Efficient Dynamic Parallelism with OpenMP on Linux SMPs

Christos D. Antonopoulos, Ioannis E. Venetis, Dimitrios S. Nikolopoulos
and Theodore S. Papatheodorou
High Performance Information Systems Laboratory
Department of Computer Engineering and Informatics
University of Patras
Rion 26 500, Patras, GREECE

Abstract *In this paper we present an integrated environment for the efficient support of dynamic parallelism with OpenMP on top of Linux-based SMPs. This environment consists of an OpenMP-compliant Fortran77 compiler, a run-time threads library and a modified Linux kernel.*

The functionality provided by our run-time threads library is used by the NanosCompiler, which converts OpenMP Fortran77 programs to equivalent Fortran77 programs with calls to the library. The NanosCompiler generated applications use a shared arena as a communication path with the OS kernel. This kind of communication facilitates the support of dynamic parallelism, resulting to performance scalability under multiprogramming.

In order to evaluate the efficiency of our approach, we have used a subset of an OpenMP implementation of the NAS benchmarks. We compared the performance of our environment with that of OmniMP. OmniMP is a free source-to-source compiler, that converts OpenMP programs written in C or Fortran77 to equivalent C programs using POSIX threads. Our environment achieves up to 6.3 times higher throughput under the presence of multiprogramming. Moreover, it performs better even on dedicated machines.

Keywords: Dynamic parallelism, OpenMP, OS support, multithreading, multiprogramming

1 Introduction

The OpenMP Application Programming Interface [9] is gaining consistently growing acceptance as the programming model of choice for shared-memory multiprocessors. OpenMP defines a portable programming interface based on directives, i.e. annotations that enclose loops and sections of code. The annotated code is executed by multiple threads, with a fork/join execution model. OpenMP offers the advantage of simplicity, since the shared-memory API can be used to construct a parallel program as a natural extension of its sequential counterpart, thus enabling incremental code development. Furthermore, OpenMP hides the architectural details and relieves the programmer from the duty of data distribution among processors. Several recent studies have demonstrated good scalability of OpenMP codes on small and medium-scale shared-memory multiprocessors [4]. Many multiprocessor vendors including Intel, Compaq, Sun, IBM, HP and SGI [9] are currently participating in the design and evolution of OpenMP, while most of them already support OpenMP on their hardware platforms, using customized compilers and run-time systems.

One of the strong advantages of OpenMP is that, since it is based on a multithreaded execution model, it enables *dynamic parallelism*, that is, OpenMP programs can be executed on a dynamically varying number of processors without modifying the source code. This can be accomplished if the run-time system at the backend of the OpenMP compiler supports a two-level scheduling model, in which the operating system controls the number of physical processors allocated to each OpenMP program, while the run-time system adjusts the number of threads to the number of processors and resumes threads that were preempted by the operating system while executing on the critical path of the program. Support for dynamic parallelism is critical on multiprogrammed production SMP servers, in which the processors are space- and time-shared among multiple sequential and parallel programs. The OpenMP standard already makes provision for the use of dynamic parallelism, using intrinsic functions[1]. However, vendor implementations of OpenMP have not paid sufficient attention to this feature. Dynamic OpenMP parallelism is either left unsupported, or supported via immature implementations.

This paper presents the implementation of an integrated kernel- and user-level infrastructure, targeted to the support of dynamic parallelism with OpenMP primitives, on top of Linux-based SMPs. Small scale SMPs based on Intel x86 microprocessors are widely acknowledged as a cost effective compute server solution. Nowadays, there is also a trend towards using this class of SMPs, combined with a high performance interconnection network such as Myrinet, Gigabit Ethernet etc., as building blocks for large-scale systems.

To the best of our knowledge, this is the first implementation on top of Linux that supports dynamic parallelism. Previous efforts (OmniMP [11], OdinMP [3], Portland Group [1]) are based on LinuxThreads [5], which is an implementation of the POSIX 1003.1c threads standard for Linux. However, LinuxThreads rely on heavy-weight kernel-level threads to implement parallel sections. Hence, they fail to exploit fine-grained parallelism opportunities. Furthermore, LinuxThreads are neither dynamic, nor aware of multiprogramming. The run-time library creates and uses as many kernel-threads as the application requests, not taking into account the variations of system workload and exploitable parallelism during the execution life of the application.

We present the user- and kernel-level implementation details of OpenMP-based dynamic parallelism in Linux, in Section 2. Next, in Section 3, we describe how our OpenMP compiler generates multithreaded code that uses our kernel services for dynamic parallelism. Section 4 provides some experimental evidence on the efficiency of our infrastructure, while the basic conclusions are outlined in Section 5.

2 User- and kernel-level implementation

The OpenMP support we provide, fully complies with the latest version of the OpenMP standard, with the exception of support for orphan directives.

Our approach differentiates from previous efforts in many aspects. The execution environment integrates a light-weight multithreading run-time system with kernel support, in order to achieve both a top-level performance in dedicated environments, as well as performance scalability under the presence of multiprogramming.

The nanothreads run-time library [7] implements parallel regions in terms of nanothreads and work descriptors. Nanothreads are very light, non-preemptive user-level threads. The execution model is dependence-driven. Each control or data dependence between two code snapshots is represented by a dependence between the corresponding threads. When a thread finishes its assigned work, it decrements the dependencies of all the threads that are dependent on it. If the decrement results to threads having no more dependencies left, those threads can be scheduled to the ready queue. In order to enhance locality, there is one local ready queue per processor. There is also a global ready queue from which all processors can take work. Every processor finishing the execution of a thread searches hierarchically its local queue and then the global queue for new threads. All queues are implemented

[1]Dynamic parallelism in OpenMP can be activated, using the `OMP_SET_DYNAMIC()` call.

using lock-free mechanisms.

The use of user-level threads allows us to exploit multilevel parallelism. However, single-level parallelism can be expressed using work descriptors [6]. A work descriptor is a structure containing a pointer to the function to be executed and its arguments. For each parallel region, a master processor creates a work descriptor and distributes it to the slave processors by putting it in a memory region local to each processor. Each slave processor examines its local region for work descriptors and executes them. The use of work descriptors relieves us from the overhead of thread creation and manipulation. The overhead is minimized to that of a function call. This makes work descriptors appropriate for fine-grain parallelism.

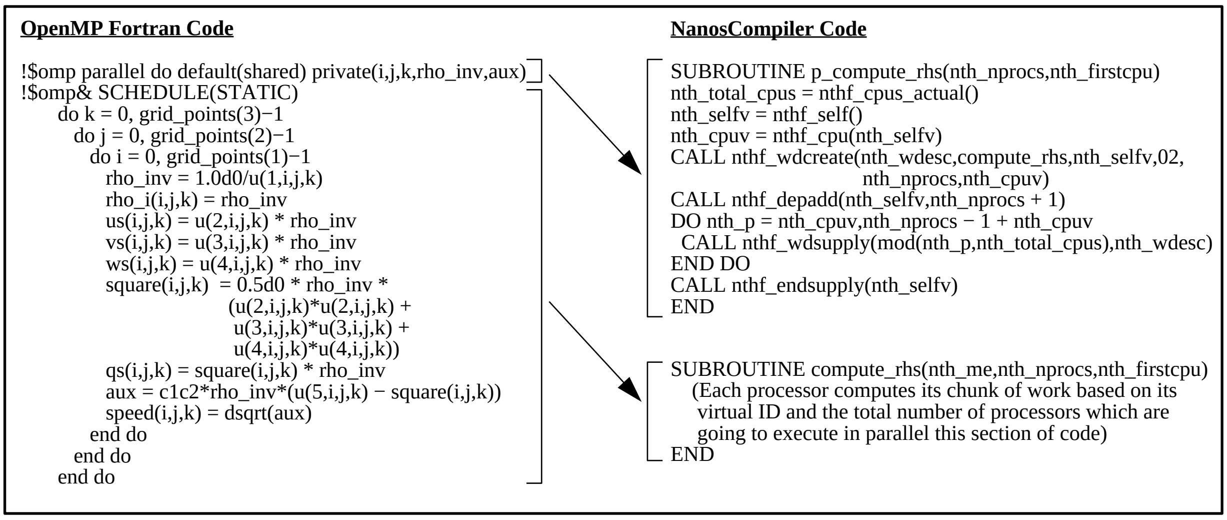
The kernel-threads are provided by the modified Linux kernel and used by the library as execution vehicles (EVs) for the user-level threads. They are created at the beginning of the execution of each application using a very fast batch cloning mechanism that we have implemented [8]. Their quantity is equal to the maximum amount of parallelism expected to be exposed by the application at any given time, but not greater than the available physical processors in the system.

The native OS scheduler of Linux has been augmented by a nanothreads scheduler controlling the applications conforming to our model. The latter works in parallel and cooperatively with the former. The main goals of the nanothreads scheduler are the preservation of EV-processor affinity as well as the fair CPU-time distribution among the nanothreading applications. For the purposes of this work, the kernel-level scheduling policy used is Step Sliding Window DSS [10], a mixed time- and space-sharing policy. It must be pointed out that we have paid extreme attention in order to ensure the fairness of CPU-time allocation between nanothreading and non-nanothreading applications. More specifically, kernel-threads that belong to nanothreading applications compete for CPU-time with the other kernel-threads in the context of the OS scheduler. In the case that the thread selected by the OS scheduler is a nanothreading one and only in this case the physical processor will be assigned to the thread suggested by the nanothreads kernel-level scheduler.

The core of dynamic parallelism support and multiprogramming scalability is a shared arena, which serves as a communication path between the OS kernel and the application. This communication path is implemented using one memory page per application, which is shared with the kernel. The application informs the kernel on its instantaneous processor needs and the state (idler or worker) of its currently running kernel-threads. The kernel communicates to the application the amount of processors currently granted to it and the existence of maliciously preempted kernel-threads, i.e. kernel-threads preempted by the OS while executing work on the critical path. The application is also informed on the exact state (Running, Voluntarily Freed, Preempted, Blocked) of all the kernel-threads it owns. Using this communication mechanism, the kernel-level nanothreads scheduler can grant CPU-time to the applications according to their actual needs. The applications can, on their turn, dynamically determine the amount of processors they are granted and create parallelism accordingly, thus avoiding unnecessary overhead.

The applications are also armed with mechanisms that assist them to make progress on their critical path. When a kernel-thread finds itself idling, it checks the shared arena for preempted kernel-threads belonging to the same application. If such a thread is found, the idler hands its processor off -via a system call- to the preempted thread. If this is not the case, the idler attempts to give its processor to a freed or blocked kernel-thread of the same application. If the idling period lasts for long, the application may just return the processor to the nanothreads scheduler, greedily hoping that there are other nanothreading applications able to effectively use it.

The mechanisms we provide are effective only during the time periods a nanothreading application is alive in the system. When these



OpenMP Fortran Code

```
!$omp parallel do default(shared) private(i,j,k,rho_inv,aux)
!$omp& SCHEDULE(STATIC)
      do k = 0, grid_points(3)−1
        do j = 0, grid_points(2)−1
          do i = 0, grid_points(1)−1
            rho_inv = 1.0d0/u(1,i,j,k)
            rho_i(i,j,k) = rho_inv
            us(i,j,k) = u(2,i,j,k) * rho_inv
            vs(i,j,k) = u(3,i,j,k) * rho_inv
            ws(i,j,k) = u(4,i,j,k) * rho_inv
            square(i,j,k)  = 0.5d0 * rho_inv *
                             (u(2,i,j,k)*u(2,i,j,k) +
                              u(3,i,j,k)*u(3,i,j,k) +
                              u(4,i,j,k)*u(4,i,j,k))
            qs(i,j,k) = square(i,j,k) * rho_inv
            aux = c1c2*rho_inv*(u(5,i,j,k) − square(i,j,k))
            speed(i,j,k) = dsqrt(aux)
          end do
        end do
      end do
```

NanosCompiler Code

```
SUBROUTINE p_compute_rhs(nth_nprocs,nth_firstcpu)
nth_total_cpus = nthf_cpus_actual()
nth_selfv = nthf_self()
nth_cpuv = nthf_cpu(nth_selfv)
CALL nthf_wdcreate(nth_wdesc,compute_rhs,nth_selfv,02,
                   nth_nprocs,nth_cpuv)
CALL nthf_depadd(nth_selfv,nth_nprocs + 1)
DO nth_p = nth_cpuv,nth_nprocs − 1 + nth_cpuv
  CALL nthf_wdsupply(mod(nth_p,nth_total_cpus),nth_wdesc)
END DO
CALL nthf_endsupply(nth_selfv)
END

SUBROUTINE compute_rhs(nth_me,nth_nprocs,nth_firstcpu)
  (Each processor computes its chunk of work based on its
   virtual ID and the total number of processors which are
   going to execute in parallel this section of code)
END
```

Figure 1: A Fortran code fragment translated from OpenMP to multithreaded code by the NanosCompiler.

mechanisms are deactivated our kernel practically corresponds to an unmodified Linux kernel, relieving the system from any overhead -even minimal- our modifications could possibly introduce.

3 OpenMP to Nanothreads Code Transformation

The applications executed on our environment are compiled using the OpenMP NanosCompiler [2]. The NanosCompiler is a parallelizing compiler that captures the parallelism expressed by the user through OpenMP directives and the parallelism automatically discovered through a detailed analysis of data and control dependencies. In our case, we have only used the front-end which converts programs written in Fortran77 that use OpenMP directives to equivalent programs that use calls to the nanothreads run-time library. In this section, we describe how these transformations are applied. Figure 1 shows a fragment of code from the SP application and the corresponding NanosCompiler generated code. Before reaching the parallel region the application requests as many processors as the user-specified maximum amount of efficiently exploitable parallelism. The compiler generates a new function which is executed by the master thread of the parallel section and corresponds to the *parallel do* OpenMP directive encountered in the original Fortran program. Another function that corresponds to the body of the parallel section is also generated. This function is provided to the consumer virtual processors via a work descriptor. The master thread reads from the shared arena the number of granted to the application physical processors (`nthf_cpus_actual()`). This number may be different from the requested processors. It then creates the appropriate work descriptor (`nthf_wdcreate()`), which is provided (`nthf_wdsupply()`) to all granted virtual processors -corresponding with a one by one relation to the granted physical processors. The appropriate dependencies have previously been added to the master thread (`nthf_depadd()`). Finally, the master thread calls `nthf_endsupply()` to execute its chunk of work and then block. When all consumers finish the execution, the dependencies of the master thread are satisfied. As a result the master gets resumed and continues its execution. In the case of multilevel parallelism, threads are used (via calls to analogous library functions) instead of work descriptors for all but the innermost level.

The scenario of a malicious preemption by the OS of a consumer virtual processor is automatically dealt with by the run-time library.

Each virtual processor that finishes its assigned chunk of work checks the shared arena for preempted virtual processors of the same application and reacts to the preemption by handing off its physical processor to the preempted virtual one. The preempted virtual processor may be resumed even earlier, in the case of a physical processor yield by another nanothreading application that considers itself unable to effectively use all the processors granted to it by the OS.

4 Performance Evaluation

We have compared the programs produced by the NanosCompiler with those of the OmniMP OpenMP compiler, version 1.0. OmniMP is a freely available front-end that translates C and Fortran77 programs that use OpenMP directives to C code. The intermediate code includes calls to POSIX 1003.1c threads functions, which are implemented, in our case, by the LinuxThreads library.

After producing the intermediate code the NanosCompiler uses the Fortran77 and the OmniMP compiler the C front-end to the GCC (GNU Compiler Collection) compiler, which creates the final executables. We have used a high level (-O4) of general, as well as processor specific (-march=i686) optimizations.

To evaluate the efficiency of our approach, we have executed a subset of an OpenMP implementation of the NAS class W benchmarks [4] using the two aforementioned environments. The subset constitutes of three computational kernels -a conjugate gradient (CG), a 3-dimensional FFT (FT) and a multigrid benchmark (MG)- and two applications-solvers of three uncoupled systems of equations: scalar pentadiagonal (SP) and block tridiagonal (BT). The benchmarks are written in Fortran77 and are parallelized using OpenMP directives.

The machine used during the evaluation is a Compaq Proliant 5500 with four Pentium-Pro processors, each one clocked at 200 MHz and equipped with 512 Kbytes level two cache, and 512 Mbytes of physical memory. The operating system is Linux 2.2.13 with the kernel modifications described earlier.

The workloads executed during the evaluation are homogeneous, i.e. they constitute of concurrently running identical copies of the same benchmark. Each benchmark requests four processors and the quantity of concurrently active benchmarks is equal to the degree of multiprogramming we want to achieve. We have experimented with multiprogramming degrees from 1 to 16 in powers of two. Each workload has been executed three times and the results reported are the average of the three executions. We measure the average turnaround time of the benchmarks, which is a metric characterizing the achieved throughput. It should be mentioned that the variation between the turnaround times of applications participating in each workload was low -in any case lower than the one achieved by OmniMP- which indicates the performance stability of our environment. The results are depicted in Figure 2. We mention in passing that comparable results heve been also attained using Portland Group pgf77 compiler instead of OmniMP and GCC. The results are not reported here due to license limitations.

The nanothreading compilation and execution environment outperforms the OmniMP environment in both the uniprogrammed and multiprogrammed versions. We believe that the efficiency of the nanothreading versions of benchmarks under uniprogramming is due to the fact that the light-weight threads and work descriptors we use are much more appropriate for loop-level parallelism than heavy kernel-threads. Attention is also paid in order to ensure the proper alignment of data in the thread stacks. Furthermore, the architecture of our run-time library allow the application to implicitly achieve high levels of data locality.

The correctness of our approach is clearly demonstrated by the multiprogramming results. The average turnaround time of nanothreading applications is from 2.3 to 6.3 times less than that of the corresponding OmniMP applications, when the multiprogramming de-

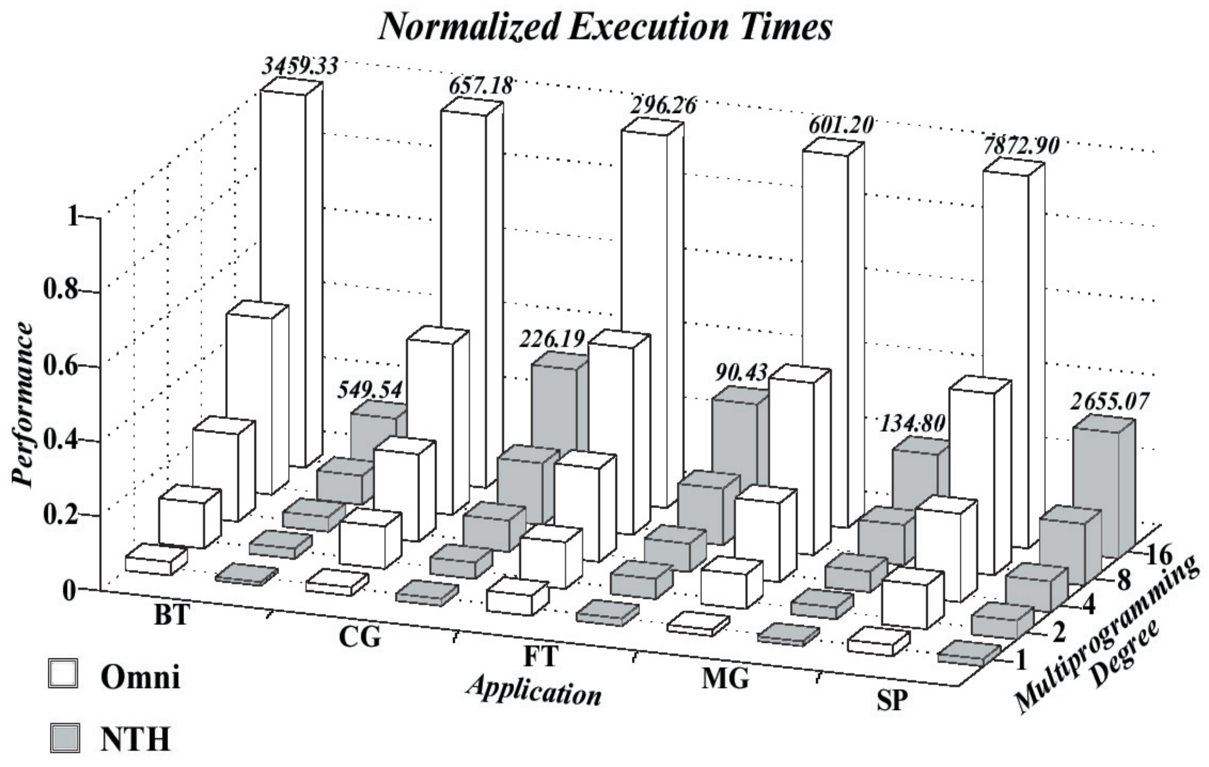


Figure 2: Normalized turnaround times of the evaluation workloads. For multiprogramming degree 16 the actual turnaround times are given in seconds.

gree ranges from 2 to 16. It is worth observing that the performance gap increases with the raise of multiprogramming degree.

Under multiprogramming, the nanothreads kernel-level scheduler limits the number of running kernel-level threads of each application, thus minimizing unnecessary context switches and synchronization overhead. In the same time, the run-time library, being aware of the actual number of running kernel-level threads, creates only the efficiently exploitable amount of parallelism thus further reducing overhead. Additionally, the handoff mechanisms provided by the kernel allow the applications to escape the consequences of malicious preemptions. The careful, lock-free implementation of the run-time library contributes to the same direction. Finally, both the kernel-level scheduling policy used, which tries to preserve affinity between processors and kernel-threads, as well as the hierarchical queue design of the run-time library result to enhanced data locality.

We must note that the OmniMP version of the BT benchmark has exhibited unexpected behavior. When executed on one processor the OmniMP BT executable is as fast as an executable created by the native GCC compiler, disregarding OpenMP directives. However, the execution of the OmniMP version on two processors results to a slowdown of two. If more processors are offered to the application the execution time degrades normally. The same problem used to occur in a previous version of our run-time library. The reason, in our case, has proven to be the incorrect alignment of data in the stacks of threads.

5 Conclusions

Previous implementations of OpenMP for Linux have ignored the need for dynamic parallelism support due to lack of appropriate kernel and run-time library extensions. We have presented an integrated compilation and execution environment for the support of dynamic parallelism in OpenMP applications running on top of Linux-based SMP systems. We have shown that an efficient bi-directional communication path between the applications and the OS kernel is of vital importance for dynamic parallelism support and multiprogramming scalability. Of the same importance has proven to be a mechanism assisting the applications make progress on their critical path in the presence of undesirable preemptions. We also consider user-level threads to be more appropriate than kernel-level threads -such as the ones provided by LinuxThreads- for efficient exploitation of parallelism.

We have evaluated the performance of our environment using a subset of an OpenMP implementation of the NAS benchmarks. Our environment has proven to be up to 6.3 times more efficient than OmniMP in terms of throughput on a multiprogrammed machine.

Acknowledgement

The authors would like to thank Eduard Ayguadé, Marc Gonzalez, Xavier Martorell, and Jose Oliver, who developed the NANOS compiler. This work was supported in part by G.S.R.T. research program 99EΔ-566 and the European Commission, through ESPRIT IV Framework, Project No. 21907 (NANOS).

References

[1] Portland Group, Inc. Web Site. http://www.pgroup.com.

[2] E. Ayguadé et al. NanosCompiler: A Research Platform for OpenMP Extensions. In *Proc. of the First European Workshop on OpenMP*, pages 27–31, October 1999.

[3] C. Brunschen and M. Brorsson. OdinMP/CCp - A portable implementation of OpenMP for C. In *Proc. of the 1st European Workshop on OpenMP (EWOMP'99)*, September 1999.

[4] H. Jin, M. Frumkin, and J. Yan. The OpenMP Implementation of NAS Parallel Benchmarks and its Performance. Technical Report NAS-99-011, NASA Ames Research Center, 1999.

[5] X. Leroy. The LinuxThreads Library. http://pauillac.inria.fr/∼xleroy/linuxthreads/index.html.

[6] X. Martorell et al. Thread Fork/Join Techniques for Multi-level Parallelism Exploitation in NUMA Multiprocessors. In *Proc. of the 1999 International Conference on Supercomputing*, June 1999.

[7] X. Martorell, J. Labarta, N. Navarro, and E. Ayguadé. A Library Implementation of the Nano-Threads Programming Model. In *Proc. of the Second International Euro-Par Conference, Vol. 2*, August 1996.

[8] D. Nikolopoulos, C. Antonopoulos, I. Venetis, et al. Achieving Multiprogramming Scalability of Parallel Programs on Intel SMP Platforms: Nanothreading in the Linux Kernel. In *Proc. of the Parallel Computing'99 (ParCo'99) Conference*, August 1999.

[9] OpenMP A.R.B. OpenMP FORTRAN API, Version 1.1. Technical report, http://www.openmp.org, November 1999.

[10] E. Polychronopoulos et al. An Efficient Kernel-Level Scheduling Methodology for Multiprogrammed Multiprocessors. In *Proc. of the 12th ISCA Conference on Parallel and Distributed Computing Systems*, August 1999.

[11] M. Sato, S. Satoh, K. Kusano, and Y. Tanaka. Design of OpenMP Compiler for an SMP Cluster. In *Proc. of the 1st European Workshop on OpenMP (EWOMP'99)*, September 1999.