Analyses for the Translation of OpenMP Codes into SPMD Style with Array Privatization

Zhenying Liu, Barbara Chapman, Yi Wen, Lei Huang,

Tien-Hsiung Weng, Oscar Hernandez

Department of Computer Science, University of Houston
{zliu,chapman,yiwen,leihuang,thweng,oscar}@cs.uh.edu

Abstract. A so-called SPMD style OpenMP program can achieve scalability on ccNUMA systems by means of array privatization, and earlier research has shown good performance under this approach. Since it is hard to write SPMD OpenMP code, we showed a strategy for the automatic translation of many OpenMP constructs into SPMD style in our previous work. In this paper, we first explain how to ensure that the OpenMP program consistently schedules the parallel loops. Then we describe the analyses required for the translation of an OpenMP program into an equivalent SPMD-style OpenMP code with array privatization. Interprocedural analysis is required to help determine the shape of the privatized array and the quality of the translation.

1 Introduction

OpenMP has emerged as a popular parallel programming interface for medium-scale high performance applications on shared memory platforms. However, there are some problems associated with obtaining high performance under this model, and they are exacerbated on ccNUMA platforms. These include the latency of remote memory accesses, poor cache memory reuse, barriers and false sharing of data in cache.

SPMD-style programs access threads and assign computations to them using the thread ID, rather than via the straightforward loop level OpenMP directives. By means of the first-touch policy, data may be allocated to the local memory of the first thread that accesses them. Therefore the affinity between the data and thread is constructed. The following loop iterations can be reorganized to reuse the data that are already in the local memory. For example, [18] showed how this method can be used to exploit memory affinity with the example of LU decomposition. The OpenMP LU code is rewritten so that each thread will reuse the data that are first accessed by the current thread. The performance of this kind of SPMD style code is better than the original OpenMP code due to the data locality [18]. However, when the reused data

This work was partially supported by the DOE under contract DE-FC03-01ER25502 and by the Los Alamos National Laboratory Computer Science Institute (LACSI) through LANL contract number 03891-99-23.

are not stored consecutively in memory as can happen, for example, in an OpenMP Fortran program, if an array is accessed frequently by row dimension simultaneously by multiple threads, the array elements required by multiple threads may co-exist on the same page. So the performance of the program may suffer from false sharing problems at the page level, depending on the size of each array dimension and the size of a page. Even if we reuse data stored consecutively in memory, there may be some page allocation overheads which are not controlled by the compiler or the user.

More aggressive SPMD style coding privatizes arrays systematically by creating private instances of (sub-)arrays. These codes show good scalability for ccNUMA systems [5, 6, 25]; they are superior to an OpenMP program with straightforward parallelization of loops and an SPMD OpenMP program taking advantage of the first touch policy running on ccNUMA systems. The false-sharing problem is alleviated when accessing the privatized data, since the privatized arrays are allocated in the local stack of the current thread. However, a number of nontrivial program modifications are required to convert a program to the SPMD style. It is thus hard for a user to write SPMD style OpenMP code, especially for a large application. Ease of program development is a major motivation for adopting OpenMP and it is important to provide some help for users who wish to improve their program performance by array privatization. One method is to provide a tool that supports the generation of SPMD style OpenMP code from an OpenMP code with loop-level parallelism. We are building just such a tool based upon the open source Open64 compiler infrastructure.

We have developed a compiler strategy in order to translate loop-parallel OpenMP code into an SPMD-like form [17]. This requires automatic privatization of a large fraction of the data or arrays, the creation of shared buffers to store the resulting *non-local* array references, storage for the non-local data accessed (the halo area), and the generation of instructions to share data between threads.

We show the components of the tool that translates OpenMP programs into equivalent SPMD ones in Fig. 1: compiler frontend, consistent loop scheduling, privatization analysis, translation, optimization and compiler backend. Oval objects represent the *frontend* and *backend* of the existing OpenMP compiler; we are using the Open64 compiler [20] due to its powerful analyses and optimizations. The rectangular objects enclosed by dashed lines represent the main steps in our SPMD translation. An OpenMP program must be analyzed to test for *consistent loop scheduling* before the *privatization analysis*, which determines the profitability of the privatization. Sometimes we may be able to enforce the consistency of loop schedules. However, our definition of consistency is relatively restrictive and there are a variety of addi-

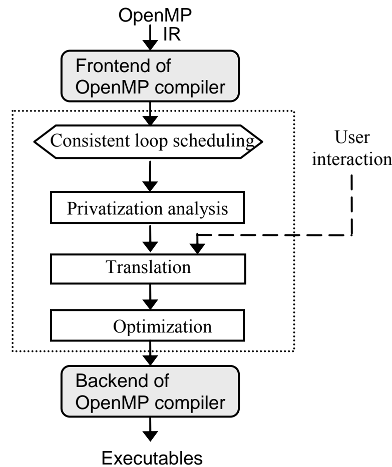

Fig. 1. The framework of SPMD translation

tional situations in which we are able to perform our SPMD transformation despite the fact that this test fails. We give two examples of code patterns that can be handled despite the fact that they are not consistent, and where the strategy used to translate them differs from the straightforward approach that can otherwise be applied. In other cases, we need to interact with the user if we are to accomplish the task of array privatization. The basic *translation* and *optimization* have been thoroughly described in [17]. In this paper, we focus on issues related to consistency of loop scheduling and privatization analysis.

2 Consistent loop scheduling for data reuse

The goal of our SPMD transformation is to help the user achieve a scalable OpenMP code. To do so, we attempt to perform array privatization by following the semantics of OpenMP loop scheduling. Therefore we need to ensure that the different parallel loops in the code lead to a consistent reuse of portions of individual arrays. If they do so, the area that they reference will be used to construct a private array. Without consistency in array accesses, array privatization will potentially lead to large private arrays and involve extensive sharing of data between threads, both of which are undesirable. We may exploit compiler technology to detect whether the loop scheduling is consistent, by using the semantics of loop scheduling directives in OpenMP to discover the loop partitioning and summarize the array sections associated with each thread within the loop using regular section analysis. In this section, we discuss the consistency test. Afterwards, we show how we deal with loops involving procedure calls, before considering how this information is exploited. Our ability to handle programs is not limited to those with consistency; this issue is explored later.

2.1 Consistency Test

In order to test the consistency of loop scheduling, we use regular section analysis to construct the array sections referenced by individual threads within parallel loops. Array section analysis is a technique to summarize rectilinear sub-regions of an array referenced in a statement, sequence of statements, loop or program unit. Here, our main focus is on forming a section to describe the references made by a thread within a parallel loop. An array section will suffice to describe the region of an array referenced by a thread as the result of a single occurrence of that array in a statement within the loop. We summarize the region of the array referenced by that thread by forming the union of the array sections that are derived from the individual references. Currently, we have decided to rely on a standard, efficient triplet notation based regular sections [10], rather than more precise methods such as simple sections [1], and region analysis [24]. However, when a union of two array sections cannot be precisely summarized, we do not summarize them immediately, but keep them in a linked list. A summary of the linked list is forced if the length of the list reaches a

certain threshold, and we mark that the access region of that parallel loop is approximate. We refer to this information as the *array section per thread* for the loop. We also compute the complete array section that is referenced in the parallel loop (by all threads), and check the consistency of the regions loop scheduling for each shared array. If the parallel loops are consistent for a shared array, we call this array *privatizable*.

Once we have created the array section per thread information for each parallel loop, we must check to determine whether or not threads consistently access the same portion of an array. Our *consistency test algorithm* works as follows. We first summarize the array section accesses for each parallel loop as indicated above. We then compare the array sections obtained for a given thread throughout the computation. We currently have strict requirements for consistency, as given in the three rules below. We hope to be able to relax them in future. Essentially, these rules describe a test between a pair of loop nests for each shared array they reference. In one case, two loop nests are operating on entirely different regions of an array, so that the data referenced by a thread in one loop is distinct from the data referenced by any thread in the second loop. An example would be a pair of loops where one loop initializes the boundary of a mesh and the second loop initializes interior elements. Another case covers the situation when the array section per thread in one loop subsumes the array section per thread in the second loop. We give an example below. There are a number of ways in which the definition of consistency might be extended to cover cases where the array sections referenced in the loops are "roughly" the same. At present, we have chosen to rely upon the following definition and to explore separately how to treat a variety of codes that only partially conform to it, or do not conform at all.

Let A_1 be the section of a shared array A that is accessed (by all threads) in a parallel loop L_1. Let A_1^t be the subsection of A_1 that is accessed by thread t. Similarly, let A_2 be the section of array A that is referenced in loop L_2 and A_2^t be the subsection of A_2 that is accessed by thread t in L_2. We consider the references to array A, and therefore the loop scheduling with respect to A, to be consistent between the parallel loops L_1 and L_2 if one of the following rules apply.

- **Rule 1**: In order to know if the loop scheduling is consistent, we first compute whether the intersection of $A_1 \cap A_2 = \phi$. If so, it is consistent; otherwise, we apply rule 2.
- **Rule 2**: We check whether the array section accessed by thread t in one of the loops contains the array section accessed by thread t in the other parallel loop by using the union operation. If $A_1^t \cup A_2^t = A_1^t$ or $A_1^t \cup A_2^t = A_2^t$, one array section contains the other. In this case, we consider the references in the pair of loops to be consistent. Otherwise, we apply rule 3 to further compute the array sections of C.
- **Rule 3**: We compute the region $C = A_1^t \cap A_2^t$, the common subsection shared by A_1 and A_2. If it is 90% of the entire set of elements A accessed by thread t in the two loops, we consider the references in the pair of loops to be consistent.

The rules for the consistency test are conservative in that they handle only some of the cases in real world applications that can be handled. Further investigation is

needed to extend this test. The test will be applied interprocedurally for each shared array.

2.2 Examples of consistent and inconsistent schedules

```
!$omp parallel default(shared) private(i,j)
!$omp do
    do j = 2 , 999
       do i = 2 , 999
         A(i,j) = ( B(i-1,j)+B(i+1,j)
               + B(i,j-1)+B(i,j+1)) * c
       end do
     end do
!$omp do
     do i=1, 1000
       do j= 1, 1000
         B(i,j) = A(i,j)
       end do
     end do
!$omp end parallel
```

Fig. 2. A Jacobi code example with inconsistent loop scheduling

```
!$omp parallel default(shared) private(i,j)
!$omp do
    do j = 2 , 999
       do i = 2 , 999
         A(i,j) = ( B(i-1,j)+B(i+1,j)
               + B(i,j-1)+B(i,j+1)) * c
       end do
     end do
!$omp do
     do j=1, 1000
       do i= 1, 1000
         B(i,j) = A(i,j)
       end do
     end do
!$omp end parallel
```

Fig. 3. Another Jacobi code example with consistent loop scheduling

In OpenMP, a parallel do loop is partitioned into several sets of iterations according to the loop scheduling clauses (which may be the default clause). Threads will be assigned their own sets of iterations and will therefore require access to the subsections of shared arrays that occur in their iterations simultaneously. As mentioned, we use the term consistent loop scheduling to imply that threads access roughly the same regions of an array within multiple parallel loops in OpenMP.

We use the Jacobi program excerpt in Fig. 2 as an example to illustrate the concept of consistency and to show how to apply the consistency test. This code contains references to two arrays. We consider array B; similar reasoning applies to A. Our first rule for consistency checks whether the pair of loops access disjoint regions B_1 and B_2 of the array. However, the intersection of these regions is not empty and so we apply rule 2. Rule 2 is not satisfied either, because $B_1^t \cup B_2^t \neq B_1^t$ and $B_1^t \cup B_2^t \neq B_2^t$. In other words, B_1 and B_2 do not contain each other. Then we calculate the region C as shown in Fig. 4(c), where thread 0 is selected to determine the consistency. Since C is far less than 90% of the entire set of elements A accessed in the two loops by thread t, the loop schedule is not consistent and B is not privatizable.

If we examine the loop nest informally, we will see that the strategy for parallelizing the first loop implies that a given thread will access a block of rows of B, whereas in the second loop nest, the thread will access a block of columns of B. This lack of consistency in access is in some sense encouraged by OpenMP, as the user is advised that loop nests may be individually parallelized. Although better strategies exist, they may not have been chosen. The program in Fig. 3 shows a different parallelization strategy. In this case, threads will access blocks of rows of A and B in each loop nest.

Furthermore, when we evaluate these regions, we determine that for array *A* the array sections per thread are identical. For array *B*, the region referenced in the first loop will subsume the region accessed in the second loop. Therefore the loop schedules are consistent with reference to both *A* and *B*, these arrays are privatizable, and we will be able to apply our privatization algorithm without further examination of the code.

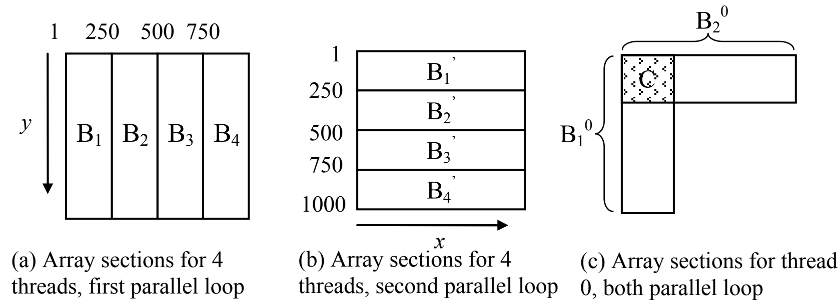

Fig. 4. Jacobi program and consistency test of Fig. 2

3 Interprocedural analysis

Our ability to determine consistency of access in a reasonable fashion relies on our ability to determine the array elements accessed by threads accurately. For this we may defer combining array sections in a loop nest. However, we also want to minimize the introduction of inaccuracies during interprocedural analysis.

3.1 Call Graph

Parallel loops may contain calls to procedures. We must also evaluate access patterns throughout the entire program. In order to do so well, our interprocedural analysis is implemented on call graphs created by a precise call graph algorithm [26]. For any data flow problem including our privatization problem, one call site may have several different sequences of actual arguments as a consequence of several different call chains that will be followed at run time. The call graphs constructed by most call graph algorithms are not able to take the multiple sequences of actual arguments into consideration [3, 9, 22]. Hence they are sometimes imprecise. To overcome this, we have developed an algorithm that enables us to precisely determine the call chains that will occur. Our call graph is a multi-graph where each procedure in the program is represented by a single node, as usual. Directed edges between nodes indicate that the source procedure invokes the sink procedure. An edge records information on the predecessor edge in a call chain of the program. For each pair of nodes in the call graph, there may be zero, one or more edges connecting them. However, no two edges will be identical with respect to the associated information.

```
      Program Jacobi
    …
!$omp parallel default(shared)
!$omp&shared(A,B,size) private(k)
       do k=1,ITER
         call sub1(B,size)
         call sub2(A,B,size)
         call sub3(A,B,size)
       end do
!$omp end parallel
      …
```

```
subroutine sub1(A,size)
integer size,i,j
double precision A(0:size,0:size)
!$omp do
    do j=0,size
      do i=0, size
          A(i,j) = 1.0
      end do
    end do
!$omp end do
end
```

```
subroutine sub2(A,B,size)
integer i,j,size
double precision A(0:size,0:size)
double precision B(0:size,0:size)
!$omp do
   do i = 1, size-2
     do j = 1, size-2
        A(i,j) = (B(i,j-1) + B(i,j+1)
               + B(i-1,j) + B(i+1,j))
     enddo
   enddo
!$omp end do
end
```

```
subroutine sub3(C,D,size)
integer m,n,size
double precision C(0:size,0:size)
double precision D(0:size,0:size)
!$omp do
     do m = 0, size-1
       do n = 0, size-1
         D(m,n) = C(m,n)
       enddo
     enddo
!$omp end do
end
```

Fig. 5. A Jacobi OpenMP program with several program units

We give a simple example to show the information that is appended to the edges in the graph. It considers a Jacobi program consisting of several program units in Fig. 5 and its call graph in Fig. 6. There are four nodes in the call graph to represent the procedures. The edges in the call graph represent the calling relations. For example, the edge e_1 depicts that *Main* calls *sub1*.

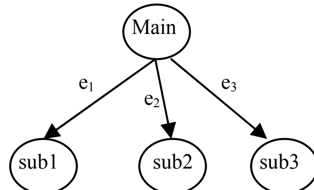

Fig. 6. The call graph for the Jacobi OpenMP program in Fig. 5

3.2 Interprocedural Algorithm

In order to handle parallel loops containing procedure calls, we must determine the array region accessed within the called procedure, using our call graph to help do so. The algorithm in Fig. 7 operates on the call graphs we introduced in Section 3.1. It traverses the call graph in a depth-first manner. The algorithm is recursive. It follows a call chain from a program entry point to the end of this call chain. Upon returning, the algorithm gathers and binds information in bottom-up order which guarantees correct binding between formal parameters and actual arguments.

```
1.  Procedure Inter_consistency_analysis(node v1, Edge eprev)
2.    if v1.complete = False then
3.      v1.privatizaion =Local_consistency_analysis(v1)
4.      v1.complete=True
5.    end if
6.    Result = 1
7.    for each edge ej where ej.prev= eprev in v1
8.      ej.privatizable=binding(Inter_consistency_analysis(ej.v2, ej), ej)
9.      Result = Result ∩ ej.privatization
10.   end for
11.   Result = v1.privatization ∩ Result
12.   return Result
13. End procedure
```

Fig. 7. The interprocedural consistency analysis algorithm

The algorithm needs to calculate local information for a procedure node only once, while the algorithm may visit a node more than once. Line 2 to line 5 of the algorithm in Fig. 7 guarantees this by examining the value of v_1.complete, which is true for completeness, and false for incompleteness. The *local consistency analysis* in line 3 involves the consistency test in Section 2.1, and records the corresponding array section information. Lines 7 to line 11 gather and assemble array section information for every edge, for which e_{prev} is their predecessor edge in the call chain. The binding function binds the formal parameters to the actual ones, where e_j stands for one of the out edges of the current procedure node v_1, and v_2 for one of the callees of the procedure v_1.

For the Jacobi example in Fig. 5, the initial invocation will be Inter_consistency_analysis(*Main*, *Null*) where *Main* is the program entry point and thus has no (*Null*) predecessor edges. In our call graph, each out edge of *Main* node has no predecessor. When we execute line 7 to line 10, each out edge of *Main* node will be examined individually. The algorithm gathers and assembles information for determining consistency from their callees and callees' successors, then binds the information to these edges. The array section per thread is finally achieved by forming the union of the individual array sections corresponding to array references. Here, we also follow the strategy of deferring the merging of array sections unless a threshold is reached. We also use this call graph and the associated information to ensure that our subsequent handling of the program is accurate, as will be explained in the previous section.

4 Privatization Analysis

4.1 Privatization Algorithm

Our privatization analysis works as follows. We first test if the OpenMP program has consistent loop scheduling. If so, we will carry out the privatization according to

OpenMP semantics. If it is largely consistent, then we may be able to modify the remaining "inconsistent" parallel loops to obtain consistency. This entails identifying which loop in the loop nest would result in the prevailing access pattern if parallelized; if dependence tests prove that this is legal, we then replace the originally parallelized loop with this one and an appropriate schedule. We may also be able to apply loop interchange to support this. Since only one level of parallelism is supported in OpenMP, we deal with one dimensional privatization at this stage. BLOCK partition is a default manner of privatization. Even if the loop scheduling is consistent, we still have to detect whether it is profitable to privatize arrays. For example, if we have large shadow areas (halos) for threads and share data extensively between threads, it is not likely to be profitable to privatize arrays. If the above methods do not enable us to privatize arrays, we may be able to detect and deal with a known special case. We illustrate this by discussing two important special cases, an LU decomposition OpenMP program and an ADI-like code, below. Although neither of these have consistent array usage, array privatization can be used in both cases to achieve good performance. In Fig. 8, we show the above process in the privatization analysis algorithm.

```
1.  Procedure Privatization_Analysis
2.     Consistency test
3.     if (inconsistent) then
4.         Data dependence test
5.         Apply loop transformation
6.     else
7.         Profitability test
8.         Translate OpenMP into SPMD style with array privation
9.     end if
10.    if ( known special case ) then handling of the special case
11.    else   report to the users
12.    end if
13. End procedure
```

Fig. 8. Privatization analysis algorithm

We use an LBE OpenMP example to illustrate the local privatization algorithm. In Fig. 9, part of the OpenMP LBE program is shown. LBE, a computational fluid dynamics code, solves the Lattice Boltzmann equation and is provided by Li-Shi Luo of ICASE, NASA Langley Research Center [11]. It uses a 9-point stencil, and the neighboring elements are updated at each iteration. The consistency test will have a positive result in this case: the loop schedule is consistent and array *f* and *fold* are privatizable, since both of the parallel loops distribute the iterations in the *j*-loop which sweeps the second dimension of array *f* and *fold*. Each thread will access a contiguous set of columns of the original array. When such results are obtained from the test, we can immediately determine the size and shape of the corresponding private arrays for individual threads: we simply use the union of the array sections referenced in the different parallel loops. In this case, the transformation will be profitable because each thread only accesses the array elements inside its individual array section. Once we privatize *f* and *fold*, only two columns of arrays are non-local and the data sharing between threads is trivial in comparison to the amount of computation.

4.2 Special case 1: LU

```
!$omp parallel
  do iter = 1, niters
!$omp do
  do j = 1, Ygrid
    do i = 1, Xgrid
            fold(i,0,j) = f(i,0,j)
             ……
             fold(i,8,j) = f(i,8,j)
    end do
  end do
!$omp end do
……
!$omp do
do j=1, Ygrid – 1
  do i=2, Xgrid -1
    f (i,0,j) = Fn(fold(i,0,j)
    f (i+1,1,j) = Fn(fold(i,1,j))
    f (i,2,j+1) = Fn(fold(i,2,j))
    ……
    f(i+1,8,j-1) = Fn(fold(i,8,j))
  end do
end do
!$omp end do
……
```

Fig. 9. OpenMP LBE code

```
!$omp parallel
  do k = 1, n-1
!$omp single
  lu(k+1:n,k) = lu(k+1:n,k)/lu(k,k)
!$omp end single
!$omp do
    do j = k+1,n
      lu(k+1:n, j) =
          lu(k+1:n,j) – lu(k,j)* lu(k+1:n,k)
    end do
!$omp end do
  end do
!$omp end parallel
```

Fig. 10. OpenMP LU decomposition

In some cases when the loop scheduling in OpenMP program is not consistent, we cannot arrive at a suitable array privatization by means of following its OpenMP semantics, for instance, the Jacobi program in Fig. 2 and the LU OpenMP program in Fig. 10. If we apply loop interchange to the second loop nest, and parallelize the new outer loop to get another Jacobi program in Fig. 3, then threads will access roughly the same set of data in each loop and will moreover reference contiguous areas in memory.

In the case of the LU decomposition, since its OpenMP program does not consistently schedule the loops, we may ask the user if we can privatize the arrays. In the LU program, the inner loop is a parallel one, while the loop bounds of the inner loop involve the upper level loop iteration variable. Therefore the inner parallel loop has dynamic loop bounds and the associated array regions for each thread changes from iteration to iteration; the size of the array regions shrinks. Although this is inconsistent loop scheduling, it is possible to obtain a privatization based upon a CYCLIC assignment of columns of arrays to threads, such that the resulting code shows little data sharing between threads.

4.3 Special case 2: ADI

ADI (Alternating Direction Implicit) application [12] is another special example in which each parallel loop sweeps a distinct dimension, and it spends almost the same execution time on each parallel loop. The data dependence in ADI prevents loop

interchange and parallelization of the other loop. Therefore we are not able to privatize arrays by directly following OpenMP semantics.

```
!$omp parallel
!$omp do
        do j=1, N
          do i= 1, N
             A(i,j)=A(i,j)-B(i,j)*A(i-1,j)
          end do
        end do
!$omp end do
!$omp do
        do i= 1,N
          do j= 1, N
            A(i,j)=A(i,j)-B(i,j) * A(i,j-1)
           end do
         end do
!$omp end do
!$omp end parallel
```

Fig. 11. The ADI-like OpenMP code

```
Double precision A(0:N,N),B(0:N,N)
!$omp parallel
        chunk = N/omp_get_num_threads()
!$omp do
        do j=1, N
          do i= 1, N
             A(i,j) = A(i,j) - B(i,j) * A(i-1,j)
          end do
         end do
!$omp end do
     sync(1:N,omp_get_thread_num())=.F.
       do i= 1,N
!$omp flush ( A, sync )
        if(id.ne.0 .and..not. sync(i,id-1))then
          do while (.not. sync(i,id-1))
!$omp flush ( A, sync )
          end do
         end if
         do j=1+id*chunk,chunk + id*chunk
              A(i,j) = A(i,j)- B(i,j) * A(i,j-1)
         end do
         sync(i,id) = .T.
!$omp flush
      end do
!$omp end parallel
```

Fig. 12. ADI-like OpenMP code in SPMD style

```
!$omp threadprivate(Aloc, Bloc)
      sync = .F.
      chunk = N/omp_get_num_threads()
!$omp parallel shared(shadow, sync)
         do j=1, chunk
           do i= 1, N
             Aloc(i,j)=Aloc(i,j)  &
                Bloc(i,j)*Aloc(i-1,j)
           end do
         end do
      do i= 1,N
!$omp flush ( shadow, sync )
          if ( id .ne. 0 ) then
             do while (.not. sync(i,id-1))
!$omp flush ( shadow, sync )
             end do
             Aloc(i,0) = shadow(i,id-1)
           end if
           do j= 1, chunk
              Aloc(i,j)=Aloc(i,j)-    &
               Bloc(i,j)*Aloc(i,j-1)
           end do
           shadow(i,id) = Aloc(i,chunk)
           sync(i,id) = .T.
!$omp flush (shadow,sync)
       end do
!$omp end parallel
```

Fig. 13. ADI-like OpenMP code in SPMD style with array privatization

However, ADI OpenMP code (Fig. 11) can be rewritten in an SPMD style with consistent loop scheduling (Fig. 12). Furthermore, we may privatize arrays according to the OpenMP semantics of this OpenMP SPMD code (Fig. 13). In Fig. 14, we show results of executing three different versions of the ADI code. The experiments were

conducted on Origin 2000 systems at the National Center for Supercomputing Applications (NCSA) in multi-user mode. The MIPSpro 7.3.1.3 Fortran 90 compiler was used with the option: -mp. We set the first touch policy and set the page migration environmental variable off. Altogether, the SPMD style code with array privatization outperforms its OpenMP and SPMD OpenMP code without array privatization.

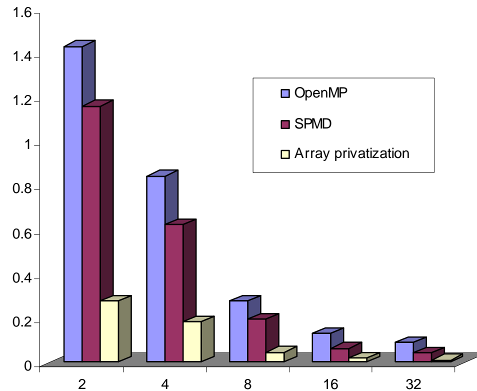

Fig. 14. The execution time different version of OpenMP programs

5 Related work

Scalability of OpenMP codes is naturally a concern on large-scale platforms. The data locality problem for OpenMP on ccNUMA architectures is well known and a variety of means have been provided by vendors to deal with it, including first touch allocation. Researchers have proposed strategies for page allocation [19] and migration; data distribution directives are implemented by SGI [23] and Compaq [2] to improve data locality, although their directives differ. Page granularity distributions are easier to handle, but may be imprecise. An element-wise (HPF-like) data distribution can distribute array elements to the desired processor or memory. For example, the DISTRIBUTE_RESHAPE directive provided by SGI uses the specification of the distribution to construct a "processor_array" which contains pointers to the array elements belonging to each processor, so as to guarantee the desired distribution. However, a consequence is that pointer arithmetic is introduced to realize accesses to the array elements, and performance of the resulting code is poor if options that optimize pointers are not selected. Instead of this approach, our strategy is to base the translation on *threadprivate* arrays which are guaranteed to be local and do not involve the use of additional pointers.

A number of researchers investigated the problem of finding an efficient data layout automatically for codes that were being compiled for distributed memory systems in the context of HPF [8, 13]. The major steps that they identified in the research for a data distribution are as follows: alignment analysis [15, 14], decision on the manner of distribution (BLOCK or CYCLIC), adjustments to the block size of dimensions for

a cyclic distribution, and determination of how many processors are to be used as the target for each distributed dimension.

But there are significant differences between this problem and ours. On distributed memory systems, each array must be distributed in order to enable parallelism: it is usually not feasible to replicate more than a few arrays, as memory costs will be prohibitive. This is not true for OpenMP programs. Even if some of the shared arrays in an OpenMP program are not privatized, we are still able to execute the code in parallel quite reasonably. Therefore, we have more options. In addition, automatic data distribution for distributed memory systems is based on sequential programs which have no analysis for potential parallelism. In contrast, OpenMP programs already include information that some loops are parallelizable. Moreover, if we follow the semantics of the user-specified OpenMP loops, we are provided with an assignment of work, and hence data references, to the individual threads. This information, in the form of array sections that are accessed by the threads, gives us initial information for determining an appropriate array privatization. Note, too, that the regions may overlap and that our analysis is greatly simplified by the fact that OpenMP specifies a simple and direct mapping of loop iterations to threads, rather than requiring us to consider each statement in a loop nest individually. While this may lead to a larger amount of data being referenced by multiple threads, it makes our problem much more tractable.

A large body of work exists that considers strategies for gathering and storing information on array sections accessed, and for summarizing these both within individual procedures and across a program. Among the best know strategies are regular sections [3], simple sections [1], atomic images and atoms [16]. We use bounded regular sections in this context. However, we have adopted a new strategy for obtaining precision, by merging regions where this does not lead to loss of precision and otherwise recording lists of references. We have also considered the problem of recording and using the correct call chains when performing this analysis.

In this paper, we have focused on the problem of array privatization for arrays that are accessed in regular (structured) patterns, which is quite widespread. In [21], the access pattern statistics are summarized for Perfect and SPEC benchmark suite. The authors note a very high percentage of simple affine subscripts (e.g., A(i)) in many applications. For instance, among the thirteen programs in its statistical table 1, four of them including *swim* and *tomcatv*, have at least 97% accesses that are simple affine subscripts.

6 Conclusions and Future Work

OpenMP codes must contend with the data locality problems incurred by multiple caches and remote memory access latency. The SPMD style enhances the capability of OpenMP to achieve better data locality. This paper discusses the analyses which will enable us to partially automate the translation of loop-parallel codes to SPMD style via a compiler. Thus the performance benefits are obtained without devolving this burden onto the users.

We have focused our attention on applications with regular data access patterns. OpenMP programs provide an explicit mapping of computation to threads and hence specify which data is needed by an individual thread; we thus have a basis for implementing a privatization algorithm. However, these codes may lead to some difficult problems. In the ADI (Alternating Direction Implicit) kernel, the access pattern of shared arrays changes for different parallel loops that makes the array privatization very hard to perform according to OpenMP semantics. We have determined that a transformation into a macro pipelined version (that can be described by using FLUSH directives) can help us obtain data locality and considerably lessen the synchronization cost for ADI-like applications. In the current state of our work, we must ask the user to help us decide how to deal with codes where the access patterns change.

More work is needed to improve our ability to perform this work automatically, to determine in some cases whether or not privatization is profitable, and to lessen the amount of global synchronization incurred in OpenMP codes. There are also many possible ways in which we could attempt to improve upon the code produced. We plan to perform experiments to learn more about these issues. We will also experiment to find out whether it is beneficial in practice to break one critical section into two or more smaller critical sections, and how beneficial it is to change global synchronization to point-to-point synchronization. We expect to replace the barriers in our generated SPMD style codes via FLUSH directives in order to improve our results. Finally, we plan to test our SPMD style codes not only on ccNUMA systems, but also in a PC cluster running Omni/Scash, where our aggressive array privatization should have a strong impact on performance.

References

1. Balasundaram, V., and Kennedy, K.: A Technique for Summarizing Data Access and Its Use in Parallelism Enhancing Transformations. Proceedings of the 1989 ACM SIGPLAN Conference on Programming Language Design and Implementation, Portland, Oregon, June 21-23, (1989) 41-53
2. Bircsak, J., Craig, P., Crowell, R., Cvetanovic, Z., Harris, J., Nelson, C. A. and Offner, C. D.: Extending OpenMP for NUMA machines. Scientific programming. Vol. 8, No. 3, (2000)
3. Callahan, D. and Kennedy, K.: Analysis of Interprocedural Side Effects in a Parallel Programming Environment. Journal of Parallel and Distributed Computing. Vol. 5, (1988)
4. Chandra, R., Chen, D.-K., Cox, R., Maydan, D. E., Nedeljkovic, N., and Anderson, J. M.: Data Distribution Support on Distributed Shared Memory Multiprocessors. Proceedings of the ACM SIGPLAN '97 Conference on Programming Language Design and Implementation, Las Vegas, NV, June (1997)
5. Chapman, B., Bregier, F., Patil, A. and Prabhakar, A.: Achieving High Performance under OpenMP on ccNUMA and Software Distributed Share Memory Systems. Currency and Computation Practice and Experience. Vol. 14, (2002) 1-17
6. Chapman, B., Patil, A., Prabhakar, A.: Performance Oriented Programming for NUMA Architectures. Workshop on OpenMP Applications and Tools (WOMPACT‘01), Purdue University, West Lafayette, Indiana. July 30-31 (2001)
7. Gonzalez, M., Ayguade, E., Martorell, X. And Labarta, J.: Complex Pipelined Executions in OpenMP Parallel Appliations. International Conferences on Parallel Processing(ICPP 2001), September (2001)

8. Gupta M. and Banerjee, P.: PARADIGM: A Compiler for Automated Data Distribution on Multicomputers. Proceedings of the 7th ACM International Conference on Supercomputing, Tokyo, Japan, July 1993.
9. Hall, M. W. and Kennedy, K.: Efficient call graph analysis. ACM Letters on Programming Languanges and Systems, Vol. 1, No. 3, (1992) 227-242
10. Havlak, P. and Kennedy, K.: An Implementation of Interprocedural Bounded Regular Section Analysis. IEEE Transactions on Parallel and Distributed Systems, Vol. 2, No. 3, July (1991) 350-360
11. He, X., and Luo, L.-S.: Theory of the Lattice Boltzmann Method: From the Boltzmann Equation to the Lattice Boltzmann Equation. Phys. Rev. Lett. E, No. 56, Vol. 6, (1997) 6811
12. Jin, H., Frumkin, M. and Yan, J.: The OpenMP Implementation of NAS Parallel Benchmarks and its Performance. NAS Technical Report NAS-99-011, Oct. (1999)
13. Kennedy, K. and Kremer, U.: Automatic Data Layout for High Performance Fortran. Proceedings of the 1995 Conference on Supercomputing (CD-ROM), ACM Press, (1995)
14. Laure, E. and Chapman, B.: Interprocedural Array Alignment Analysis. Proceedings HPCN Europe 1998, Lecture Notes in Computer Science 1401. Springer, April (1998)
15. Li, J. and Chen, M.: Index domain alignment: Minimizing cost of cross-referencing between distributed arrays. Proc. Third Symp. on the Frontiers of Massively Parallel Computation, IEEE. October (1990): 424-433
16. Li, Z., and Yew, P.-C.: Program Parallelization with Interprocedural Analysis, The Journal of Supercomputing, Vol. 2, No. 2, October (1988) 225-244
17. Liu, Z., Chapman, B., Weng, T.-H., Hernandez, O.: Improving the Performance of OpenMP by Array Privatization. In the Workshop on OpenMP Applications and Tools (WOMPAT 2002), Fairbanks, Alaska, August (2002)
18. Nikolopolous, D. S., Artiaga, E., Ayguadé, E. and Labarta, J.: Exploiting Memory Affinity in OpenMP through Schedule Reuse. Third European Workshop on OpenMP (EWOMP 2001), (2001)
19. Nikolopoulos, D. S., Papatheodorou, T. S., Polychronopoulos, C. D., Labarta, J., Ayguadé, E.: Is Data Distribution Necessary in OpenMP? Proceedings of Supercomputing 2000, Dallas, Texas, November (2000)
20. The Open64 compiler. http://open64.sourceforge.net/
21. Paek, Y., Navarro, A., Zapata, E., Hoeflinger, J., and Padua, D.: An Advanced Compiler Framework for Non-Cache-Coherent Multiprocessors, IEEE Transactions on Parallel and Distributed Systems. Vol. 13, No. 3, March (2002) 241-259
22. Ryder, B. G.: Constructing the Call Graph of a Program. IEEE Transactions on Software Engineering, Vol. 5, No. 3, (1979) 216-225
23. Silicon Graphics Inc. MIPSpro 7 FORTRAN 90 Commands and Directives Reference Manual, Chapter 5: Parallel Processing on Origin Series Systems. Documentation number 007-3696-003. http:// techpubs.sgi.com
24. Triolet, R., Irigoin, F., and Feautrier, P.: Direct Parallelization of CALL statements. Proceedings of ACM SIGPLAN '86 Symposium on Compiler Construction, July (1986) 176-185
25. Wallcraft, A. J.: SPMD OpenMP vs. MPI for Ocean Models. Proceedings of First European Workshops on OpenMP (EWOMP'99), Lund, Sweden, (1999)
26. Weng, T.-H., Chapman, B., Wen, Y.: Practical Call Graph and Side Effect Analysis in One Pass. Technical Report, University of Houston, Submitted to ACM TOPLAS (2003)