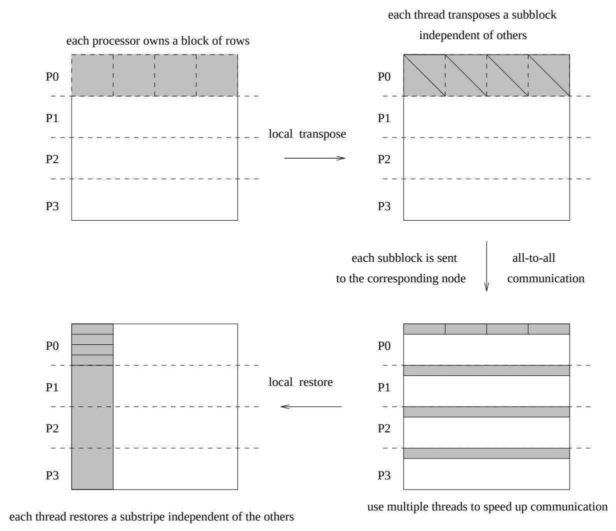

each processor owns a block of rows
each thread transposes a subblock independent of others
P0
P1
P2
P3
local transpose
each subblock is sent to the corresponding node
all-to-all communication
local restore
each thread restores a substripe independent of the others
use multiple threads to speed up communication

Figure 4. Matrix transpose on four nodes (P0-P3), with each node utilizing four threads. For clarity of illustration, this figure only shows data movement for node P0.

```
 of the target buffer */
{

  /* local memory copy:
     copy nbytes of data from source buffer to target
     buffer in the section corresponding to my_node */

  1. copy(source, dest+nbytes*my_node, nbytes);

  /* communicate with other SMP nodes */
  2. for (i=1; i<num_nodes; i++) {
       exchange_node = my_node^i;
       recv_addr = dest + nbytes*exchange_node;
       send_addr = source;
       send_and_receive(exchange_node,recv_addr,nbytes,
                        exchange_node,send_addr,nbytes);
  }
}
```

Multithread Allgather Suppose there are k CPUs on each SMP node, and $k < num\_nodes$. Then the all-to-all communication can be further parallelized by using k threads – one thread for the copy statement and $(k-1)$ threads for the communication loop, with each thread handles the communication with $num\_nodes/(k-1)$ SMP nodes. If source and target are different arrays, then each thread accesses different memory location and so no memory locking is required.

Allscatter performs the inverse function of Allgather. The semantics of Allscatter is identical to all-to-all personalized communication.

3.1.5 Layout Conversion

Row-Column Conversion The conversions between row distribution and column distribution all require all-to-all communication. In addition, it also requires pre- and post-communication memory copying to maintain correct memory ordering. Figure 5(a) and 5(b) illustrate the message passing patterns and local memory storage patterns for conversion between row distribution and column distribution on four nodes.

Block-Cyclic Conversion Conversions between block distribution and cyclic distribution also involve either all-to-all or all-to-some communication, and both pre- and post-communication memory copying. Figure 5(c) shows the message passing patterns and local memory storage patterns for conversion between row block distribution and row cyclic distribution.

Both kinds of layout conversion can be parallelized by using multiple threads within each SMP node, similar to that in the matrix transpose routine.

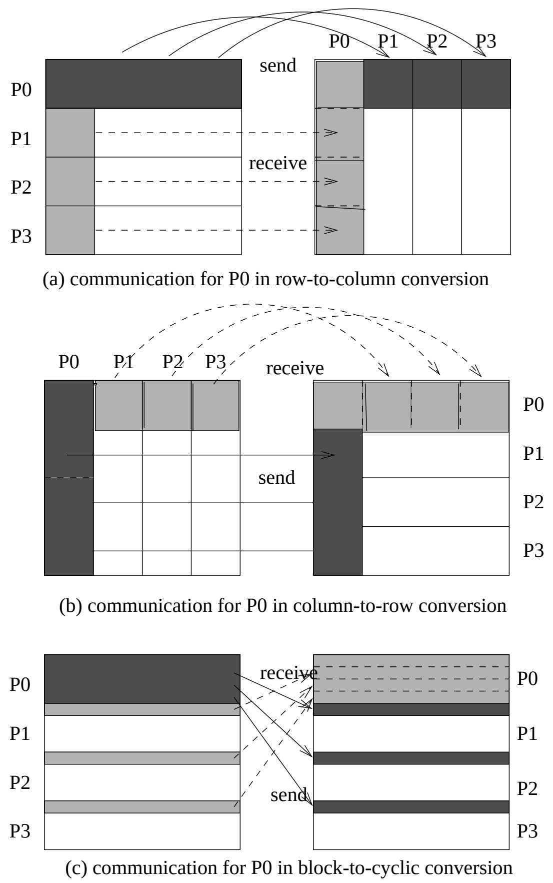

P0
P1
P2
P3
send
receive
(a) communication for P0 in row-to-column conversion
(b) communication for P0 in column-to-row conversion
(c) communication for P0 in block-to-cyclic conversion

Figure 5. Data Layout Conversion on four nodes

3.2. Computation Primitives

The set of computation primitives aim to facilitate data-parallel programming in multithread environment. The difficulty of data-parallel programming with threads lies in the need to maintain separate address space for each thread, to manage the mapping between subscripted data references and thread indices, and to synchronize concurrent accesses to shared resources. Current implemenation of our library supports a set of query functions that can help ease the first two tasks. Management of synchronization is left to the user. Some of the computation primitives are described in the following.

Thread Index. This set of query functions give the total number of threads on current node (`num-of-thread`()) and the rank of current thread among the total number of threads on current node (`thread-rank`()).

Address Space. The query function domain-of-thread(size-of-global-domain, partition-strategy, number-of-thread) gives the range of the subdomain (lower bound, upper bound, and stride for each dimension) that is assigned to current thread under the specified partitioning strategy.

Data Ownership. The query function owner-thread(domain-of-data-references) gives the ranks of the threads whose address spaces intersect with the range of the data references.

3.3. Example: 2D Fast Fourier Transform

Two-dimensional FFT can be implemented by a FFT along one dimension followed by another FFT along the other dimension. An efficient parallel implementation that entirely eliminates communication in the butterfly stages is to partition the data array at the dimension that is parallel to the FFT operation. This is typically done by transposing the data array after performing the first FFT and restoring the data array by another transpose operation. Therefore, a 2D FFT is decomposed into two computation phases and two communication phases.

If the data array is initially partitioned into blocks of rows, then each node will be assigned with n/p rows of data, where n is the data size at one dimension and p is the number of processing nodes. If there are r CPUs on each node, then the computation phase may be further parallelized by using r threads, with each thread computes $n/(p*r)$ rows of data, independent of others. In the communication phase, the r threads carry out the transpose operation as described in the multithread matrix-transpose algorithm. The pseudo code segment is outlined below.

```
/* perform 2d FFT on a n by n array t */
main()
{

 global-domain = the lower and upper bounds of the
 portion of array t that is assigned to current node.

/* create nthr threads, and each thread executes
   fft2d when created */
 for (i=1,nthr) {
   create_thread(fft2d,t)
 }
wait for all threads to complete
}

fft2d()
{

/* determine the address space of current thread */
 my-rows =
  domain-of-thread(global-domain,ROW-PARTITION,nthr)

/* perform 1d fft on my portion of array t, independent
   of other threads and other nodes */
 call fft1d(my-rows,t)

/* transpose array t and save result to array tr,
   this requires communication with other nodes */
 call matrix-transpose(t,tr)

 call fft1d(my-rows,tr)
/* restore to array t */
 call matrix-transpose(tr,t)
}
```

4. Experimental Result

The experiments were conducted on a network of four Sun UltraSparc-II workstations located in the Institute of Information Science, Academia Sinica. The workstations are connected by a fast Ethernet network capable of 100M bps per node. Each workstation is a SMP with two 296MHz CPUs, running the multithread OS SUNOS 5.5.1. The communication library in the framework is implemented on top of the MPICH implementation of MPI.

We compared two implementation approaches, one using the hybrid thread+MPI methodology, and the other using MPI only. For the hybrid version, we created four SPMD processes by the command “mpirun -np 4”, which maps each of the four processes to a distinct workstation. Each process launches two threads, which were then automatically assigned to different CPUs by the Operating System. Threads within a process communicate by shared variables and memory locks, while processes communicate with each other by passing messages. For the MPI-only version, eight processes, each running a single thread, were created using the command “mpirun -np 8”, which assigns each process to a distinct CPU. All processes communicate with each other by passing messages.

Figure 6(a) shows the effectiveness of using multiple threads in global reduction. Using two threads per process improves performance by about 70%. The result also shows that the overhead of shared-memory synchronization in the hybrid approach for summing up partial reduction results from multiple CPUs within a SMP node is negligible compared with the MPI-only implementation.

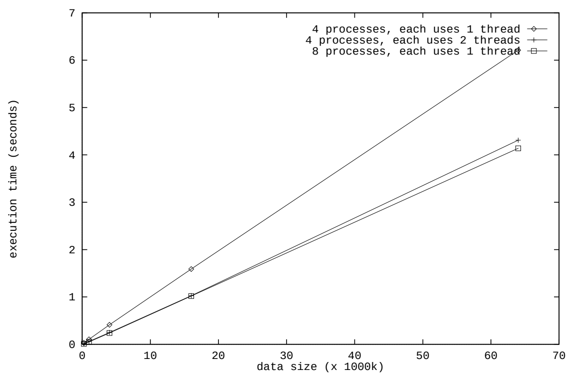
Figure 6(b) reports the execution time of all-to-all communication. Using two threads per process achieves a speedup factor of 2, due to the effect of multithreading in overlapping local memory copying with inter-process communication. The advantage of the hybrid approach over MPI-only implementation is even more significant. This is because by the hybrid approach, each process needs to sends/receives three messages, while the MPI-only implementation requires that each process sends/receives seven messages. We expect more performance improvement on large number of SMP nodes, where the advantage of reducing number of messages will become more evident.

Figure 6(c) reports the execution time of matrix transpose. Using two threads per process achieves a speedup factor of 2. The hybrid approach also outperforms MPI-only implementation, however, the speedup factor does not scale up propotionally with problem sizes. Possible reason is that the hybrid approach partitions data array into smaller number of processes (equivalent to the number of SMP nodes), which results in larger local subarray size in each process, and hence less efficient cache performance for local transpose of the subarray. Nevertheless, this performance penalty can be easily avoided by tiling the loop into a number of smaller ones.

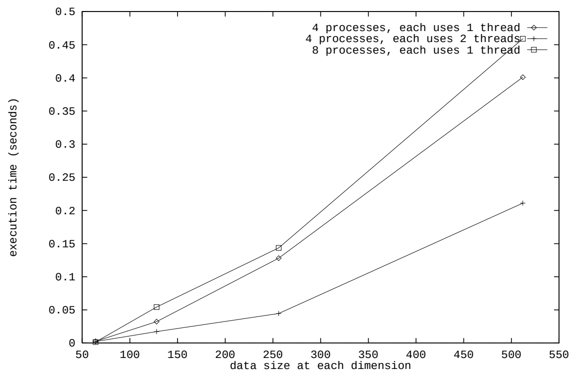
5. Related Work

A number of thread package support interprocessor communication mechanisms. Nexus [5] provides a thread abstraction that is capable of interprocessor communication in the form of Active Messages (asynchronous remote procedure calls). Nexus has not supported either point-to-point communication or collective communication. Panda [1] has similar goals and approaches to Nexus. It also supports distributed threads. Chant [6] is a thread-based communication library which supports both point-to-point primitives and remote procedure calls. Ropes [7] extends Chant by introducing the notion of "context", a scoping mechanism for threads, and providing group communication between threads in a specific context. Some simple kinds of collective operations (e.g. barrier synchronization, broadcast) are supported. The MPC++ Multithread Template Library on MPI [10] is a set of templates and classes to support multithreaded parallel programming. It includes support for multi-threaded local/remote function invocation, reduction, and barrier synchronization.

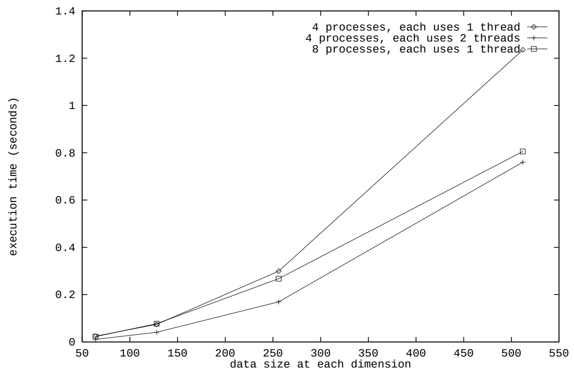
On the part of Message Passing Interface, Skjellum etcl.



(a) global reduction



(b) all-to-all communication



(c) matrix transpose

Figure 6. Execution time on a cluster of four dual-cpu workstations. Compare three versions: hybrid thread+MPI with single thread per process, hybrid thread+MPI with two threads per process, and MPI-only (8 processes, no multithreading).

[4] addresses the issue of making MPI thread-safe with respect to internal worker threads designed to improve the efficiency of MPI. Later, the same group propose many possible extensions to the MPI standard, which allows user-level threads to be mixed with messages [16, 15].

Just recently, there are some new results from the Industry. For example, new version of HP MPI (released in late June this year) [13] incorporates support for multithreaded MPI processes, in the form of a thread-compliant library. This library allows users to freely mix POSIX threads and messages. However, it does not perform optimization for collective communication like we do.

6. Conclusion

In this paper, we have presented a hybrid threads and message-passing methodology for programming SMP clusters. We have also described the implementation of a set of collective communications based on this methodology.

Our experimental results for some collective communication routines on a cluster of four Sun UltraSparc-II workstations demonstrate performance advantage of the hybrid methodology against their MPI-only counterpart. We expect more performance improvement on larger systems, where the effect of utilizing multiple threads to either avoid or hide message latency will become more significant.

Although some trade-off between cache performance and inter-process communication latency due to the hybrid approach was observed in the experiemental result for matrix transpose, we argue that the performance penalty can be avoided by standard optimization techniques such as loop tiling.

As threads are commonly used in support of parallelism and asynchronous control in applications and language implementations, we believe that the hybrid threads/message-passing methodology has advantage of both flexibility and efficiency.

References

[1] R. Bhoedjang, T. Ruhl, R. Hofman, K. Langendoen, H. Bal, and F. Kaashoek. Panda: A portable platform to support parallel programming languages. In *Symposium on Experiences with Distributed and Multiprocessor Systems*, San Diego, CA, 1993.

[2] S. H. Bokhari. Complete Exchange on The iPSC-860. Technical report, ICASE, NASA Langley Research Center, 1991.

[3] S. H. Bokhari. Multiphase Complete Exchange on A Circuit Switched Hypercube. Technical report, ICASE, NASA Langley Research Center, 1991.

[4] A. K. Chowdappa, A. Skjellum, and N. E. Doss. Thread-safe message passing with p4 and mpi. Technical Report TR-CS-941025, Computer Science Department and NSF Engineering Research Center, Mississippi State University, April 1994.

[5] I. Foster, C. Kesselman, R. Olson, and S. Tuecke. Nexus: An interoperability layer for parallel and distributed computer systems. Technical report, Argonne National Labs, Dec. 1993.

[6] M. Haines, D. Cronk, and P. Mehrotra. On the design of Chant: a talking threads package. In *Proceedings of Supercomputing'94*, 1994.

[7] M. Haines, P. Mehrotra, and D. Cronk. ROPES: Support for collective operations among distributed threads. Technical report, ICASE, NASA Langley Research Center, May 1995.

[8] C.-T. Ho and S. L. Johnsson. Optimal Algorithms for Stable Dimension Permutations on Boolean Cubes. In *The Third Conference on Hypercube Concurrent Computers and Applications*, pages 725–736. ACM, 1988.

[9] C.-T. Ho and S. L. Johnsson. The Complexity of Reshaping Arrays on Boolean Cubes. In *proceedings of 5th Distributed Memory Computing Conference*, 1990.

[10] Y. Ishikawa. Multithread template library in c++. http://www.rwcp.or.jp/lab/pdslab/dist.

[11] S. L. Johnsson and C.-T. Ho. Spanning graphs for optimum broadcasting and personalized communication in hypercubes. *IEEE Trans. Computers,* 38(9):1249–1268, September 1989.

[12] W. E. Johnston. The case for using commercial symmetric multiprocessors as supercomputers. Technical report, Information and Computing Sciences Division, Lawrence Berkeley National Laboratory, 1997.

[13] H. Packer. Hp message passing interface. http://www.hp.com/go/mpi.

[14] T. Schmiermund and S. R. Seidel. A Communication Model for the Intel iPSC/2. Technical report, Michigan Technological University, 1990.

[15] A. Skjellum, N. E. Doss, K. Viswanathan, A. Chowdappa, and P. V. Bangalore. Extending the Message Passing Interface (MPI). Technical report, Computer Science Department and NSF Engineering Research Center, Mississippi State University, 1997.

[16] A. Skjellum, B. Protopopov, and S. Hebert. A Thread Taxonomy for MPI. Technical report, Computer Science Department and NSF Engineering Research Center, Mississippi State University, 1996.