# Improving the Performance of OpenMP by Array Privatization

Zhenying Liu, Barbara Chapman, Tien-Hsiung Weng, Oscar Hernandez

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

**Abstract.** The scalability of an OpenMP program in a ccNUMA system with a large number of processors suffers from remote memory accesses, cache misses and false sharing. Good data locality is needed to overcome these problems whereas OpenMP offers limited capabilities to control it on ccNUMA architecture. A so-called SPMD style OpenMP program can achieve data locality by means of array privatization, and this approach has shown good performance in previous research. It is hard to write SPMD OpenMP code; therefore we are building a tool to relieve users from this task by automatically converting OpenMP programs into equivalent SPMD style OpenMP. We show the process of the translation by considering how to modify array declarations, parallel loops, and showing how to handle a variety of OpenMP constructs including REDUCTION, ORDERED clauses and synchronization. We are currently implementing these translations in an interactive tool based on the Open64 compiler.

## 1 Introduction

OpenMP has emerged as a popular parallel programming interface for medium scale high performance applications on shared memory platforms. Strong points are its ability to support incremental parallelization, portability, and ease of use. However, the obstacles to scale an OpenMP code to hundreds or thousands of processors, as may be configured in ccNUMA systems, are latency of remote memory access, poor cache memory reuse, a large number of barriers and false sharing of data in cache. Good data locality is needed to overcome these performance problems.

It is possible to obtain data locality, as well as to minimize false sharing between threads on a ccNUMA system, via manual privatization of the arrays in an OpenMP program. However, a systematic use of array privatization that separates array elements that are shared by multiple threads from those elements of the same arrays that are not shared, requires extensive program modification. The so-called SPMD style of OpenMP programming is a systematic realization of this approach and it has been

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.

shown to provide scalable performance [2, 3, 20] that is superior to a straightforward parallelization of loops for ccNUMA systems.

SPMD style OpenMP code is distinct from ordinary OpenMP code: In most OpenMP programs, shared arrays are declared and PARALLEL DO (or FOR) directives are used to realize a distribution of work among threads possibly via explicit loop scheduling. In the SPMD style, a systematic privatization of arrays, by creating private instances of (sub-)arrays, provides opportunities to spread computation among threads in a manner that ensures data locality. When data that have been privatized need to be shared by two or more threads, the programmer must insert additional data structures and code to achieve this. This is because in OpenMP programs, interactions between threads occur via shared variables only; each thread may moreover access its own private variables which are not visible to other threads. Thus 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 the performance of a rapidly created program by adopting SPMD style. One approach is to provide a tool that supports the generation of SPMD style OpenMP code, either from a sequential program or from an OpenMP code with loop-level parallelism. We are developing just such a tool. It is being implemented using the open source Open64 compiler [14] infrastructure. The tool is interactive, because we believe that user involvement is essential for good results.

This paper is organized as follows. In the next section, we describe the basic translation to SPMD style, using a simple Jacobi code fragment to show the process step by step. The subsequent sections discuss the translation of OpenMP directives and clauses that may be encountered in the original OpenMP program. After this, an overview of related work is given and some conclusions are reached.

## 2 Overview of SPMD Translation

Programs written using SPMD style with array privatization have shown superior performance [2, 3] on the SGI Origin 2000, a typical ccNUMA architecture. Indeed, their superior data locality has even been shown to improve performance on small SMPs [2]. Our goal is to help the user translate OpenMP programs into equivalent SPMD ones, which are expected to achieve data locality and high performance in ccNUMA systems, without introducing extensions to OpenMP. We follow a translation strategy that privatizes a program's arrays automatically or semi-automatically by following the OpenMP semantics of loop scheduling. The work described here assumes we know which arrays are to be privatized and the manner of privatization.

SPMD (Single Program Multiple Data) fashion code is parameterized by the thread number and loaded by all the target processors. Arrays are generally distributed among threads in such a manner that the region of the array assigned to a thread becomes a private data structure and is allocated locally. If appropriately carried out, this permits a high degree of data locality. This implies that additional work is needed

whenever privatized array elements are used by other threads. In the following, we outline the general techniques required to translate OpenMP code into SMPD style for execution on a ccNUMA machine. While doing so, we point out the difference between the array privatization required by this style and the privatization commonly used in OpenMP codes.

### 2.1 SPMD Style with Array Privatization

Our privatization approach in support of SPMD translation differs from other work on array privatization [19]. First, in the previous research, the following condition must be satisfied in order to privatize an array: every fetch to an array element in a loop must be preceded by a store to the element in the same iteration of the loop. In other words, there may be no dependences between loop iterations involving this array. This condition does not need to hold in order for us to privatize an array to generate SPMD style code. Second, a thread may access array elements that are private to other threads in SPMD style code, while this is not allowed in previous array privatization. We must naturally perform additional code modifications in order to realize these accesses.

**Basic translation.** In this section, we describe the process of translating loop-parallel OpenMP programs into SPMD style code. We indicate the modifications needed for array declarations when privatizing arrays, the basic modification of loop nests, and a strategy for sharing the privatized data between threads when it is required by the code. We show a simple example of a Jacobi code which approximates the solution of a partial differential equation discretized on a grid, and translate this straightforward OpenMP program in Fig. 1 into corresponding SPMD style in Fig. 2. We discuss the translation of this code fragment to illustrate our approach in the following subsections.

**The declaration of private arrays.** In order to obtain an SPMD program, the major arrays must be distributed among the threads. Within the context of OpenMP, data distribution is achieved via privatization. The approach is similar to that of MPI in the sense that a data distribution is chosen by the user or a tool and modified data structures are declared that have a shape and size corresponding to the (thread-) local portion of the array. In general, our translation strategy will choose to create threadprivate data structures, as threadprivate variables are globally accessible within the code. As a consequence of the above, the size of the threadprivate arrays at run time will depend on the manner in which we have partitioned and distributed the arrays, as well as their original size and the number of threads. The thread that declares and defines a section of a privatized array is called the owner of that section.
In our example code fragment, we privatize arrays *A* and *B*. The manner in which we do so is not arbitrary, but depends on the approach taken by the user to parallelize the original OpenMP code. Here, the *j*-loop has been parallelized (how to achieve the loop bounds *start_y* and *end_y* of the *j*-loop in Fig. 2 is further explained below) and the threads each access a contiguous segment of the second dimension of arrays *A* and

*B*, whereas all elements in the first dimension are accessed. (In the current OpenMP definition, only one loop nest may be parallelized and the parallelization of two or more loops, which will often correspond to distribution of two or more array dimensions, is only possible via collapsing of loops. We expect this to change in the future.) Array section analysis computes the regions of *A* and *B* that are read and written by individual threads. We use the array sections that are written by threads to decide how to distribute the arrays. For example, in *S1* statement of Fig. 2, the array section *A*[1:1000,*id*×1000/*numthreads*:(*id*+1)×1000/*numthreads*] is defined by the thread with identifier *id*, where *numthreads* is the total number of threads, since the default block schedule is used. Therefore we distribute array *A* by block in the second dimension according to the loop scheduling in the OpenMP program. In this case, the parallelization strategy enables us to distribute data in a manner that privatizes almost all references to both arrays. In a more realistic program, this may not be the case. Since OpenMP by its nature does not expect the user to compare the data usage implied by the parallelization strategy, it is possible that the loops chosen for parallelization imply different data distributions in different parts of the program. Since this will result in many nonlocal accesses, no matter which distribution we base our translation on, it will seriously detract from the performance benefits of our approach. This is a challenge for our translation process and is one of the reasons why we do not expect to fully automate this translation. We return to this topic below. But note that achieving consistency in thread data usage is likely to benefit even small programs executed on just a few CPUs [2, 3].

```
double precision A(1000,1000)
double precision B(1000,1000)
   call omp_set_num_threads(4)
!$omp parallel default(shared)
!$omp& private(i,j)
   do k=1, 20
!$omp do
     do j = 2 ,999
       do i = 2 , 999
S1:        A(i,j) = ( B(i-1,j)+B(i+1,j)
               + B(i,j-1)+B(i,j+1)) * c
       end do
     end do
!$omp do
     do j=1, 1000
       do i= 1, 1000
S2:        B(i,j) = A(i,j)
       end do
     end do
   end do
!$omp end parallel
```

**Fig. 1.** Jacobi kernel in OpenMP

```
double precision Aloc(1000,0:251),Bloc(1000,0:251)
double precision buf_upper(1000,-1:5)
double precision buf_lower(1000,-1:5)
!$omp threadprivate(Aloc, Bloc)
!$omp parallel default(shared)
!$omp& private(shtart_y,end_y,i,j,k,id)
      id = omp_get_thread_num()
      do k = 1, 20
S1':    buf_upper(1:1000, id)=Bloc(1:1000,250)
S2':    buf_lower(1:1000, id) = Bloc(1:1000, 1)
!$omp barrier
S3':    Bloc(1:1000,0)=buf_upper(1:1000,id -1)
S4':    Bloc(1:1000,251)=buf_lower(1:1000,id+1)
        do j=start_y, end_y
          do i=2, 999
S5':          Aloc(i,j) = ((Bloc(i-1,j) + Bloc(i+1,j)
                      +  Bloc(i,j-1) + Bloc(i,j+1)) * c
          end do
        end do
        do j=1, 250
          do i=1, 1000
S6':            Bloc(i,j) = Aloc(i,j)
          end do
        end do
      end do
!$omp end parallel
```

**Fig. 2.** Jacobi kernel in OpenMP SPMD style

In our example translation (Figures 1 and 2), we have assumed that the program is executed by four threads for the sake of simplicity. The parallelization strategy assigns each thread the task of updating a contiguous block of columns of each of arrays *A* and *B*. We accordingly distribute a contiguous block of columns of *A* and *B* to each thread; the corresponding private arrays have been renamed *Aloc* and Bloc. Note that the user has not specified a loop schedule; with such a schedule, the distribution of loop iterations, and hence of array updates, would change. We are able to declare the threadprivate arrays statically in this case, since the number of threads at run time is fixed (Fig. 1). Dynamic arrays have to be used if the user does not explicitly specify how many threads will execute the code. It is obviously more flexible to compile the program for an unspecified number of threads. We must use the relationship between an original globally shared array and the privatized array to translate array references. For example, *Aloc*(1:1000,1:250) for thread 0, 1, 2, and 3 in Fig. 2 represents *A*(1:1000,1:250), *A*(1:1000:1,251:500), *A*(1:1000,501:750) and *A*(1:1000,751:1000) in the original OpenMP program in Fig. 1 respectively. The translation process actually introduces a common block, since it is needed to pass privatized arrays across the procedures in SPMD code. In the examples of this paper, we simplify our discussion by only declaring the privatized array in a threadprivate directive without introducing a new common block.

```
!$omp parallel private(start_y, end_y, id, numthreads)
      numthreads = omp_get_num_threads()
      if( id .eq.0) then
        start_y = 2;     end_y   = 250
      else if (id .eq. (numthreads-1)) then
        start_y = 1;     end_y   = 249
      else
        start_y = 1;      end_y   = 250
      end if
      do j = start_y,  end_y
            …
      end do
!$omp end parallel
```

**Fig. 3.** Translation of loop bounds (start_y, end_y) of the first j-loop in Fig. 1 into SPMD style

**Loop translation.** Parallel loops are the major source of parallelism in most programs and they drive our translation into SPMD style. The translation process must not only translate array references, but also deal with DO directives, the do loop control statement and the loop body. Some DO and END DO directives can be removed if a privatized array is inside a parallel region. In general, assignment statements are surrounded by guards (if-constructs) in SPMD code [18] so that they are only executed by the thread owning the privatized array element on the left hand side. In the case of statements within parallel loops, these loop bounds are modified so that the guards are redundant. We adapt the loop bounds so that each thread executes the portion assigned to it by the loop schedule. We may choose to retain global coordinates and use the thread id to parameterize it, or we may translate the array bounds and hence all references to the loop variable in the original loop (as in our code). In each case, the

loop control variable can be directly used as subscript for privatized arrays without modification. The original loop bounds of *j*, 2 to 999, are reduced to *start_y* and *end_y* in Fig. 2, whose calculation is demonstrated in Fig. 3.

**Sharing elements of privatized data between threads.** In OpenMP, threads interact via references to shared data. In the original OpenMP program, multiple threads may reference part of an array that has been privatized in our SPMD translation process. Since entire loop iterations are executed by a thread, it may occur that a thread needs to read data that have been privatized and are not local; it may also need to write non-local data as in an LBE program we previously studied [3]. For example, in our Jacobi code fragment, some elements of array *B* are accessed by more than one thread. Since *B* has been privatized in the SPMD version, some array elements private to a thread must be accessed (read, in this case) by another thread. We thus have the task of identifying such array elements and of modifying the code to ensure that the data are available: the owner thread must explicitly "export" this data. In our example, the privatized array *Bloc* on thread 1 will correspond to the original *B*(1:1000,250:500). This is equivalent to the region of the array that is written by it. However, thread 1 also needs to read the array section *B*(1:1000,249:501). When we compare this with the privatized region, we observe that two columns of the original array are used by thread 1 but are not immediately accessible to it: the non-local array elements *B*(1:1000,249) and *B*(1:1000,501) must be shared between threads as a result.

Wherever privatized data must be shared between threads, we require a translation from the original OpenMP program into SPMD style in the following steps:

(1) Shared buffers must be declared to store the non-local array references. The size of the shared buffers depends on the total number of threads, the number of array sections of non-local elements and the dimensions that are not privatized. For instance, in Fig. 2, *buf_lower* and *buf_upper* are declared for this purpose. Their declaration is double precision *buf_lower*(1000,4), *buf_upper*(1000,4). Because each of four threads accesses one column of *B* from neighbor threads, the total size of the buffer is one column multiplied by total number of threads. If the number of threads is not fixed, two dynamic arrays have to be declared to hold the data shared with neighbor threads at left and right boundaries.

(2) The size of the privatized array must be increased to create space to store the non-local data accessed. The adjusted size of the privatized array is obtained from the non-local access pattern. For example, *Aloc* and *Bloc* must be expanded by one column at both sides. The declarations of arrays *Aloc* and *Bloc* are thus modified to double precision *Aloc*(1:1000, 0:251), *Bloc*(1:1000,0:251). The additional memory spaces (1:1000, 0) and (1:1000, 251) are referred to as the shadow area [15], or ghost area. It is also common to append this storage directly to the local array in MPI SPMD programs, since it simplifies the translation process and is likely to provide better performance than using separate data structures.

(3) Executable instructions are generated to implement the sharing of private data between threads. Statements are inserted into the code to ensure that the owner thread writes data that are needed by another thread into shared buffers. In our example, *S1'* and *S2'* in Fig. 2 are inserted to copy two columns from array *Bloc* to *buf_upper* and *buf_lower*.

(4) Synchronization directives are inserted to ensure correct order of accesses to data in the shared buffers. In Fig. 2, data are written to, and should subsequently be retrieved from, shared buffers *buf_upper* and *buf_lower*; we need to insert a barrier to enforce this order of access. In general, we have several alternatives for implementing this, including barriers or a more loosely synchronous code.

(5) Executable instructions are generated to store the contents of shared buffers into the shadow area of the reading thread. Hence in Fig. 2, statements *S3'* and *S4'* are inserted. Since the local copies are part of a private data structure, we can benefit from its data locality in subsequent use. Note that these steps are only slightly modified if a thread writes non-local data.

### 2.3 General Strategy for SPMD Translation

In the previous section, we outlined the basic translation steps using our Jacobi example. Now we discuss a more general translation technique for SPMD style code, because we need to apply this strategy not only when the program is much more complex, but also in the presence of other OpenMP constructs.

**The transformation between global and local representations.** For a given parallel loop in an OpenMP program, we may need to translate references to both privatized arrays as well as arrays that remain shared. Basically, we follow the method of [18] that we will first remove the DO directive and reduce the loop bounds. The resulting array subscript is local to each thread. This method is described in the translation of Jacobi program. However, whenever the loop control variable appears outside a reference to a privatized array, we have to recover its original "global" value to ensure correct translation. This requires a local to global conversion (see) of subscripts. To show this, we choose to privatize array *A* only in the example of Fig. 4(a). The code is transformed into the SPMD style in Fig. 4(b) using the index conversion method; the global loop index *i* is transformed into local *iloc*, where *i* is equal to *iloc+id*chunk*. Thus we must replace the loop variable by the latter expression wherever it is not referred to in a privatized data structure.

```
double precision A(200)
double precision B(200)
call omp_set_num_threads(10)
!$omp parallel do shared(A,B)
    do i=1,200
      A(i) = i + B(i)
    end do
!$omp end parallel do
…
```

(a) OpenMP example code

```
double precision Aloc(20),B(200)
!$omp threadprivate(Aloc)
   call omp_set_num_threads(10)
!$omp parallel shared(B)
   id = omp_get_thread_num()
   chunk = 200/omp_get_num_threads()
   do iloc=1,chunk
     Aloc(iloc)=iloc + id*chunk
              +B(iloc+id*chunk)
    end do
!$omp end parallel
…
```

(b) SPMD style with loop bounds reduction

```
double precision Aloc(10), B(200)
!$omp threadprivate(Aloc)
     call omp_set_num_threads(10)
!$omp parallel shared(B)
     id = omp_get_thread_num()
   chunk=200/omp_get_num_threads()
!$omp do schedule(static, chunk)
    do i=1,200
       Aloc(i-id*chunk) = i + B(i)
     end do
!$omp end do
!$omp end parallel
…
```

(c) SPMD style with transformed subscripts for privatized array

**Fig. 4.** An OpenMP example code, and its SPMD style code with loop bounds reduction and reduced array size. Array *A* is privatized into *Aloc*, while *B* is not to be privatized.

In a few cases, the above method does not work. Hence an alternative translation is required. In this approach, we retain the original loop bounds and thus it is the subscript of the privatized array that is modified. The loop schedule ensures that threads execute the required iterations. Although the global *i* is untouched, we need the local index to access privatized arrays. The global to local conversion as described in Table 1 is thus required to deal with references to the privatized array *Aloc*; we substitute i-*id***chunk* for global *i*. Code generated using this method is displayed in Fig. 4(c). We can see the difference between these translations by comparing Fig. 4(b) and (c). This second strategy is used to translate the OpenMP ORDERED directive in Section 3.1 below since we must keep the DO directive.

Table 1 gives formulae used to adapt loop lower and upper bounds, and transform between local and global address spaces for the most common loop schedules. It can be used as a reference to translate parallel loops into SPMD style when the loop scheduling of OpenMP programs is the straightforward default chunk size and we follow this loop scheduling to privatize arrays.

**Table 1.** Creation of loop bounds, local and global indices[18]

| Transformation | Block | Cyclic |
|---|---|---|
| Global to local lower loop bounds | *MAX((id×chunk)+1, L) –id×chunk* | *lb=((L -1)/numthreads + 1*<br>*If ( id < MOD(L-1, numthreads ) ) lb = lb +1* |
| Global to local upper loop bounds | *MIN((id+1)×chunk,U) –id×chunk* | *ub = ((U-1)/numthreads ) + 1*<br>*If ( id > MOD(U-1, numthreads) ) ub = ub – 1* |
| Global to local | *i - id × chunk* | *(i – 1)/chunk + 1* |
| Local to global | *i+ id × chunk* | *( (i-1)×chunk ) + 1 + id* |
| Global index to owned thread | *CEIL(N/numthreads) – 1* | *MOD(i-1,numthreads)* |

*numthreads* = total number of threads; *id* = the thread number;
*chunk* = chunk size of local iterations; *L*, *U*: original lower, upper bounds;

*lb*, *ub*: transformed loop lower and upper bound

## 3 Translating OpenMP Constructs

In the previous section, we focused on the basic SPMD translation that permits us to convert an OpenMP program with some shared arrays into an equivalent SPMD style OpenMP program with threadprivate arrays. In this section, we discuss how efforts have to be exerted to translate other OpenMP directives and clauses that may appear in the code. We consider several DO directive clauses and other OpenMP constructs including synchronization directives.

### 3.1 Other Issues in Loop Translation

We have discussed the basic loop translation in Section 2. However, more work has to be done when some clauses, including REDUCTION and ORDERED of parallel do loops, are encountered.

```
    S= 0.0
!$omp parallel do default(shared)
!$omp& reduction(+:S)
    do i = 1, N
      S = S + A(i) * B(i)
     end do
!$omp end parallel do
```

(a) An OpenMP code with REDUCTION

```
!$omp threadprivate(Aloc, Bloc)
      S = 0.0
!$omp parallel reduction(+:S) default(shared)
      Sloc = 0.0
      do iloc=1,ub
        Sloc=Sloc + Aloc(i)*Bloc(i)
      end do
      S = S+ Sloc
!$omp end parallel
```

(b) The SMPD code

**Fig. 5.** Translation of reduction operations in OpenMP

**REDUCTION Clause**. If the arrays to be privatized are encountered in a parallel do loop with a REDUCTION clause, a new private variable has to be introduced to save the local result of the current thread; we must perform a global reduction operation among all the threads to combine the local results. The REDUCTION clause will be moved so that it is associated with the parallel region although it may be associated with a DO directive in original OpenMP program. The SPMD transformations for the do loops and privatized array are still needed. Fig. 5(a) and (b) depict the translation of a REDUCTION clause. Sloc, which is a private scalar, is introduced to accumulate the local sum for each thread taking advantage of privatized arrays *Aloc* and *Bloc*, and *S* is the result of a global sum of the local *Sloc* among all the threads. Variable *ub* has to be calculated according to Table 1.

**ORDERED in a parallel loop.** The ORDERED directive is special in OpenMP as neither CRITICAL nor ATOMIC can ensure the sequential execution order of loop iterations. So we have to retain not only the ORDERED directive, but also the corre-

sponding PARALLEL and DO directives in the generated code due to the semantics of ORDERED. The translation of the parallel do loop follows the second method introduced in section 2.3, where the DO directive and loop control statement remain the same, while the index of the privatized array is converted from global to local format. A schedule that permits threads to access private array elements as far as possible is declared explicitly. The example in Fig. 6 shows an OpenMP program with the ORDERED directive and corresponding SPMD code. The generated loop in Fig. 6(b) has to sweep from 1 to *N* to ensure the correct distribution of iterations to threads. We use an IF construct inside the do loop to isolate the wasteful first and last iteration.

```
    integer N
    parameter(N=1000)
    double precision A(N)
    call omp_set_num_threads(10)
!$omp parallel do ordered
   do i=2, N-1
      …
!$omp ordered
     write (4,*) A(i)
!$omp end ordered
      …
   end do
!$omp end parallel do
```

(a) An OpenMP code with ORDERED

```
   double precision Aloc(100)
!$omp threadprivate(Aloc)
!$omp parallel
   id = omp_get_thread_num()
   chunk = N/omp_get_num_threads()
!$omp do ordered schedule ( static, chunk )
   do i=1, N
     if ( i .ge. 2 .and. i .le. N-1 ) then
           …
!$omp ordered
      write (4,*) Aloc(i-id*chunk)
!$omp end ordered
       …
      end if
     end do
!$omp end do
!$omp end parallel
```

(b) The SPMD code

Fig. 6 Translation of ORDERED directive into SPMD style

**Different loop scheduling and different number of threads.** In an OpenMP program, we may encounter parallel loops whose execution requires that each thread access arrays in a pattern that is quite different from the array elements that they will need to access during the execution of other parallel loops. This is generally detrimental to performance, since it will not permit the reuse of data in cache. However, it may be a suitable way to exploit parallelism inherent in the code. The ADI (Alternating Direction Implicit) kernel provides a common example of this problem. Since this could also introduce substantial inefficiencies into an SPMD program, we have considered two solutions to this problem. In one of these, we create two private arrays, each of which is privatized in first and second dimension separately. When the loop scheduling is changed between these two dimensions, we have to transfer the content of the first privatized array to the second through a shared buffer. This is similar to performing data redistribution. In the other solution, a private array is used for one kind of loop scheduling; whenever the loop scheduling changes, the contents of the private array are transferred to a shared buffer, which is subsequently referenced. Both methods require a good deal of data motion. From previous experiments [9], we know that such sharing is very expensive unless there is a great deal of computation to amortize the overheads or a large number of processors are involved. In practice we may ask the user to decide how to privatize.

In our outline of the basic translation strategy, we discussed the case of static scheduling and a fixed number of threads. In our SPMD translation, we disable the dynamic, guided scheduling and dynamic number of threads including NUM_THREADS clause in order not to degrade the performance. It is hard to determine which iterations are executed by a thread under these scheduling methods and when there is a dynamic number of threads. The size of privatized arrays may change and a large number of privatized array elements may need to be shared between threads. All of these will prevent us from generating efficient code.

### 3.2 Translation of Other OpenMP Constructs

Parallel regions must replace serial regions to enable access to privatized arrays within them, since the thread number and threadprivate arrays that are essential for SPMD style code can only be obtained in a parallel region. After constructing the parallel region, we need to decide two things: which thread executes each statement of the parallel region, and whether we need to share the privatized data between threads.

During the translation, only assignment statements may be modified, while control statements are kept without modifications in the generated code. If a privatized array appears on the left hand side of an assignment statement, then the owner thread of the privatized array element executes this statement. If the private array is accessed on both sides of a statement, the owner thread of the privatized array element on the right hand side will execute this statement; data sharing statements are inserted if non-local array elements are referenced.

```
   double precision A(100), B(200)
   call omp_set_num_threads(4)
!$omp parallel shared(A,B)
…
!$omp master
      B(120) = A(50)
!$omp end master
…
```

(a) An OpenMP code with MASTER

```
   double precision A(100), Bloc(50)
!$omp threadprivate (Bloc)
    call omp_set_num_threads(4)
!$omp parallel shared(A)
…
      if ( id .eq. 2 ) then
         Bloc(20) = A(50)
      end if
…
```

(b) The SPMD code

**Fig. 7.** Translation of MASTER directive into SPMD style

**MASTER and SINGLE directives.** The semantics of the MASTER directive and sequential regions is that the master thread performs the enclosed computation. We can follow the OpenMP semantics in such program regions by modifying the code so that privatized array elements are shared between the master thread and the owner thread, or we can let the owner thread of a privatized array element compute instead. Since both of these methods ensure correctness, we select the latter in our implementation for performance reasons, although it does not entirely follow the semantics of the original OpenMP program. For example, we elect to privatize *B* but not *A* in Fig. 7(a). The translated code for the MASTER directive is shown in Fig. 7(b). We can see that

*B*(120) is mapped to *Bloc*(20) for thread 2, so thread 2 executes the code enclosed within the MASTER directive. More conditionals may have to be introduced for each statement of the code enclosed by the MASTER directive; these IF constructs may be merged during the subsequent optimization process. The code enclosed within a SINGLE directive could be translated in a similar way to that of the MASTER directive. The only difference between translating SINGLE and MASTER directives is that we have to replace END SINGLE with a BARRIER if NOWAIT is not specified.

**Synchronization directives**. If a private array appears in the code enclosed by CRITICAL, ATOMIC and FLUSH directives, the enclosed code is translated according to the strategy outlined in Section 2. Since the owner of a privatized array element on the left hand side usually executes a statement, accesses to privatized arrays are serialized. This may allow us to move the statements associated with private arrays outside the synchronization directives, and improve the performance to some extent. Unfortunately, sometimes the control flow and data flow of the program will not allow us to carry out this improvement.

## 4 Current Implementation

The performance of the generated SPMD style OpenMP code will be greatly enhanced by precise analyses and advanced optimizations [16]. Many compiler analyses are required for the SPMD translation, for instance, dependence analysis in a do loop, array section analysis within and between the loops and between the threads, parallel data flow analysis for OpenMP programs, the array access pattern analysis, affinity analysis between the arrays [7], etc. Interprocedural analysis is important to determine the array regions read and written in loop iterations. We need to calculate accesses to array elements accurately for each thread in order to determine array privatization and shared data elements, even in the presence of function calls within a parallel loop. Interprocedural constant propagation can increase our precision. For example, if we do not know whether a reference ( for example, $A(i,j+k)$ ), is local, we have to make a conservative assumption that they are not, and share it between the owner thread and accessing thread, although this is not efficient. $A(i,j+k)$ may be known to be local to the current thread at compile time if the value of variable $k$ can be fixed by interprocedural constant propagation.

Our translation from OpenMP into SPMD style OpenMP is being realized within our Dragon tool based on the Open64 compiler [14], a suite of optimizing compilers for Intel Itanium systems running on Linux that is a continuation of the SGI Pro64 compiler. Our choice of this system was motivated by a desire to create a robust, deployable tool. It is a well-structured code that contains state-of-the-art analyses including interprocedural analysis [8], array region analysis, pointer and equivalence analyses, advanced data dependence analysis based on the Omega test, and a variety of traditional data flow analyses and optimization. It also has a sophisticated loop nest optimization package. A version of the Dragon tool with limited functionality has been made widely available [5]. This extended version is aimed at helping the user

generate SPMD code interactively, by providing them with on-demand analysis and assisting in the code generation/optimization process. The high-level forms of WHIRL, the intermediate representation (IR) in the Open64 compiler, are able to explicitly represent OpenMP constructs. We transform a standard OpenMP program into an equivalent SPMD style one at the high level, i.e. before the IR constructs corresponding to OpenMP directives and clauses are lowered. At that point, we may either unparse the resulting WHIRL into source OpenMP code, or we may continue by lowering the IR for the SPMD style code into low-level WHIRL and then object code for the IA-64. We will discuss our implementation, including the analyses and optimizations required for SPMD translation, in more detail in future publications.

## 5 Related Work

The idea of data privatization can be dated to the first version of OpenMP language. However, it is still challenging to write an OpenMP program with scalable performance for ccNUMA architectures due to the data locality problem. In fact, data locality issues and the strongly synchronous nature of most OpenMP programs pose problems even on small SMP systems architectures. Most related work to date has addressed the problem of obtaining performance under OpenMP on ccNUMA platforms.

Researchers have presented several strategies to solve this problem automatically, taking advantage of the first-touch policy and page migration policy provided either by some operating systems such as those on SGI [17] and Compaq systems [1], or by the user-level run-time libraries [12]. Hence program changes are minor. Besides, the SCHEDULE clause in OpenMP may be extended to schedule the computation in a cache friendly way for regular and irregular applications [13]. These means attempt to minimize the intervention from users. But the operating system or run-time libraries are not able to know precisely when to migrate the page. Besides, the users are not able to fully control the behavior of operating systems and tune the performance.

Data distribution directives give users facilities for determining how to allocate data and arrange the computation locally. Decomposition of data in terms of page granularity and element granularity is supported by SGI and Compaq. Data and thread affinity are enabled by SGI to ensure the mapping of data with the corresponding computation; similar functionality is provided by Compaq's NUMA directive. There are some differences; SGI maps data to processors, whereas Compaq maps data to node memories. Extending OpenMP with data distribution directives may improve the data locality and decrease the remote memory access, but important features of OpenMP - incremental parallelism, and portability and ease for programmer – will suffer as a result. Our method is different from the data distribution method since we attempt to achieve the same result, data locality, without introducing more directives. The work distribution already implies a strategy for data distribution. Our goal is to permit the user to retain full control of the translation process by making them aware of the performance implications of their parallelization strategy and helping them change it as desired.

Some translation techniques used in this paper are similar to those developed for HPF compilation [18, 21], as both are targeted to generating SPMD code. For instance, HPF compilers also perform loop bounds reduction, global to local and local to global conversion. There are several major differences between this approach and HPF compilation, however. First, our SPMD code is not based on message passing, but on the use of shared data to exchange values. A pair of assignment statements with synchronization between them is enough to affect this exchange. Also the further optimizations of data sharing in OpenMP and optimizations of communication in MPI are different. Second, in HPF compilation, all the variables in the generated code must be local or private, whereas some variables could possibly remain shared in the translated SPMD style OpenMP code. This means that the user or tool may have several alternative strategies available when adapting a program in this way.

## 6 Conclusion and Future Work

In this paper, we have described a basic strategy for translating OpenMP code into SPMD style with array privatization for ccNUMA systems. Privatization is achieved via the threadprivate feature of OpenMP, and may introduce the need to account for sharing of privatized array elements between the threads. The translation of several OpenMP constructs, directives, clauses and their enclosed code into SPMD style is presented.

Our method of array privatization is based on mapping the data to threads. A disadvantage of this method is that we cannot ensure that logical neighbor threads are physically near to each other. On the other hand, if we want to share the private data among (say) two threads, we require global synchronization between the copies to and from a shared buffer and will have to deal with false sharing between them. Without local synchronization, the generated code will be much less efficient. Unstructured or irregular problems are not covered in this paper.

Our translated codes are primarily targeted to ccNUMA platforms. On the other hand, we also plan to integrate SPMD OpenMP style code with Global Arrays [11], a portable efficient shared-memory programming model for distributed memory systems, so that we can implement OpenMP for distributed memory computers.

## References


1. 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)
2. 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
3. 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)

4. Chapman, B., Weng, T.-H., Hernandez, O., Liu, Z., Huang, L., Wen, Y., Adhianto, L.: Cougar: An Interactive Tool for Cluster Computing. 6th World Multiconference on Systemics, Cybernetics and Informatics. Orlando, Florida, July 14-18, (2002)
5. The Dragon analysis tool. http://www.cs.uh.edu/~dragon
6. Eggers, S. J. , Emer, J. S., Lo, J. L., Stamm, R. L. and Tullsen, D. M.: Simultaneous Multithreading: A Platform for Next-Generation Processors. IEEE Micro, Vol. 17, No. 5, (1997) 12-19
7. Frumkin, M., Yan, J.: Automatic Data Distribution for CFD Applications on Structured Grids. The 3rd Annual HPF User Group Meeting, Redondo Beach, CA, August 1-2, 1999. Full version: NAS Technical report NAS-99-012, (1999)
8. Hall, M. W., Hiranandani, S., Kennedy, K., and Tseng, C.-W.: Interprocedural Compilation of FORTRAN D for MIMD Distributed-Memory Machines. Proceedings of Supercomputing 92', Nov. (1992) 522-534.
9. Marowka, A., Liu, Z., Chapman, B.: OpenMP-Oriented Applications for Distributed Shared Memory. In the Fourteenth IASTED International Conference on Parallel and Distributed Computing and Systems. November 4-6, 2002, Cambridge, (2002)
10. Muller, M.: OpenMP Optimization Techniques: Comparison of FORTRAN and C Compilers. Third European Workshop on OpenMP (EWOMP 2001), (2001)
11. Nieplocha, J., Harrison, R. J., and Littlefield, R. J.: Global Arrays: A portable `shared-memory' programming model for distributed memory computers. Proceedings of Supercomputing, (1994) 340-349
12. Nikolopoulos, D. S., Papatheodorou, T. S., Polychronopoulos, C. D., Labarta, J. and Ayguade, E.: Is data distribution necessary in OpenMP. Proceedings of Supercomputing, Dallas, TX, (2000)
13. Nicolopoulos, D. S., Ayguadé, E.: Scaling Irregular Parallel Codes with Minimal Programming Effort. Proceedings of Supercomputing 2001 (SC'01), the International Conference for High Performance Computing and Communications, Denver, Colorado, November 10-16, (2001)
14. The Open64 compiler. http://open64.sourceforge.net/
15. Sato, M., Harada, H., Hasegawa A. and Ishikawa Y.: Cluster-Enabled OpenMP: An OpenMP Compiler for SCASH Software Distributed Share Memory System. Scientific Programming Vol. 9, No. 2-3, Special Issue: OpenMP, (2001): 123-130
16. Satoh, S., Ksano K. and Sato M.: Compiler Optimization Techniques for OpenMP Programs. Scientific Programming Vol. 9, No. 2-3, Special Issue: OpenMP, (2001) 131-142
17. 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
18. Tseng, C.-W.: An Optimizing FORTRAN D Compiler for MIMD Distributed-Memory Machines. PhD thesis, Dept. of Computer Science, Rice University, January (1993)
19. Tu P. and Padua, D.: Automatic Array Privatization. Proc. Sixth Workshop on Languages and Compilers for Parallel Computing, Portland, OR. Lecture Notes in Computer Science., Vol. 768, August 12-14, (1993) 500–521
20. Wallcraft, A. J.: SPMD OpenMP vs. MPI for Ocean Models. Proceedings of First European Workshops on OpenMP (EWOMP'99), Lund, Sweden, (1999)
21. Zima, H. and Chapman, B.: Compiling for Distributed Memory Systems, Proceedings of the IEEE, Special Section on Languages and Compilers for Parallel Machines, Vol. 81, No. 2, Feb. (1993) 264-287