Design of OpenMP Compiler for an SMP Cluster

Mitsuhisa Sato, Shigehisa Satoh, Kazuhiro Kusano and Yoshio Tanaka
Real World Computing Partnership, Tsukuba, Ibaraki 305-0032, Japan
E-mail:{msato,sh-sato,kusano,yoshio}@trc.rwcp.or.jp

Abstract

In this paper, we present a design of OpenMP compiler for an SMP cluster. Although clusters of SMPs are expected to be one of the cost-effective parallel computing platforms, both of inter and intra node parallelism must be exploited to achieve high performance. These two levels of structure complicate parallel programming. The OpenMP is an emerging standard for parallel programming on shared-memory multiprocessors. We extend the OpenMP model for an SMP cluster by "compiler-directed" software distributed shared memory system. Our OpenMP compiler instruments an OpenMP program by inserting remote communication primitives to keep consistency of memory between different nodes, and provides a view of shared memory model on the SMP cluster. Different from multithreaded programs on conventional software DSMs, an OpenMP program is so well-structured that it allows the compiler to analyze extent of parallel region for the optimization of efficient communication and synchronization. We report some preliminary results of OpenMP programs on a Pentium Pro based SMP cluster, COMPaS.

1 Introduction

Recent progress in microprocessors and interconnection networks motivates high performance computing using clusters out of commodity hardware. Clusters using fast networks, such as Myricom's Myrinet, can provide high performance equal to MPPs. Symmetric multi-processor systems (SMPs) are also becoming widespread, both as compute servers and as platforms for high performance parallel computing. Clusters of SMPs are expected to be one of the cost-effective parallel computing platforms.

SMP clusters have a mixed configuration of shared memory and distributed memory architecture. One way to program the SMP clusters is *all message passing model*. In this approach, the message passing operation is used even for intra-node communication. It makes parallel programming simple, but may lose the advantage of shared memory in an SMP node. Message passing systems requires programmers to explicitly code the communication and makes writing parallel programs cumbersome.

In SMP clusters, both of inter and intra node parallelism must be exploited to achieve high performance. While multi-threaded programming is often used to take advantage of shared memory within an SMP node, explicit communication is required between nodes. So far, we proposed a *hybrid programming model* of shared memory and distributed memory in order to take advantage of locality in each SMP node. Intra-node computations utilize a multi-threaded programming style and inter-node programming is based on message passing and remote memory operations. We found, however, that two levels of structure of an SMP cluster make programming complicated because the programmer must take care of both programming models.

Another way to program for an SMP cluster is *all shared memory models* using a software distributed shared memory (SDSM) system such as TreadMarks[4] and SCASH[2]. Distributed shared memory systems provide a more elegant and easier to program alternative. However, these systems may suffer from much coherence-maintenance network traffic. The most SDSM systems do not exploit application specific data access pattern to minimize communication, because communication for shared data is unknown until run time.

In this paper, we presents a design of the OpenMP compiler for an SMP cluster. The OpenMP Application Programming Interface (API)[7] is an emerging standard for parallel programming on shared-memory multiprocessors. While OpenMP is originally designed as a programming model for shared memory multiprocessors, we extend the OpenMP model for an SMP cluster by "compiler-directed" distributed shared memory system. Our OpenMP compiler instruments an OpenMP program by inserting remote communication primitives to keep consistency of memory between different nodes. Different from

multithreaded programs on conventional SDSMs, the OpenMP program is so well-structured in parallel programming that it allows the compiler to analyze the extent of parallel region for the optimization of efficient communication and synchronization. For example, the compiler can analyze the loop to hoist out contiguous access to array elements in a remote node inside the loop to a bulk access code outside. Our important goal is to understand how to exploit the performance potential by the compiler optimization.

The rest of this paper is organized as follows. Section 2 describes our experimental prototype OpenMP C compiler, **Omni OpenMP compiler**, and reports some performance results on an SMP node. Section 3 describes a design of OpenMP compiler and the runtime system to support distributed shared memory facility and how to optimize communication by the compiler. We have built a PC-based SMP cluster called COMPaS[9] with 8 nodes 4-processor Intel Pentium-Pro SMP as our platform. Section 4 describes our COMPaS briefly and presents some preliminary results of OpenMP programs on the system. Section 5 concludes this paper with the current status of the project and our future work.

2 The Omni OpenMP Compiler

The Omni OpenMP compiler is an experimental prototype compiler for OpenMP. In this section, we describe the OpenMP API briefly and present the overview of Omni OpenMP compiler as background.

2.1 The OpenMP API

The OpenMP API defines a set of directives that augment standard C/C++ and Fortran 77/90. In contract to previous API such as POSIX threads and MPI, OpenMP facilitates an incremental approach to the parallelization of sequential program. The programmer may add a parallelization directives to loops or statements in the program. OpenMP provides three kinds of directives: parallelism/work sharing, data environment, and synchronization. We only explain the directives relevant to this paper. Refer to the OpenMP standard for the full specification.

OpenMP uses the fork-join model of parallel execution. An OpenMP program begins execution as a single process, called the master thread of execution. The fundamental directive for expressing parallelism is the `parallel` directive. It defines a *parallel region* of the program, that is executed by multiple threads. When the master threads enters a parallel region, it forks a team of t threads (one of them being the master thread), and work is continued in parallel among these threads. Upon exiting the parallel construct, the threads in the team synchronize (join the master), and only the master continues execution. The statements in the parallel region, including functions called from within the enclosed statements, are executed in parallel by each thread in the team. The statements enclosed lexically within a construct define the *static extent* of the construct. The *dynamic extent* further includes the functions called from within the construct.

All threads replicate the execution of the same code, unless a work sharing directives is specified within the parallel region. Work sharing directives, such as `for`, divide the computation among threads. For example, `for` directive (`DO` directive in Fortran) specifies that the iterations of the associated loop should be divided among threads so that each iteration is performed by a single thread. It should be noted that an OpenMP compliant program guarantees that all threads in the team reach and execute the work sharing construct at the same time.

The data environment directives control the sharing of program variables defined outside of a parallel region. Variables default to `shared`, which means shared among all threads in a parallel region. A `private` variable has a copy per thread. The `reduction` directives identifies reduction variables.

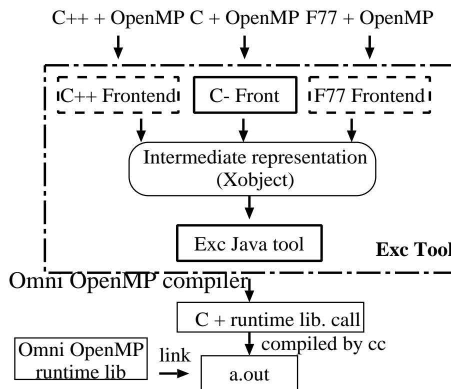

Figure 1: Overview of Omni OpenMP compiler

2.2 Omni OpenMP compiler and Omni Exc toolkit

The compiler is a translator which takes OpenMP programs as input to generate the multithreaded C program with runtime library calls. The generated program is compiled by the native back-end compiler linked with the runtime library. Figure 1 illustrates the overview of the Omni OpenMP compiler.

The Omni Exc toolkit is a set of programs and

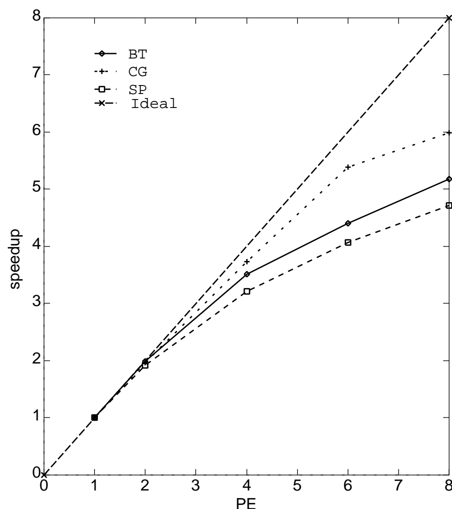

Figure 2: Performance of NPB-1 in Sun SC1000

Java libraries to construct a system that analyzes and transforms a program. C-front is a front-end program that parses a C source code into a intermediate code, called Xobject code. The front-ends for Fortran and C++ is under development. Exc Java tool is a Java class library that provides classes and methods to analyze and modify the program easily with a high level representation, and to unparse the Xobject code into a C program. The representation of Xobject code can be manipulated by the Exc Java tool is a kind of AST (Abstract Syntax Tree) with data type information, which each node is a Java object that represent a syntactical element of the source code, and that can be easily transformed. The Exc java tool includes several class libraries to transform SSA form and perform data flow analysis for optimization. The Omni OpenMP compiler is implemented as a part of the Omni Exc toolkit.

To translate a sequential program annotated with parallel directives into a fork-join parallel program, the compiler encapsulates each parallel region into a separate function, called "parallel region function". The master thread calls the runtime library to invoke the slave threads which executed this function in parallel. Pointers to shared variables with auto storage class are copied into shared memory heap and passed to slaves at the fork. Implementation of private variables is straightforward: it is redeclared in the functions generated by compiler. The work sharing and synchronization constructs are translated into codes which contain the corresponding runtime library calls. This transformation pass is written by Java in the Exc Java tool.

In an SMP, the runtime library uses either POSIX threads and Solaris threads as an operating system thread.

2.3 Performance on SMP

Figure 2 shows the performance of the OpenMP version of NAS parallel benchmark (version 1) on Sun Sparc Center 1000 (8 processors).The problem sizes are class A. We found that the overhead of OpenMP version parallelized by our compiler is less than 10% in CG.

3 OpenMP Design for an SMP cluster

3.1 Compiler-directed Approach

Since the OpenMP is designed for shared memory model, a straightforward approach is to generate multithreaded code on an SDSM system such as TreadMarks[4]. This approach, however, can not exploit application specific data access pattern because all coherence-maintenance communication occurs on demand.

Our approach relies on the runtime DSM library and the compiler analysis of OpenMP programs. In general multithreaded programs by thread library, a compiler cannot analyze the data dependency between threads exactly because the behavior of threads is controlled by the library. The structured parallel description of OpenMP allows the compiler to capture the behavior of threads easily. For an SMP cluster, we extend the OpenMP compiler and runtime library as follows:

- To execute OpenMP program in distributed memory environment, the compiler first insert memory coherence maintenance codes of the base DSM runtime system in every access to shared data. The base DSM runtime system keeps memory consistency between SMP nodes.
- The compiler analyze the memory access pattern to optimize memory coherence code. In data-parallel context in OpenMP programs, efficient collective communication can be generated by compiler analysis.
- The runtime library of thread management also extended in an SMP cluster environment to fork threads in different SMP nodes and synchronize between nodes.

Since the base DSM runtime system is implemented entirely in software, it provides flexibility in the design of cache coherence protocol and opportunity for

optimization of communication with the knowledge by compiler analysis. The SDSM implementation by inserting memory coherence maintenance codes is similar to Shasta[8]. Shasta also attempts to batch together checks for contiguous multiple load and store by the compiler. While the Shasta compiler analysis is limited at assembly level, our compiler can use more higher level knowledge of OpenMP programs.

In ADSM and its compiler optimization[5], a page-based SDSM is used for shared read, and explicit user-level consistency management codes are used for shared write. The compiler analyzes a multi-threaded program and optimizes shared write codes to reduce the overhead. The ADSM system is different from our system because it is based on a page-based SDSM and is targeted for general-purpose multithreaded programs such as SPLASH2 benchmark suite. A compiler-directed SDSM is also proposed in [3].

3.2 Base DSM runtime system

The base DSM runtime system for distributed memory environment maps data space into the fixed same address in each SMP nodes. Static data segment is also allocated in the same virtual address at linking time. Data in the shared space may be cached by multiple nodes at the same time, with copies residing at the same virtual address on each node. The copies are naturally shared between threads in the same SMP node by hardware shared memory mechanism.

The DSM runtime system divides the shared space into a small fixed-size block of memory, called *cache line* (typically 64 bytes or 128 bytes). The shared data in each line has a status bit at each node, which indicates if the shared data in the line is valid or not at the node. Communication is required if a thread attempts to read data that in invalid state. The following primitives, called *check codes* are inserted by the compiler before each load and store to/from the shared data space to keep coherent:

- `check_before_read(p,size)` — check before reading the shared data from the address `p` with its size `size`. If the lines which contain the data are in invalid state, the data is fetched from the home node and turn them into valid state.
- `check_before_write(p,size)` — check before writing the shared data. If the line for the data is not in valid state and a part of the data in the line is to be not actually written, the data is fetched as `check_before_read` because all data in a line must be in the same state and kept coherent as a unit.
- `check_after_write(p,size)` — check after writing the shared data. The base DSM runtime system can support both invalidate protocol and update protocol. In the invalidate protocol, the data is written back to its home node and invalidate the lines in other nodes except the home node by broadcasting the valid flag to them. In the update protocol, the data is broadcasted to every nodes. For simplicity, the directory information is not maintained in the current implementation, and all data are broadcasted.

These primitives can be inlined to reduce the overhead. We use NICAM communication library[6] as a communication layer. NICAM is a simple communication library on our SMP cluster, COMPaS, described in the next section. It supports one-sided remote memory transfer and synchronization between nodes.

3.3 Optimization

To obtain good performance on distributed memory environment, it is important to efficiently manage communication between nodes. Making communication efficient is a major challenge for the compilers. The base DSM runtime system works as a default mechanism if the compiler optimization is failed. The following optimization can be performed by the compiler.

3.3.1 Parallel extent detection

In OpenMP program, only codes in parallel region is executed in parallel. By making call graph, we can recognize the codes in static extent of parallel region and the functions which may be executed in the dynamic extent. The compiler eliminates the check codes outside the parallel region and in the functions which are not executed in the dynamic extent. These code is executed as in its sequential program only in the master thread's nodes.

3.3.2 Redundant check code elimination

In OpenMP programs, we can assume relaxed memory consistency model. This means that shared write may be delayed until a thread reaches at a *flush directive*. The flush directive specifies a "cross-thread" sequence point at which the system ensures that all threads in a team have a consistent view of certain objects. The flush directive ensure that shared writes must be reflected in shared memory. The flush directive is executed implicitly at the barrier synchro-

nization, at the end of work sharing construct and at reference to volatile variables.

A read check code is redundant if the data is already available by the preceding read check at the same location. A write check code is redundant if the succeeding write check at the same data location will be executed. Data-flow analysis is done for each statement in parallel region to determine the earliest possible read check code and the latest possible write check code. To eliminate these redundant code, the compiler computes the following dataflow information on each statement i along with the control flow graph:

- Availability for read checks: In all paths which precede the statement i, the data set is available by the read check or the write check, and is not killed by flush directive. If the data set is not modified at all the path, the data set is not killed by flush directive.
- Anticipatability for write checks: In all paths which succeed the statement i, the data set is modified by write checks before the data set is killed by the flush directive.

After computing these data flow sets, the compiler eliminates the read check in the availability sets and the write checks in the anticipatability sets.

3.3.3 Merging multiple check codes

In scientific applications, array elements are often referenced contiguously in a loop. The check codes for such access can be hoisted out from the loop converting into one check codes for a large block. For example, consider the following loop.

```
for(i = 0; i < n; i++) a[i] = c * b[i];
```

If array `a` and `b` are shared, the compiler inserts check codes.

```
for(i = 0; i < n; i++){
  check_before_read(&b[i],s);
  check_before_write(&a[i],s);
  a[i] = c * b[i];
  check_after_write(&a[i],s);
}
```

Since the loop does not contain any flush directives, the check code can be hoisted out as follows:

```
check_before_read(&b[0],s*n);
check_before_write(&a[0],s*n);
for(i = 0; i < n; i++)  a[i] = c * b[i];
check_after_write(&a[0],s*n);
```

This optimization can reduce the overhead of calling check codes. If the stride of access is a constant, message vectorization is possible in the runtime library.

3.3.4 Data-parallel communication optimization

The optimization on check codes described above can be applied for multithreaded programs. Different from general multithreaded execution model, the OpenMP API guarantee that work sharing directives are executed by all threads in the team. For the codes that is unknown to be executed by all threads, we can use the complication technique to optimize the communication in data parallel languages such as HPF. For example, consider the following OpenMP program.

```
#pragma omp parallel
do {
  ...
#pragma omp for
  for(i = 1; i < n-1; i++)
      y[i] = (x[i+1]+x[i-1])*0.5;
#pragma omp for
  for(i = 1; i < n-1; i++) x[i] = y[i];
 ...
} while(check_convergence());
```

If the compiler determines the data mapping of `x` and `y` to be one-dimensional block mapping, the code can be optimized as follows:

```
... Setup the data of x and y
...    in one-dimensional mapping.
... Compute the loop bounds,
...    lb and ub at the same time.
do {
  ...
  update x[lb-1] and x[ub] in local nodes
  for(i = lb; i < ub; i++)
      y[i] = (x[i+1]+x[i-1])*0.5;
  for(i = lb; i < ub; i++) x[i] = y[i];
 ...
} while(check_convergence());
... Write back x and y if required
```

Because there is no data mapping directives in the OpenMP, the compiler determines the data mapping according to the scheduling of the loop which references the data if the loop is statically scheduled. Exact loop bounds and strides may not be compile time constants in OpenMP because the number of threads is given at runtime. The compiler inserts calling to the data mapping runtime library primitives which determine the loop bounds and data sets to be communicated. For each data set, a data mapping descriptor is

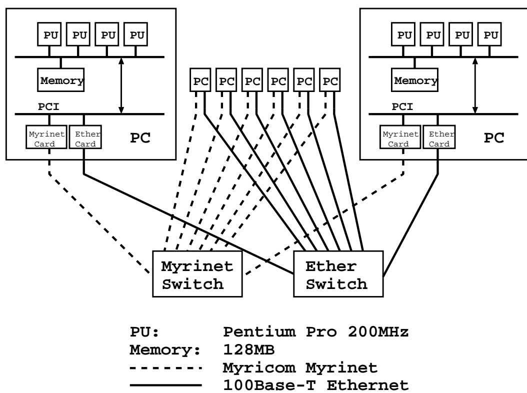

Figure 3: Configuration of COMPaS

allocated at runtime to keep the information of data mapping. When the data mapping and loop scheduling is the same in the preceding loop, the same mapping can be reused. The compiler analyzes the availability of the mapping and scheduling compiled by the previous runtime call to reduce the overhead of calling the runtime library. Agrawal et. al.[1] proposes the interprocedural optimization of runtime scheduling preprocessing.

The all check codes on the data sets managed by the data mapping runtime library are removed because it is guaranteed that the data is available on the node.

3.3.5 Collective communication optimization

In OpenMP, the programmer can specify the reduction variables as data scope attribute in parallel region. The reduction on the variable is done between nodes at the end of parallel regions or work sharing constructs by the collective communication runtime library.

4 Preliminary Evaluation

4.1 COMPaS: a PC-based SMP Cluster

We have built a cluster of SMPs, **COMPaS (Cluster Of Multi-Processor Systems)** with PC-based SMPs. Figure 3 illustrates the configuration of COMPaS. COMPaS consists of eight 4-processor Pentium Pro PC servers (Toshiba GS700, 450GX chip-set, 200MHz, 16KB L1 cache, 512KB L2 cache, 128MB Main Memory) connected to Myrinet. The operating system on each node is Solaris 2.5.1. For communication between nodes, we consider remote memory operations to be more suitable than message passing on SMP clusters, because message passing suffers from the handling of incoming messages. We designed a

```
double *A, *b, *x, *y, omega, err, epsilon;
int N, i, j;
#pragma omp parallel private(j)
do {
#pragma omp barrier
#pragma omp single
  err = 0.0;
#pragma omp for reduction(+:err) schedule(static)
  for (i=0;i<N;i++) {
    y[i]=b[i];
    for (j=0;j<N;j++)
      if (j!=i) y[i]=y[i]-A[i*N+j]*x[j];
    y[i]=x[i]+omega*(y[i]/A[i*N+i]-x[i]);
    err+=(x[i]-y[i])*(x[i]-y[i]);
  }
#pragma omp for schedule(static) nowait
  for (i=0;i<N;i++) {
    x[i]=y[i];
  }
} while (err>epsilon);
```

Figure 4: The kernel of Jacobi over relaxation solver

user-level communication layer for Myrinet, **NICAM**. NICAM transfers data using the direct memory access (DMA) engine on the Myrinet Network Interface (NI) which utilizes not the central processor but microprocessor on the NI. The minimum latency of for small messages is about 20 microseconds and the call overhead of data transfer is 5.7 microseconds. The maximum bandwidth is about 105 MB/s and and $N_{1/2}$ is 2KB.

4.2 Preliminary result

As a preliminary result, we show the performance of the Jacobi over relaxation solver of dense matrix on COMPaS. Figure 4 shows a fragment of the OpenMP program of the solver.

The program solves the equation $A \times x = b$ by the iterative method. In an iteration of the solver, the most time-consuming computation is the matrix-vector multiply $A \times x$. After scaling y by the diagonal of A, x is updated by y for the next iteration.

In the results in this section, we solved an equation where the size of matrix A is 6144 and the solution x is obtained in ten iterations. We measured the execution times for the iterative computation part.

Figure 5 shows performance of the base DSM system. Horizontal coordinates are the product of the number of nodes and the number of threads in each node, that is, the total number of threads. Vertical coordinates are the speedup over the serial execution. The serial version is the program obtained by compiling the same OpenMP program ignoring OpenMP

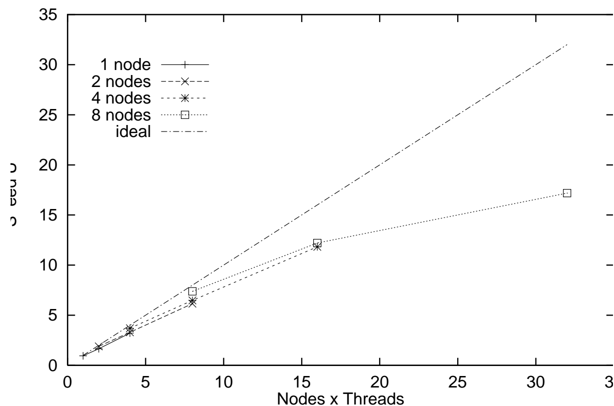

Figure 5: Speed Up of Jocobi (base SDSM)

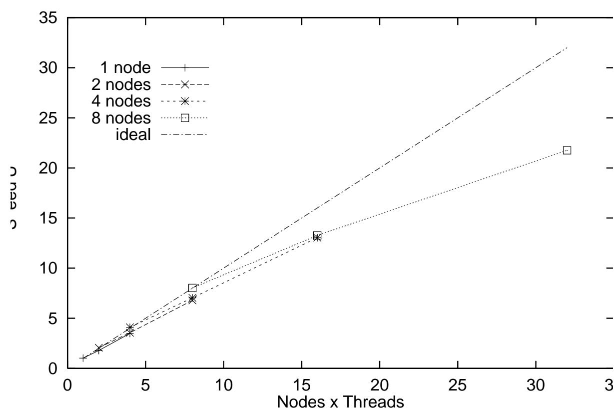

Figure 6: Speed Up of Jocobi (Optimized)

directives.

In the base DSM system, all shared data are kept coherent at runtime. Optimizations except data-parallel communication optimization are performed. For example, runtime checks for shared data which is not modified in the parallel loop are removed, and checks for array elements which are accessed sequentially are merged into a single check, and so on.

We developed two versions of coherence algorithms, namely update-based version and invalidation-based version, and the figure shows the performance of the former one. In the update-based version, check codes broadcast new data at each write access for shared data. In Jacobi solver, the update-based version is better than invalidate-based one because most accesses of shared data are read access.

Figure 6 shows the performance of the optimized program. In this case, the speedup of the eight quad-processor SMP nodes is 21.76 over serial execution. This improvement is due to the following optimization. In the matrix-vector multiply $y = A \times x$, a part of y is computed by each threads. The modified part of y is written back to the home node (node 0). To update x with y, the entire y is fetched because y is modified. This optimization is possible because these operations are executed by work sharing constructs in data-parallel manner. This additional optimization improved the performance by about 27% over the base DSM system.

5 Concluding Remarks and Future Work

We are currently working on implementing the optimization as a part of Omni OpenMP compiler system being developed at RWCP. We extend the OpenMP for parallel programming on an SMP cluster by “compiler-directed” SDSM. The compiler first inserts check codes of the base DSM runtime library to keep memory consistency. Besides check code optimization by merging and hoisting the check codes, the compiler analyzes the data-parallel computation in work sharing constructs to optimize communication in a distributed memory environment of the SMP cluster. For a programmer, the OpenMP programming model supported by our “compiler-directed” SDSM offers a comfortable and easy-to-program, yet efficient programming environment by hiding the complicated configuration of an SMP cluster.

Currently, the SMP version of Omni OpenMP compiler is available at our web site:

`http://pdplab.trc.rwcp.or.jp/Omni/`

References

[1] G. Agrawal and J. Saltz. Interproedural Communication Optimization for Distributed Memory Compilation. In *Proc. of Workshop on Lang. and Compiler for Parallel Computing 94*, pages 19.1–19.16, 1994.

[2] H. Harada, H. Tezuka, A. Hori, S. Sumimoto, and Y. Ishikawa. Implementation and Evaluation of Memory Barrier on Software Distributed Shared Memory on Myrinet. In *JSPP'99*, pages 237–244. , June 1999. (In Japanese).

[3] T. Chiueh and M. Verma. A Compiler-directed Distributed Shared Memory System. In *Proc. of ICS '95*, pages 77–86, 1995.

[4] Y.C. Hu, H. Lu, A. L. Cox, and W. Zwaenepel. OpenMP for Networks of SMPs. In *Proc. of IPPS*, 1999.

[5] T. Inagaki, J. Niwa, T. Matsumoto, and K. Hiraki. Supporting Software Distributed Shared Memory with an Optimizing Compiler. In *Proc. of ICPP'98*, 1998.

[6] M. Matsuda, K. Kubota, Y. Tanaka and M. Sato. Network Interface Active Messages for Low Overhead Communication on SMP PC Clusters. In *Proc. on HPCN'99*, April 1999.

[7] OpenMP Architecture Review Board. *OpenMP: Simple, Portable, Scalable SMP Programming*, 1997. `http://www.openmp.org/`.

[8] D. J. Scales and K. Gharachorloo. Desgin and Performance of the Shasta Distributed Shared Memory System. In *Proc. of ICS '97*, pages 245–252, 1997.

[9] Y. Tanaka, M. Matsuda, M. Ando, K. Kazuto and M. Sato, COMPaS: A Pentium Pro PC-based SMP Cluster and its Experience. IPPS Workshop on Personal Computer Based Networks of Workstations, LNCS 1388, pp. 486–497, 1998.