Compiler Optimization Techniques for OpenMP Programs

Shigehisa Satoh, Kazuhiro Kusano, and Mitsuhisa Sato

Tsukuba Research Center, Real World Computing Partnership
1-6-1 Takezono, Tuskuba, Ibaraki 305-0032, JAPAN
{sh-sato,kusano,msato}@trc.rwcp.or.jp

Abstract

In this paper, we present some compiler optimization techniques for explicit parallel programs using OpenMP API. To enable optimizations across threads, we designed dataflow analysis techniques in which interaction between threads is effectively modeled. Structured description of parallelism and relaxed memory consistency in OpenMP make the analyses effective and efficient. We show algorithms for reaching definitions analysis, memory synchronization analysis, and cross-loop data dependence analysis for parallel loops. Our primary target is a compiler-directed software DSM system where aggressive compiler optimizations for software-implemented coherence scheme are crucial to obtain good performance. We also show optimizations applicable to general OpenMP implementations, namely redundant barrier removal and privatization of dynamically allocated objects. We consider compiler optimizations are beneficial for performance portability across various platforms and non-expert programmers. Experimental results for the coherency optimization in a compiler-directed software DSM system shows that aggressive compiler optimizations are quite effective for a shared-write intensive program because coherence-induced communication volume in such a program is much larger than the those for shared-read intensive programs.

1 Introduction

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

We first present our framework of dataflow analysis for OpenMP programs and concrete dataflow algorithms. We use internal representation called *Parallel Flow Graph(PFG)* which models both intra-thread and inter-thread flow of data. Dataflow analysis algorithms described in this paper are reaching definition analysis, memory synchronization analysis and cross-loop data dependence analysis for parallel loops. Dataflow information obtained by these analyses are useful for our optimizations.

We present optimization techniques to reduce synchronization and coherence overheads, and to improve data locality. These techniques are applicable to any OpenMP implementation while our primary target is a compiler-directed software DSM system[20]. The compiler-directed software DSM system provides shared memory image on top of distributed memory parallel computers assisted by the compiler which analyzes communication pattern and optimize coherence control codes. In such a system, aggressive compiler optimizations are crucial to obtain good performance because software-implemented coherence scheme has larger coherence overhead than SMPs and hardware DSMs.

We present preliminary performance results using hand-translated codes and runtime library prototype. From the experimental results, we found that the coherence optimizations have quite large impact on a shared-write intensive program because large amount of communications are reduced by the optimizations.

This paper is organized as follows. Firstly we discuss performance bottlenecks in OpenMP programs and optimizations to overcome these issues. Then we present dataflow analysis techniques in Section 3. Section 4 describe compiler optimizations especially for the compiler-directed software DSM system and present experimental results. In section 5, related work on program analyses and optimizations for explicit parallel programs and software DSMs are presented. Section 6 summa-

rize 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.

Control of threads are synchronized by directives such as the `barrier` directive and the `critical` directive. The `parallel` directive also 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 improve control synchronization overhead, OpenMP API provides means such as the `nowait` clause, orphaned directives and SPMD style parallelism. Here, SPMD style means that multiple parallel tasks are put in a single large parallel region to reduce overheads due to the creation of separate parallel regions. Therefore programmers can optimize control synchronizations at source level. However, there are still situations in which it is difficult or impossible to optimize programs at source level. For example, to add `nowait` clause to an orphaned `for` directive, the programmer must be sure that it is valid for all calling contexts.

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

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

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

There are two approaches to deal with platform dependent issues. Firstly, programmers can use non-standard features such as vender-specific directives sacrificing portability. The second is to put such burden to OpenMP implementations, i.e. compilers and runtime systems. The latter is preferable because it does not sacrifice portability. Since many OpenMP programs are written in very structured manner, it is easier to analyze and optimize programs effectively compared with other multi-threading API such as POSIX threads.

The another reason to validate compiler optimizations is the fact that there are many application programmers who are not experts of parallel programming. Such non-expert programmers do not want to spend much time to improve their programs and tend to write less efficient programs with simple structures, e.g. using loop-parallelism only.

3 Parallel Dataflow Analysis

We present dataflow analysis techniques for OpenMP programs to enable aggressive optimizations. Our analyses use internal representation which models both intra-thread flow of control and synchronization between threads. The semantics of OpenMP directives are also considered in each analysis. The relaxed memory consistency in OpenMP enables efficient and effective dataflow analyses for parallel programs because we need to take account of interaction between threads only at a synchronization point.

3.1 Internal Representation

We first define internal representation called *Parallel Flow Graph(PFG)* to model both flow of control 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 entry node and exit node of the program respectively.

Figure 1 shows an OpenMP program and its parallel flow graph.

```
#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;
}
```

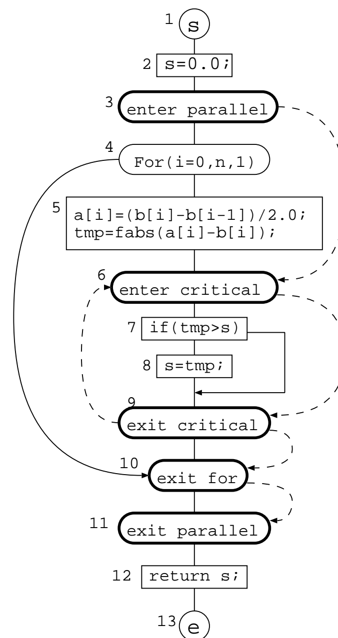


Figure 1: An OpenMP Program and its PFG

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 similar to nodes in 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, namely `barrier` and `flush`, is represented by a single directive node. A directive used as a construct is represented by two directive nodes which represents entry and exit of the construct, respectively. In the figure, sequential nodes and directive nodes are represented by rectangles and ovals, respectively. Directive nodes which implies flush operations are also called *synchronization nodes* and represented by thick ovals in the figure.

An edge in E represents either flow of control or ordering of synchronization events. *Control edges*, represented by solid arrows, represent flow of control in a single thread. Edges between sequential nodes are created similar to 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 the figure 1, we create edges as if at most one iteration can be executed, in other words, we do not create a back edge. This is due to the semantics of the `for` construct that no loop-carried dependence is assumed without explicit synchronizations.

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

We can model many parallel dataflow problems using parallel dataflow analysis framework which is comprised of parallel flow graph, a lattice of dataflow information and a set of transfer functions, similar to dataflow analysis frameworks for sequential programs. In such PDA frameworks, interactions between threads are reflected by propagating information via synchronization edges and transfer functions associated with directive nodes where the semantics of OpenMP directives and clauses are considered. In the following sections, we present algorithms for concrete dataflow analysis problems.

3.2 Reaching Definitions Analysis

Reaching definitions are widely used dataflow information. Reaching definitions analysis finds possible definitions which can be reached to each program point without intervening definitions of the same variable. Here we consider reaching definitions taking account of both intra-thread and inter-thread flow of data. We first describe a reaching definitions algorithm for scalar variables to explain basic idea of dataflow analysis for OpenMP programs.

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

We also compute Gen sets and $Kill$ sets for each node in PFG. These sets for sequential nodes are computed similar 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 clauses associated with it.

Let us consider construction of Gen sets 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 values of those shared variables become undefined. We add dummy definitions at the exit node for variables appeared in the `lastprivate` clause or the `reduction` clause of the `for` construct. Other clauses to the `for` construct do not affect to Gen sets and $Kill$ sets of the directive nodes.

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

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

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

Notice that a set of predecessors of node n in PFG, denoted by $pred(n)$, include predecessors with respect to both sequential edges and synchronization edges. We can solve the dataflow equations by iterative method over PFG. Though the dataflow equations are identical to 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 an extension of a dataflow problem for sequential programs to parallel setting. In contrast, memory synchronization analysis described in this section does not have sequential counterpart. Memory synchronization analysis finds variables which must be synchronized at each synchronization point.

Shared data in OpenMP programs is synchronized by the flush operation. The flush operations are performed only at the `flush` directives or other directives which implies flush operations. Therefore, between such synchronization points, a compiler can optimize programs without considering interaction between threads. However, extra memory accesses and memory barrier instructions are inserted by the compiler at synchronization points. This incurs data synchronization overheads.

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 and it is a serious problem for software DSMs. The purpose of the memory synchronization analysis is to find minimal set of memory synchronizations required for correct execution and remove or improve memory synchronization operations. If a compiler detects shared data which 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 shown in 4.3.

Our memory synchronization analysis algorithm finds sufficient condition of memory synchronization for the 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 which reach node n without intervening definitions or flush operations of the same variable. Let $RDefGen(n)$ be a set of direct reaching definitions for node n.

2. Analyze exposed uses for each synchronization node. Exposed uses for node n are uses which reach node n without intervening definitions of the same variable. Notice that such uses may be executed by a thread different from a thread which executes synchronization node n. Let $RUse(n)$ be a set of exposed uses for node n.

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

$$WSync(n) := RDefGen(n) \cap RUse(n)$$

Here we computed *may* information and *must* information can be computed by using *must* informations for direct reaching definitions and exposed uses. A set of uses which may be need to be synchronized at synchronization node n, $RSync(n)$, are also computed similarly.

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

3.4 Cross-Loop Data Dependence Analysis for Parallel Loops

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

Parallel loops in OpenMP programs are `doall` style loops, i.e. there are no loop-carried data dependence without explicit synchronization. Therefore data dependence analysis within a single parallel loop is not so important. Instead, cross-loop data dependence analysis for parallel loops is effective for communication optimizations for software-implemented coherence scheme.

We first extend the reaching definition analysis algorithm in 3.2 so as to deal with array sections. We denote an array section of array A as $A[lb : ub]$ if the lower bound of the section is lb and the upper bound of the section is ub. Array section analysis algorithms are similar to sequential counterpart except OpenMP constructs. For example, we compute array sections referenced in a parallel loop in the following steps:

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

2. Replace the loop counter in the array bounds with the loop bound and thus access to an array element is extended to an access to an array section.

If precise information can not be computed, then we can approximate it in a conservative manner. All definitions in a parallel loops reaches successive memory synchronization point, unless these definitions are killed by intervening definitions to that array. Similarly, all uses in a parallel loops are considered to be exposed to previous memory synchronization points, unless that uses are killed by intervening definitions to that array.

Array sections computed for each parallel and serial loops are propagated over parallel flow graph and we compute Def-Use chains for arrays. Then we perform cross-loop data dependence analysis for each Def-Use pair.

Cross-loop data dependence analysis for parallel loops is performed similar to the analysis for sequential loops[28]. We first adjust loop bounds so as to both loops have identical loop bounds. Then we compute cross-loop dependence distance from subscript expressions.

We can find possible inter-thread data dependence from cross-loop data dependence. Cross-loop data dependence exposes inter-iteration data dependence and thus we can find possible data dependence between chunks. In any scheduling, only array elements which have inter-chunk data dependence are the cause of inter-thread data dependence. Moreover, inter-thread data dependence can be completely computed at compile-time if scheduling policy is `static`. In this way, we find data sharing pattern of arrays and use that information for optimizations.

4 Optimizations in a Compiler-Directed Software DSM

4.1 System Overview

We presented basic design of a compiler-directed software DSM(CD-SDSM) at EWOMP'99[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 using source-level information. Such optimizations are crucial to obtain good performance on such systems.

Our CD-SDSM consists of an OpenMP compiler and a runtime library. The OpenMP compiler translates OpenMP programs into multi-threaded C programs with runtime library calls. Generated C programs are compiled by a native C compiler and linked with libraries. Generated executable code is executed on each SMP node in a SPMD fassion. A part of address space in each process running on SMP nodes is virtually shared between nodes and coherence of shared space is maintained by software. Shared space is divided into 64-byte segments(*lines*) and these lines are unit of coherence control. As a basis, the compiler insert coherence control code(*check codes*) for each shared data access. The compiler optimizes check codes using source-level information so as to reduce coherence overheads. In the rest of this section, we present

```
double u[102][102],uu[102][102];
double err;
#pragma omp parallel
{
  int i,j,k;
  double err_local,tmp;
  do {
#pragma omp for nowait
    for (i=1;i<=100;i++)
      for (j=1;j<=100;j++)
        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<=100;i++)
      for (j=1;j<=100;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);
}
```

Figure 2: Laplace Equation Solver

optimizations for our CD-SDSM, but more or less these technique can be applied to other platforms such as SMPs, CC-NUMAs, and page-based software DSMs.

4.2 Coherence Optimizations

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

Let us explain coherence optimizations using example OpenMP program in Figure 2. This program, Laplace, is a stencil code which solves Laplace equation by iterative method. There are three shared data used in this program. Two arrays u and uu are 2-dimensional square arrays and most elements of them are modified and read in each iteration of the outer-most loop. Scalar variable err is used to hold the value of the norm.

Figure 3 shows translated code fragment of the first parallel loop in the program in Figure 2. This

```
for (i=lb;i<=ub;i++) {
  // update lines if they are stale
  _check_before_read(&n,sizeof(int));
  for (j=1;j<=100;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 home and invalidate others
    _check_after_write(&uu[i][j],sizeof(double));
}
_barrier();
```

Figure 3: Non-Optimized Code

```
// 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)*100);
  _check_before_write(&uu[i][1],
             sizeof(double)*100);
  for (j=1;j<=100;j++)
    uu[i][j] = u[i][j];
  _check_after_write(&uu[i][1],
             sizeof(double)*100);
}
_barrier();
```

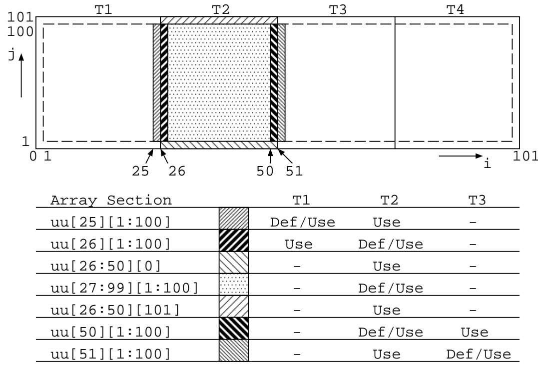
Figure 4: Optimized without PDA

code shows naive translation and check codes are inserted to each shared data access. Function `_check_before_read()` checks status flags of the lines to be accessed and update lines if they have stale copy. Function `_check_before_write()` is similar to the former function except that a line is not updated if entire line is to be modified. Function `_check_after_write()` write modified lines back to the home node and invalidate copies on other nodes. These three check codes are basic ones and more sophisticated check codes are available for optimizations.

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

Figure 4 shows translated code where optimizations without parallel dataflow analysis are performed. In this case, the check code for loop invariant variable is hoisted out of the loop, and check codes for consecutive data are merged into a check for a larger region.

Parallel dataflow analysis enables further optimizations. Figure 5 shows reference pattern of array uu in the case of four thread execution. The



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

Figure 5: Reference Pattern of Array uu

```
for (i=lb;i<=ub;i++)
  for (j=1;j<=100;j++)
    uu[i][j] = u[i][j];
// update copy of the previous thread
if (_is_first_thread_on_node()
    && _my_node_no > 0)
    _update(&uu[lb][1],
            sizeof(double)*100,
            _my_node_no-1);
// update copy of the next thread
if (_is_last_thread_on_node()
    && _my_node_no < _n_nodes-1)
    _update(&uu[ub][1],
            sizeof(double)*100,
            _my_node_no+1);
_barrier();
```

Figure 6: Optimized with PDA

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. Thread T3 and T4 do not access this section. Therefore coherence of this section have to be maintained for only T1 and T2. Moreover, it is found by reaching definition analysis for array sections that array section $uu[25][1:100]$ is modified by thread T1 in the first parallel loop and used by thread T1 and T2 in the second parallel loop. This implies that communication required for coherency occur at the end of the first parallel loop and needed operation is a copy from thread T1 to thread T2. Using such information, the compiler can generate explicit writer-initiated communication to update copies on other nodes.

Figure 6 shows fully optimized code for the same parallel loop. Here explicit remote copies are generated for the accesses to array uu. Since our platform is an SMP cluster and thread running on the same node have consecutive thread number, adjacent threads may be executed on the same node. So explicit remote copy is performed only when the reader thread is executed on the different 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 entire parallel region. In other words, elements of array u is 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, we also show optimizations beneficial on many platforms in the next two sections.

4.3 Redundant Barrier Removal

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

- Barrier synchronization will incur large overhead for programs which have poor load balance.
- Coherence overhead associated with barrier synchronization on software DSMs is much larger than those for SMPs or hardware DSMs.
- SPMD style parallelism is better than parallelizing each loop separately by the `parallel for` directive. When a compiler merges multiple parallel sections into one larger parallel section, there will be many redundant barriers.

Memory synchronization analysis presented in 3.3 can be used for detecting redundant barriers. If $WSync(S) \cap RSync(S) = \emptyset$ for barrier S, then no inter-thread flow dependences require barrier synchronization at S, because if there exists an inter-thread data dependence across S for variable v, then the writer of v must flush v at synchronization point preceding barrier S and the reader of v must flush v at synchronization point succeeding barrier S. Possible inter-therad output depdendence and anti dependence can also be found using sets computed in the memory synchronization analysis.

Figure 7 shows an example of redundant barrier detection. Variables a, b and c referred in the code fragment are shared variables. In this code fragment, implicit barrier $S2$ does not required for inter-thread flow dependence because $WSync(S2) = \{b\}$ and $RSync(S2) = \emptyset$, thus $WSync(S2) \cap RSync(S2) = \emptyset$. There are no output and anti dependences across barrier $S2$ because $RDefGen(S2) \cap RDefGen(S3) = \emptyset$ and

```
#pragma omp for          WSync        RSync
   for (i=0;i<n;i++) {
     a[i] = ...;
   } // barrier S1      {a[0:n-1]}  {a[0:n-1]}
#pragma omp single
{
   for (i=0;i<n;i++)
     ... = a[i];
   b = ...;
}    // barrier S2      {b}         { }
#pragma omp for
   for (i=0;i<n;i++) {
      c[i] = ...;
   } // barrier S3      {c[0:n-1]}  {b}
    ... = b;
```

Figure 7: Redundant Barrier Removal

$RDefGen(S3) \cap RUseGen(S1) = \emptyset$. Therefore the removal of barrier $S2$ does not introduce race conditions or other unexpected flow of data.

In practice, redundant barrier removal is expected to be effective in the following cases:

- Combination with optimizations to merge multiple parallel regions into large single parallel region
- Orphaned barrier directives where redundancy depends on calling context of the function

4.4 Privatization of Dynamically-Allocated Objects

The OpenMP specification does not support privatization nor selective flush operations for dynamically-allocated objects so far. This leads to large redundant coherence overheads for such objects. We find dynamically allocated private data and optimize the program in the following way. If no shared pointers may point to a dynamically allocated data, that data can not 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 that object. This optimization improves not only coherence overheads but also data locality.

4.5 Experimental Results

We evaluated effectiveness of compiler optimizations for our compiler-directed software DSM using two data-parallel application with different memory access characteristics. These programs do not have redundant barriers nor dynamically-allocated private data. So we evaluate effectiveness of coherence optimizations solely. We used hand-compiled codes for experiments because our compiler is under development.

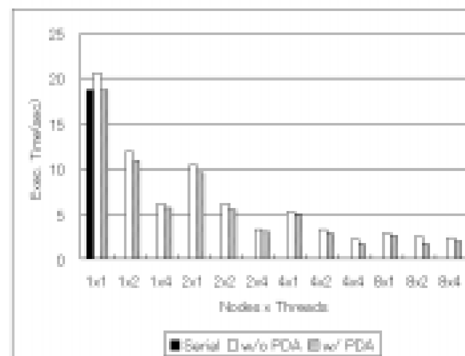


Figure 8: Execution Times of Laplace

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

The first example, explicit laplace equation solver, is a stencil code presented in Figure 2. Size of arrays uu and u are both 2048×2048 in this experiment. It takes 20 iterations of the outer-most loop.

Figure 8 shows execution times of Laplace for different configurations of the number of nodes and the number of threads on each node.

In Laplace, most shared data are modified and read in each iteration of the outer-most loop. Therefore the 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 coherence operations for most shared data. It should be noted that we have poor scalability within SMP nodes due to poor bandwidth of the shared bus.

The second example is Jacobi overrelaxation solver(JOR). This program solves linear equation $Ax = b$ by an iterative method. The size of matrix A is 4096×4096 and iterate 10 times. Structure of the program is similar to Laplace, except that we compute value of vector x rather than 2-dimensional matrix u in Laplace. Therefore most shared data, 2-dimensional 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 is opti-

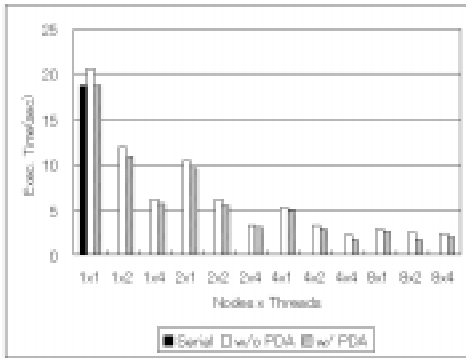


Figure 9: Execution Times of JOR

mized without PDA.

Figure 9 shows execution times of JOR. In this case, performance improvement due to aggressive optimizations using parallel dataflow analysis is mainly the reduction of execution overhead of check codes, that have small impact on performance compared with the reduction of communication volume.

5 Related Work

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

General frameworks for analyzing explicit parallel programs are presented by Chow and Harrison[2], Grunwald and Srinivasan[5], and Ferrante et al.[4]. In their work, `cobegin/coend` parallelism[2], event 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 are proposed by Srinivasan et al.[25]. Novillo et al. [16] and Lee et al.[15] present optimization algorithm using concurrent SSA form. Collard [3] propose array SSA form for parallel programs.

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

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

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

Tseng[27] and Han and Tseng[6] show effectiveness of compiler optimizations for DSMs by eliminate or lessen control synchronizations. Koufaty and Torrellas[12] show compiler optimizations for data forwarding.

6 Conclusion

We presented parallel dataflow analysis (PDA) and optimization algorithms for OpenMP programs. We designed internal representation which represents both intra-thread and inter-thread flow of data. Semantics of OpenMP directives and clauses are also considered when constructing transfer functions. Using such dataflow analysis framework, we can analyze dataflow information across multiple threads. Optimization techniques using parallel dataflow information are 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 shows that our coherence optimizations have great impact on execution performance for a shared-write intensive program.

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

References

[1] T.Chiueh et al. A Compiler-Directed Distributed Shared Memory System. *ICS'95*, pp. 77–86, 1995.

[2] J-H.Chow and W.L.Harrison. Compile-time Analysis of Parallel Programs that Share Memory. *POPL'92*, pp. 130–141, 1992.

[3] J-F.Collard. Array SSA for Explicitly Parallel Programs. *Euro-Par'99*, 1999.

[4] J.Ferrante, et al. Computing Communication Sets for Control Parallel Programs. *LCPC'94*, pp. 316–330, 1994.

[5] D.Grunwald and H.Srinivasan. Data Flow Equations for Explicitly Parallel Programs. *PPoPP'93*, pp. 159–168, 1993.

[6] H.Han and C-W. Tseng. Compile-time Synchronization Optimizations for Software DSMs. *IPPS'98*, 1998.

[7] D.Hisley, et al. Porting and Performance Evaluation of Irregular Codes using OpenMP. *EWOMP'99*, 1999.

[8] Y.C.Hu, et al. OpenMP for Networks of SMPs. *IPPS/SPDP'99*, April 1999.

[9] T.Inagaki, et al. Supporting Software Distributed Shared Memory with an Optimizing Compiler. *ICPP'98*, 1998.

[10] J.Knoop and B.Steffen. Code Motion for Explicitly Parallel Programs. *PPoPP'99*, pp. 13–24, 1999.

[11] J.Knoop. Parallel Data-Flow Analysis of Explicitly Parallel Programs. *EuroPar'99*, pp. 391–400, 1999.

[12] D.Koufaty and J.Torrellas. Compiler Support for Data Forwarding in Scalable Shared-Memory Multiprocessors. *ICPP'99*, 1999.

[13] A.Krishnamurthy and K.Yelick. Optimizing Parallel Programs with Explicit Synchronization. *PLDI'95*, pp. 196–204, 1995.

[14] K.Kusano, et al. Performance Evaluation of the Omni OpenMP Compiler, *WOMPEI'00*, Oct. 2000(to appear).

[15] J.Lee, et al. Basic Compiler Algorithms for Parallel Programs. *PPoPP'99*, pp. 1–12, 1999.

[16] D.Novillo, et al. Concurrent SSA Form in the Presence of Mutual Exclusion. *ICPP'98*, 1998.

[17] W.Pugh and E.Rosser. Iteration Space Slicing and Its Application to Communication Optimization. *ICS'97*, pp. 221–228, 1997.

[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):82–86, 1999.

[20] M.Sato, et al. Design of OpenMP Compiler for an SMP Cluster. *EWOMP'99*, 1999.

[21] M.Sato, et al. OpenMP Compiler for a Software Distributed Shared Memory System SCASH. *WOMPAT2000*, July 2000.

[22] D.J.Scales, et al. Fine-Grain Software Distributed Shared Memory on SMP Clusters. *HPCA'98*, 1998.

[23] A.Scherer, et al. Transparent Adaptive Parallelism on NOWs using OpenMP. *PPoPP'99*, pp.96-106, 1999.

[24] I.Schoinas, et al. Sirocco: Cost-Effective Fine-Grain Distributed Shared Memory. *PACT'98*, 1998.

[25] H.Srinivasan, et al. Static Single Assignment for Explicitly Parallel Programs. *POPL'93*, pp. 260–272, 1993.

[26] O.Tatebe, et al. Impact of OpenMP Optimizations for the MGCG Method. *WOMPEI'00*, Oct. 2000(to appear).

[27] C-W.Tseng. Compiler Optimizations for Eliminating Barrier Synchronization. *PPoPP'95*, July 1995.

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