Compiler optimization techniques for OpenMP programs

Shigehisa Satoh[a,*], Kazuhiro Kusano[b] and
Mitsuhisa Sato[c]

*Tsukuba Research Center, Real World Computing
Partnership, 1-6-1 Takezono, Tsukuba, Ibaraki
305-0032, Japan*
E-mail: {sh-sato,kusano,msato}@trc.rwcp.or.jp
Present addresses:
[a]*Systems Development Laboratory, Hitachi, Ltd.,
1099 Ohzenji, Asao, Kawasaki, Kanagawa 215-0013,
Japan*
[b]*1st Computers Software Division, NEC Solutions,
NEC Corporation, 1-10 Nissin-cho, Fuchu, Tokyo
183-8501, Japan*
[c]*Center for Computational Physics, University of
Tsukuba, 1-1-1 Tennodai, Tsukuba, Ibaraki 305-8577,
Japan*

We have developed compiler optimization techniques for explicit parallel programs using the OpenMP API. To enable optimization across threads, we designed dataflow analysis techniques in which interactions between threads are effectively modeled. Structured description of parallelism and relaxed memory consistency in OpenMP make the analyses effective and efficient. We developed algorithms for reaching definitions analysis, memory synchronization analysis, and cross-loop data dependence analysis for parallel loops. Our primary target is compiler-directed software distributed shared memory systems in which aggressive compiler optimizations for software-implemented coherence schemes are crucial to obtaining good performance. We also developed optimizations applicable to general OpenMP implementations, namely redundant barrier removal and privatization of dynamically allocated objects. Experimental results for the coherency optimization show that aggressive compiler optimizations are quite effective for a shared-write intensive program because the coherence-induced communication volume in such a program is much larger than that in shared-read intensive programs.

*Corresponding author: Shigehisa Satoh, 3-16-8-402 Fujimi-Cho, Chofu-shi, Tokyo 182-0033, Japan. Tel.: +81 424 41 4058; Fax: +81 424 41 4058; E-mail: sh-sato@acm.org.

1. Introduction

The OpenMP API is an emerging standard for shared-memory parallel programming. It provides a simple and incremental way to write parallel programs, particularly for data-parallel applications. However, naive parallelization may not achieve good performance due to poor locality, large synchronization overhead, or other reasons. In addition, the current specification of the OpenMP API does not provide enough means for controlling data locality or memory coherency, because these features are platform-dependent. Compiler optimization for OpenMP programs should be effective for achieving performance portability across various platforms and for non-expert programmers.

We first present our framework of dataflow analysis for OpenMP programs and some concrete dataflow algorithms. We used an internal representation called *Parallel Flow Graph* (*PFG*) to model both the intra- and inter-thread flows of data. We describe analysis algorithms called reaching definition analysis, memory synchronization analysis, and cross-loop data dependence analysis for parallel loops to obtain dataflow information for our optimizations.

We developed optimization techniques to reduce the synchronization and coherence overheads and to improve data locality. While these techniques are applicable to any OpenMP implementation, our primary target is a compiler-directed software distributed shared memory (DSM) system [20]. This system provides a shared memory image on top of distributed memory parallel computers with the assistance of a compiler that analyzes communication patterns and optimizes coherence control codes. In such a system, aggressive compiler optimizations are crucial to obtaining good performance because a software-implemented coherence scheme has larger coherence overhead than symmetric multi-processors (SMPs) and hardware DSMs.

We obtained preliminary performance results using a prototype system. From the experimental results, we found that coherence optimization greatly impacts

on the performance of shared-write intensive programs because the large amount of communication is reduced by the optimization.

This paper is organized as follows. First, we discuss performance bottlenecks in OpenMP programs and optimizations that overcome them. Then, we present dataflow analysis techniques in Section 3. Section 4 describes compiler optimizations particularly useful for compiler-directed software DSM systems and presents experimental results. In Section 5, related work on program analyses and optimizations for explicit parallel programs and software DSMs are presented. Section 6 summarizes this work.

2. Motivation

There can be several performance bottlenecks in OpenMP programs. We discuss here bottlenecks related to control synchronization, data synchronization, and data locality.

Threads are synchronized by directives such as the `barrier` directive and the `critical` directive. The `parallel` directive synchronizes threads by forking or joining them. Control synchronization reduces parallelism in a program by forcing threads to wait until a certain condition holds. To reduce control synchronization overhead, the OpenMP API provides such means as a `nowait` clause and orphaned directives. Multiple parallel tasks can be put in a single large parallel region to reduce overheads caused by the creation of separate parallel regions for each parallel task. Doing this enables programmers to optimize control synchronization at the source level. However, there are still situations in which it is difficult or impossible to optimize programs at the source level. For example, to add a `nowait` clause to an orphaned `for` directive, the programmer must be sure that it is valid in all calling contexts.

In addition to control synchronization, shared data must be synchronized at synchronization points induced by flush operations to maintain a coherent view of shared data. Implementation of data synchronization, or coherence, varies from platform to platform, so the OpenMP API provides only a generic interface for data synchronization. Compiler optimizations such as register allocation and code motion for shared data are inhibited at synchronization points, and the compiler must insert appropriate memory barrier instructions at these points. Data synchronization tends to be over-performed because a programmer cannot specify shared data that must be synchronized by an implicit flush operation, that may be frequently executed. Many shared data items, especially array elements, do not need to be synchronized by an implicit flush operation in practice.

The exploitation of data locality is also important, especially for distributed shared memory systems in which memory access latency varies depending on the location of the data to be accessed. Programmers can use data placement or data distribution pragmas if such features are supported by the implementation, or they can write programs taking account of the default data placement policy, such as first-touch placement. The alternative is compile-time analysis of data access patterns and automatic data placement or distribution. Privatization of dynamically allocated objects is also beneficial because the current OpenMP API do not support dynamic allocation of private data, even though such data objects may be allocated on remote nodes in a DSM.

In some cases a programmer can overcome these problems by careful design of the algorithms and appropriate use of the OpenMP API [7,26]. However, there are cases in which programmers cannot control program behavior at the source level because the OpenMP API is designed independently of the platform architecture and thus do not provides means to utilize the platform-dependent features.

There are two approaches to dealing with platform-dependent issues. First, programmers can use non-standard features such as vendor-specific directives, sacrificing portability. The second is to put such burden on the OpenMP implementation, i.e., the compiler and runtime system. The latter approach is preferable because it does not sacrifice portability. Since many OpenMP programs are written in a very structured manner, it is easier to analyze and optimize programs effectively compared with parallel programs using other multi-threading APIs such as POSIX threads.

In addition to the portability, another reason to validate compiler optimization is that many application programmers are not experts at parallel programming. Such non-expert programmers do not want to spend much time improving their programs and tend to write less efficient programs with simple structures, e.g., by using loop-level parallelization only.

3. Parallel dataflow analysis

We developed dataflow analysis techniques for OpenMP programs to enable aggressive optimization.

```
#include <math.h>
double sub(double *a, double *b, int n) {
  double s = 0.0;
#pragma omp parallel
{
  int i;
  double tmp;
#pragma omp for
  for (i = 1; i < n; i++) {
    a[i] = (b[i] - b[i-1])/2.0;
    tmp = fabs(a[i]-b[i]);
#pragma omp critical
      if (tmp > s) s = tmp;
    }
  }
  return s;
}
```

Fig. 1. An example OpenMP program.

Our analyses use an internal representation that models both intra-thread flow of control and data synchronization across threads. They also consider the semantics of OpenMP directives. The relaxed memory consistency in OpenMP enables efficient and effective dataflow analyses for parallel programs because interactions between threads need to be accounted for only at synchronization points.

3.1. Internal representation

We first define the internal representation, called *Parallel Flow Graph* (*PFG*), we use to model control flow within each thread and synchronization between threads.

The PFG of an OpenMP program is a tuple (N, E, s, e) where:

- N is a set of nodes,
- E is a set of directed edges, and
- s and e are the entry and the exit nodes of the program, respectively.

Figure 1 shows an example OpenMP program, and its parallel flow graph is depicted in Fig. 2.

A node in N represents either a basic block or an OpenMP directive. We call the nodes representing basic blocks *sequential nodes* and the nodes representing OpenMP directives *directive nodes*. Sequential nodes are created in a manner similar to that of nodes in a control flow graph for sequential programs. Directive nodes are created according to the usage of the directive. A directive used as if a single statement, e.g., `barrier` and `flush`, is represented by a single directive node. A directive used as a construct is represented by a pair of directive nodes representing the entry and exit of the construct. In the figure, sequential nodes
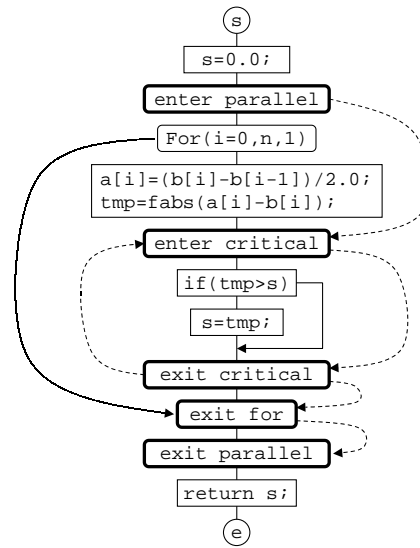

Fig. 2. A parallel flow graph.

and directive nodes are represented by rectangles and ovals, respectively. A subset of directive nodes, which imply flush operations, are also called *synchronization nodes* and are represented by thick ovals in the figure.

An edge in E represents either flow of control or the ordering of synchronization events. *Control edges*, represented by solid arrows, represent possible flow of control in a single thread. Edges between sequential nodes are created in a manner similar to that of sequential programs. However, edges from or to directive nodes are created somewhat differently according to the semantics of the directive. For example, for the `for` directive in Fig. 2, we create edges as if at most one iteration can be executed; in other words, we do not create a back edge. Instead, a edge from the entry of the construct to the exit of that is created to represent the fact that some thread may not be assigned any loop iterations. Such representation of the `for` directive reflects the semantics of the `for` construct: all iterations of the loop are executed independently and no loop-carried dependence is assumed, unless explicit synchronization is performed in the loop body.

Synchronization edges, represented by dashed arrows, represent event-ordering constraints between synchronization nodes. A synchronization edge from node n to node m indicates that the synchronization represented by node m may occur immediately after the synchronization event represented by node n, in either the same or in different threads.

We can model many parallel dataflow problems by using a parallel dataflow analysis (PDA) framework

composed of a parallel flow graph, a lattice of dataflow information, and a set of transfer functions, in a manner similar to that of dataflow analysis frameworks for sequential programs. In such PDA frameworks, interactions between threads are reflected by propagating dataflow information via synchronization edges. Transfer functions associated with directive nodes represents the semantics of OpenMP directives and clauses associated with them. In the following sections, we present some algorithms for concrete dataflow analysis problems.

3.2. Reaching definitions analysis

Reaching definitions analysis is a widely used dataflow problem. This finds definitions that can possibly reach each program point without intervening definitions of the same variable. In our analysis, reaching definitions are considered to take account of both intra- and inter-thread flow of data. We first describe a reaching definitions algorithm for scalar variables to explain the basic idea of dataflow analysis for OpenMP programs.

A lattice of dataflow information $L(V, \cap, \cup)$ is constructed as follows. Let V be a set of definitions within the given program. Operators $\cap$ and $\cup$ are intersection and union, respectively. We consider only references to shared variables; any definition of a privatized variable is not a member of V.

We also compute Gen and $Kill$ sets for each node in the PFG. These sets for sequential nodes are computed similarly to sequential programs, except that references to privatized variables are ignored. For directive node, we compute these sets considering the semantics of each directive and the clauses associated with it.

Let's consider the construction of Gen and $Kill$ sets for a `for` directive as an example. We have two directive nodes for a `for` construct: one for entry and one for exit. At the entry node, all definitions for variables privatized in the construct are killed because the values of those shared variables become undefined. We also add dummy definitions at the exit node for variables appearing in the `lastprivate` clause or the `reduction` clause of the `for` construct. Other clauses of the `for` construct do not affect the Gen and $Kill$ sets of the directive nodes.

Using Gen and $Kill$ sets for each node, we can construct dataflow equations for reaching definitions analysis as follows:

$$In(n) = \cup_{p \in pred(n)} Out(n) \qquad (1)$$

$$Out(n) = Gen(n) \cup (In(n) - Kill(n)). \qquad (2)$$

Note that the set of predecessors of node n in the PFG, denoted by $pred(n)$, include predecessors with respect to both sequential and synchronization edges. We can solve the dataflow equations by the iterative algorithm over the PFG. Though the dataflow equations are identical to that of the reaching definitions analysis for sequential programs, parallelism is introduced by the construction of transfer functions and propagation via synchronization edges.

3.3. Memory synchronization analysis

Reaching definitions analysis, described in the previous section, is the extension of a dataflow problem for sequential programs into a parallel setting. In contrast, memory synchronization analysis does not have a sequential counterpart. Memory synchronization analysis finds the variables that must be synchronized at each synchronization point.

Shared data in OpenMP programs is synchronized by the flush operations which are performed only at `flush` directives or other directives that imply flush operations. Therefore, between such synchronization points, a compiler can optimize programs without considering interactions between threads. At synchronization points, however, extra memory accesses and memory barrier instructions are inserted by the compiler, resulting in data synchronization overhead.

In practice, there are many implicit flush operations executed in an application, and all visible shared data are synchronized at such operations. This leads to a large amount of redundant memory synchronization, a serious problem for software DSMs. The purpose of memory synchronization analysis is to find the minimal set of memory synchronizations required for correct execution and to remove or improve memory synchronization operations. If a compiler detects shared data that do not have to be synchronized at a certain synchronization point, that data can be allocated to a register or moved across that synchronization point. Such information can also be used for detecting redundant control synchronizations, as explained in Section 4.3.

Our memory synchronization analysis algorithm finds a sufficient condition for memory synchronization that ensures correct execution. The algorithm is comprised of the following steps.

1. Analyze direct reaching definitions for each synchronization node. *Direct reaching definitions for node n are definitions that possibly reach node*

n without intervening definitions or flush operations of the same variable. Let $RDefGen(n)$ be the set of direct reaching definitions for node n.

2. Analyze exposed uses for each synchronization node. *Exposed uses* for node n are uses that are reachable from node n without intervening definitions of the same variable. Note that such uses may be executed in a thread different from that in which synchronization node n is executed. Let $RUse(n)$ be the set of exposed uses for node n. $RUse$ sets can be computed by the backward propagation of uses over the PFG.

3. Let $WSync(n)$ be the set of definitions that may need to be synchronized at synchronization node n. Then $WSync(n)$ is computed by the following formulae:

$$
\begin{aligned}
&WSync(n) \\
&= RDefGen(n) \cap RUse(n)
\end{aligned}
\tag{3}
$$

Here, *may* information for $WSync$ sets is computed because *may* information for the $RDefGen$ and $RUse$ sets are used; *must* information can be computed using the *must* information for direct reaching definitions and exposed uses. The set of uses that may need to be synchronized at synchronization node n, $RSync(n)$, is computed in a similar manner.

When a definition in set $WSync(n)$ is executed, the modified variable must be synchronized *before* executing node n, because other threads may use the value assigned by those definitions after synchronization at node n. Conversely, when a use in set $RSync(n)$ is executed, the variable to be read must have been synchronized *after* executing node n, because other threads may have modified the variables to be used before synchronization at node n.

3.4. Cross-loop data dependence analysis for parallel loops

So far we have described dataflow analysis techniques for scalar variables. In this section we deal with array dataflow analysis.

Parallel loops in OpenMP programs are `doall` type loops, i.e., there are no loop-carried data dependencies without explicit synchronization. Therefore, data dependence analysis within a single parallel loop is not so important. In contrast, cross-loop data dependence analysis for parallel loops is effective for optimizing communications for software-implemented coherence schemes.

We extended the reaching definition analysis algorithm in Section 3.2 to deal with array sections. We denote a section of array A as $A[lb : ub]$, where the lower bound of the section is lb and the upper bound is ub. Array section analysis algorithms are similar to their sequential counterpart, except for the way to deal with OpenMP constructs. For example, we compute array sections defined in a parallel loop as follows:

1. Compute array elements defined in the loop body. In this step, the loop counter of the parallel loop can appear in the array bounds. Other variables appearing in the array bounds must be loop invariant.

2. At the barrier synchronization immediately succeeding the parallel loop, replace the loop counter in the array bounds with the loop bound, thereby extending an access to an array element to an access to an array section. Before that barrier, it is not guaranteed that all definitions in the loop are observable by all thread.

If precise information cannot be computed, we can approximate it in a conservative manner. All definitions in a parallel loop reach successive memory synchronization points, unless they are killed by intervening definitions of that array. Similarly, all uses in a parallel loop are exposed to previous memory synchronization points, unless they are killed by intervening definitions of that array.

The array sections computed for each parallel and serial loop are propagated over a parallel flow graph and Def-Use chains are computed for the arrays. Cross-loop data dependence analysis is then performed for each Def-Use pair.

Cross-loop data dependence analysis for parallel loops is performed in a manner similar to that for sequential loops [28]. First, the loop bounds are adjusted so that both loops have identical bounds. The cross-loop dependence distance from the subscript expressions is then computed.

We can find possible inter-thread data dependence from the cross-loop data dependence, because the cross-loop data dependence exposes inter-iteration data dependence and thus we can find possible data dependence between chunks. In any scheduling policy, only array elements that have inter-chunk data dependence cause inter-thread data dependence. Moreover, inter-thread data dependence can be completely computed at compile-time if the scheduling policy is `static`. In this way, we find data sharing patterns in arrays and use that information for optimization.
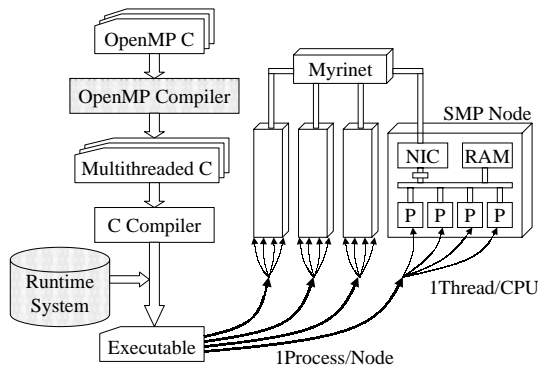
Fig. 3. The compiler-directed software DSM system.

4. Optimization in a compiler-directed software DSM

4.1. System overview

We are developing a compiler-directed software dstributed shared memory system (CD-SDSM) for OpenMP [20]. The CD-SDSM transparently executes OpenMP programs on a cluster of SMPs. Since fine-grain coherence control is performed by software, the compiler can optimize coherence operations by using source-level information. Such optimization is crucial to obtaining good performance on such systems.

Our CD-SDSM consists of an OpenMP compiler and a runtime library. Figure 3 depicts components of the CD-SDSM. The compiler translates OpenMP programs into multi-threaded C programs with runtime library calls. The generated C programs are compiled by a native C compiler and linked with our runtime library and others. Generated executable code is executed on each SMP node in an SPMD fashion. A part of the address space in each process running on SMP nodes is virtually shared between nodes, and the coherence of the shared space is maintained by software. The shared space is divided into 64-byte segments called *lines*, and these lines are the units of coherence control. In naive instrumentation, the compiler insert coherence control codes (*check codes*) for each shared data access. The compiler optimizes these codes by using source-level information so as to reduce the coherence overhead. In the rest of this section, we describe optimizations for our CD-SDSM. These techniques can more or less be applied to other platforms such as SMPs, CC-NUMAs, and page-based software DSMs.

4.2. Coherence optimization

Coherence overhead is a serious problem for software DSMs. The memory synchronization analysis described above finds minimal synchronization operations for correct execution. Therefore, we flush shared variables at each synchronization point only if that data needs to be synchronized at that point. This optimization greatly reduces the coherence overhead for implicit flush operations. We also optimize coherence operations for arrays according to their sharing patterns.

We will explain coherence optimization by using the example OpenMP program shown in Fig. 4. This program is a stencil code that solves Laplace equations by an iterative method. Three variables are shared across threads in this program. Arrays u and uu are two-dimensional arrays and most of their elements are modified and read in each iteration of the outermost loop. Scalar variable err holds the value of the norm.

Figure 5 shows the translated code fragment of the first parallel loop in the program shown in Fig. 4. This code was naively translated and check codes were inserted into each shared data access. Function _check_before_read() checks the status flags of the lines to be accessed and updates the lines if they hold stale values. Function _check_before_write() is similar except that a line is not updated if the entire line is to be modified. Function _check_after_write() writes modified data back to the home node and invalidates copies on other nodes. These three check codes are basic ones; more sophisticated check codes are available for optimizations.

In this non-optimized code, the execution overhead of the check codes is very large, and single thread execution is ten times or more slower than serial execution. Therefore, optimization of check codes is crucial for obtaining good performance.

Figure 6 shows the another translated code fragment in which optimizations without parallel dataflow analysis are performed. In this case, the check code for a loop invariant variable n is hoisted out of the loop, and check codes for consecutive data are merged into a check code for a larger region.

Parallel dataflow analysis enables further optimization. Figure 7 shows the reference pattern of array uu in the case of four thread execution. We also assume that the value of variable n is 100. The figure shows references for each array section accessed by thread T2. For example, array section $uu[25][1:100]$ is modified and read by thread T1 and read by thread T2. Threads T3 and T4 do not access this section. Therefore, the

```
double u[2048+2][2048+2], uu[2048+2][2048+2], err;
int n;
#pragma omp parallel shared(u,uu,err,n)
{
  int i,j,k;
  double err_local,tmp;
  do {
#pragma omp for nowait
    for (i=1;i<=n;i++)
      for (j=1;j<=n;j++)            first parallel loop
        uu[i][j]=u[i][j];
    err_local=0.0;
#pragma omp single
    err=0.0;
    /* implicit barrier */
#pragma omp for nowait
    for (i=1;i<=n;i++)
      for (j=1;j<=n;j++) {
        u[i][j]=(uu[i-1][j]+uu[i+1][j]+uu[i][j-1]+uu[i][j+1])/4.0;
        tmp=fabs(u[i][j]-uu[i][j]);
        if (tmp>err_local) err_local=tmp;
      }
#pragma omp critical
    /* implicit flush */
    if (err_local>err) err=err_local;
    /* implicit flush */
#pragma omp barrier
  } while (err>1.0e-5);
}
```

Fig. 4. A Laplace equation solver.

```
for (i=lb;i<=ub;i++) [
  // update lines if they are stale
  _check_before_read(&n,sizeof(int));
  for (j=1;j<=n;j++)
    // update lines if they are stale
    _check_before_read(&u[i][j], sizeof(double));
    _check_before_write(&uu[i][j], sizeof(double));
    uu[i][j] = u[i][j];
    // write back to the home and invalidate other copies
    _check_after_write(&uu[i][j], sizeof(double));
  }
  _barrier();
}
```

Fig. 5. Non-optimized code.

coherence of this section has to be maintained only for T1 and T2. Moreover, reaching definition analysis for the array sections showed that array section $uu[25][1:100]$ is modified by thread T1 in the first parallel loop and used by threads T1 and T2 in the second parallel loop. This implies that communication required for coherency occurs at the end of the first parallel loop and that the needed operation is a copy from T1 to T2. Using such information, the compiler can generate explicit writer-initiated communication to update copies on other nodes.

Figure 8 shows fully optimized code for the same parallel loop. In it, explicit remote copies are generated for the accesses to array uu. Since our platform is
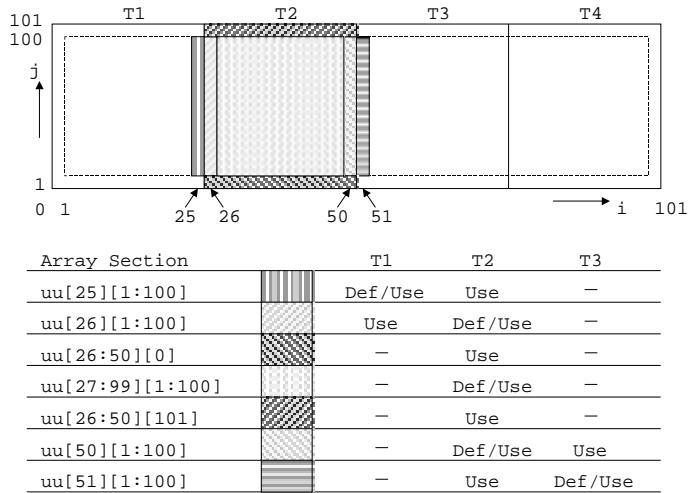
```
// check for a loop invariant variable
_check_before_read(&n, sizeof(int));
for (i=lb; i<=ub; i++) {
  // merged check codes
  _check_before_read(&u[i][1], sizeof(double)*n);
  _check_before_write(&uu[i][1], sizeof(double)*n);
  for (j=1; j<=n; j++)
    uu[i][j] = u[i][j];
  _check_after_write(&uu[i][1], sizeof(double)*n);
}
_barrier();
```

Fig. 6. Optimized code without PDA.

an SMP cluster, and threads running on the same node have consecutive thread numbers, adjacent threads may be executed on the same node. Therefore explicit re-

Array Section		T1	T2	T3
uu[25][1:100]		Def/Use	Use	—
uu[26][1:100]		Use	Def/Use	—
uu[26:50][0]		—	Use	—
uu[27:99][1:100]		—	Def/Use	—
uu[26:50][101]		—	Use	—
uu[50][1:100]		—	Def/Use	Use
uu[51][1:100]		—	Use	Def/Use

Fig. 7. Reference pattern of array uu.

```
for (i=lb;i<=ub;i++)
  for (j=1;j<=n;j++)
    uu[i][j] = u[i][j];
// update the copy on the previous node
if (_is_first_thread_on_node() && _my_node_no > 0)
  _update(&uu[lb][1], sizeof(double)*n, _my_node_no-1);
// udpate the copy on the next node
if (_is_last_thread_on_node() && _my_node_no < _n_nodes-1)
  _update(&uu[ub][1], sizeof(double)*n, _my_node_no+1);
_barrier();
```

Fig. 8. Optimized code with PDA.

mote copy is performed only when the reader thread is executed on an adjacent node.

For accesses to array u, no check codes are inserted because each element of array u is accessed by only a single thread within the entire parallel region. In other words, elements of array u are not shared between threads.

In this way, we can optimize coherence operations aggressively using parallel dataflow information. Though this kind of optimization is specific to compiler-directed software DSMs, optimizations presented in the next two sections are beneficial on many platforms.

4.3. Redundant barrier removal

Compiler optimization to find and remove redundant barriers is beneficial for several reasons:

– Barrier synchronization imposes a large overhead on programs that have poor load balance.

– Coherence overhead associated with barrier synchronization on software DSMs is much larger than that on SMPs and hardware DSMs.
– Coarse-grain parallelization is better than parallelizing each loop separately using the `parallel for` directive. When a compiler merges multiple parallel regions into one larger parallel region, there will be many redundant barriers.

The memory synchronization analysis presented in Section 3.3 can be used to detect redundant barriers. If $WSync(S) \cap RSync(S) = \emptyset$ for barrier S, no inter-thread flow dependencies require barrier synchronization at S, because if there is an inter-thread flow dependence across S for variable v, the writer of v must flush v at the synchronization point preceding barrier S and the reader of v must flush v at the synchronization point following barrier S. Possible inter-thread output dependence and anti dependence can also be found using the sets computed in the memory synchronization analysis.

Figure 9 shows an example of redundant barrier removal. Variables a, b, and c in the code fragment

```
#pragma omp for
  for (i=0;i<n;i++) {
    a[i] = ...;
  } // barrier S1

#pragma omp single
{
  for (i=0;i<n;i++)
    ... = a[i];
  b = ...;
}    // barrier S2

#pragma omp for
  for (i=0;i<n;i++) {
    c[i] = ...;
  } // barrier S3
  ... = b;
```

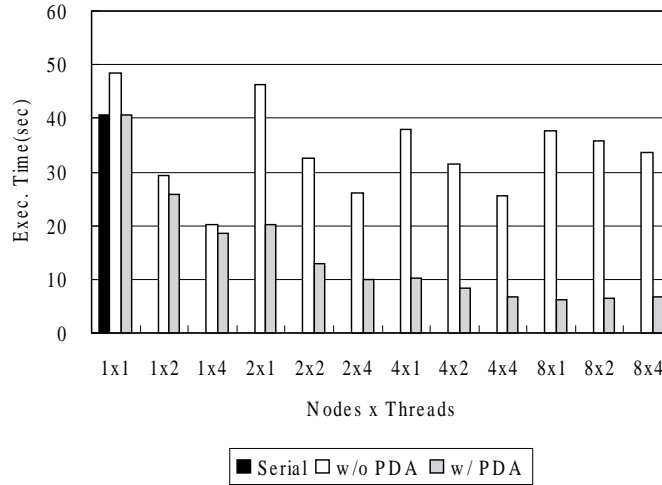| barrier	WSync	RSync
S1	{a[0:n-1]}	{a[0:n-1]}
S2	{b}	{ }
S3	{c[0:n-1]}	{b}

Fig. 9. Redundant barrier removal.

Fig. 10. Execution times of Laplace.

are shared variables. In this fragment, implicit barrier $S2$ is not required for inter-thread flow dependence because $WSync(S2) = \{b\}$ and $RSync(S2) = \emptyset$, so $WSync(S2) \cap RSync(S2) = \emptyset$. There are no output or anti dependencies across barrier $S2$ because $RDefGen(S2) \cap RDefGen(S3) = \emptyset$ and $RDefGen(S3) \cap RUseGen(S1) = \emptyset$. Therefore, the removal of barrier $S2$ does not introduce race conditions or other unexpected flow of data.

In practice, redundant barrier removal should be effective in the following cases:

– in combination with optimizations to merge multiple parallel regions into a large single parallel region,
– in orphaned barrier directives where redundancy depends on the calling context of the function.

4.4. Privatization of dynamically allocated objects

The OpenMP specification does not support privatization or selective flush operations for dynamically allocated objects. This leads to large redundant coherence overheads for such objects. We can find dynamically allocated private data and optimize the program as follows. If no shared pointers point to a dynamically allocated data, that data cannot be accessed by threads other than the thread in which that data is allocated. We allocate such objects in a private space, e.g., a private heap or a stack, and remove any coherence operations for such objects. This optimization improves not only the coherence overheads but also the data locality.

4.5. Performance evaluation

We evaluated the effectiveness of compiler optimizations for our compiler-directed software DSM by using

Table 1
Execution times and speedups of the Laplace program

#Nodes	without PDA		with PDA	
×#CPUs	Time (sec.)	Speedup	Time (sec.)	Speedup
Serial	40.58	1.00	–	–
1 × 1	48.53	0.84	40.53	1.00
1 × 2	29.27	1.39	25.80	1.57
1 × 4	20.10	2.02	18.51	2.19
2 × 1	46.20	0.88	20.22	2.01
2 × 2	32.66	1.24	12.94	3.14
2 × 4	26.06	1.56	9.88	4.11
4 × 1	37.96	1.07	10.13	4.01
4 × 2	31.50	1.29	8.44	4.81
4 × 4	25.48	1.59	6.71	6.05
8 × 1	37.80	1.07	6.27	6.47
8 × 2	35.72	1.14	6.41	6.33
8 × 4	33.59	1.21	6.60	6.15

Table 2
Execution times and speedups of the JOR program

#Nodes	without PDA		with PDA	
×#CPUs	Time (sec.)	Speedup	Time (sec.)	Speedup
Serial	18.72	1.00	–	–
1 × 1	20.59	0.91	18.80	1.00
1 × 2	11.80	1.59	10.86	1.72
1 × 4	6.08	3.08	5.64	3.32
2 × 1	10.36	1.81	9.47	1.98
2 × 2	6.06	3.09	5.52	3.39
2 × 4	3.25	5.76	2.98	6.28
4 × 1	5.27	3.55	4.80	3.90
4 × 2	3.24	5.78	2.86	6.55
4 × 4	2.17	8.63	1.73	10.82
8 × 1	2.83	6.61	2.55	7.34
8 × 2	2.53	7.40	1.72	10.88
8 × 4	2.39	7.83	1.89	9.90

two data-parallel applications with different memory access characteristics. Since these programs does not have redundant barriers or dynamically allocated private data, we evaluated effectiveness of coherence optimizations solely. We used prototype compiler and runtime system for experiments, and a part of optimization is performed by hand.

We used a 200-MHz PentiumPro-based SMP cluster, COMPaS [19]. It has eight 4-way SMP nodes, connected via a Myrinet network interface. We used the Solaris 2.5.1 operating system and the Solaris thread library for intra-node parallelism. For communication via Myrinet, we used the NICAM communication library.

The first example, an explicit Laplace equation solver, is the stencil code presented in Fig. 4. Arrays uu and u are both 2048×2048 in this experiment, and it takes 20 iterations of the outer-most loop. Figure 10 shows the execution times for different numbers of nodes and for different numbers of threads on each node. Table 1 shows execution times and speedup.

In the Laplace program, most shared data are modified and read in each iteration of the outer-most loop. Therefore, performance is unacceptable due to coherence overhead when all shared data are kept coherent at each synchronization point. However, our optimization using parallel dataflow analysis removes the coherence operations for most shared data. The differences between serial execution time and single-thread execution times represents the execution overhead of check codes. By using PDA, that overhead is greatly reduced because check codes for non-shared data are removed. Optimization using PDA also reduces communication overhead when multiple nodes are used. When the case of without PDA, redundant update of shared lines occured frequently and most of them are removed when

the case of with PDA. It should be noted that we had poor scalability within our SMP nodes due to the small bandwidth of the shared bus and poor performance of barrier and lock operations limited the performance when larger number of nodes are used.

The second example is a Jacobi overrelaxation solver (JOR). This program solves linear equations $Ax = b$ by an iterative method. Matrix A is 4096×4096 and it iterates eleven times. The structure of the program is similar to that of the Laplace program, except that it computes the value of vector x rather than of two-dimensional matrix u. Therefore, most shared data, matrix A and vector b, are not modified in the parallel region. This means that the volume of coherence-induced communications is much smaller even if the program was optimized without PDA.

Figure 11 and Table 2 shows the execution times and speedup of the JOR program. In this case, the performance improvement due to the aggressive optimizations using parallel dataflow analysis was mainly the reduction of the execution overhead of the check codes. It had a smaller impact on performance than reducing the communication volume. Similar to Laplace, we had poor performance on 8-node configuration due to poor performance of barrier and lock operations.

5. Related work

Research on analyses or optimizations of OpenMP programs is still rare, but there are several articles on other explicit parallel programs.

General frameworks for analyzing explicit parallel programs were presented by Chow and Harrison [2], Grunwald and Srinivasan [5], and Ferrante et al. [4]. In their work, `cobegin/coend` parallelism [2], event
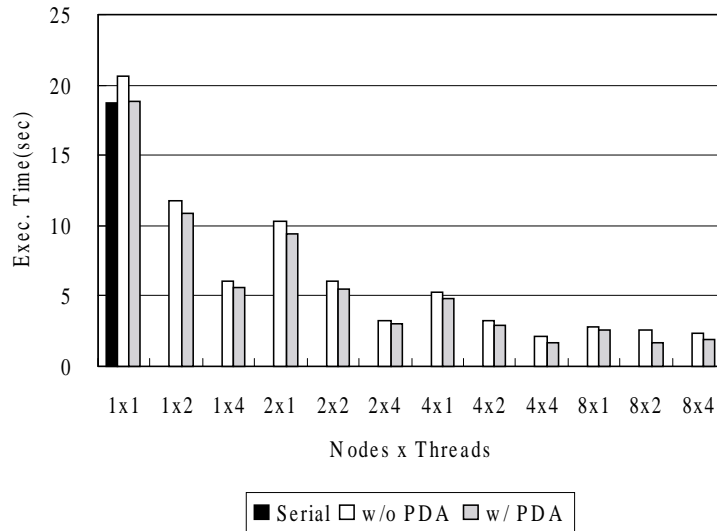
Fig. 11. Execution times of JOR.

synchronizations [5], and `doall` loops [4] are considered. In addition, Grunwald and Srinivasan [5] point out the advantages of relaxed memory consistency for compiler optimizations.

Static single assignment (SSA) form for parallel programs was proposed by Srinivasan et al. [25]. Novillo et al. [16] and Lee et al. [15] presented optimization algorithms using concurrent SSA form. Collard [3] proposed array SSA form for parallel programs.

Knoop and Steffen [10] presented a code motion algorithm for `cobegin/coend` parallel programs. Krishnamurthy and Yelick [13] proposed communication optimization for SPMD programs with shared variables. Pugh and Rosser [17] presented communication optimization within a single parallel loop.

Previous work on analyses and optimizations for explicit parallel programs dealt with programs whose parallelization constructs or semantics are different from those of the OpenMP API. The differences in the semantics of the parallel loops and memory consistency model make the algorithms quite different.

Chiueh and Verma [1], Scales et al. [22], Schoinas et al. [24], and Inagaki et al. [9] presented software DSMs that rely on aggressive compiler optimizations. However, the compiler optimizations in these papers are limited to optimizations within the interval between adjacent synchronization points and thus cannot optimize across threads, as our method can. Hu et al. [8], Scherer et al. [23], and Sato et al. [21] described page-based software DSMs for OpenMP but did not mention compiler optimizations.

Tseng [27] and Han and Tseng [6] demonstrated the effectiveness of compiler optimizations for DSMs by eliminating or lessening the control synchronizations. Koufaty and Torrellas [12] described compiler optimizations for data forwarding. We can perform similar optimizations for OpenMP programs by using our analysis techniques.

6. Conclusion

We presented parallel dataflow analysis and optimization algorithms for OpenMP programs. We designed an internal representation that represents both intra- and inter-thread flows of data. The semantics of OpenMP directives and clauses were considered when constructing transfer functions of dataflow equations. Using such a dataflow analysis framework, we can analyze dataflow information across multiple threads. Optimization techniques using parallel dataflow information were originally designed for a compiler-directed software DSM system, but they are also applicable to other OpenMP implementations. Preliminary performance results on a compiler-directed software DSM show that our coherence optimization greatly impacts the execution performance of shared-write intensive programs.

We are implementing the algorithms described in this paper in an OpenMP compiler [18], and we will evaluate the effectiveness of our optimization techniques by using more applications.

Acknowledgements

References

[1] T. Chiueh et al., A Compiler-Directed Distributed Shared Memory System, in *Proc. of the 9th ACM Int'l Conf. on Supercomputing* (*ICS'95*), 1995, pp. 77–86.

[2] J.-H. Chow and W.L. Harrison. Compile-time Analysis of Parallel Programs that Share Memory. in *Proc. of 19th ACM SIGPLAN-SIGACT Symp. on Principles & Practice of Programming Languages* (*PPoPP'92*), 1992, pp. 130–141.

[3] J.-F. Collard, Array SSA for Explicitly Parallel Programs, in *Proc. of 5th European Conf. on Parallel Processing* (*EuroPar'99*), 1999.

[4] J. Ferrante et al., Computing Communication Sets for Control Parallel Programs, in *Proc. of 7th Int'l Workshop on Languages and Compilers for Parallel Computing* (*LCPC'94*), 1994, pp. 316–330.

[5] D. Grunwald and H. Srinivasan, Data Flow Equations for Explicitly Parallel Programs, in *Proc. of 4th ACM SIGPLAN Symp. on Principles & Practice of Parallel Processing* (*PPoPP'93*), 1993, pp. 159–168.

[6] H. Han and C.-W. Tseng, Compile-time Synchronization Optimizations for Software DSMs, in *Proc. of Int'l Parallel Processing Symposium* (*IPPS98*), 1998.

[7] D. Hisley et al., Porting and Performance Evaluation of Irregular Codes using OpenMP, in *Proc. of 1st European Workshop on OpenMP* (*EWOMP'99*), 1999.

[8] Y.C. Hu et al., OpenMP for Networks of SMPs, in *Proc. of 12th Int'l Parallel Processing Symp. and 9th Symp. on Parallel and Distributed Processing* (*IPPS/SPDP'99*), 1999.

[9] T. Inagaki et al., Supporting Software Distributed Shared Memory with an Optimizing Compiler, in *Proc. of Int'l Conf. on Parallel Processing* (*ICPP98*), 1998.

[10] J. Knoop and B. Steffen, Code Motion for Explicitly Parallel Programs, in *Proc. of 7th ACM SIGPLAN Symp. on Principles & Practice of Parallel Processing* (*PPoPP'99*), 1999, pp. 13–24.

[11] J. Knoop, Parallel Data-Flow Analysis of Explicitly Parallel Programs, in *Proc. 5th European Conf. on Parallel Processing* (*EuroPar'99*), 1999, pp. 391–400.

[12] D. Koufaty and J. Torrellas, Compiler Support for Data Forwarding in Scalable Shared-Memory Multiprocessors, in *Proc.*

Int'l Conf. on Parallel Processing (*ICPP'99*), 1999.

[13] A. Krishnamurthy and K. Yelick, Optimizing Parallel Programs with Explicit Synchronization, in *Proc. 1995 ACM SIGPLAN Conference on Programming Language Design and Implementation* (*PLDI'95*), 1995, pp. 196–204.

[14] K. Kusano et al., Performance Evaluation of the Omni OpenMP Compiler, in *Proc. of Int'l Workshop on OpenMP: Experiences and Implementations* (*WOMPEI'00*), 2000.

[15] J. Lee et al., Basic Compiler Algorithms for Parallel Programs, in *Proc. of 7th ACM SIGPLAN Symp. on Principles & Practice of Parallel Programming* (*PPoPP'99*), 1999, pp. 1–12.

[16] D. Novillo et al., Concurrent SSA Form in the Presence of Mutual Exclusion, in *Proc. 1998 Int'l Conf. on Parallel Processing* (*ICPP'98*), 1998.

[17] W. Pugh and E. Rosser, Iteration Space Slicing and Its Application to Communication Optimization, in *Proc. 1998 Int'l Conf. on Supercomputing* (*ICS'97*), 1997, pp. 221–228.

[18] Omni: RWCP OpenMP compiler project, (http://pdplab.trc. rwcp.or.jp/pdperf/Omni/).

[19] M. Sato, COMPaS: a PC-based SMP cluster, *IEEE Concurrency* **7**(1) (1999), 82–86.

[20] M. Sato et al., Design of OpenMP Compiler for an SMP Cluster, in *Proc. of 1st European Workshop on OpenMP* (*EWOMP'99*), 1999.

[21] M. Sato et al., OpenMP Compiler for a Software Distributed Shared Memory System SCASH, in *Proc. of Workshop on OpenMP Application and Tools* (*WOMPAT2000*), 2000.

[22] D.J. Scales et al., Fine-Grain Software Distributed Shared Memory on SMP Clusters, in *Proc. of 4th Int'l Symp. on High-Performance Computer Architecture* (*HPCA'98*), 1998.

[23] A. Scherer et al., Transparent Adaptive Parallelism on NOWs using OpenMP, in *Proc. of 7th ACM SIGPLAN Symp. on Principles & Practice of Parallel Processing* (*PPoPP'99*), 1999, pp. 96–106.

[24] I. Schoinas et al., Sirocco: Cost-Effective Fine-Grain Distributed Shared Memory, in *Proc. of Int'l Conf. on Parallel Architecture and Compilation Technique* (*PACT'98*), 1998.

[25] H. Srinivasan et al., Static Single Assignment for Explicitly Parallel Programs, in *Proc. of 20th ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages* (*POPL'93*), 1993, pp. 260–272.

[26] O. Tatebe et al., Impact of OpenMP Optimizations for the MGCG Method, in *Proc. of Int'l Workshop on OpenMP: Experiences and Implementations* (*WOMPEI'00*), 2000.

[27] C.-W.Tseng, Compiler Optimizations for Eliminating Barrier Synchronization, in *Proc. of 5th ACM SIGPLAN Symp. on Principles & Practice of Parallel Processing* (*PPoPP'95*), 1995.

[28] M. Wolfe, *High Performance Compilers for Parallel Computing*, Addison-Wesley, 1995.