

Implementing OpenMP using Dataflow Execution Model for Data Locality and Efficient Parallel Execution [†]

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

Abstract

In this paper, we show the potential benefits of translating OpenMP code to low-level parallel code using a data flow execution model, instead of targeting it directly to a multi-threaded program. Our goal is to improve data locality as well as reduce synchronization overheads without introducing data distribution directives to OpenMP. We outline an API that enables us to realize this model using SMARTS (Shared Memory Asynchronous Run-Time System), describe the work of the compiler and discuss the benefits of translating OpenMP to parallel code using data flow execution model. We show experimental results based part of the Parallel Ocean Program (POP) code and Jacobi kernel code running on an SGI Origin 2000.

1. Introduction

OpenMP [2] is an industrial standard for shared memory parallel programming agreed on by a consortium of software and hardware vendors. It consists of a collection of compiler directives, library routines, and environment variables that can be easily inserted into a sequential program to create a portable program that will run in parallel on shared-memory architectures. It is easier for a non-expert programmer to develop a parallel application under OpenMP than in the de facto message passing standard MPI. OpenMP also permits the incremental development of parallel code. Thus it is not surprising that OpenMP has quickly become widely accepted for shared-memory parallel programming.

However, it is up to the user to ensure that performance does not suffer as a result of poor cache locality or high synchronization overheads: these have a particularly large im-

act on cache coherent Non-Uniform Memory Access (ccNUMA) machines. An additional potential cause of performance degradation on such 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 across the network. 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.

The method provided for enforcing locality is to use the PRIVATE clause or THREADPRIVATE directive. However, systematically applied privatization may require a good deal of programming effort, akin to writing an MPI program. This problem has been investigated by many researchers and several approaches have been proposed to provide performance with modest programming effort, including extending OpenMP by providing data distribution directives and directives to link parallel loop iterations with the node on which specified data is stored [7] [4], dynamic page migration by the operating system [9], user-level page migration [10], a combination of OpenMP and MPI [13] [6], and compiler optimization [11] [12].

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 that with plain OpenMP. Moreover, there is no agreed standard for specifying data distributions in OpenMP; existing ccNUMA compilers have implemented their own sets of data distribution directives. Thus, when they are used, performance may well be improved, but portability will be sac-

[†]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.

rificed. So OpenMP users may be forced to choose between simplicity and performance.

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 therefore the operating system 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, with the help of the compiler or performance tools to identify the regions of the program that are the most computation-intensive and call the page migration runtime system to take care of such regions. 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. OpenMP and MPI have been combined to improve data locality. This scheme works, but with the price of considerable additional program complexity. In addition, the program will be more difficult to maintain.

Many compilers translate OpenMP directly to multi-threaded code with little or no program analysis. A more aggressive compiler optimization technique has been proposed to improve the performance of OpenMP applications by removing barriers and improving cache reuse by means of privatization [12]. Our approach is to put work into compiler optimization and to use a runtime system that can better exploit data locality, provide higher degrees of parallelism, and reduce synchronization overheads. We target OpenMP to a data flow execution model realized using the SMARTS runtime system developed at Los Alamos National Laboratory. In order to do so, we first evaluated SMARTS as a runtime system for OpenMP on SMPs and ccNUMA systems and then developed a Fortran API for SMARTS, which serves to pass information from compiler to runtime system.

2. SMARTS runtime system

SMARTS [16] (Shared-Memory Asynchronous Runtime System) was developed by ACL at Los Alamos National Laboratory to support both task and data parallelism. It is a runtime system written in Object-Oriented C++ and is originally designed to be the runtime system for POOMA (Parallel Object-Oriented Method and Application). The novel aspect is its macro-dataflow approach [15]. It extends the master-slave programming model and the single-program-multiple data (SPMD) model to allow multiple parallel loops to be executed in a depth-first manner which will reuse the cached data more effectively on multiprocessors with deep memory hierarchies.

SMARTS is able to support the efficient execution of

code, for which both the data and the work has been decomposed, on shared memory multiprocessors and ccNUMA systems by applying the data flow concept of exploiting temporal locality. It relies upon a prior partition of the original loop iteration space into so-called *iterates*, and a partition of the arrays in the program into *data-objects*. The array partitions defined and used in each iterate must be determined by the programmer or parallelizing compiler. Based upon the data accesses, a task graph is constructed that captures the dependences between iterates. When an iterate requires data that is (partially) created by another iterate, the original execution order must be preserved; similarly, anti- and output dependences between iterates impose an ordering on their execution. So long as these dependences are respected, iterates that are not involved in such relationships can be scheduled concurrently for out-of-order execution. The decomposition of an entire loop into many small sets of iterations may result in having many independent iterates available for execution at any given time. Note that the execution behavior depends on both the data and the work decomposition. It is the task of the user and/or compiler to find a good decomposition. To reduce the overheads associated with the creation of more than one thread in SMARTS, threads are created only once at the beginning of the program and a join is carried out at the end of the program.

In the master slave model, the master thread distributes work to slave threads. A thread is bound to a processor and always runs on that processor, whereas an iterate will be assigned to a thread statically or dynamically. A pool of parallel tasks is implemented by multiple work queues to avoid contention on a shared queue and to promote cache locality of the task queues. When the workload is well distributed across work queues, contention is minimized. When a slave thread is idle, it first grabs work from its own work queue for execution, but if its queue is empty, it may steal work from another work queue. SMARTS uses the first touch policy by default. Under this policy, the process that first touches (that is, writes to, or reads from) a page of memory causes that page to be allocated in the node on which the process is running. Two variants of affinity scheduling have been employed: soft and hard affinity scheduling. With soft affinity or hint affinity scheduling, the user or parallelizing compiler can specify where an iterate is to be bound, but the scheduling is up to the SMARTS runtime system with work stealing when there is an imbalanced work load. On the other hand, hard affinity scheduling means that an iterate is statically bound to a thread under the control of the user or parallelizing compiler without work stealing.

3. Cougar compiler

The Cougar compiler is a prototype software tool being developed at the University of Houston to compile OpenMP

Fortran applications. It also provides a powerful graphical user interface to displays both source code and related information in text and graphical format. After the Cougar preprocessor handles include files, the compiler performs standard analysis including data flow and data dependence analysis. It is currently being extended to translate OpenMP programs to parallel programs that will be executed using the data flow execution model provided by SMARTS. Among other things, this requires us to extend our techniques to parallel data flow analysis and to improve the array section analysis.

Our strategy is to gather as much of the information required by the runtime system as possible to reduce runtime overhead. The compiler decomposes do loops into iterates; this may be based on a user specified strategy (using OpenMP clauses) or it may be determined by the compiler based on an array usage analysis, using techniques originally developed in the search for global automatic data decomposition techniques.

```

call OMP_SET_NUM_THREADS(4)
!$OMP PARALLEL
  do k=0, ITER
!$OMP DO
    do j=1, size-2
      do i=1, size-2
        A(i,j)= (B(i,j-1)+B(i,j+1)+
                B(i-1,j)+B(i+1,j)) / 4.0
      enddo
    enddo
!$OMP END DO
!$OMP DO
    do n=0, size-1
      do m=0, size-1
        B(m,n) = A(m,n)
      enddo
    enddo
!$OMP END DO
  end do
!$OMP END PARALLEL

```

Figure 1. Standard OpenMP version of Jacobi kernel

The generated target code must hand off information to the runtime system such as iterate identification, its read/write requests to data objects, and possibly identify the thread to which the iterate will be bound. Based on these, the SMARTS runtime scheduler maintains request queues with iterate read/write requests. These are later used to schedule iterates for execution in a manner that preserves the dependencies of the program. On the other hand, when there is no dependency between iterates, those iterates can be executed in parallel or can be reordered for execution for maximum cache reuse.

The compiler must further compute the list of data ob-

jects read and written by an iterate, and use these to construct the iterate dependence graph as shown in figures 5 and 8. Array section analysis is a technique to summarize array sub-regions affected by a statement, sequences of statements, or loops, and quality interprocedural array section analysis is required to create an accurate graph. Currently, our compiler relies on a standard, efficient triplet notation based on regular sections [8], rather than more precise methods such as simple section [3], and region analysis [14]. However, when a union of two array regions cannot be precisely summarized, we do not summarize them immediately, but keep them in a linked list. A summary is computed when the length of the list reaches a certain threshold, and we mark the iterate. The dependences between two iterates can be computed by an intersection operation, which returns true or false. Since our compiler is an interactive tool, we can display potential false dependences between iterates to the user. We plan to experiment with a more accurate analysis similar to simple section analysis to evaluate the impact of an improvement in precision.

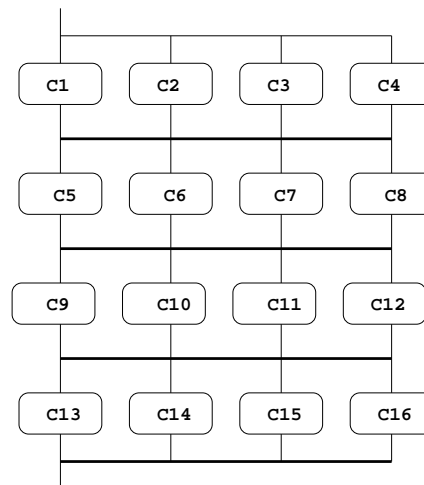


Figure 2. Runtime execution model for Fig. 1 (C stand for chunk or iterate and horizontal bold lines represent barrier)

The generation of the iterate dependence graph at compile time is essential for iterate mapping with soft or hard affinity scheduling. For hard affinity scheduling, each iterate is assigned a thread identification and will be executed by that thread when its data become available. We utilize the iterate dependence graph to do so. When the compiler cannot determine the iterate mapping, the iterate affinity is set to -1 which allows the runtime system to make its own decision.

To illustrate this, we show a simple OpenMP Jacobi kernel in figure 1. Figures 3 and 4 show the corresponding code parallelized using calls to SMARTS via the API we have

created. Figure 5 illustrates the iterate dependence graph constructed by the compiler (it may be compared with the execution model associated with the original OpenMP code in figure 2). Based upon this graph, the compiler might assign the iterates, or chunks of work, to threads as follows: C1 to the master thread, C2 to thread 1, C3 to thread 2, and C4 to thread 3, then C5 to the master thread, C6 to thread 1, C7 to 2, C8 to 3 in order to reuse cache.

```

call SM_concurrency(n)
call SM_define_array(info)
schedtype = hintaffinity
call SM_startgen()
do k = 1, iter
  do il = 1, sqit
    do j1 = 1, sqit
      call SM_def_readwrite(loop0)
      call SM_getinfo(loop0,affin,datinfo)
      call SM_handoff(0,schedtype,affin,datinfo)
    end do
  end do
  do il = 1, sqit
    do j1 = 1, sqit
      call SM_def_readwrite(loop1)
      ! handOff loop 1
      call SM_handoff(1,schedtype,affin,datinfo)
    end do
  end do
end do
call SM_run()
call SM_end()

```

Figure 3. Main program using SMARTS

```

subroutine fortran_loop0(datinfo)
  use share_DATA
  call SM_comput_bound(datinfo,lb1,ub1,lb2,ub2)
  do j = lb1, ub1
    do i = lb2, ub2
      A(i,j) = (B(i,j-1)+ B(i,j+1)+
                B(i-1,j)+ B(i+1,j)) /4.0
    enddo
  enddo
end subroutine fortran_loop0

subroutine fortran_loop1(datinfo)
  use share_DATA
  call SM_comput_bound(datinfo,lb1,ub1,lb2,ub2)
  do n = lb1,ub1
    do m = lb2, ub2
      B(m,n) = A(m,n)
    enddo
  enddo
end subroutine fortran_loop1

```

Figure 4. Parallel loops using SMARTS

Figure 3 also illustrates the structure of the generated SMARTS code and our API. The main program is first executed by the master thread which creates additional threads,

performs initializations, and passes read/write access information for iterates to the runtime system, along with iterate affinity, and iterate identification, etc. The call to SM_concurrency(n) creates an additional n - 1 slave threads in which all iterates will be run. Information such as array partition information is passed via the API function SM_define_array; SM_startgen tells the scheduler that the master thread is starting a new data parallel statement. Information on read/write access of iterates to data objects is then passed to the runtime system achieved via SM_def_readwrite. Next, the API routine SM_handoff takes arguments such as iterate, type of affinity scheduling, and a thread ID that specifies the iterate mapping to slave threads and passes it to the runtime scheduler. SM_run is called to tell the scheduler that no more iterates will be handed off and the scheduler starts to work. SM_getinfo obtains information such as iterate affinity and the bounds of each iterate from data that is computed at compile time in each of the parallel loops.

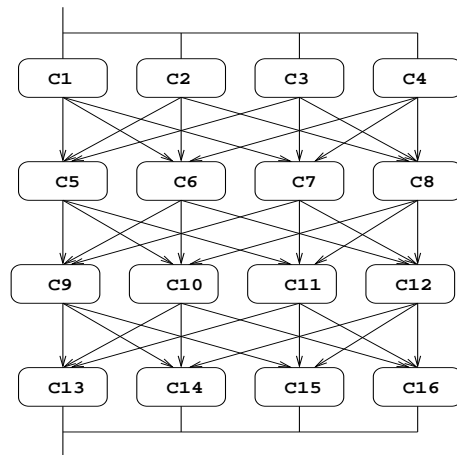


Figure 5. Dataflow execution model associated with Fig. 1

As soon as iterates have been handed off, the SMARTS runtime scheduler checks if their data are available. If they are, the scheduler immediately schedules the iterate to the specified slave queue for execution. When the slave thread is idle, it grabs the iterate and the parallel loops routine associate with the iterate as figure 4 will be invoked. In figure 5, C1, C2, C3, and C4 correspond to subroutine fortran_loop0, and C5,C6, C7, and C8 correspond to subroutine fortran_loop1. At the end of the parallel region, SM_end is invoked to perform a join, which destroys the slave threads.

4. The benefits of a dataflow execution model for OpenMP

Although there is not a lot of potential for speedup with our Jacobi code, the above example already hints at the potential for program improvement if SMARTS is used as a runtime system for OpenMP. Our interest in using SMARTS results from its ability to exploit additional parallelism in a code and from its efficient implementation. Specific benefits of targeting OpenMP to the dataflow execution model include *cross-loop out-of-order execution*, *temporal data locality*, and *reduction of synchronization overheads*.

```

$OMP PARALLEL
!$OMP DO
  do i=1,imt
    RHOKX(imt,i) = 0.0
  enddo
!$OMP ENDDO
!$OMP DO
  do i=1, imt
    do j=1, jmt
      if (k .le. KMU(j,i)) then
        RHOKX(j,i)=DXUR(j,i)*p5*RHOKX(j,i)
      endif
    enddo
  enddo
!$OMP ENDDO
!$OMP DO
  do i=1, imt
    do j=1, jmt
      if (k > KMU(j,i)) then
        RHOKX(j,i) = 0.0
      endif
    enddo
  enddo
!$OMP ENDDO
  if (k == 1) then
!$OMP DO
    do i=1, imt
      do j=1, jmt
        RHOKMX(j,i) = RHOKX(j,i)
      enddo
    enddo
!$OMP ENDDO
!$OMP DO
  do i=1, imt
    do j=1, jmt
      SUMX(j,i) = 0.0
    enddo
  enddo
!$OMP ENDDO
endif
!$OMP SINGLE
  factor = dzw(kth-1)*grav*p5
!$OMP END SINGLE
!$OMP DO
  do i=1, imt
    do j=1, jmt
      SUMX(j,i)=SUMX(j,i)+factor * &
        (RHOKX(j,i) + RHOKMX(j,i))
    enddo
  enddo
!$OMP ENDDO

```

Figure 6. Part of the code that compute the gradient of hydrostatic pressure

4.1. Reduction of synchronization overheads

OpenMP employs a fork-join programming model. In this model, a master thread executes the sequential regions

of the program. When a parallel region is encountered, the master thread forks additional worker threads to share in the work it contains. The model is both flexible and simple, but it may incur significant barrier synchronization overheads. In figure 1, the first parallel DO construct requires a barrier to maintain the correctness of the program since there is a true dependence between the first loop and the second loop. The corresponding runtime execution schema is shown in figure 2, where the iterations performed by each thread are assumed to be a chunk. In this case, each chunk of the second parallel DO construct (C5, C6, C7 and C8) must wait until all of C1, C2, C3, and C4 have completed. On the other hand, in the dataflow execution model as shown in figure 5, each of the parallel loops in the second parallel loop construct only wait for three iterates (chunks) of the first parallel DO construct to finish. As a result, this approach can reduce the barrier synchronization overhead.

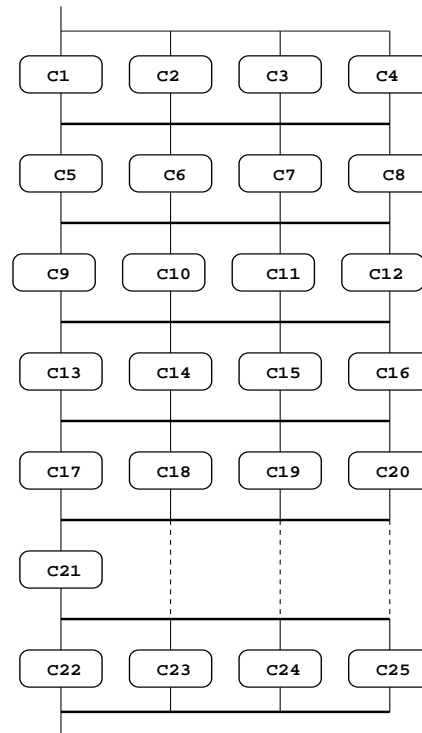


Figure 7. Runtime execution model for Fig. 6

4.2. Data locality

As a shared-memory programming model, OpenMP does not exploit data locality. It is up to the programmer to write code that makes good use of cache and avoids false sharing of cache lines by appropriately privatizing data, possibly padding shared data and blocking loops. Unfortunately, this is responsible for a good deal of the complexity

of using this paradigm. The SMARTS runtime system uses the first touch policy and affinity scheduling with or without

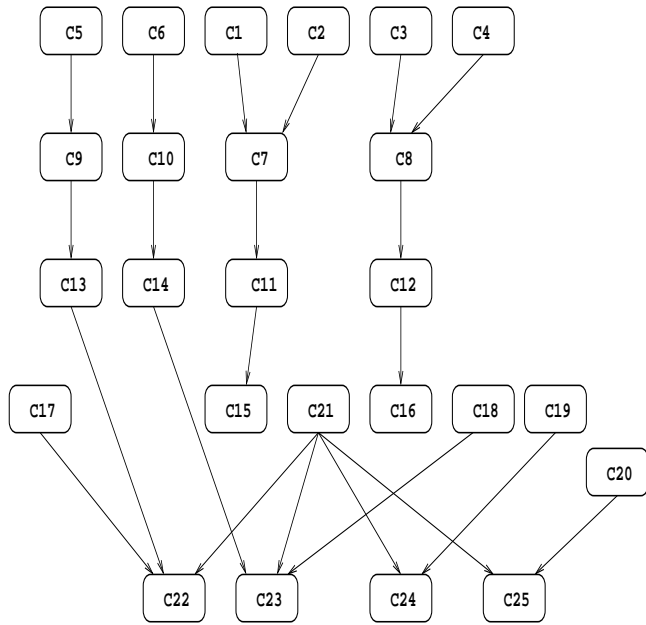


Figure 8. Dataflow execution model associated with translated code in Fig. 6

work stealing to support locality. Work stealing is clearly useful for load balancing, but if it is too eagerly applied, it may lead to performance degradation, given the potential conflict with cache reuse, and this effect is easily observed.

Affinity scheduling is useful in both static and dynamic forms. In the static case, we perform as much compiler optimization as possible to improve data locality by utilizing the iterate dependence graph to map iterates to processors for the best cache reuse. Here, we assign cache reuse a higher priority than load balancing. The idea is to schedule iterates that use the same data to the same processors. For the dynamic case, when the array subscript variable is unknown at compile time, the iterate mapping will be deferred to runtime. This requires the generation of a call to a routine that completes array section computation at runtime. Regular sections are employed to keep the overheads of this runtime computation as low as possible.

4.3. Out-of-order execution

Traditional OpenMP compilation will lead to execution of one parallel construct (DO or SECTIONS) at a time in parallel. SMARTS allows an out-of-order execution not only within but also between parallel constructs if and only if there are no data dependencies between the parallel constructs. Therefore it is necessary to compute the data depen-

dencies between parallel constructs by applying the appropriate array section analysis and array data flow analysis. With the standard OpenMP execution model shown in figure 7, for instance, C17 can only be executed after C13 finishes and C21 can only be executed when C17, C18, C19, and C20 have completed. A translation from OpenMP to SMARTS provides an opportunity for out-of-order execution. As illustrated in figure 8, under this approach C17 no longer needs to wait for C13. Further, the chunks can be reordered to permit cache reuse.

5. Experimental results

We show results of comparisons between OpenMP and SMARTS versions of a simple application based upon the Jacobi kernel as well as one in which we translated part of a Fortran90 OpenMP version of the POP code to a corresponding parallel code based on SMARTS. POP (the Parallel Ocean Program) [1] is an explicit finite difference model developed in FORTRAN 90 by the Los Alamos ocean modeling research group. It makes extensive use of modules, array syntax, and WHERE constructs. Our target platform was an Origin 2000 at the National Center for Supercomputing Applications (NCSA). They were compiled with SGIs MIPSpro Fortran 90 compiler under the option `-64 -Ofast -IPA` and run on MIPS R10000 processor at 195 MHz with 32 Kbytes of split L1 cache and 4 Mbytes of unified L2 cache per processors and 4 Gbytes of DRAM memory. We made use of the strategy provided by SGI to obtain good performance under OpenMP. First touch allocation, where a datum is stored on the node where it is first accessed, was used in each case. We also set `_DSM.MIGRATION` (page migration) environment variable to OFF.

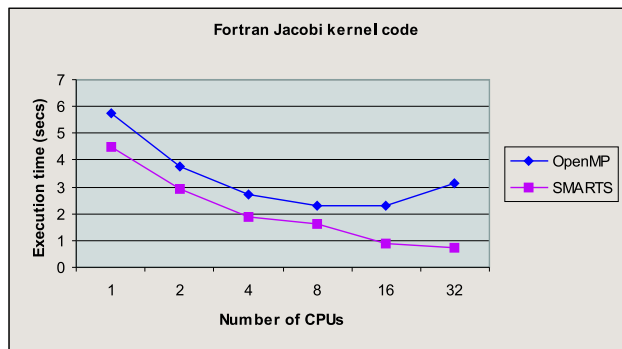


Figure 9. Execution time for Fortran 90 OpenMP Jacobi and translated Fortran 90 Jacobi using SMARTS running on SGI Origin 2000 with array size of 2048x2048

Different implementations of OpenMP on the avail-

able hardware platforms incur different overheads for the OpenMP directives. The measurement of both synchronization and loop scheduling overhead has been conducted by [5]. The PARALLEL DO directive with REDUCTION clause has one of the worst overheads under SGI's MIP-Spro f90 compiler on the SGI Origin 2000. Even though the overhead of the barrier directive itself is minor, the time spent waiting for all threads to finish could be significant when large numbers of processors are involved. We show results for a Jacobi computation based upon runs with a 2048 by 2048 double precision matrix in figure 9. With an increasing number of processors, the barrier and the cache misses clearly affect performance; from 8 processors on, it does not give good speed up. On the other hand, under data flow execution, the Jacobi program obtains good speedup up to 32 processors. When both programs are compiled with O2, the difference between the two is much larger.

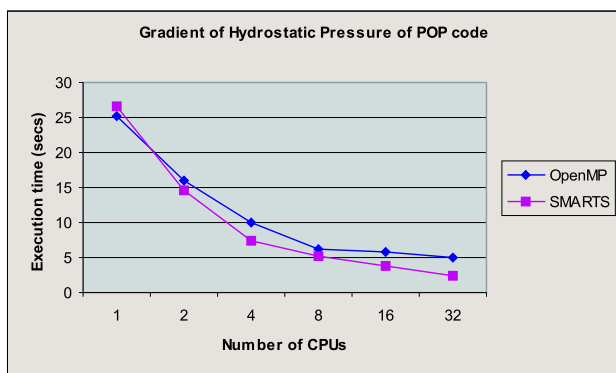


Figure 10. Execution time for Fortran 90 routine of code computing Gradient of Hydrostatic Pressure of POP code and its translated Fortran 90 code using SMARTS on SGI Origin 2000 with array size of 2048

Figure 10 illustrates our results on a routine from the POP code that computes the gradient of hydrostatic pressure at level k . For our measurement, we executed the code with an array of size 2048 by 2048. Number of data objects and read/write requests are greater than that of Jacobi kernel code. Hence, the initialization overhead caused by the passing of information to the runtime system, leads to a longer execution time for the translated code on one processor. This overhead is amortized by the function of the problem size and the number of iterations.

6. Conclusion

In this paper, we show that by using the SMARTS runtime system as an execution target for OpenMP, we can im-

prove performance of the resulting code by improving data locality, reducing synchronization overhead as well as exploiting out of order execution to maximize parallelism. An API has been designed to interface between Fortran and the SMARTS C++ library. It serves as the target for our compiler. We perform an experiment and compare OpenMP with a translated code using SMARTS, our result clearly show that the translated Fortran code using SMARTS outperform the OpenMP code and it also scale well. However, we can do better by further developing the compiler to improve the analysis required for the translation. Our approach avoids the use of data distribution directives, which can add complexity to programming model as well as destroying the portability. Our experimental results consistently show a benefit from this approach.

Acknowledgement

We thank Suvas Vajracharya for introducing us to SMARTS, discussing its functionality with us and encouraging us to use the library.

References

- [1] *Parallel Ocean Program (POP)*, At www.acl.lanl.gov/client/models/pop.
- [2] OpenMP Architecture Review Board, Fortran 2.0 and C/C++ 1.0 Specification. At www.openmp.org.
- [3] V. Balasundaram and 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.
- [4] J. Bircsak, P. Craig, R. Crowell, J. Harris, C. A. Nelson, and C. D. Offner. Extending OpenMP for NUMA Machines: The Language. *WOMPAT 2000, Workshop on OpenMP Applications and Tools, San Diego*, July 2000.
- [5] J. M. Bull. Measuring Synchronisation and Scheduling Overheads in OpenMP. *Proceedings of the First European Workshop on OpenMP*, pages 199–105, Sept. 1999.
- [6] F. Cappello and D. Etiemble. MPI versus MPI+OpenMP on the IBM SP for the NAS Benchmarks. *Proc. of Supercomputing (SC)*, 2000.
- [7] B. Chapman, P. Mehrotra, and H. Zima. Enhancing OpenMP with Features for Locality Control. *Proc. ECMWF Workshop Toward Tera Computing*, November 1998.
- [8] P. Havlak and K. Kennedy. An Implementation of Interprocedural Bounded Regular Section Analysis. *IEEE Transactions on Parallel and Distributed Systems*, 3(2):350–360, July 1991.
- [9] 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.

- [10] D. S. Nikolopoulos, T. S. Papatheodorou, C. D. Polychronopoulos, J. Labarta, and E. Ayguade. User-Level Dynamic Page Migration for Multiprogrammed Shared-Memory Multiprocessors. *Proc. 29th International Conference on Parallel Processing, Toronto, Canada, August 2000*.
- [11] M. Sato and et al. OpenMP Compiler for a Software Distributed Shared Memory System SCASH. *WOMPAT2000*, July 2000.
- [12] S. Satoh, K. Kusano, and M. Sato. Compiler Optimization Techniques for OpenMP Programs. *Second European Workshop on OpenMP (EWOMP 2000)*, pages 14–15, September 2000.
- [13] 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*, pages 6–9, Sept. 1999.
- [14] R. Triolet, F. Irigoien, and P. Feautrier. Direct Parallelization of CALL Statements. *Proceedings of ACM SIGPLAN'86 Symposium on Compiler Construction*, pages 176–185, July 1986.
- [15] S. Vajracharya, P. Beckman, S. Karmesin, K. Keahey, R. Oldehoeft, and C. Rasmussen. Programming Model for Cluster of SMP. *PDPTA*, January 1999.
- [16] S. Vajracharya, S. Karmesin, P. Beckman, J. Crotinger, A. Malony, S. Shende, R. Oldehoeft, and S. Smith. Exploiting Temporal Locality and Parallelism through Vertical Execution. *ICS*, 1999.