An Implementation and Evaluation of Thread Subteam for OpenMP Extensions *

Lei Huang, Barbara Chapman and Chunhua Liao
University of Houston
4800 Calhoun Rd.
Houston, TX 77204
(leihuang,chapman,liaoch)@cs.uh.edu

ABSTRACT

OpenMP provides a portable programming interface on shared memory multiprocessors (SMPs). The set of features in the current OpenMP specification provides essential functionality that was selected mostly from existing shared-memory parallel application programming interfaces (APIs). Although this interface has proven successful for small SMPs, it requires greater flexibility in light of the steadily growing size of individual SMPs and the recent advent of multithreaded chips. In this paper, we introduce the syntax and semantics of a proposed OpenMP extension for facilitating the expression of worksharing on the emerging chip multithreading architectures. We also describe our implementation in the OpenUH reference OpenMP compiler and the runtime library. We then evaluate the new feature using a kernel of seismic data processing application.

Categories and Subject Descriptors

D.3.4 [**Programming Languages**]: Parallel languages; D.3.2 [**Programming Languages**]: Compilers

General Terms

Languages, Compiler, Experiments

Keywords

Parallel Programming Language, OpenMP, Compiler

1. INTRODUCTION

OpenMP [11] is a standard high-level parallel programming API for shared memory multiprocessors (SMPs). It consists of a set of compiler directives, runtime library and environment variables, which can be used to create parallel region, to define work sharing, to manage the data environment, and to specify synchronizations. Strong points of OpenMP include the ability to support incremental parallelization, portability, and ease-of-use. Although OpenMP has been mainly used in high performance computing applications development, it is being deployed more and more to general business software to exploit the parallelism when the multicore architectures are becoming mainstream.

As computer components decreasing in size, architects have begun to consider different strategies for exploiting the space on a chip and sought for more efficient ratio of performance and power consuming. Chip MultiThreading (CMT) architectures provide support for the simultaneous execution of two or more threads within one chip. It may be implemented through several physical processor cores in a chip (Chip MultiProcessor, CMP) [9], a single core with replication of features to maintain the state of multiple threads simultaneously (Simultaneous multithreading, SMT) [15] or their combination [6, 7]. A hierarchical multithreading architecture results from using several of these chips in a single SMP. However, OpenMP was not designed for such hierarchical parallelism. It does not provide a mechanism to allow a programmer assigning different kinds of workloads to sibling threads in order to avoid resource contention. The OpenMP *sections* construct restricts only one thread execution in each section; it is not flexible to express work sharing. Moreover, OpenMP was traditionally targeted to computational intensive applications, where large amounts of calculations are performed in loops. In the execution of OpenMP programs, all available threads are managed using only one team and largely treated equally. The one-team-for-all scheme makes work sharing among regular computation loop iterations very easy and fits symmetric multiprocessor platforms well. However, the increasing deployment of chips with multi-threading capabilities is likely to lead to more diverse parallel applications based on OpenMP. Programmers will need language mechanisms that are able to express complex algorithms and facilitate scalable parallel programming on these hierarchical systems, including flexibility in the assignment of work to threads.

2. OPENMP EXTENSIONS

The set of features in existing specifications of the API provide essential functionality mostly selected from existing shared memory parallel APIs. In this paper, we describe a proposed OpenMP extension for facilitating the expression of worksharing on the emerging chip multithreading architectures. We then describe our implementation of the new feature in the OpenUH reference OpenMP compiler, following some preliminary experiments. Our goal is to iden-

*(This work was supported by the DOE under contract DE-FC03-01ER25502.

tify language features that help improve application performance while preserving ease of programming. The new feature may broaden application space for OpenMP from high performance computing architectures to ordinary computers, even to embedded systems.

The motivation of the OpenMP extension came from our experience of parallelizing different applications with OpenMP [2]. Our inability to control the assignment of work to subsets of threads in the current thread team and to orchestrate the work of different threads artificially limited the performance that we could achieve. In order to overcome the first difficulty, we propose a new clause for worksharing constructs that assigns the work to a subteam of the existing threads. There is much to be explored here, such as notation for naming a subteam, team-to-team synchronizations, and thread topology. This paper presents a minimal change of current OpenMP specification and its implementation in a reference compiler.

2.1 Thread Subteam Syntax

We have proposed to add one clause *ONTHREADS* [2] on OpenMP worksharing directives, which include *omp for/do*, *omp section*, *omp single*, and *omp workshare*. The clause specifies a subteam of threads that share the workloads; it does not imply the thread-to-processor mapping. The clause supports flexible worksharing among the logical thread subteams. The syntax of the clause is as following:

```
onthreadsclause :  onthreads ( threadset )

threadset : scalar integer expression
          | subscript-triplet
          | threadset , threadset

subscript : scalar integer expression
subscript-triplet : [subscript]:[subscript][:stride]
stride    : scalar integer expression
```

We could also define a name for a subteam and reuse it later. For the current implementation, our goal is to demonstrate the usage of the feature, so we keep it as simple as possible. Meanwhile, we need the following runtime functions to support the subteam notation.

- *omp_get_subteam_numthreads()* returns the number of threads participating in the current subteam.
- *omp_get_subteam_threadnum()* returns the logical thread number within the current subteam.
- *omp_set_number_subteam(int number)* sets the total number of subteams in the current parallel region
- *omp_subteam * omp_set_subteam(int start,int end, int stride)* sets the starting thread number, the ending thread number, and the stride in a subteam

The code enclosed within the work-sharing region will be executed by the threads specified in the threadset, which consists of some or all of the threads that encounter the construct. Other threads may proceed past this construct, much as they would if they encountered an *omp single* with a *nowait*. We would consider all threads to have "encountered" any implicit barrier associated with the construct, but only the threads specified in the threadset wait on the barrrier. If no *onthreads* clause is present, the threadset is equal to all threads in the current team (in other words, there is no change).

The subscripts in the threadset description refer to the threads in the current team via their ids. The section notation introduced in Fortran 90 has been used here to conveniently refer to sets of threads, e.g. onthreads (2:6:2) refers to threads 2, 4 and 6. If the stride is omitted, it is assumed to have a value of 1. If "onthreads (:)" is specified, this refers to all threads in the current team, which is also the default. In order to keep the correct control flow for all threads, we must assume that implicit barriers at the end of worksharing constructs are encountered by all threads in a team that reach the construct.

The associated work will only be executed on the threads specified that are part of the current team of threads. If the threadset contains values that do not correspond to thread ids in the current team, this will not prevent execution of the remaining threads, if any.

2.2 Implementation in OpenUH

The OpenUH compiler [12] is developed and maintained by University of Houston for OpenMP related research. It is a branch of the Open64 compiler [10], which was originally developed and put into the public domain by Silicon Graphics Inc. OpenUH is a portable OpenMP compiler based on the Open64 compiler infrastructure combining a state-of-the-art optimizing infrastructure with a source-to-source approach. OpenUH supports C/C++/Fortran 90 and includes a complete implementation of OpenMP 2.5. It is able to generate executables for Intel Itanium IA-64 architectures, and achieves the portability by generating a compilable source code. OpenUH includes state-of-the-art analysis and optimzations, such as interprocedural analysis, data flow analysis, data dependence analysis and array region analysis. We are using the open source compiler infrastructure to perform our OpenMP and compiler research.

Fig. 1 depicts an overview of the OpenUH compiler. It consists of the frontends, optimization modules, OpenMP transformation module, a portable OpenMP runtime library, a code generator and IR-to-source tools. Most of these modules are derived from the corresponding original Open64 module. OpenUH has five levels of a tree-based IR called WHIRL to facilitate the implementation of different analysis and optimization phases. They are classified as being Very High, High, Mid, Low, and Very Low levels, respectively. Most compiler optimizations are implemented on a specific level of WHIRL. It is a complete native compiler for Itanium platforms, for which object code is produced, and may be used as a source-to-source compiler for non-Itanium machines using the IR-to-source tools. The translation of a submitted OpenMP program works as follows: first, the source code is parsed by the appropriate extended language frontend and translated into WHIRL IR with OpenMP prag-

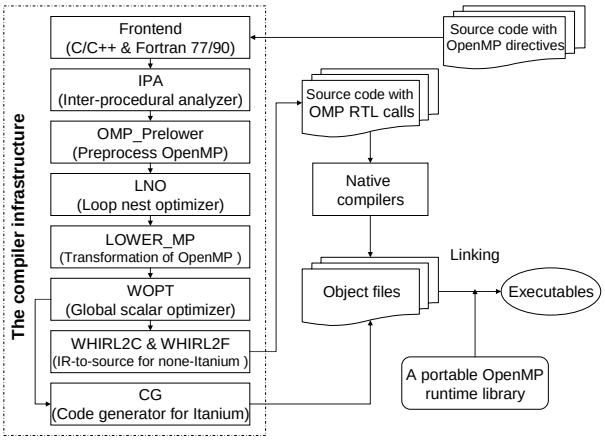

Figure 1: OpenUH: an optimizing and portable OpenMP compiler based on Open64

mas. The next phase, the interprocedural analyzer (IPA), is enabled if desired to carry out interprocedural alias analysis, array section analysis, inlining, dead function and variable elimination, interprocedural constant propagation and more. After that, the loop nest optimizer (LNO) will perform many standard loop analyses and optimizations, such as dependence analysis, register/cache blocking (tiling), loop fission and fusion, unrolling, automatic prefetching, and array padding. The transformation of OpenMP, which lowers internal IR with OpenMP pragmas into IR representing multithreaded code with OpenMP runtime library calls, is performed in two steps: OMP_Prelower and LOWER_MP. The first phase preprocesses OpenMP pragmas while the latter does the major transformation. Please refer to [8] for more information about how OpenUH implements OpenMP. In the following subsections, we explain the extensions to OpenUH in order to support thread subteam.

2.2.1 Frontends

OpenUH has the multiple language frontends supporting C/C++ and Fortran 90. Currently, we only implemented the OpenMP subteam clause into its Fortran frontend, and we will extend it to C/C++ later. The Fortran frontend uses the Cray Fortran 90 to parse the Fortran syntax, to check the semantics, and to generate the intermediate representation (IR) of the program. The generated IR is in Cray F90 IR format and is converted to OpenUH's own IR. We extended the Cray F90 frontend to handle the *ONTHREADS* clause and to check the semantics. The semantics checking ensures that the *ONTHREADS* clause is only used on OpenMP worksharing directives, and verifies the correct semantics of its parameters. And then we modified the OpenUH IR conversion functions to convert the Cray F90 IR of *ONTHREADS* to OpenUH high level WHIRL.

2.2.2 OpenMP Translation

An OpenMP implementation transforms code with OpenMP directives into corresponding multithreaded code with runtime library calls. Most open source compilers including Omni [13] and OdinMP/CCp [1] translates OpenMP parallel regions using outlining approach, which generates a separate procedure to encapsulate the work for each parallel region. The OpenUH compiler uses a lightweight approach to handle parallel regions by generating a nested procedure into the original function. We extended the translation by processing the new *ONTHREADS* clause. It generates a new function call *ompc_subteam_create* whenever the translation encounters the clause. The function creates a new global data structure for subteam and return its address to a pointer. Note that the subteam structure needs to be shared by the team of threads in order to synchronize them. We then pass the pointer into all related OpenMP runtime functions, such as scheduling and barrier. To preserve the backward compatibility, we pass a NULL pointer to these functions if there is no subteam specified on an OpenMP worksharing directive. Therefore, the runtime functions are able to distinguish if only a subteam of threads or the whole team of threads need to participate in the worksharing.

```
!$OMP DO ONTHREADS(1:NUMTHREADS-1:1)
      do i=1,N
         A(i) = i
      enddo
!$OMP END DO
```

(a) The original OpenMP program with subteam

```
tmp0 = ompc_subteam_create(mp_subteam_7937,
 %val(mplocal_t$5), %val(mplocal_t$6), %val(mplocal_t$7))
tmp0 = ompc_static_init_4(%val(ompv_temp_gtid),
 %val(1), ompv_temp_do_lower, ompv_temp_do_upper,
  mpv_temp_do_stride, %val(1), %val(mplocal_t$8),
  mp_subteam_7937)
DO WHILE(ompv_temp_do_lower .LE. temp_limit)
 IF(ompv_temp_do_upper .GT. temp_limit) THEN
  ompv_temp_do_upper = temp_limit
 ENDIF
 DO mplocal_I = ompv_temp_do_lower, ompv_temp_do_upper, 1
  A(mplocal_I) = mplocal_I
 END DO
 ompv_temp_do_lower = (ompv_temp_do_lower
   + ompv_temp_do_stride)
 ompv_temp_do_upper = (ompv_temp_do_upper
   + ompv_temp_do_stride)
END DO
tmp0 = ompc_barrier(mp_subteam_7937)
```

(b) The corresponding compiler translated code

Figure 2: A compiler translated OpenMP code with the subteam clause.

Figure 2 shows a OpenMP Do construct with the subteam clause specified and its corresponding compiler translated program. The source code is generated using the source-to-source capability of OpenUH after it processed the OpenMP constructs. A subteam structure *mp_subteam_7937* is created and passed into scheduling and barrier functions. The scheduling function *ompc_static_init_4* returns an empty workload to threads that are not belonging to the current subteam. These threads are not waiting in the barrier function *ompc_barrier* either.

2.2.3 OpenMP Runtime Library

An OpenMP runtime libary implements all user-level routines defined in the OpenMP language specification such as *omp_set_lock()*, *omp_set_num_threads()* and *omp_get_wtime()*. More over, it provides a layer of abstraction for the underlying thread manipulation (to perform tasks such as thread creation, synchronization, suspension and wakeup) and deal with repetitive tasks (such as internal variable bookkeeping, calculation of chunks for each thread used in different scheduling options). As a result, compiler writers are freed from tedious chores that arise in OpenMP translation. A standalone library also enables library developers to do op-

timizations without delving into details of the complex compiler. All OpenMP runtime libraries are fairly similar in term of the basic functionality, but they are very different in details since the division of work between the compiler and runtime library is highly implementation-dependent.

OpenUH's OpenMP runtime library is based on the one shipped with the ORC-OpenMP compiler [3] and is extended to support OpenMP 2.5 specification [8]. Written in C, it relies on the Pthreads API to manipulate underlying threads as well as to achieve portability. The runtime library initializes a thread pool when it first encounters a parallel region. To minimize the overhead of repetitive thread creation and cancellation, all threads in the pool are reused across the execution of an entire OpenMP program. A Pthread's condition variable is used to suspend and active threads between two consecutive parallel regions.

The runtime system of OpenUH has been extended to support the thread subteam with additional internal variables, runtime calls as well as updated existing runtime calls. In general, a thread team tree is used to represent the default thread team containing all threads (the root node) and the corresponding thread subteams (leaf nodes). The subteam is created based on the parameters of *ONTHREADS* clause. The node of the thread team tree (*omp_team_t*) is passed to worksharing and synchronization runtime calls as one additional parameter to indicate the desired participating thread subteams or the default thread team. Most existing runtime calls have been modified to judge if current thread id belongs to the subteam argument passed in before carrying out actual work. In this way, irrelavant threads will bypass the workload. For example, the loop schedulers only schedule the iteration chunks among the threads belonging to a thread team node; a call to barrier only affects threads in a team node also. A thread that is not belonging to the specified subteam bypasses the worksharing construct.

3. EXPERIMENTS

We have installed and tested the OpenUH compiler on NCSA's Cobalt SGI Altix machine which consists of two SMP systems running the Linux operating system. Each system has 512 Intel Itanium2 1.6 GHz processors, and the two SMPs are connected with InfiniBand. The system will allow us to test the thread subteam feature under a large number of threads and evaluate its impacts. We have performed our first experiment based upon commercial seismic data processing and interpretation software, Kingdom Suite from Seismic Micro-Technology, Inc. Kingdom Suite is an integrated geosciences interpretation software package for Windows Systems used by the energy industry in the search for oil. Due to the confidentiality of the software, we have created a program kernel to simulate the execution.

Figure 3 illustrates the kernel of the seismic data process application. The application repeatedly reads seismic data, processes it, and then writes the result into another file. In order to keep the sequential order of I/O, we use the pipelined execution approach to overlap the I/O operations and computation. A dependence between the seismic data processing functions prevents parallelization of the outer loop (j-loop). The innermost, work-shared loop includes an implicit barrier at its end. Thus although plenty of com-

```
      call readdata(1)
!$omp parallel
      num_threads = omp_get_num_threads()
      do k=1,N1
!$omp master
        call readdata(k+1)
!$omp end master
        do j=1,N2
!$omp do onthreads(2:num_threads-1:1)
          do i=1, N3
            call process(j)
          enddo
!$omp end do
        enddo

!$omp barrier

!$omp single onthreads(1:1:1)
        call writedata(k)
!$omp end single nowait
      end do

!$omp end parallel
```

Figure 3: Seismic data processing code kernel.

putation remains, the computing threads must wait at the implicit barrier until the I/O has completed. Thus I/O operations and computation are not fully overlapped. Unfortunately, exchanging the order of the loops in the nest would, if possible, require a complete rewrite.

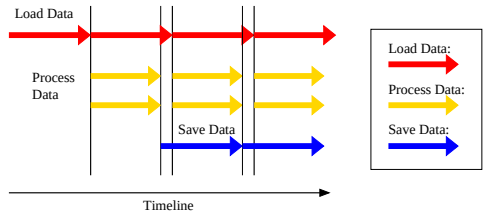

Figure 4: Overlapping I/O with computation in the parallel seismic program

One way to alleviate the penalty of implicit barrier with pure OpenMP is to introduce a nested parallelism in the j loop. Unfortunately, each iteration of the outer k-loop requires a new parallel region and the overheads for these are potentially high. Moreover, there is an implicit barrier at the end of nested parallelism that can not be removed. Another way is to write the SPMD stype OpenMP program, in which each thread executes its work explicitly based on the thread ID. However, it requires a lot more programming efforts. Our *ONTHREADS* clause can solve the problem easily by adding it to the *OMP DO* directives and specifying a subteam of threads to the worksharing construct. Applying the clause on worksharing directives, OpenMP programmers do not need to dramatically modify their sequential applications into the SPMD style by specifying different workloads to different threads explicitly. Using the clause, we achieved a comparable speedup with the SPMD style OpenMP approach in the seismic application, but with much less programming efforts. Fig. 4 illustrates the overlapping I/O and computation during the execution, and Fig. 5 shows the performance improvement after using the *onthread* clause. We are currently working on more benchmarks using a large number of threads to evaluate the subteam usage and performance.

4. RELATED WORK

The NanosCompiler team has proposed the creation of groups of threads in association with parallel regions [5, 4]. Their notation permits the user to specify the number of join-

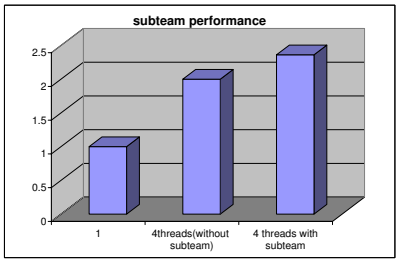

Figure 5: Overlapping I/O with computation in the parallel seismic program

t/disjoint teams of threads that will be created. Since these thread groups are associated with the parallel region, additional notation is required to assign work to the individual groups. They also propose extensions to easily express the precedence relations that originate pipelined computations. These extensions are also valid in the scope of nested parallelism. In contrast, we specify the target of a worksharing directive, i.e., the group of threads that share the work. We refer to threads explicitly. Furthermore, our proposal does not involve nested parallelism and the associated overhead and restrictions.

Zhang [16] has proposed a thread mapping and grouping for OpenMP extensions recently. His proposal mixed the thread mapping with grouping that may complicate the language. And he has not mentioned the formal syntax and semantics of the extension, and implementation either. The SGI compiler for the Origin platforms [14] provides limited support by accepting the SGI NEST clause on the OMP DO directive. The NEST clause requires at least 2 variables as arguments to identify indices of subsequent DO-loops. The identified loops must be perfectly nested.

5. CONCLUSIONS

OpenMP is a shared memory programming API that offers the promise of performance and ease of use. It is currently deployed on both small and large SMPs, including systems with distributed global memory and new platforms with hyperthreading. It seems possible that the judicious addition of language features that increase the power of expressivity might also improve the achievable performance of a variety of OpenMP codes. In this paper, we introduce a unified notation for sharing work among subteams of threads and evaluate its implementation in the OpenUH compiler. This ongoing work could be extended in a variety of ways. Additional notation could be defined to name groups of threads, to more easily specify multiple different orderings, to handle synchronization between thread subteams, and perhaps to help a compiler and run time system find an appropriate data layout. We will explore more on the impacts of the notation and investigate the integration with nested parallelism, thread and processors mapping, machine abstract representation, and more.

6. REFERENCES

[1] C. Brunschen and M. Brorsson. OdinMP/CCp - a portable implementation of OpenMP for C. *Concurrency - Practice and Experience*, 12(12):1193–1203, 2000.

[2] B. Chapman, L. Huang, H. Jin, G. Jost, and B. R. de Supinski. Extending openmp worksharing directives for multithreading. In *Proceedings of Europar 2006*, August, 2006.

[3] Y. Chen, J. Li, S. Wang, and D. Wang. ORC-OpenMP: An OpenMP compiler based on ORC. In *International Conference on Computational Science*, pages 414–423, 2004.

[4] M. Gonzalez, E. Ayguade, X. Martorell, and J. Labarta. Defining and supporting pipelined executions in openmp. In *Proceedings of WOMPAT 2001*, July, 2001.

[5] M. Gonzalez, J. Oliver, X. Martorell, E. Ayguade, J. Labarta, and N. Navarro. Openmp extensions for thread groups and their run-time support. In *LCPC '00: Proceedings of the 13th International Workshop on Languages and Compilers for Parallel Computing-Revised Papers*, pages 324–338, London, UK, 2001. Springer-Verlag.

[6] R. Kalla, B. Sinharoy, and J. Tendler. Ibm power5 chip: a dualcore multithreaded processor. *IEEE Micro*, 24(2):40–47, 2004.

[7] P. Kongetira, K. Aingaran, and K. Olukotun. Niagara: A 32-way multithreaded Sparc processor. *IEEE-MICRO*, 25(2):21–29, March/April 2005.

[8] C. Liao, O. Hernandez, B. Chapman, W. Chen, and W. Zheng. OpenUH: An optimizing, portable OpenMP compiler. In *12th Workshop on Compilers for Parallel Computers*, 2006.

[9] K. Olukotun. The case for a single-chip multiprocessor. In *Intl. Conf. on Architec-tural Support for Programming Languages and Operating Systems*, pages 2–11, 1996.

[10] The Open64 compiler. `http://open64.sourceforge.net`, 2005.

[11] OpenMP: Simple, portable, scalable SMP programming. `http://www.openmp.org`, 2006.

[12] The OpenUH compiler project. `http://www.cs.uh.edu/~openuh`, 2005.

[13] M. Sato, S. Satoh, K. Kusano, and Y. Tanaka. Design of OpenMP compiler for an SMP cluster. In *the 1st European Workshop on OpenMP(EWOMP'99)*, pages 32–39, Sept. 1999.

[14] Silicon Graphics Inc. *MIPSpro 7 FORTRAN 90 Commands and Directives Reference Manual*, 2002.

[15] D. M. Tullsen, S. Eggers, and H. M. Levy. Simultaneous multithreading: Maximizing on-chip parallelism. In *Proceedings of the 22th Annual International Symposium on Computer Architecture*, pages 392–403, 1995.

[16] G. Zhang. Extending the openmp standard for thread mapping and grouping . In *Proceedings of IWOMP 2006*, June, 2006.