Compiler-Assisted Distributed Shared Memory Schemes Using Memory-Based Communication Facilities

Takashi Matsumoto, Junpei Niwa, Kei Hiraki
Department of Information Science, Faculty of Science
University of Tokyo
7-3-1 Hongo, Bunkyo-ku, Tokyo, Japan
{tm, niwa, hiraki}@is.s.u-tokyo.ac.jp

Abstract *To execute shared-memory-based parallel programs efficiently, we introduce two compiler-assisted software cache schemes which are well-suited to automatic optimizations of remote communications. One scheme is a full user-level software cache (User-level Distributed Shared Memory: UDSM) and another is a page-based cache (Asymmetric Distributed Shared Memory: ADSM) which exploits TLB/MMU only in the cases of read-access-misses. Under these schemes we can apply several optimizing techniques, which exploit capabilities of the middle-grained or coarse-grained remote-memory-accesses, to reduce the number and the amount of communications. We also introduce a high-speed user-level communication and synchronization scheme "Memory-Based Communication Facilities (MBCF)" for providing the capabilities in a general-purpose system with off-the-shelf communication-hardware. In this paper, we explain outline of our approach, the UDSM and the ADSM, the MBCF, and optimizing techniques for remote communications. Finally we show experimental results on effects of our proposed approach using our prototype optimizing compiler "Remote Communication Optimizer (RCOP)" and the MBCF on Fast Ethernet.*

Keywords: Software DSM, Optimizing Compiler, Memory-Based Communication Facilities, Asymmetric Distributed Shared Memory, User-level Distributed Shared Memory

1 Introduction

It is difficult to execute shared-memory-based parallel programs (e.g. SPLASH-2[1]) efficiently on distributed-memory parallel systems without hardware-remote-cache mechanisms. In the usual cases we need to modify and rewrite those programs to fit them into the distributed-memory systems. A way in which we can avoid the rewriting is to utilize software Distributed Shared Memory (DSM) mechanisms supported by the Operating System (OS). We can receive the benefits of caches from the conventional software DSM, but the cache-miss penalty and the coherence overhead are still large. Because of the penalty and the overhead, we cannot attain good results through the approach using conventional software DSM.

In this paper we introduce a brand-new approach to solve this problem. Our approach is a combination of user-level cache emulation and optimizing compiler. Our goal is to realize efficient execution of shared-memory-based parallel programs with automatic optimizations for remote communications under a general-purpose operating system on a stock network of workstations. If we use some conventional OS-based (page-based) software DSM, procedures for cache maintenances are invisible from user-level codes and it is difficult for optimizing compilers to optimize the procedures with user-level application codes. On the other hand, user-level codes for cache maintenances increase opportunities of code optimizations. Therefore, we adopt user-level cache emulations (i.e. user-level software DSMs).

We have developed an optimizing compiler (**RCOP**: Remote Communication Optimizer) for our approach and it has run-time libraries for the user-level cache emulations. RCOP analyzes memory accesses in a target application program, finds requests for remote memory accesses, inserts check and maintenance codes for caches of remote data, reduces redundant cache maintenance codes, and reduces the number of communications by merging packets.

We also introduce a novel protected and virtualized high-speed user-level communication and synchronization scheme "Memory-Based Communication Facilities (MBCF)" for a general-purpose distributed-memory system with off-the-shelf communication-hardware. The MBCF is suited to our approach of efficient executions of shared-memory-based parallel programs. For programmers and compilers, the MBCF provides methods for the direct remote memory accesses in user task spaces.

2 Our Approach

Our basic strategies for efficient executions of shared-memory-based parallel programs are as follows.

1. Automatic translations/compilations from source programs
2. Shared-memory-based data allocation
3. Inserting user-level cache emulation codes
4. Using explicit communication codes for remote accesses
5. Optimizing remote communications
    - Reducing cache maintenance codes
    - Reducing the number of communications
    - Tuning data-size of a communication packet

Our automatic optimizer exploits the information of source programs. To handle memory pointers to the shared objects elegantly, we assmue the shared-memory-based data allocation. It supposes that the local virtual addresses of a shared data object are identical in every node of the system, so that the system and applications easily can identify shared data objects. However, we assume that there is neither remote-memory access hardware nor hardware DSM in the system. Therefore, we must replace remote-memory-accesses with alternative codes which invoke explicit operations to perform them. If we use an executable object where every remote access is replaced with an explicit remote communication code, there are too many fine-grained communications and the very large accumulated overhead degrades the execution speed. We need to reduce the number of communications as much as possible. The first step of the reduction is to introduce a kind of cache system in our execution environment. We adopt the user-level (application-level) implementation of software DSM, since user-level codes for cache maintenances increase opportunities of code optimizations. We need to insert explicit cache-emulation codes at the candidate points where remote accesses occur. If our analyzer cannot find whether a memory access at a candidate point is remote or local, our optimizer inserts an address-range checking code before cache-emulation code in order to decide necessity of a remote access. Cache-emulation codes perform explicit remote communications if necessary.

Simple replacements from remote-accesses to cache-emulation codes are still redundant and leave room to optimize remote communications. We will describe the details of our optimizing techniques in section 5.

We assume that commodities (e.g. Fast ethernet) are used as the communication hardware of the system. Such mechanisms have non-negligible overhead of communication and hence they are poor at fine-grained communications. Since there is a limit for the size of the sending buffer in the communication hardware, they cannot efficiently send large data over the limit at a time. While keeping the meaning of parallel programs, it is advantageous on performance to adjust the data-size of communication packets statically or dynamically.

3 UDSM and ADSM

Considering code optimizations for inter-node communications, the full user-level cache scheme (User-level DSM:**UDSM**), where application programs only use user-level codes to maintain software-remote-caches, is better than OS-based software DSMs. In other words, the UDSM scheme is more suitable to exploit opportunities for the optimizations of communications and executions than OS-based DSMs.

In the UDSM case, however, the user-level executable code must explicitly maintain, check and modify software-controlled-cache tags. It is a little difficult to develop optimizing compilers that are sophisticated enough to hide and/or reduce the overhead of handling cache tags. Inter-node communications are required only at shared-write or cache-miss situations. If we use a large area of memory as a software-controlled-cache we can keep the rate of cache-miss small. Besides, in usual applications the number of shared-writes is much less than the number of shared-reads. These characteristics suggest us a brand-new remote cache scheme "Asymmetric Distributed Shared Memory (**ADSM**)" [2, 3].

In conventional page-based (i.e. OS-based) DSMs, not only read-cache-misses but also shared-writes are supported by the TLB/MMU mechanisms of node processors using write-protection traps and page-fault traps. Though the ADSM is one of page-based cache schemes, only read-cache-misses are supported by the TLB/MMU mechanisms. For each shared-write in the ADSM scheme, a proper sequence of instructions which maintains the cache consistency of the system is inserted into the user-level executable code by the optimizing compiler. The user-level code-sequences include the explicit communication procedures and invalidate (or update) remote caches while modifying the local cache-states.

The ADSM has less opportunities for the optimizations of communications and executions than the UDSM, because codes for shared-reads are implicitly supported by OS and the cache block-size of the ADSM should be equal to the size of a memory-page. In the ADSM, however, there is still large room for various optimizations using inserted cache maintenance codes. The strategy for handling shared-reads (read-cache-misses) and that for handling shared-writes are different. Therefore we call this scheme the "asymmet-

ric" DSM.

Which of the UDSM and the ADSM should we use? The answer depends on system parameters: optimization level of the compiler, memory access pattern of application programs, software cost of checking cache tags, and cost of the page-fault trap.

4 Memory-Based Communication Facilities

4.1 Needs of the MBCF

Conventional user-level communication interfaces (e.g. socket library or MPI) of existing operating systems are useless for realizing our goal because of their large overheads. Therefore, we proposed a protected and virtualized high-speed user-level communication and synchronization scheme "Memory-Based Communication Facilities (MBCF)"[2, 4] for a general-purpose system with off-the-shelf communication-hardware. There are two factors which produce the major part of overheads on the conventional interfaces. The first one is a methodological aspect (including functionalities, protocols, packet format). Conventional communication interfaces are message-passing-type ones, and their functions are limited to remote-write operations into a few specific message-buffer addresses in the kernel-space. To break out of this limitations, we adopt memory-based operations where arbitrary target addresses and a wide variety of functions can be used. Moreover, a middle-grained or coarse-grained direct remote-memory-access capability provided by the MBCF is fit to communication optimizations of the UDSM and the ADSM. Middle-size or large-size amount of data can be handled in an MBCF operation and multiple MBCF operations can be merged into one communication packet by communication optimizations. The other factor of overhead is a software engineering aspect (implementation methodology). In the conventional OSs, communications and synchronizations among nodes (machines) are regarded as usual I/O events like disk operations, and device-drivers for communications and synchronizations have the same data and control structures as device-drivers for other I/O devices. Consequently, the device-drivers suffer large overheads that are not necessary for functions of communication and synchronization. To realize high-performance implementations, the MBCF-dedicated system-calls and the MBCF-dedicated interrupt routines have been developed and used, then there is no operation irrelevant to the functions of the MBCF.

4.2 Outline of the MBCF

In the MBCF scheme, communications and synchronizations are performed through virtual inter-node memory locations. An address of some location is specified the combination of a logical task-ID and a logical address in the target logical task. The task is an abstraction of a processor's activity and has its own memory-space, and it belongs to a node in the MBCF system. In the MBCF system, a task is specified by the combination of a physical node-ID and a physical task-ID in the physical-node. In user-level application programs, only the logical task-ID is used to specify a task. The OS for the MBCF system maintains one translation table for each task, and translates a logical task-ID into the combination of a physical node-ID and a physical task-ID.

In the MBCF system remote memory accesses are invoked by explicit system-calls for the MBCF functions. First a user-program prepares an MBCF packet in user-mode and executes the MBCF requesting system-call. The packet includes a target logical task-ID, a target logical address, an access-key of the target task, a command (type of memory operation), a few parameters of the command (optional), a return address of status reply (optional), and the size of data and data to be sent. Secondly the kernel-level routine of the MBCF-dedicated requesting system-call makes a inter-node communication-style packet and transmit it using conventional Network Interface Cards. Finally the MBCF-dedicated interrupt routine at the target node receives the packet, checks the access-key and directly executes the remote access specified in the packet and returns a reply if necessary. The command variation of the MBCF includes simple remote-memory-accesses (read and write), remote-memory-accesses with flag operations, atomic operations (swap, test&set, compare&swap), multi-casting operations (invalidate and update), memory-based fifo[5, 4], and memory-based signal[5, 4].

By adopting memory-based operations, protection and virtualization in communications and synchronizations can be replaced with those of memory accesses. This replacement makes high-speed implementations of the MBCF scheme feasible, since advanced architectural mechanisms of processors for memory-accesses can be exploited. Each entry of the Translation Look-aside Buffer (TLB) in a recent processor has a field for context identifier (context-ID) and the OS for the processor can switch contexts without clearing entries of the TLB. In the MBCF interrupt routine, each MBCF packet requires a few page-entries to be used for direct memory-accesses to the target space. With this type of advanced TLB, the MBCF interrupt routine replaces at most only a few entries of the TLB. If the MBCF operations access the same locations frequently, it is likely that the page entries corresponding to the locations are resident in the TLB. The cost of the replacement itself is small and the influence of the MBCF operations is also

very small after the MBCF interrupt.

5 Optimizing Techniques

RCOP deals with a parallel shared memory program based on lazy release consistency (LRC) model[6]. The input program is written in C extended by PARMACS[7][1]. RCOP analyzes the shared memory program and translates it into a instrumented C program which explicitly contains consistency management codes for the ADSM. The output C code is compiled by the backend compiler, then linked with the ADSM runtime library to generate executable code. We used gcc 2.7.2 (the optimizing level is -O2) as the backend compiler.

In the ADSM runtime system, write transactions on shared locations are represented as a pair of a shared address and a byte size of a written block. This pair is generated by a commitment of shared write and is stored directly at runtime (we call this pair as a *write commitment*). Though it is implicit to commit shared write, contents of written shared location are also required. Consequently, we place a write commitment after the shared write in question.

In order to reduce both communication overheads and instruction overheads for consistency management codes, RCOP requires to insert valid and optimal write commitments. First, we have to detect all the shared write operations. Second, we calculate redundancy among shared write operations.

5.1 Detecting Shared Write

In order to detect which write operation accesses to a shared location, we have to identify which pointer value points to a shared address. We adopted interprocedural points-to analysis[8, 9] to detect pointers to shared locations.

Applying the points-to analysis to shared write detection is straightforward. We track pointer values created by `G_MALLOC` (dynamic shared memory allocation), and enumerate write operations using them.

5.2 Eliminating Redundant Write Commitments

In LRC models, commitments of modifications in shared locations are delayed until a synchronization point to avoid frequent invalidations and communications. Therefore we can place a write commitment anywhere from the shared write in question to the next synchronization primitive.

Let us look the following code fragments:

```
a[ii][jj]=((double)lrand48())/MAXRAND;
if (i == j)
    a[ii][jj] *= 10;
```

Suppose the location `a[ii][jj]` is shared, both assignments require the same consistency management code (WC). But, if we delay the commitment after the conditional statement, the consistency management code in the conditional clause is redundant.

```
a[ii][jj]=((double)lrand48())/MAXRAND;
if (i == j)
    a[ii][jj] *= 10;
WC (&a[ii][jj], 1);
```

This optimization is formalized as *write commitment redundancy elimination* like common subexpression elimination[10]. For simplicity, we consider one fixed shared write operation. We represent a statement

$$
\begin{aligned}
\mathrm{AVIN}(i) &= \prod_{p \in \mathrm{pred}(i)} \mathrm{AVOUT}(p) \\
\mathrm{AVOUT}(i) &= \mathrm{COMP}(i) + \mathrm{TRANS}(i) \cdot \mathrm{AVIN}(i) \\
\mathrm{ANTOUT}(i) &= \prod_{s \in \mathrm{succ}(i)} \mathrm{ANTIN}(s) \\
\mathrm{ANTIN}(i) &= \mathrm{COMP}(i) + \mathrm{TRANS}(i) \cdot \mathrm{ANTOUT}(i) \\
\mathrm{INSERT}(i) &= \mathrm{AVOUT}(i) \cdot \neg \left(\prod_{s \in \mathrm{succ}(i)} \mathrm{AVOUT}(s) \right) \\
&\quad \cdot \neg \mathrm{ANTOUT}(i)
\end{aligned}
$$

Figure 1: Dataflow equation to remove redundant write commitments

in a procedure as i. We can consider that i is a node of a control flow graph (CFG) of the procedure. To remove redundant commitments, we calculate the following two dataflow variables for each statement i

Availability All preceding statements to i write into the shared location.

Anticipatability All succeeding statements to i write into the shared location.

Both availability and anticipatability take values of true or false, and are calculated from two kinds of constants about i.

$\mathrm{COMP}(i)$ Statement i writes into the shared location.

$\mathrm{TRANS}(i)$ Statement i propagates information above and below.

Transparency is false if i is a synchronization primitive or if i modifies parameters of the shared write.

In order to minimize the number of shared write commitments, we have only to insert a write commitment

[1] PARMACS is a parallel macro construction that contains shared memory allocation, thread creation, and synchronization

where 1) the shared write is available, 2) the shared write is not available at its successors and 3) the shared write is not anticipatable.

We denote availability before and after execution of the statement i as AVIN(i) and AVOUT(i). Similarly, we represent anticipatability as ANTIN(i) and ANTOUT(i). The dataflow variable INSERT(i) reflects whether or not the write commitment is really inserted after i. Each dataflow variable is computed from the equations described in Figure1[2].

In order to compute interprocedurally, we reflect AVOUT at the exit of the callee procedure to the COMP at the call site of the caller procedure. When the availability of the callee can not be propagated to the caller, we insert write commitments at the exit of the callee. An *open*[11] procedure does not propagate availability to the call sites. Therefore, we can consider the call graph is acyclic. RCOP simply calculates interprocedural availability with bottom-up traversal of the call graph.

5.3 Handling Multiple Shared Writes

Because a write commitment contains a size information, we can calculate redundancy between multiple shared write operations. For example, suppose `a` is a shared pointer in the following code.

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

Instead of inserting a write commitment after each write operation, we can generate the following code:

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

The advantage of this optimization is as follows. First, the instruction overhead for consistency management is alleviated. Second, the ADSM runtime system utilize the size information for message vectorization.

It is convenient to denote multiple write commitments as a *shared write set*. A *shared write set* W consists of a tuple (f, s, C). f and s are respectively the shared address and the size of each contiguous region to be written, and C is a set of inequalities to generate instances. Usually, f, s, and C contain common index variables, so $f = f(\vec{i})$, $s = s(\vec{i})$, $C = C(\vec{i})$. Informally, a shared write set corresponds to a sequence of consistency management codes. f and s are arguments of one write commitment, and C corresponds to enclosing loops. We describe optimization methods using shared write sets.

[2] pred(i) is a set of preceding statements to i and succ(i) is a set of succeeding statements to i.

5.3.1 Coalescing

This is applicable when write commitments onto contiguous locations are issued in a loop. For a shared write set $W = (f(i), s, C(i))$, suppose that we want to test whether we can coalesce about an induction variable i, whose stride is c. This is examined by an equation $f(i + c) - f(i) = s$. When this condition holds, we replace i by its initial value and multiply s by number of iterations and eliminate inequalities about i. For the above code, the transformation is like this:

$$W = (\&a[i], 1, \{0 \leq i < n\}) \rightarrow W' = (a, n, \emptyset)$$

Even if the index is not an induction variable, it is possible to perform this translation when the index is only contiguous.

5.3.2 Fusion

Originally, each shared write set corresponds to each shared write operation in the program. Sometimes we can merge a series of shared write sets into a new large one. We call this transformation *fusion*, and express it by a binary operator "$\circ$". The most simple case is when we can detect contiguous shared write accesses from locally like this:

```
for (i=0; i<n1; i++) {
    x[2*i] /= N; x[2*i+1] /= N;
}
```

Suppose `x` points to a shared location, each write operation generates shared write sets in the `for` loop,

$$W = (\&x[2 * i], 1, \emptyset), \quad W' = (\&x[2 * i + 1], 1, \emptyset).$$

Using similar test to coalescing, we can obtain

$$W \circ W' \rightarrow W'' = (\&x[2 * i], 2, \emptyset).$$

5.3.3 Redundant Index Elimination

When the location of a shared write set is loop-invariant, the shared write set is represented as a single write commitment whose size is the maximum of all the iterations. We can detect the above case using the Fourier-Motzkin pairwise elimination[13]. The Fourier-Motzkin elimination is also applicable to non-linear but monotonous index expressions. For example,

$$W = (x, 2 * 2^q * (N/2^q), \{1 \leq q \leq M\})$$

Since x is loop-invariant about q and $2 * 2^q * (N/2^q)$ is monotonously decreasing function of q, we can eliminate q from the C of W as follows.

$$W \rightarrow W' = (x, 4 * (N/2), \emptyset)$$

5.3.4 Putting All Together

We integrate the above optimizations: coalescing, redundant index elimination and fusion using shared write sets. The dataflow variables in the Figure1 take values of shared write sets. The logical operations are considered as set operations. Just after points-to analysis, each write set includes only one write commitment, i. e., $s = 1, C = \emptyset$.

We use interval analysis[14, 15] to calculate dataflow equations. We create a summary of each interval or call site in a bottom up manner. Coalescing and redundant index elimination are applied to a summary of an interval and to a summary of a call site which is created by a parameter mapping on a summary of a procedure. Fusion is tried when we merge multiple shared write sets at a merge point in the program or between propagated values and computed values.

6 Performance Evaluation

6.1 ADSM

We have implemented a prototype of RCOP and the runtime system of the ADSM on the SSS–CORE/NOW ver.1.0. Current SSS–CORE is working on 8 nodes of Axil 320 model 8.1.1 (which is compatible with Sun SPARCstation 20, with one 85MHz SuperSPARC). Each node is equipped with Fast Ethernet SBus Adapter2.0, and are connected by Fast Ethernet with a 100BASE-TX switch. The SSS–CORE implements the MBCF on the Fast Ethernet. Its measured peak bandwidth is 11.2MB/s and its round-trip latency is 49μs.

Because the SSS–CORE/NOW ver.1.0 has not supported user-level page-fault handlers yet, we cannot detect page fault via traps. In order to detect that the processor attempts to access the shared page which is not allocated or invalid, we insert the code checking the corresponding page's validity before each shared access. The messages which request memory-copies or cache invalidations are serviced through the memory-based signal mechanism of the MBCF. As for the cache consistency protocol, we adopt the SAURC protocol[2] which is a variation of the LRC protocols[6] and emulates the AURC protocol[16] with explicit communication codes.

We evaluate the performance on LU-Contig and Radix of SPLASH-2 benchmark suit[1] using 8 nodes. The problem size of LU-Contig is a 512×512 matrix with 16×16 blocks and that of Radix is 1M sorting keys. Table1 shows the results of LU-Contig. Table2 shows the results of Radix.

For each table, we evaluate three optimization methods for the ADSM: "NO" describes no optimizations, "MB" denotes the runtime packet combining and "IA" represents the static interprocedural redundancy elimination. The "Opt" column in the tables expresses the combination of the optimization methods which are applied to the corresponding measurement. The "#CM" column shows the number of consistency management codes.

Since LU-Contig is a simple application in which the processor accesses the contiguous locations, RCOP can find many opportunities to perform coalescing. Consequently the number of write commitments (i.e., consistency management codes) is greatly reduced. As for Radix, there are a few opportunities to perform coalescing optimization using induction variables. But we can perform coalescing by using a contiguous variable and this produces good results. The runtime level packet combining is also effective for both applications.

6.2 UDSM

We can apply similar optimization to the write commitment redundancy elimination for shared page table checking. Successive checks of contiguous shared locations are coalesced and executed before the enclosing loop. We perform this shared-read optimization manually by rewriting output codes which RCOP for the ADSM generates.

In order to show how efficient the UDSM scheme is, we evaluate the performance on the above benchmarks using not only the SSS–CORE but also a multicomputer Fujitsu AP1000+[17]. In the AP1000+, each node consists of 50MHz Super SPARC with 20 Kbytes I-cache, 16 Kbytes D-cache and 16MB memory. Nodes are interconnected by a 2-D torus network whose bandwidth is 25 Mbytes/sec per link. The AP1000+ has dedicated hardware which executes remote block transfer operation (put/get interface). Request messages from remote nodes are serviced through polling mechanism. We insert polls at every loop backedge and every function call.

Table3 shows the input problem size and sequential execution time. It also shows the single processor execution time with shared page table checking[3], along with the percentage increase in the time over the original sequential time. In LU-Contig, because the processor accesses the contiguous locations, many shared page table checks are coalesced and the check overhead is reduced.

Figure2 shows the execution time of LU-Contig on 1 to 8 nodes. Figure3 shows the results of Radix. The left part of each graph gives the results on the AP1000+. The right part of each graph gives the results on the SSS–CORE. "Task" means execution time of the original application codes. Page table checking and polling

[3] Polling overheads are also included in AP1000+

Table 1: Effects of optimization methods on LU-Contig (n=512,b=16)

Opt	Exec Time (sec)	#CM	Number Of Packets	Data Traffic (MByte)
NO	28.199637	5592384	5206567	47.728956
MB	14.351349	5592384	83515	113.002224
IA	2.174304	1430	7731	9.424220
MB & IA	2.160257	1430	7599	9.274476

Table 2: Effects of optimization methods on Radix (#key = 1M)

Opt	Exec Time (sec)	#CM	Number Of Packets	Data Traffic (MByte)
NO	21.904937	792831	3220108	76.717636
MB	12.131433	792831	75832	101.082236
IA	1.565786	2079	19457	13.471372
MB & IA	1.236287	2079	10148	13.626328

Table 3: Sequential execution time(in seconds) and checking overheads

		SSS–CORE		AP1000+	
program	problem size	sequential time	with check overhead	sequential time	with check overhead
LU-Contig	1024^2 doubles	59.000	60.096(2%)	115.673	125.00(8%)
Radix	1M integer keys	1.817	2.240(18%)	4.325	5.535(21%)

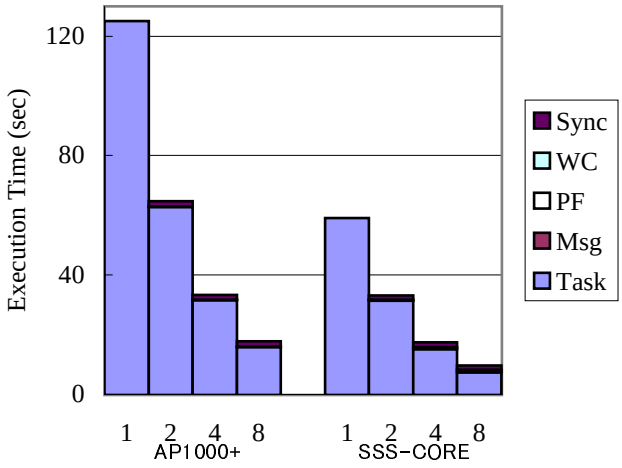

Figure 2: Execution time of LU-Contig on 1 to 8 nodes

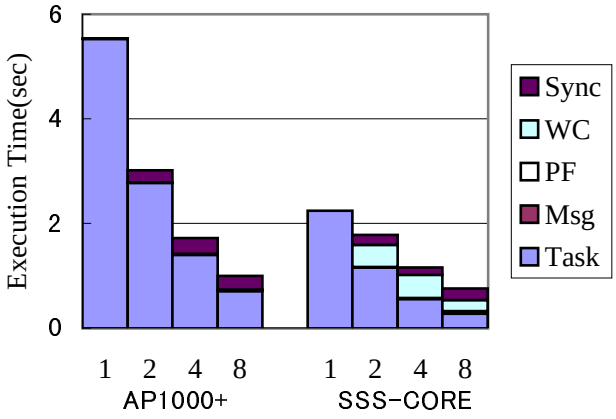

Figure 3: Execution time of Radix on 1 to 8 nodes

overhead are also included in “Task”. “Sync”, “WC”, and “PF” respectively mean execution time spent in synchronization, write commitment, and page fault handler. “Msg” means time in remote message handlers. Note that “Msg” also contains handling time of messages which are caught by polling in synchronization, write commitment, and page fault. Consequently, “Sync”, “WC”, and “PF” consist of their inherent local overhead and pure idle time waiting for responses from other processors.

Overall, the speedups achieved by the UDSM optimization are quite promising. This result indicates that when the MBCF is given, the performance of a workstation cluster with off-the-shelf communication-hardware is as good as that of a multicomputer with special high-speed communication-hardware.

As for LU-Contig, in both platforms WC time is negligible by the UDSM optimization. About Radix, WC time in the SSS–CORE is relatively high, but in the AP1000+ it is negligible. The write commitments are merged as large as possible and delayed as far as possible in order to detect redundancy. This optimization

policy does not always produce good results for the platforms that have poor communication-hardware comparing with processor power because they cannot send large data over the limit at a time. The result indicates that the relation between processor power and communication-hardware must be reflected in the dataflow equation.

7 Conclusions

To execute shared-memory-based parallel programs efficiently, we introduced two compiler-assisted software cache schemes and a user-level memory-based communication mechanism. The cache schemes are User-level Distributed Shared Memory (UDSM) and Asymmetric Distributed Shared Memory (ADSM). Under these cache schemes we can apply several optimizing techniques to reduce the number and the amount of communications. The communication mechanism is a high-speed user-level communication and synchronization scheme "Memory-Based Communication Facilities (MBCF)" for a general-purpose system with off-the-shelf communication-hardware. The MBCF is superior to conventional communication interfaces and suited to optimizations for the UDSM and the ADSM. We have developed our prototype optimizing compiler "Remote Communication Optimizer (RCOP)" for the ADSM scheme and made experiments to show the effects of the combination of our cache schemes, optimizing techniques and the MBCF. The result shows that our approach is promising to realize efficient executions of shared-memory parallel programs on a general-purpose system like a network of workstations which has no dedicated communication hardware.

Acknowledgments

This work is partly supported by Advanced Information Technology Program (AITP) of Information-technology Promotion Agency (IPA) Japan and by Real World Computing Partnership (RWCP) Japan. Authors thank Mr. Tatsushi Inagaki for his collaboration on our compiler research.

References

[1] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta. The SPLASH-2 Programs: Characterization and Methodological Considerations. In *Proc. of the 22nd ISCA*, pages 24–36, June 1995.

[2] T. Matsumoto, T. Komaarashi, S. Uzuhara, S. Takeoka, and K. Hiraki. A general-purpose massively-parallel operating system: Sss–core — implementation methods for network of workstations —. *IPS Japan SIG Reports 96-OS-73*, 96(79):115–120, aug, 1996. (In Japanese).

[3] T. Matsumoto and K. Hiraki. Memory-Based Communication Facilities and Asymmetric Distributed Shared Memory. In *Innovative Architechture for Future Generation High-Performance Processors and Systems*, pages 30–39. IEEE Computer Society Press, April 1998.

[4] T. Matsumoto and K. Hiraki. MBCF:A Protected and Virtualized High-Speed User-Level Memory-Based Communication Facility. In *Proc. of the 1998 Int. Conf. on Supercomputing*, July 1998. to appear.

[5] T. Matsumoto and K. Hiraki. A shared-memory architecture for massively parallel computer systems. *IEICE Japan SIG Reports CPSY*, 92(173):47–55, August, 1992. (In Japanese).

[6] P. Keleher, A. L. Cox, and W. Zwaenepoel. Lazy Release Consistency for Software Distributed Shared Memory. In *Proc. of the 19th ISCA*, pages 13–21, May 1992.

[7] J. Boyle et al. *Portable Programs for Parallel Processors*. Holt, Rinehart and Winston, Inc., 1987.

[8] M. Emami, R. Ghiya, and L. J. Hendren. Context-Sensitive Interprocedural Points-to Analysis in the Presence of Function Pointers. In *Proc. of '94 Conf. on PLDI*, pages 242–256, June 1994.

[9] R. P. Wilson and M. S. Lam. Efficient Context-Sensitive Pointer Analysis for C Programs. In *Proc. of '95 Conf. on PLDI*, pages 1–12, June 1995.

[10] J. Cocke. Global Common Subexpression Elimination. *Proc. of a Symp. on Compiler Optimization, SIGPLAN Notices*, 5(7):20–24, July 1970.

[11] F. C. Chow. Minimizing Register Usage Penalty at Procedure Calls. In *Proc. of the SIGPLAN '88 Conf. on PLDI*, pages 85–94, June 1988.

[12] J. Niwa, T. Inagaki, T. Matsumoto, and K. Hiraki. Efficient Implementation of Software Release Consistency on Asymmetric Distributed Shared Memory. In *Proc. of the 1997 ISPAN*, pages 198–201, December 1997.

[13] G. B. Dantzig and B. C. Evans. Fourier-Motzkin Elimination and Its Dual. *Journal of Combinatorial Theory*, A(14):288–297, 1973.

[14] J. Cocke and J. T. Schwartz. *Programming Languages and Their Compiler*. New York University Press, 2nd edition, April 1970.

[15] M. Burke. An Interval-Based Approach to Exhaustive and Incremental Interprocedural Data-Flow Analysis. *ACM Trans. on Programming Languages and Systems*, 12(3):341–395, July 1990.

[16] L. Iftode, C. Dubnicki, E. W. Felten, and K. Li. Improving Release-Consistent Shared Virtual Memory using Automatic Update. In *Proc. of the 2nd HPCA*, February 1996.

[17] O. Shiraki et al. Architecture of Highly Parallel Computer AP1000+. In *Proc. of the 3rd Parallel Computing Workshop*, pages P1–G–1 – P1–G–8, November 1994.