A Hybrid Multithreading/Message-Passing Approach for Solving Irregular Problems on SMP Clusters

Jan-Jan Wu

Institute of Information Science
Academia Sinica
Taipei, Taiwan, R.O.C.

Chia-Lien Chiang Nai-Wei Lin

Dept. Computer Science
National Chung Cheng Univ.
Chia-Yi, Taiwan, R.O.C.

Abstract *This paper reports the design of a runtime library for solving irregularly structured problems 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 runtime library supporting the* inspector/executor *model of execution for solving irregular problems. Experimental results on a cluster of Sun UltraSparc-II workstations are reported.*

Keywords: data-parallel programming, workstation clusters, symmetric multiprocessor, multithreaded message-passing, collective communication

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 [9].

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.

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.

Our previous experiences with the hybrid methodology for regular data-parallel computation (i.e. where reference patterns of each data element are uniform and predictable) are reported in [3]. In this paper, we extend our previous work to support irregular computations, where reference patterns may depend on input data or runtime-computed values. This paper gives an overview of the hybrid methodology for solving irregular problems and reports the algorithmic design of a runtime library based on this hybrid methodology. The library collects a set of computation and communication primitives that form the runtime kernel for supporting the *inspector/executor* model of execution on distributed-memory multiprocessors.

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 languages (e.g. HPF2) targeting for SMP clusters. Our preliminary experimental results on a cluster of Sun UltraSparc-II dual-CPU workstations demonstrate performance improvement of the hybrid methodology against message-passing implementation.

2 Overview

Preprocessing for Irregular Computation

The most commonly used model for explicit data-parallel programming on distributed-memory multiprocessors is the *Single Program Multiple Data* model, where data are distributed among available processors, and the same program is executed concurrently on the processors with each processor only responsible for the computation of a portion of the data.

For cases where data are uniformly distributed, the location of an data element can be determined statically. In many situatations dynamic and irregular data distribution is required to obtain load balancing and data locality. It is not possible to express irregular data distribution in a simple way. The mapping of data is usually described by a *distributed mapping table* defined by a partitioning algorithm.

During the preprocessing or *inspector* phase, we carry out a set of interprocessor communications that allows us to anticipate exactly which send and receive communication calls each processor must execute [1]. Once preprocessing is completed, we are in a position to carry out the problem's communication and computation; this phase is called the *executor* [1]. The exchange of interprocessor distributed array elements data follows the plains established by the inspector. Off-processor data are appended to the end of the local array. Once a communication phase is over, each processor carries out its computation entirely on the locally stored data without further communication. Writes to off-processor data elements are also appended to the end of the local array. When the computational phase is finished, array elements to be stored off-processor are obtained from the end of the local array and sent to the appropriate off-processor locations.

We use the Parti library (single thread on each process) as the basis for our multithreaded implementation. The Parti/Chaos runtime library developed by [1, 11] aims to support the inspector/executor style of programming using

arrays as the main data structures. The Parti runtime primitives include communication routines designed to support irregular data distribution and irregular patterns of distributed array access (Figure 3). Figure 1(a) shows the data-parallel code segment for a sparse matrix-vector multiply. Figure 1(b) depicts the SPMD implementation with the Parti library.

In the inspector phase, `index-translation` translates the indirect data reference subscripts (given by array `col`) to pairs of process index and local index (`Proc`, `Loc`). `build-schedule` determines which process should send/receive which data elements to/from which processes (called communication schedule). In the executor phase, off-processor data are gathered from remote processors according to the planned communication schedule and stored at the end of the local array `x`. The local index array `Loc` is updated accordingly to reflect the actual location of the remotely fetched data in local array `x`. Then the two nested loops carry out the matrix-vector multiply entirely on the local array `x`.

```
(a) data-parallel program
do i=1, NRows
 sum = 0
 do j=1,ncols(i)
     sum = sum + f(j,i)*x(cols(j,i))
 enddo
 y(i) = y(i) + sum
enddo

(b) SPMD program
!! the inspector phase
call index-translation(Proc,Loc,cols)
sched = build-schedule(Proc,Loc)
!! the executor phase
call gather(x,x,sched,Loc)
count=0
do i=1, LocNRows
 sum = 0
 do j=1,ncols(i)
   sum = sum + f(j,i)*x(Loc(count))
   count += 1
 end do
 y(i) = y(i) + sum
enddo
```

Figure 1: Inspector/Executor implementation of sparse matrix-vector multiply

Hybrid 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.

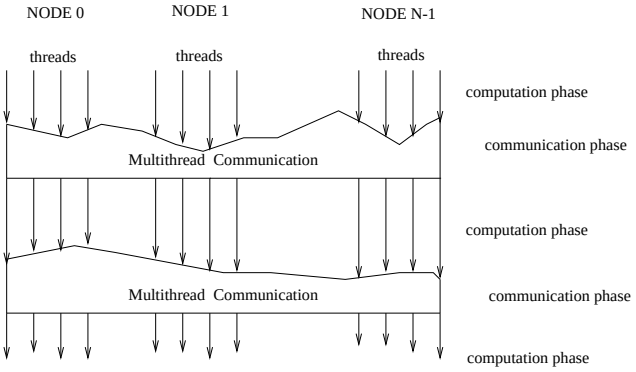


Figure 2: Loosely synchronous execution model for SMP cluster

Multithreaded Runtime Library

Based on the loosely synchronous execution model, we design a multithreaded runtime 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 multithreaded runtime library. POSIX threads primitives are used to parallelize local memory computation and message buffering within a SMP node. MPI primitives are used for message passing between SMP nodes.

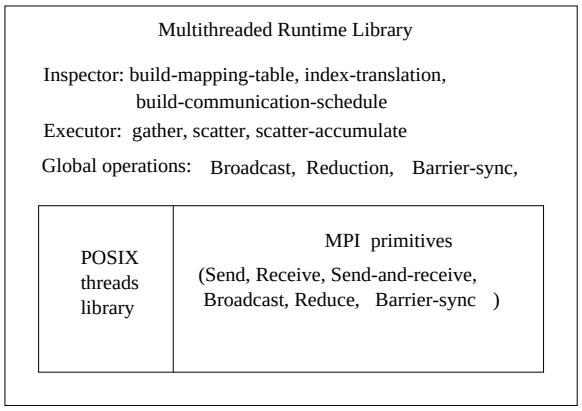


Figure 3: Organization of the multithreaded runtime library

3 Runtime Primitives

Currently, the library supports a set of global operations (barrier synchronization, broadcast, reduction), inspector operations (index translation, build communication schedule) and collective communication (gather, scatter, scatter-accumulate), which alltogether form the runtime kernel for multithreaded execution of the *inspector/executor* model. This section describes the algorithms for some of the primitives.

3.0.1 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:

   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
}
```

3.0.2 Gather/Scatter Communication

Data movement, such as gather and scatter, in irregular computation can all be formulated as *all-to-some* personalized communication, in which each SMP node transmits a distinct block of data to a subset of SMP nodes. The following algorithm performs all-to-some gather communication with single thread per SMP node. The data structure "sched" describes the communication schedule (including the number of data elements and the local indices of the data elements to send/receive) on this node. A node sends $sched.n\_data\_to\_send[j]$ of data elements in the source array, indexed by $sched.n\_send\_list[i]$, to node j and store the received data into proper locations in the target array.

```
Algorithm gather(source,dest,sched)
void *source, void *dest;
SCHEDULE *sched;
{
  /* send data */
  for(i=0;i<num_nodes;i++){
    if(sched.n_data_to_send[i]>0) {
      copy the requested data elements from the
      source array to ith section of the buffer,
      and send the buffer data to node i
  }}

  /* receive data */
  for(i=0;i<sched.n_msgs_to_recv;i++){
   poll for incoming message,
   copy the received data to proper locations
   in target array
 }
}
```

Multithreaded Gather Suppose there are k CPUs on each SMP node, and k is smaller than the number of SMP nodes (num_nodes), then the gather communication can be further parallelized by using k threads – each thread is responsible for the execution of num_nodes/k send-loop iterations. That is, each thread handles the communicationm with num_nodes/k SMP nodes. Multithreading also parallelizes local memory copy to message buffers. 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 gather communication may be tuned to achieve optimal performance.

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 4(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.

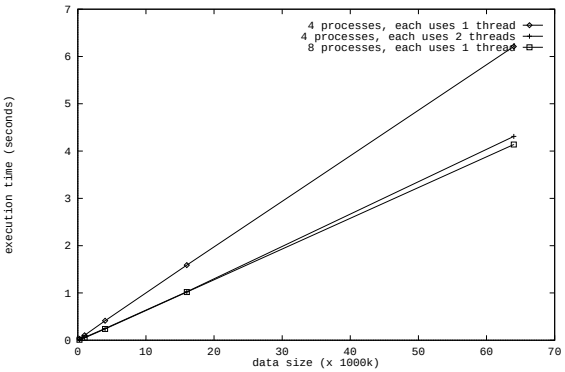
Figure 4(b) reports the execution time of all-to-some gather communication. Using two threads per process achieves a speedup factor

of 1.5, due to the effect of multithreading in overlapping local memory copying with interprocess 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 less messages than the MPI-only implementation. We expect more performance improvement on large number of SMP nodes, where the advantage of reducing number of messages will become more evident.

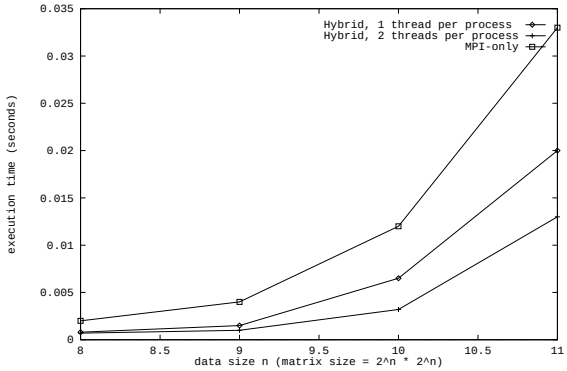
Figure 4(c) shows the execution time of the sparse matrix-vector multiply by partitioning the sparse matrix into blocks of rows. Using two threads per process shortens the communication time for both the inspector phase and the executor phase. The computation time is competitive to that of the MPI-only implementation, resulting in overall speedup factor of 1.5.

5 Related Work

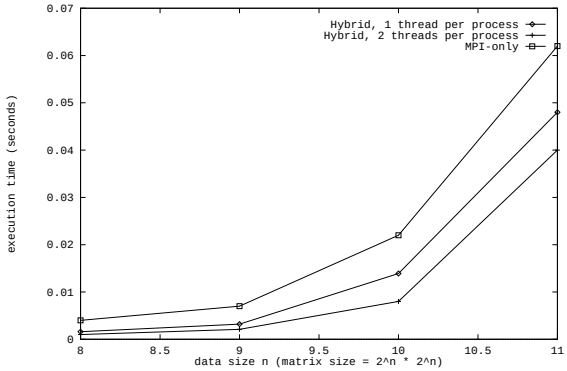
A number of thread package support interprocessor communication mechanisms. Nexus [5] provides a thread abstraction that is capable of interprocessor communication in the form of Active Messages (asynchronous remote procedure calls). Nexus has not supported either point-to-point communication or collective communication. Panda [2] 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 [8] 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 syn-



(a) global reduction



(b) all-to-some communication



(c) sparse matrix-vector multiply

Figure 4: Execution time of some collective communications 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).

chronization.

On the part of Message Passing Interface, Skjellum etcl. [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 [13, 12].

Just recently, there are some new results from the Industry. For example, new version of HP MPI (released in late June this year) [10] 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 present a hybrid threads and message-passing methodology for solving irregular problems on SMP clusters. We have also described the implementation of a set of collective communications based on this methodology.

Our preliminary experimental results on a network of Sun UltraSparc-II shows that the hybrid methodology improves performance of global reduction by about 70% and the performance of all-to-some gather by about 80%, compared with MPI-only, non-threaded implementations.

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.

Support for this work is provided by National Science Council of Taiwan under grant 88-2213-E-001-004.

References

[1] H. Berryman, J. Saltz, and J. Scroggs. Execution time support for adaptive scientific algorithms on distributed memory architectures. *Concurrency: Practice and Experience*, 3(3):159–178, 1991.

[2] 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.

[3] C.-L. Chiang, J.-J. Wu, and N.-W. Lin. Toward Supporting Data-Parallel Programming for Clusters of Symmetric Multiprocessors. In *International Conference on Parallel and Distributed Systems*, Tainan, Taiwan, 1998.

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

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

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

[11] R. Ponnusamy. A manual for the chaos runtime library. Technical Report TR93-105, Computer Science Dept. Univ. of Maryland, Dec. 1993.

[12] 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.

[13] 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.