

Toward Optimization of OpenMP Codes for Synchronization and Data Reuse

Tien-hsiung Weng and Barbara Chapman
Computer Science Department
University of Houston
Houston, Texas 77024-3010
thweng, chapman@cs.uh.edu

Abstract

In this paper, we present the compiler transformation of OpenMP code to an ordered collection of tasks, and the compile-time as well as runtime mapping of the resulting task graph to threads for data reuse. The ordering of tasks is relaxed where possible so that the code may be executed in a more loosely synchronous fashion. Our current implementation uses a runtime system that permits tasks to begin execution as soon as their predecessors have completed. A comparison of the performance of two example programs in their original OpenMP form and in the code form resulting from our translation is encouraging.

1 INTRODUCTION

OpenMP [1] succeeds as a popular parallel programming interface for medium scale high performance applications on shared memory systems due to its portability, ease-of-use and support for incremental parallelization. Writing OpenMP programs is relatively easy, in contrast to other parallel programming models such as MPI. Users may achieve reasonable performance by adding just a few OpenMP directives. However, it requires a non-trivial effort to obtain scalable OpenMP codes for ccNUMA systems. In order to make OpenMP a parallel programming model for a system consisting of a large number of processors, synchronization in the code must be minimized. An additional potential cause of performance degradation on ccNUMA (cache coherent Non-Uniform Memory Access) systems is that a thread may need to access a remote memory location; despite substantial progress in interconnection systems, this still incurs significantly higher latency than accesses to local memory, and the cost may be exacerbated by network contention. If data and threads are unfavorably mapped, cache lines may ping-pong between locations. However, OpenMP provides few features for managing data locality. In particular, it does not enable the explicit binding of parallel loop iterations to a processor executing a given thread; such a binding might allow the distribution of work such that threads can reuse data.

An alternative approach that can provide high performance on ccNUMA platforms is for the user to rewrite the OpenMP code, decomposing the data structures into portions that will be private to each thread, creating buffers to hold shared data and arranging for the appropriate synchronization. Yet this so-called Single Program Multiple Data (SPMD) OpenMP style requires considerable reprogramming. We attempt to relieve users from the extra burden of creating SPMD code and to help them achieve good performance for their original programs by transforming standard OpenMP codes to tasks, relaxing synchronization between tasks where possible, and mapping them to processors for data reuse.

2 OUR APPROACH

We target OpenMP to a data flow execution model realized using the SMARTS runtime system [7][8] developed at Los Alamos National Laboratory. At runtime, SMARTS employs a master thread to orchestrate the work of slave threads that execute the tasks. The tasks, a graph that represents the ordering of their execution and the data read and written by each task are passed to the runtime system. Before a task can run, it must make read/write requests for the data it needs. As soon as the data is available, the master thread schedules the task to a slave queue to be executed. The slave thread may be chosen by the compiler or by the runtime system. SMARTS also supports work stealing, where a slave thread takes one or more tasks from other queues.

The compiler transforms an OpenMP code to a collection of sequential tasks and a task graph, which represents the dependences between the tasks. This translation is based heavily on array region analysis, which must be applied interprocedurally. One of our goals is to perform as much work as possible to avoid expensive runtime computation of dependences; where necessary, code will be generated to complete the task graph and finalize the mapping to processors at run time. The performance of the resulting code will be determined by the suitability of the code for this approach, the quality of the translation, and the amount of data reuse by each processor to compensate the overheads incurred by the SMARTS runtime system. These overheads include managing requests for read and write data, as well as scheduling of tasks into task queues by SMARTS.

The compiler may specify the mapping of tasks to processors (or equivalently, to threads that are bound to processors) or may leave this to the runtime system. Finding a good compile-time mapping of nodes of the task graph to the processors for data locality reuse is the main focus of this paper. We use this mapping when we generate parallel code that includes calls to SMARTS. The resulting code is executed using the so-called *first touch policy*, where data will be stored so that it is local to the task that first accesses it. All tasks that use this data will be assigned to the same thread wherever possible to maximize data locality. We describe and discuss the mapping algorithm that we have developed below.

This paper is organized as follows. In the next section, we first discuss the generation of tasks and the task graph. We then present the algorithm for scheduling the tasks in the graph to threads for data locality reuse. Our strategy for code generation is also discussed. Then, in Section 4, we illustrate our work using an LU kernel. We also use an ADI code example to illustrate how our mapping algorithm enables macro-pipelined computations. Then, experimental results are presented. Finally, Section 5 presents related work and Section 6 our conclusions and future work.

3 CREATION AND MAPPING OF THE TASK GRAPH

For the purposes of this paper, we call a unit of *sequential* code that will be executed by a single thread a task. Such a task is an arbitrary code sequence, such as a set of iterations of a loop nest, or one or more procedures, without synchronization constructs. An OpenMP program will be decomposed into an ordered collection of tasks according to the semantics of the language. The ordering will be represented by the task graph. A task graph for an OpenMP program, denoted by $G(N,E)$ consists of a set of nodes $N=\{t_1, t_2, \dots, t_m\}$, where each node represents a task in the decomposed program, and a set of edges E between nodes, where $e_{i,j}$ is an edge from node t_i to node t_j . Each edge represents synchronization in the sense that the source node must be executed before the sink node at run time. As part of our translation strategy, we will eliminate as many edges as possible from this graph in order to reduce the amount of synchronization imposed on the executing code. We will also use this graph and our analysis to map the tasks in the program to threads. Note that we bind exactly one thread to each target processor so that we may refer to

threads and processors interchangeably. We assume that processors (and threads) are numbered consecutively, beginning with 1.

3.1 Generation of tasks and task graph

OpenMP does not restrict work-sharing constructs to be within the lexical scope of the parallel region; they can occur within a subroutine or a procedure that is invoked, either directly or indirectly from inside a parallel region. Parallel regions may contain long call chains. Thus the work required to create tasks is of necessity interprocedural. We have developed an efficient call graph construction algorithm [34] to support recursive calls and dynamically bound calls arising from invocations of procedure-valued formal parameters. It provides a precise traversal of the generated call graph, since additional traversal information is saved. It has been implemented in our Dragon tool [36] that is based on Open64 [35]. The traversal algorithm is used to create task identification and gather array section information in an exact call chain of the program call graph. For each task, we collect the following information: 1) read/write access to array sections; 2) the code segment; 3) information on (possible) loop iterations; 4) synchronization constraints.

We follow the semantics of OpenMP worksharing constructs (DO, SECTIONS, MASTER, and SINGLE, etc.) to generate the sequential tasks. For example, the iterations of a parallel loop with a default schedule are assigned to threads so that each has a contiguous set of iterations of approximately the same size. If the loop does not contain any further worksharing or synchronization constructs, each such set of iterations would be a task in our program representation. A SINGLE or MASTER construct each represents a task, while a CRITICAL SECTION will be translated to one task per thread, each with the same code segment. Sequential regions of a code may be divided into basic blocks, each of which may form a task. The complete sequence of tasks of a program is generated interprocedurally in a traversal of our call graph. A typical OpenMP program will contain barrier synchronizations, and possibly atomic updates or critical sections to ensure ordered access to shared data. These synchronizations will be represented explicitly in the task graph that is also constructed. However, many of the resulting edges are in fact superfluous in this representation. For instance, an OpenMP code may require a barrier between two parallel loop nests, although the data reuse between them only occurs between two of the resulting tasks. All other tasks can be executed asynchronously.

In order to determine which edges can be eliminated from the graph, the compiler performs an array section analysis to determine as accurately as possible each task's write and read accesses to shared data objects in the program. If we can use this analysis to prove that a pair of threads that are connected by an edge do not access the same data (or such accesses are reads only), then we may remove the corresponding edge from the task graph.

We have chosen to use standard regular sections, or the triplet notation, to represent data read and written, because of the efficiency with which we can compare them ([15], [9]); however more accurate analyses have been proposed, e.g., simple sections [10], regions [11][12][13], and the Omega test [17] and these are potential alternatives for compile time analysis when precision is critical. Efficiency and practicality are key for runtime synchronization analysis and regular sections are the natural choice for our runtime strategy where we must avoid overheads.

In order to improve the accuracy of our analysis, we use a delayed union approach when determining the region of an array that is accessed by a thread. Under this approach, two referenced sections of an array are merged if the merge does not lead to a loss of accuracy. Otherwise, each individual section is kept in a list. If a new reference cannot be merged with an existing regular section, it is added to the list until a threshold number of entries is achieved, in which case union will be enforced. This method may not improve accuracy in many cases.

The need for synchronization is given by comparing the data written by a thread with that which is read and written by a thread to which it is linked via an edge in the task graph. Where

compile-time analysis can prove that there is no common data, the edge is deleted from the graph. Where results are imprecise, runtime tests may be inserted to determine the need for an edge during program execution. It is worthwhile to do so when the test is fast, such as checking the range of a specific constant value.

When the number of threads is known in advance, task generation is straightforward. Otherwise, more work is needed at runtime. In the former case, each task is created, ordering relationships are determined based upon OpenMP semantics, and the data read and written by each task is computed. Task sequences and its read/write accesses to regular sections of arrays are also obtained. For example, we present the algorithm for translating the DO construct in Figure 1.

```

Let  $NT$  be the number of threads and
       $t_k$  be the last task before this loop
DO  $I = 1, NT$ 
       $k = k + I$ 
      Compute bounds of  $t_k$  according to loop schedule
      Compute array accesses by tasks  $t_k$ 
      Add  $t_k$  to  $N$ 
ENDDO

```

Figure 1 The generation of tasks from an OpenMP parallel loop

```

!$OMP PARALLEL
...
!$OMP DO
DO I= 1,100
  DO J=100
    A(J,I) =
  ENDDO
ENDDO
!$OMP SINGLE
  A(1,1) =
!$OMP END SINGLE
!$OMP END PARALLEL

```

Figure 2 Example of DO

To illustrate the above algorithm, we use the code in Figure 2 and assume that NT is 2. Tasks will be created by subdividing the work in the outer loop according to the default schedule. If eight tasks have already been created when this loop is encountered, then it will generate two new tasks, t_9 and t_{10} , to perform the work in the loop. The first of these tasks will perform the first 50 iterations and the second will perform the remaining 50. Accordingly, task t_9 has bounds $I=1,50$ and $J=1,100$ and will write the array section $A[1:100, 1:50]$. Task t_{10} has bounds $I=51,100$ and $J=1,100$ and writes to $A[1:100, 51:100]$. These tasks must both wait until all tasks involved in the previous construct have been completed, unless the NOWAIT clause was specified there.

Each of the SINGLE, MASTER and SECTION constructs corresponds to one task. So for the code in Figure 2, task t_{11} will be created to execute the code within the SINGLE construct that has write access to array section $A[1:1,1:1]$. Both t_9 and t_{10} will be predecessors of t_{11} . By applying

```

For each  $e_{ij}$  in  $E$  do
  If  $(t_i.RWaccess \cap t_j.RWaccess) = \emptyset$  then
    Remove  $e_{i,j}$  from  $E$ 
    For each  $t_k \in pr(t_i)$  do
      If  $(t_k.RWaccess \cap t_j.RWaccess) \neq \emptyset$  then
        Add  $e_{k,j}$  to  $E$ 
        Compute  $e_{k,j}.reuseLevel$ 
      Endif
    Endfor
  Else
    Compute  $e_{i,j}.reuseLevel$ 
  Endif
Endfor

```

Figure 3 Algorithm for eliminating edges from task graph

the algorithm of Figure 3, we can remove edge $e_{10,11}$.

Let there be an edge from t_i to t_j in the task graph and let R_i be the data read by task t_i and W_i be the data it writes. As already described, we must compare the data written in each with the data read and written in order to determine whether we can remove the edge. If we can do so, we must compare the data accessed by task t_j with that accessed by all predecessors of task t_i in the graph. For the purpose of scheduling, we are also interested in situations where two tasks read the same data. This information will also be recorded in the form of a special edge.

If there is an edge between node t_i and t_j in the modified task graph, the compiler must further determine the level of data reuse between the two tasks; the result will be appended to the edge and denoted by $e_{i,j}.reuseLevel$. The reuse level is determined by the amount of data accessed by both source and sink tasks, and the type of dependence between them. For example, if both tasks write the data, the reuse level will be higher than if both read the same data. The execution time of each task t_i in the task graph, $w(t_i)$ is statically estimated by the number and type of operation and operand, available information on loop bounds, etc. An alternative would be to provide a profile option. We may also be able to provide better estimates at run time, when symbolic bounds etc. can be evaluated.

Note that tasks may be subdivided to improve their cache behavior as well as to isolate iterations that give rise to synchronization, and that reordering of computation within a task may support the former but not the latter. Hence, techniques to determine appropriate granularity and to move constructs causing synchronization are required. This topic is not discussed in this paper.

3.2 Mapping algorithm for data locality reuse

We use a heuristic algorithm to map tasks in the task graph to processors (Figure 4). It is based upon a list scheduling approach [33]. The goal of the algorithm is to map tasks that use the same data to the same processor. However, at each step of the way, it must decide which one is the next task to map: it maps “more important” tasks first when possible, where this is determined by the weight of a node.

The algorithm consists of an initialization and an iterative step. In the initialization step, the algorithm first computes the level of each task t_i by determining the sum of the weights on paths from t_i to the terminal node, taking both edge and task weights into account. It sets $level(t_i)$ to the greatest weight on such a path. It also computes $\#ipr(t_j)$, the number of immediate predecessors of t_j . The function $Procs(t_i) = p_k$ records mapping decisions, i.e., task t_i is mapped to processor p_k ; initially, it is set to *NULL*.

At any stage during the process, the ready queue will hold those tasks that are ready for mapping but not yet assigned to processors. The algorithm initially puts all tasks without any predecessors in the task graph (“initial tasks”) into the ready queue (*RQ*) and orders them based upon their level; then it selects the highest priority (highest level) task $t_{current}$ from *RQ* as the next ready task, assigns it to processor 1, and removes $t_{current}$ from *RQ*.

The iterative step successively selects tasks and allocates them. In step 2.1, the algorithm will select the most important of the immediate successors of $t_{current}$ by examining those for which data reuse between it and $t_{current}$ is highest. If this task is subsequently chosen for mapping, it will be allocated to the same processor as $t_{current}$. It uses the two functions *list_max_isucs()* and *max_ipred()* to do so. The function *list_max_isuccs()* will return a list of those immediate successors of t_{temp} for which the highest reuse level is to be found on the edge from $t_{current}$. In other words, suppose t_s is one of $t_{current}$'s immediate successors with two incoming edges from $t_{current}$ and some other task t_k . If $e_{k,current}.reuseLevel > e_{s,current}.reuseLevel$, then t_s will not be included in the list returned by this function. The function *Max_ipred()* selects the task from those returned by *list_max_isuccs()* that corresponds to the edge from $t_{current}$ with the highest reuse level. This edge,

t_{temp} , will be temporarily mapped to the processor that executes $t_{current}$, and its incoming edge from $t_{current}$'s is marked as visited. The visited edges only affects $list_max_isucs()$ and $max_ipred()$.

We next compare the priority (weight) of the successor task chosen with the highest priority task in the ready queue. Whichever one has highest priority will be mapped next; we call it t_{next} . Before continuing, we now add any tasks to RQ if all their successors have already either been assigned to processors or placed in RQ . In step 4, we remove the next task from RQ , if necessary, and fix its mapping. If it is an immediate successor of $t_{current}$, then it will be mapped to the same processor. Otherwise, we call function *Locate_Processor* to check if it has been temporarily mapped to a processor; if it has, this function will map it to that processor. Otherwise, the function will find a processor that is idle for mapping. Finally, we set t_{next} to be the new $t_{current}$ before beginning the next iteration.

Input: Task Graph $G(N,E)$ and m processors of distributed shared memory system.

Output: $procs(t_i) = p_k$ for each task t_i , the assignment of tasks to processors

Begin

1. **Initialization step:**
 - 1.1 For each task t_i in G do
 - 1.1.1 Compute $level(t_i)$ and $\#ipr(t_i)$
 - 1.1.2 Initialize $Procs(t_i)$ to *NULL*
 - 1.2 All initial tasks are added to **RQ** (Ready Queue) ordered by level number
 - 1.3 Get highest priority task in **RQ** and set it to $t_{current}$. Remove it from **RQ** .
 $Procs(t_{current}) = I$

Repeat

2. **Select next task for mapping:**
 - 2.1 **Select best immediate successor of $t_{current}$:**
If $t_{current}$ has immediate successor **then**
 $t_{temp} = max_ipred(list_max_isuccs(t_{current}))$
temporary map t_{temp} to $procs(t_{current})$
mark_visited($inedge(t_{current})$)
else $t_{temp} = NULL$
Endif
 - 2.2 **Compare priority with tasks in RQ and select t_{next} :**
 $t_{next} = maxweight(t_{temp}, RQ)$ /* task with max priority of t_{temp} and RQ */
3. **Process successors of $t_{current}$:**
 - 3.1 **For each** $t_i \in isuccs(t_{current})$ **Do**
 $\#ipr(t_i) = \#ipr(t_i) - 1$
If ($\#ipr(t_i) = 0$) **then**
 Add t_i to **RQ** in priority order
Endif

Endfor

4. **Map next task:**
 - 4.1 **If** t_{next} is in **RQ** **then**
 Remove t_{next} from **RQ**
If t_{next} is immediate successor of $t_{current}$ **then**
 $Procs(t_{next}) = Procs(t_{current})$
else
Locate_processor(t_{next} , P_L); $Procs(t_{next}) = Procs(P_L)$
Endif
If not *visited*($inedge(t_{next})$) **then** *Mark_visited*($inedge(t_{next})$)
Endif

5. **Update current task:**

$t_{current} = t_{next}$

Until all tasks in G are assigned

End

Figure 4 Data reuse mapping algorithm

3.3 Code Generation

3.3.1 Static case

In the static case, the compiler collects as much information as possible to avoid runtime overheads and to generate code that passes complete information to the runtime system. Such information includes task identification, possibly loop iterations, task mapping information, read/write access information (pass task graph information if other runtime system is used). This is necessary so that the task can be executed on the processors for data reuse and can be out-of-order execution for those independent tasks.

After compile-time task mapping has been performed, we can generate source code such as that in Figure 5 as an example from Figure 2. The main program is first executed by the master thread, which creates additional threads that each bounds to processor and pass information stated above. In Figure 6, *task_loop0* and *task_single* are routines for tasks correspond to DO construct and SINGLE construct of Figure 2, respectively.

```
call SM_concurrency(n) /*to create n-1 slave threads to execute tasks*/
call SM_define_array(info) /* SMARTS Initialization */
schedtype = hardaffinity
call SM_startgen()
do i = 1, iter
  k = k + 1
  call SM_def_readwrite(task_loop0, k)
  call SM_getinfo(task_loop0, affin, datinfo)
  call SM_handoff(task_loop0, schedtype, procs(task_loop0,k), datinfo)
end do
k = k + 1
call SM_def_readwrite(task_single,k)
call SM_handoff(task_single, schedtype, procs(task_single,k), datinfo)
call SM_run()
call SM_end()
```

Figure 5 Main program using SMARTS

```
subroutine task_loop0(datinfo)
  use shared_DATA
  call SM_compute_bound(datinfo,lb1,ub1,lb2,ub2)
  do j = lb1, ub1
    do i = lb2, ub2
      A(i,j) =
    enddo
  enddo
end subroutine task_loop0

subroutine task_single(datinfo)
  use shared_DATA
  A(1,1) =
end subroutine task_single
```

Figure 6 Parallel loops using SMARTS

3.3.2 Dynamic case

Most application programs have a significant number of input-data dependent unknowns, and the number of processors may also be unknown at compile time. Compile-time analyses can

consequently produce symbolic expressions, despite the application of interprocedural constant propagation and other techniques for maximizing available information about variables. Therefore, symbolic analysis is required to support compile time analyses as well as to reduce the cost of runtime analysis. Several techniques have been proposed for evaluating symbolic expressions and for expressing information about program variables at arbitrary program points, as well as for aggressive simplification of a system of constraints of symbolic expressions, determining the relationships between symbolic expressions, and computing their lower and upper bounds.

Blume and Eigenmann [27] use abstract interpretation to extract information about variable ranges. Haghghat [25] presents a variety of symbolic analysis techniques based on abstract interpretation. Fahringer [26] presents a unified symbolic evaluation framework that statically determines the values of variables and symbolic expressions. He uses the Omega library[18] for simplifying first order logic expressions and constraints.

We are working on the generation of a routine that computes the array sections of those symbolic bounds and symbolic array sections that have not been resolved at compile time. (At compile time, the symbolic analysis consists of an aggressive simplification of a system of constraints of symbolic expression, determining the relationship between symbolic expressions, computing lower and upper bound of symbolic expression.) In this case we may need to (partially) construct the task graph and perform the task mapping algorithm at run time.

4 EVALUATION AND EXPERIMENTAL RESULTS

To illustrate how the mapping algorithm of Figure 4 works, we present a simple example task graph in Figure 7. After step 1.1, the *levels* of the nodes for tasks t_1 , t_2 , t_3 , t_4 , and t_5 are 81, 101, 1, 1, and 1, respectively, and the numbers of their immediate predecessors ($\#ipr$) are 0, 0, 1, 2, and 2, respectively. In step 1.2, two independent tasks: t_1 and t_2 are put into the RQ ordered by the node level or priority. In step 1.3, task t_2 is selected and removed from RQ as ready task $t_{current}$, and t_2 is assigned to processor $p1$.

In step 2.1, t_2 as $t_{current}$ is passed as input parameter to $list_max_isucss()$, in which t_2 has two immediate successors: t_4 and t_5 . Task t_4 has two incoming edges: $e_{1,4}$ and $e_{2,4}$ with weights of 80 and 100, respectively. Since $e_{2,4}.reuseLevel$ is the greatest, it is in the list. Similarly, t_5 has two incoming edges: $e_{1,5}$ and $e_{2,5}$ with their weights of the same value, so $e_{2,5}$ is also in the list; $list_max_isucss(t_a)$ returns $\{(e_{2,4}), (e_{2,5})\}$ as input to $Max_ipred()$, and since $e_{2,4}.reuseLevel$ has the maximum value, it returns t_4 to t_{temp} and is temporarily mapped to $p1$. It then marks $e_{2,4}$ and $e_{1,4}$ as visited, indicated by the dotted line in Figure 8, needed for the internal workings of the function $Max_ipred()$. In step 2.2, t_4 is compared with task t_1 that has highest priority in RQ , t_1 is assigned to t_{next} . In step 3, $\#ipred(t_3)$ and $\#ipred(t_4)$ are decremented to 1. In step 4, t_1 is removed from RQ . Task t_1 is not the immediate successor of t_2 , therefore $Locate_processor()$ return $p2$, since t_2 has not been temporarily mapped to any processor. Next, t_1 become $t_{current}$.

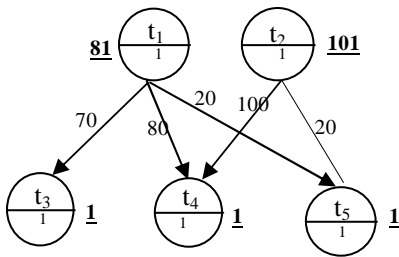


Figure 7 Example task graph

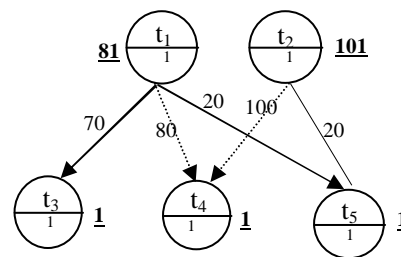


Figure 8 After Mark visited

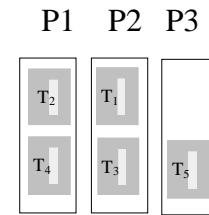


Figure 9 Mapping of Fig. 7

In the next iteration of step 2.1, the best successor of t_1 is t_3 , so t_{temp} is t_3 , which is temporarily mapped to $p2$. It marks $e_{1,3}$ visited. In step 2.2, t_3 is compared with the maximum priority task in RQ ; since RQ is empty, t_3 is assigned to t_{next} . Now in step 3, $\#ipr(t_3), \#ipr(t_4)$, and $\#ipr(t_5)$ are decremented to 0, they are put into RQ . In step 4, since t_3 is in RQ , it is removed. Because t_3 is immediate successor of t_1 , t_3 is mapped to thread that executes t_1 , which is $p2$. t_3 becomes $t_{current}$.

In next iteration of step 2.1, t_3 does not have a successor, so t_{temp} is $NULL$. In step 2.2, t_{next} is t_4 . In step 4, t_4 is in RQ , and is thus removed from RQ ; since t_4 has been temporarily mapped, $Locate_processor()$ map it to $p1$. Then, t_4 becomes $t_{current}$. Finally, in the last iteration of step 2.1, t_4 does not have a successor, so t_{next} is t_5 . In step 4.1, t_5 is removed from RQ and mapped to $p3$ by $locate_processor()$ since it has not been temporarily mapped. The result is shown in Fig. 9.

In the following, we discuss two small codes, one an LU decomposition and the other an ADI iteration, that have been translated using our approach.

4.1 LU example code to demonstrate the data reuse

We present the OpenMP LU example in Figure 10 to illustrate the generation of tasks, task graph, and the data locality reuse scheduling algorithm. Task t_k is generated for k from 1 to 5 respectively, corresponds to the SINGLE construct. When k is 1, the memory access pattern of task t_1 involves write access to section $[2:5,1:1]$ of array A and read reference to regular section $[1:5,1:1]$ of array A is obtained from $[2:5,1:1] \cup [1:1, 1:1]$. Similarly, task t_2 involves write access to array $A[3:5,1:1]$ and read access to $[2:5,1:1]$. All array sections in the example can be exactly represented when we further decompose a block of data into several columns. This requires delayed union operation.

```

Program      LU
integer n
parameter (n=5)
double precision a(n,n)
!$OMP PARALLEL
do k=1,n
!$OMP SINGLE
    do m=k+1, n
        a(m,k) = a(m,k) / a(k,k)
    enddo
!$OMP END SINGLE
!$OMP DO PRIVATE(i,j)
    do j=k+1,n
        do i=k+1,n
            a(i,j)=a(i,j)-
                a(i,k)*a(k,j)
        enddo
    enddo
!$OMP ENDDO
!$OMP END PARALLEL

```

Figure 10 Simple OpenMP LU code example

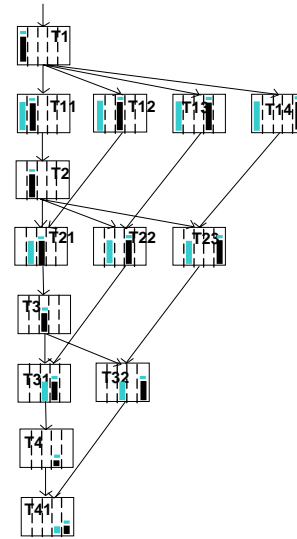


Figure 11 Task graph and memory access pattern

A collection of tasks t_{kc} where $k=1,5$ and c is the chunk number which is obtained from the OpenMP DO construct, is generated from the code of Figure 10. It depends on the loop size, the number of threads and the scheduling chosen; in the present example, it is static scheduling, the chunk size is equal to one, and the number of threads is 4. The reference access to array A of t_{11} is $[2:5,2:2] \cup [2:5,1:1] \cup [1:1,2:2]$. Our delayed union algorithm does not summarize $[2:5,1:2] \cup$

[1:1,2:2] into one representation. Similarly, access to array A by t_{12} is $[2:5,3:3] \cup [2:5,1:1] \cup [1:1,3:3]$ which can be precisely summarized only to $[1:5,3:3] \cup [2:5,1:1]$. If t_{12} is summarized directly to $[1:5,1:3]$, then there will be dependences between t_{12} and t_2 , which is a false dependence. Hence, the delayed union operation provides superior accuracy in this case. Figure 11 shows the task graph of the corresponding program example in Figure 10. It also shows the access pattern of each task. The dark shade represents read and write accesses, while the lighter shade represents read only accesses to array A .

In the OpenMP static scheduling shown in Figure 12, we assume that the number of threads is four, and the work is executed by the first touch policy. It is clear that the data accessed by a processor will not be the same as that accessed in the next iteration of the k loop. Consider the scheduling of OpenMP in Figure 12; after t_1 and t_{11} have been scheduled to processor $p1$, both tasks were the first that touch columns 1 and 2 respectively. Task t_{12} in processor $p2$ first touches column 3 of array A , $p3$ first touches column 4, and $p4$ first touches column 5. There is no problem with t_2 , since it reuses column 2. But t_{22} , t_{23} absolutely cannot reuse the data that has been brought into the local memory, since in $p2$, t_{22} reads column 2, and reads and updates column 4, where its previous t_{12} first touched the column 3 on the same processor. Hence there is no reuse and it needs to access data from the remote memory.

In addition, there are synchronization overheads in OpenMP, for instance, t_2 needs to wait for all t_{11} , t_{12} , t_{13} , and t_{14} to finish; therefore, t_2 can only be executed after t_{14} . We remove synchronization overhead by translating the OpenMP code to task graph; then t_2 only needs to wait for t_{11} , it does not need to wait for t_{12} , t_{13} , and t_{14} to finish. Our mapping is scheduled at runtime by SMARTS runtime system and the result is shown in Figure 13. It can be easily observed that our results will schedule for data reuse, giving better locality reuse.

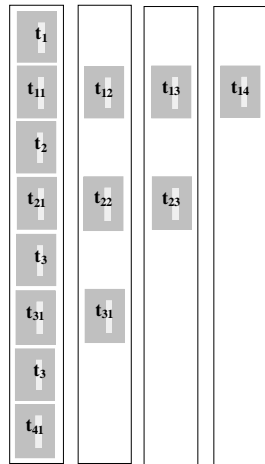


Figure 12 Static scheduling on OpenMP on four processors

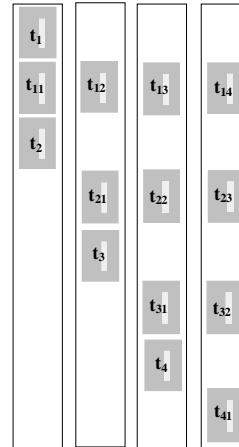


Figure 13 Our scheduling based on data locality reuse on four processors

4.2 ADI example exploiting macro-pipelined computation for data reuse

Pipelining computation and communication have been exploited to improve the performance of many scientific applications [21][23]. Brandes and Desprez [21] integrated pipelined computation successfully within an HPF using compiler called ADAPTOR and gave impressive experimental results. Pipelining has been used by Chamberlain et al. [29] to exploit parallelism in wavefront computations. They introduce extensions to array languages to support pipelining

wavefront computation. Gonzalez et al. [23] propose a set of extensions to OpenMP to allow complex pipelined computations. This is done by defining directives in terms of thread groups and precedence relations among tasks originated from work-sharing constructs, as ours are. It assumes that heuristic scheduling is available to achieve this. But they do not propose any extensions explicitly for *macro-loop* pipelined computation.

Figure 14 presents a Fortran ADI kernel written in OpenMP. The outer loop of the first parallel loop is parallelized so that each thread accesses a contiguous set of columns of the array *A*. The outer loop of the second loop is also parallelizable, but as a result each thread accesses a contiguous set of rows of *A*. The static OpenMP standard execution model is shown in Figure 15, and it can be easily seen that it has poor data locality. Even though we can interchange the second loop so that it also accesses *A* in columns, parallelism is limited as a result of the overheads of the DO loop and the lack of data locality; and the effects of this can be easily observed. Our strategy for handling this situation is to create a version in which threads access the same data and the second loop is executed in a macro-pipelined manner. The task graph that was created for this transformed code is shown in Figure 16. When we apply our mapping algorithm to this task graph, it realizes a *macro-loop* pipeline wavefront computation that enables data locality reuse.

```

!$OMP PARALLEL
!$OMP DO
do j= 1, N
do i= 2, 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= 2, N
A(i,j)=A(i,j)-B(i,j) * A(i,j-1)
end do
end do
!$OMP END DO
!$OMP END PARALLEL

```

Figure 14 ADI Kernel code using OpenMP

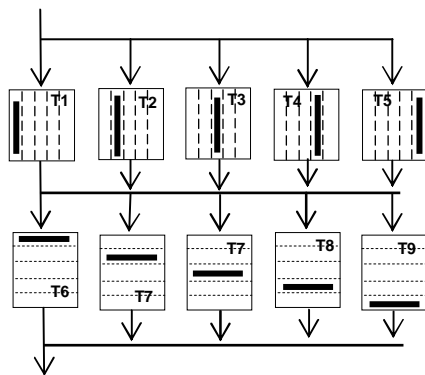


Figure 15 OpenMP standard execution of Figure 14

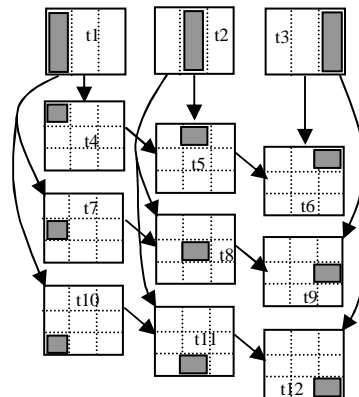


Figure 16 Macro-pipelined computations of Figure 14

4.3 Experimental result

We compare OpenMP versions of the LU kernel and the ADI kernel with the code translated according to our scheme and using SMARTS as a runtime for code. Our target platform was an Origin 2000 at the National Center for Supercomputing Applications (NCSA). All codes were compiled with SGI's MIPSpro Fortran 90 compiler under the options `-64 -Ofast -IPA` and run on MIPS R10000 processor at 195 MHz with 32 Kbytes of split L1 cache, 4 Mbytes of unified L2 cache per processors and 4 Gbytes of DRAM memory. We made use of the first touch allocation strategy provided by SGI, where a datum is stored on the node where it is first accessed. We also set `_DSM_MIGRATION` (page migration) OFF.

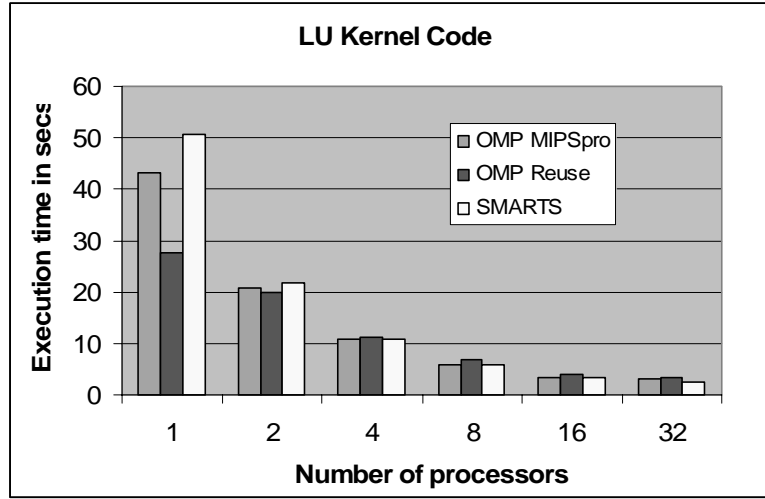


Figure 17 LU kernel Code

We show results for the LU computation using a 1024 by 1024 double precision matrix in Figure 17. We compile the original OpenMP LU code compiled with MIPSpro is labeled MIPSpro. The special hand written OpenMP LU code for scheduling reuse compiled with MIPSpro compilers. Here we see that the SMARTS runtime system introduces non-trivial overheads. The master/slave scheme controls the distribution of work to slave threads. The coordination incurs the overhead for the runtime system. In addition, SMARTS does not accept task graph information in the form of precedence constraints between tasks; instead, it requires information on the read/write access tasks makes to data objects, using these to (re)compute the data dependences at runtime. Therefore, it incurs significant overheads while recomputing information that is already available. We initially chose SMARTS in order to experiment with various strategies for runtime scheduling. Unfortunately, the high overheads in this case make it hard to show the benefits of our approach. In future, we plan to experiment with other runtime systems that suit this need. At the same time, we also want to investigate and develop a method to reduce these overheads possibly by minimizing the number of data objects passing to the runtime system, while avoiding the false dependences caused by decomposition of array into appropriate size.

In this experiment, the only array A in the program it is declared as shared. In addition, we avoid false dependences by breaking the work into smaller tasks. In SMARTS terminology, each array region accessed by a task is a data object. Since the SMARTS runtime system needs to handle many small data objects as a result of our work decomposition, the initialization overheads caused by passing information to the runtime system leads to a slightly longer execution time for the translated code on small numbers of processors. Even though the overhead incurred by the

runtime system is high in this case, it is amortized by the good data reuse as the problem size and the number of iterations increases.

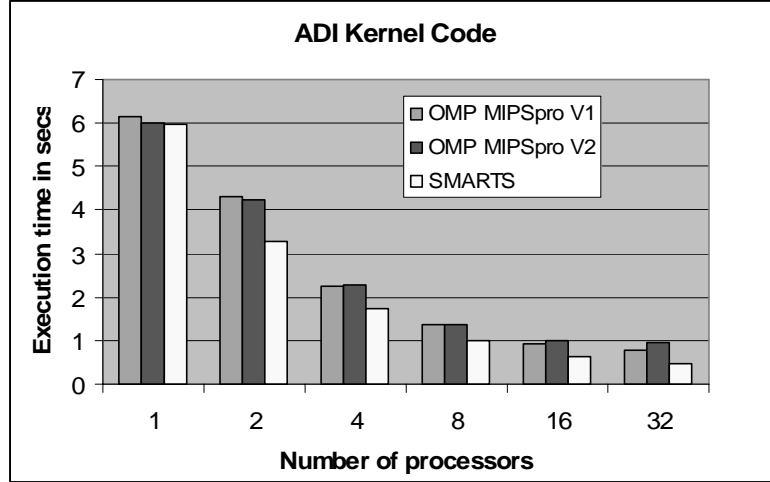


Figure 18 ADI kernel code

Figure 18 illustrates our results on ADI kernel code. There are two version of OpenMP code. Version one parallelizes the outer loop along i loop of Figure 14. Where version two interchanges the loop and parallelizes the inner loop along i loop. For our measurement, we executed the code with an array of size 1024 by 1024. Our translated code to dataflow execution model enables macro-pipelined computation labeled as SMARTS. In the translated code, the number of data objects and read/write requests depend on the decomposition of array A and B . Since it does not need to avoid false dependence by the decomposition of arrays into smaller block size for this code, there is no substantial overhead, for even with a few processors. With an increasing number of processors, translated code also gave a better performance and scale well up to 32 processors.

5 RELATED WORK

The method provided by OpenMP for enforcing locality is to declare variables to be private to threads. One of the best ways to get high performance is to write code in the SPMD (Single Program Multiple Data) style which involves a systematic privatization of program data [19][20]. However, this may require significant reprogramming, which undermines one of the major benefits of the OpenMP API. The use of OpenMP and MPI [5] together also works, but at the price of considerable additional program complexity. In addition, the program will be more difficult to maintain. Several approaches have been proposed to help the user obtain good performance with modest programming effort, including extending OpenMP by data distribution directives and directives to link parallel loop iterations with the node on which specified data is stored [2][3], first touch memory allocation, dynamic page migration by the operating system [4], user-level page migration [4], and compiler optimization [6].

Dynamic page migration has been used in ccNUMA machines to improve data locality. In this scheme, the operating system keeps track of memory page counters to check whether to move a page to a remote node that frequently accesses it. However, this information cannot be associated with the semantics of the program, and if compile-time information is not passed to the operating

system, the latter may move a page when it is not helpful to do so, or it may not move a page that should be migrated. User-level page migration is then required to improve this approach. The compiler or performance tools are needed to help identify the regions of the program that are the most computation-intensive so that calls to page migration routines may be inserted accordingly. Although this is likely to lead to an improvement, it still cannot precisely represent the semantics of the program without compile-time analysis of data access patterns. Programming using data distribution directives has the potential to help the compiler ensure data locality; however, it means that programmers should be aware of the pattern in which threads will access data across large program regions. Hence the programming task remains more complex than with plain OpenMP. Moreover, there is no agreed standard for specifying data distributions in OpenMP; existing ccNUMA compilers have implemented their own data distribution directives, sacrificing portability. The most successful existing strategy is also the simplest: first-touch data allocation places data so that it is local to the thread that first allocates it. Our algorithm also uses this technique, provided by SGI, HP and Sun among others, to ensure appropriate initial storage for data. Thus our work can be viewed as attempting to maintain this locality.

We translate OpenMP to tasks that may share data and map these tasks to a cc-NUMA platform. Most researchers who focus on task mapping problems have performed work to handle the mapping of processes on distributed systems [32][30]. Our tasks may be finer grained than these. Moreover, most distributed memory tasks already have a clear communication structure, which helps in the evaluation of alternatives and generally forms the basis for the solution proposed. Very little work has considered the problem of mapping shared memory tasks on distributed shared memory systems, such as cc-NUMA, where there is a lower penalty for making non-local accesses and the code does not make non-local data references explicit. Liang et al. [31] proposed a static task mapping method for handling shared memory programs on a DSM system based on a two-dimensional Hopfield neural network to map a group of related tasks to a group of workstations that provide DSM. However, the neural network method is too costly to apply to dynamic cases. They also have difficulty estimating the cost that is incurred by remote memory accesses and cache misses, since the actual location of shared data depends on the operating system. We use the first touch approach to minimize the last of these problems.

6 CONCLUSION AND FUTURE WORKS

Our goal is to search for compiler techniques that improve the performance of OpenMP codes without the necessity of manual program modification, especially on large ccNUMA systems. Difficulties include the cost of non-local data accesses, which remain significantly higher than local references, the high amount of global synchronization in most OpenMP codes, and false sharing of data in cache. To achieve our aims, we translate OpenMP codes to a form that may be executed in a dataflow fashion that enables large-scale out-of-order execution, and explore the ability of the compiler to reduce synchronization overheads, as well as to improve data reuse in the resulting ordered collection of tasks [16]. In this paper, we present a practical algorithm for the generation and improvement of a task graph that represents the necessary synchronization in an OpenMP code. We propose an algorithm for mapping the resulting tasks to processors for data reuse. From this, we can generate code scheduling tasks for reuse at runtime. We also demonstrate that this scheme can enable macro-wavefront pipelined computation. Our approach relies on compiler technology and a suitable runtime system; thus there is no need for user intervention.

Despite using a relatively inefficient runtime system in our experiments, the results with LU and ADI kernels show that our strategy has improved performance. However, there is more to be learned. Our mapping algorithm is sensitive to the method for weighting of nodes and edges, which must represent data transfer and cache effects; we intend to continue to experiment with these parameters. Our current effort does not take the problem of false sharing of cache data into account and this needs to be considered. Our future work also includes exploiting symbolic analysis to

improve our ability to handle more difficult cases and to lower the overheads of dynamic task scheduling.

Acknowledgement

This work has been supported by the Los Alamos Computer Science Institute under grant number LANL 03891-99-23 (DOE W-7405-ENG-36) and by the NSF via the computing resources provided at NCSA. We thank Suvas Vajracharya for introducing us to SMARTS, discussing its functionality with us and encouraging us to use the library.

REFERENCES

- [1] OpenMP Architecture Review Board, Fortran 2.0 and C/C++ 1.0 Specifications. At www.openmp.org.
- [2] B.Chapman, P.Mehrotra, and H.Zima, Enhancing OpenMP with Features for Locality Control. Proc. ECMWF Workshop "Towards Teracomputing - The Use of Parallel Processors in Meteorology". Reading, England, November 1998.
- [3] J. Bircsak, P. Craig, R. Crowell, J. Harris, C. A. Nelson, C. D. Offner, Extending OpenMP For NUMA Machines: The Language, WOMPAT 2000, Workshop on OpenMP Applications and Tools, San Diego, July 2000.
- [4] D. S. Nikolopoulos, T. S. Papatheodorou, C. D. Polychronopoulos, J. Labarta and E. Ayguade, Is Data Distribution Necessary in OpenMP? Proc. of the IEEE/ACM Supercomputing'2000: High Performance Networking and Computing Conference, Dallas, Texas, November 2000.
- [5] L. A. Smith and P. Kent, Development and Performance of a Mixed OpenMP/MPI Quantum Monte Carlo Code, Proceedings of the First European Workshop on OpenMP, Lund, Sweden, Sept. 1999, pages. 6-9.
- [6] S. Satoh, K. Kusano, and M. Sato, Compiler Optimization Techniques for OpenMP Programs, Second European Workshop on OpenMP (EWOMP 2000), Edinburgh, September 14-15, 2000.
- [7] S. Vajracharya, S. Karmesin, P. Beckman, J. Crotinger, A. Malony, S. Shende, R. Oldehoeft, and Stephen Smith, Exploiting Temporal Locality and parallelism through Vertical Execution. ICS '99.
- [8] S. Vajracharya, P. Beckman, S. Karmesin, K. Keahey, R. Oldehoeft, and C. Rasmussen, Programming Model for Cluster of SMP. PDPTA '99.
- [9] P. Havlak and K. Kennedy. An implementation of interprocedural bounded regular section analysis. IEEE Transactions on Parallel and Distributed Systems, 2(3), pages 350-360, July 1991.
- [10] V. Balasundaram, K. Kennedy, A Techniques for Summarizing Data Access and its Use in Parallelism Enhancing Transformations, Proceedings of ACM SIGPLAN'89 Conference on Programming Language Design and Implementation, Jun. 1989.
- [11] R. Triolet, F. Irigoien, P. Feautrier, Direct Parallelization of CALL Statements, Proceedings of ACM SIGPLAN '86 Symposium on Compiler Construction, Palo Alto, CA, July 1986, pages 176-185.
- [12] B'eatrice Creusillet. IN and OUT array region analyses. In Workshop on Compilers for Parallel Computers, June 1995.
- [13] B. Creusillet and F. Irigoien. Interprocedural array region analyses. In Lecture Notes in Computer Science - Languages and Compilers for Parallel Computing, pages 46-60. Springer-Verlag, August 1995.
- [14] Michael Hind, Michael Burke, Paul Carini, and Sam Midkiff. An emperical study of precise interprocedural array analysis. Scientific Programming, 3(3), pages 255-271, 1994.
- [15] Zhiyuan Li and Junjie Gu and Gyungho Lee. Symbolic array dataflow analysis for array privatization and program parallelization. In Supercomputing 95, 1995.
- [16] Tien-Hsiung Weng, Barbara M. Chapman. Implementing OpenMP using Dataflow execution Model for Data Locality and Efficient Parallel Execution. In Proceedings of the 7th workshop on High-Level Parallel Programming Models and Supportive Environments, (HIPS-7), IPDPS, April 2002, Ft. Lauderdale.
- [17] P. Tang. Exact Side Effects for Interprocedural Dependence Analysis. Communications of the ACM, 35(8), pages 102- 114, August 1992.
- [18] William Pugh. The Omega test: a fast and practical integer programming algorithm for dependence analysis; Communications of the ACM, 35, pages 102-114, August 1992.
- [19] Zhenying Liu, Barbara Chapman, Tien-hsiung Weng, Oscar Hernandez. Improving the Performance of OpenMP by Array Privatization. WOMPAT 2002, LNCS 2716, pages 244-256.
- [20] Zhenying Liu, Barbara Chapman, Yi Wen, Lei Huang, Tien-hsiung Weng, Oscar Hernandez. Analyses for the Translation of OpenMP Codes into SPMD Style with Array Privatization. WOMPAT 2003, LNCS 2716, pages 26-41.

- [21] T. Brandes and F. Desprez. Implementing pipelined computation and communication in an HPF compiler. In L. Boug'e, P. Fraigniaud, A. Mignotte, and Y. Robert, editors, Euro-Par'96 Parallel Processing, number 1123 in Lecture Notes In Computer Science, pages 459-462, Lyon, France, August 1996. Springer.
- [22] F. Desprez. A Library for Coarse Grain Macro-Pipelining in Distributed Memory Architectures. In Proceedings of the IFIP 10.3 Conference on Programming Environments for Massively Parallel Distributed Systems, pages 365⁺ 371, Monte Verita, Ascona, CH, April 1994.
- [23] S. Hiranandani, K. Kennedy, and C-W. Tseng. Evaluating compiler optimizations for Fortran D. *Journal Of Parallel and Distributed Computing*, 21, pages 27⁺ 45, 1994.
- [24] Gonzalez, M., Ayguade, E., Martorell, X. And Labarta, J.: Complex Pipelined Executions in OpenMP Parallel Applications. International Conferences on Parallel Processing(ICPP 2001), September (2001)
- [25] Mohammad R. Haghighat and Constantine D. Polychronopoulos. Symbolic analysis for parallelizing compilers. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 18, number 4, pages 477-518, 1996.
- [26] Thomas Fahringer and Bernhard Scholz. Symbolic evaluation for parallelizing compilers. In Proceedings of the 11th international conference on Supercomputing, pages 261-268, Vienna, Austria, 1997.
- [27] W. Blume and R. Eigenmann. The range test: A dependence test for symbolic, non-linear expressions. In *Proceedings of Supercomputing '94*. IEEE Press, November 1994.
- [28] D. S. Nikolopoulos and E. Artiaga and E. Ayguadé and J. Labarta. Exploiting memory affinity in OpenMP through schedule reuse. *ACM SIGARCH Computer Architecture News*, vol.29 (5), pages 49-55, 2001.
- [29] B. Chamberlain, C. Lewis and L. Snyder. Array Language Support for Wavefront and Pipelined Computations In Workshop on Languages and Compilers for Parallel Computing, August 1999.
- [30] T. Yang, A. Gcrasoulis, PYRROS: Static task scheduling and code generation for message passing multiprocessors, in: International Conference on Supercomputing, 1992.
- [31] Liang, T. Y., Shieh, C. K., and Zhu, W. P., Task Mapping on Distributed Shared Memory 25 Systems Using Hopfield Neural Network, In Communication Networks and Distributed Systems Modeling and Simulation Conference, Western Multi-Conference '97 (January 1997), pages 37-43.
- [32] Marco Vanneschi. Heterogeneous HPC Environments. Euro-Par'98, pages 21-34.
- [33] Yang, T., Gerasoulis, A. List Scheduling with and without Communication. *Parallel Computing Journal* (19), pages 1321-1344, 1993.
- [34] Weng, T.-H., Chapman, B., Wen, Y.: Practical Call Graph and Side Effect Analysis in One Pass. Technical Report, University of Houston, Under preparation.
- [35] The Open64 compiler. <http://open64.sourceforge.net/>
- [36] Barbara Chapman, Oscar Hernandez, Lei Huang, Tien-hsiung Weng, Zhenying Liu, Laksono Adhianto, Yi Wen. *Dragon: An Open64-Based Interactive Program Analysis Tool for Large Applications*. To appear in the Proceedings of the 4th International Conference on Parallel and Distributed Computing, Applications and Technologies (PDCAT '03).