Alistair P. Rendell
Barbara M. Chapman
Matthias S. Müller (Eds.)

OpenMP in the Era of Low Power Devices and Accelerators

LNCS 8122

9th International Workshop on OpenMP, IWOMP 2013
Canberra, ACT, Australia, September 2013
Proceedings

Springer

Lecture Notes in Computer Science 8122

Commenced Publication in 1973
Founding and Former Series Editors:
Gerhard Goos, Juris Hartmanis, and Jan van Leeuwen

Alistair P. Rendell Barbara M. Chapman
Matthias S. Müller (Eds.)

OpenMP in the Era of Low Power Devices and Accelerators

9th International Workshop on OpenMP, IWOMP 2013
Canberra, ACT, Australia, September 16-18, 2013
Proceedings

Volume Editors

Alistair P. Rendell
Australian National University
Research School of Computer Science
Bldg 108, North Road, 0200 Canberra, ACT, Australia
E-mail: alistair.rendell@anu.edu.au

Barbara M. Chapman
University of Houston
Department of Computer Science
4800 Calhoun Road, Houston, TX 77204, USA
E-mail: chapman@cs.uh.edu

Matthias S. Müller
RWTH Aachen University
Lehrstuhl für Hochleistungsrechnen und Rechen- und Kommunikationszentrum
Seffenter Weg 23, 52074 Aachen, Germany
E-mail: mueller@rz.rwth-aachen.de

ISSN 0302-9743 e-ISSN 1611-3349
ISBN 978-3-642-40697-3 e-ISBN 978-3-642-40698-0
DOI 10.1007/978-3-642-40698-0
Springer Heidelberg New York Dordrecht London

Library of Congress Control Number: 2013946472

CR Subject Classification (1998): C.1, D.1, F.2, D.4, C.3, C.4

LNCS Sublibrary: SL 2 – Programming and Software Engineering

Typesetting: Camera-ready by author, data conversion by Scientific Publishing Services, Chennai, India

Printed on acid-free paper

Springer is part of Springer Science+Business Media (www.springer.com)

Preface

OpenMP is a widely accepted, standard application programming interface (API) for high-level shared-memory parallel programming in Fortran, C, and C++. Since its introduction in 1997, OpenMP has gained support from most high-performance compiler and hardware vendors. Under the direction of the OpenMP Architecture Review Board (ARB), the OpenMP specification has evolved up to the upcoming release of version 4.0. This version will include several new features like accelerator support for heterogeneous hardware environments, an enhanced tasking model, user-defined reductions and thread affinity to support binding for performance improvements on non-uniform memory architectures.

The evolution of the standard would be impossible without active research in OpenMP compilers, runtime systems, tools, and environments. OpenMP is both an important programming model for single multicore processors and as part of a hybrid programming model for massively parallel, distributed memory systems built from multicore or manycore processors. In fact, most of the growth in parallelism of the upcoming Exascale systems is expected to be coming from an increased parallelism within a node. OpenMP offers important features that can improve the scalability of applications on such systems.

The community of OpenMP researchers and developers in academia and industry is united under cOMPunity (www.compunity.org). This organization has held workshops on OpenMP around the world since 1999: the European Workshop on OpenMP (EWOMP), the North American Workshop on OpenMP Applications and Tools (WOMPAT), and the Asian Workshop on OpenMP Experiences and Implementation (WOMPEI) attracted annual audiences from academia and industry. The International Workshop on OpenMP (IWOMP) consolidated these three workshop series into a single annual international event that rotates across Asia, Europe, and America. The first IWOMP workshop was organized under the auspices of cOMPunity. Since that workshop, the IWOMP Steering Committee has organized these events and guided the development of the series. The first IWOMP meeting was held in 2005, in Eugene, Oregon, USA. Since then, meetings have been held each year, in Reims, France, Beijing, China, West Lafayette, USA, Dresden, Germany, Tsukuba, Japan, Chicago, USA, and Rome, Italy. Each workshop has drawn participants from research and industry throughout the world. IWOMP 2013 continued the series with technical papers, tutorials, and OpenMP status reports. The IWOMP meetings have been successful in large part due to the generous support from numerous sponsors.

The cOMPunity website (www.compunity.org) provides access to the talks given at the meetings and to photos of the activities. The IWOMP website

(www.iwomp.org) provides information on the latest event. This book contains proceedings of IWOMP 2013. The workshop program included 14 technical papers, 2 keynote talks, a tutorial on OpenMP and a report of the OpenMP Language Committee about the latest developments of OpenMP. All technical papers were peer reviewed by at least three different members of the Program Committee.

September 2013

Alistair P. Rendell
Barbara M. Chapman
Matthias S. Müller

Organization

Program and Organizing Chair

Alistair Rendell	Australian National University, Australia

Program Co-chairs

Matthias Müller	RWTH Aachen University, Germany
Barbara Chapman	University of Houston, USA

Tutorials Chair

Ruud van der Pas	Oracle America, USA

Local Organizing Committee

Eric McCreath	Australian National University, Australia
Josh Milthorpe	Australian National University, Australia
Alistair Rendell	Australian National University, Australia

Program Committee

Dieter an Mey	RWTH Aachen University, Germany
Eduard Ayguadé	Barcelona Supercomputing Center, Spain
Mark Bull	EPCC, UK
Nawal Copty	Oracle America, USA
Rudi Eigenmann	Purdue University, USA
Larry Meadows	Intel, USA
Alistair Rendell	Australian National University, Australia
Bronis R. de Supinski	NNSA ASC, LLNL, USA
Mitsuhisa Sato	University of Tsukuba, Japan
Christian Terboven	RWTH Aachen University, Germany
Ruud van der Pas	Oracle America, USA
Michael Wong	IBM, Canada

Steering Committee Chair

Matthias S. Müller	RWTH Aachen University, Germany

Steering Committee

Dieter an Mey	RWTH Aachen University, Germany
Eduard Ayguadé	BSC/UPC, Spain
Mark Bull	EPCC, UK
Barbara M. Chapman	University of Houston, USA
Rudolf Eigenmann	Purdue University, USA
Guang R. Gao	University of Delaware, USA
William Gropp	University of Illinois, USA
Ricky Kendall	Oak Ridge National Laboratory, USA
Michael Krajecki	University of Reims, France
Rick Kufrin	NCSA/Univerity of Illinois, USA
Kalyan Kumaran	Argonne National Laboratory, USA
Federico Massaioli	CASPUR, Italy
Larry Meadows	Intel, USA
Arnaud Renard	University of Reims, France
Mitsuhisa Sato	University of Tsukuba, Japan
Sanjiv Shah	Intel, USA
Bronis R. de Supinski	NNSA ASC, LLNL, USA
Ruud van der Pas	Oracle America, USA
Matthijs van Waveren	Fujitsu, France
Michael Wong	IBM, Canada
Weimin Zheng	Tsinghua University, China

Table of Contents

Proposed Extensions to OpenMP

Applications

Accelerators

Scheduling

Tools

A Proposal for Task-Generating Loops in OpenMP*

Xavier Teruel[1], Michael Klemm[2], Kelvin Li[3], Xavier Martorell[1], Stephen L. Olivier[4], and Christian Terboven[5]

[1] Barcelona Supercomputing Center
[2] Intel Corporation
[3] IBM Corporation
[4] Sandia National Laboratories
[5] RWTH Aachen University
{xavier.teruel,xavier.martorell}@bsc.es, michael.klemm@intel.com, kli@ca.ibm.com, slolivi@sandia.gov, terboven@rz.rwth-aachen.de

Abstract. With the addition of the OpenMP* tasking model, programmers are able to improve and extend the parallelization opportunities of their codes. Programmers can also distribute the creation of tasks using a worksharing construct, which allows the generation of work to be parallelized. However, while it is possible to create tasks inside worksharing constructs, it is not possible to distribute work when not all threads reach the same worksharing construct. We propose a new worksharing-like construct that removes this restriction: the `taskloop` construct. With this new construct, we can distribute work when executing in the context of an explicit task, a `single`, or a `master` construct, enabling us to explore new parallelization opportunities in our applications. Although we focus our current work on evaluating expressiveness rather than performance evaluation, we present some initial performance results using a naive implementation for the new `taskloop` construct based on a lazy task instantiation mechanism.

Keywords: OpenMP, Task, Worksharing, Loop, Fork/Join.

1 Introduction

The proliferation of multi-core and many-core architectures necessitates widespread use of shared memory parallel programming. The OpenMP* Application Program Interface [9], with its cross-vendor portability and straightforward directive-based approach, offers a convenient means to exploit these architectures for application performance. Though originally designed to standardize the expression of loop-based parallelism, the addition of support for *explicit tasks* in OpenMP has enabled the expression of divide-and-conquer algorithms and applications with irregular parallelism [1]. At the same time, OpenMP worksharing has been recast in the specification to use the task model. A `parallel` region is said to create a set of *implicit tasks* equal to the number of threads in the team.

A.P. Rendell et al. (Eds.): IWOMP 2013, LNCS 8122, pp. 1–14, 2013.

Each implicit task is tied to a different thread in the team, and iterations of a worksharing loop are executed in the context of these implicit tasks.

However, no interaction has been defined between explicit tasks and worksharing loops. This leads to an asymmetry since the implicit tasks of a worksharing construct can create explicit tasks, while explicit tasks may not encounter a worksharing construct. Hence it becomes cumbersome for programmers to compose source code and libraries into a single application that uses a mixture of OpenMP tasks and worksharing constructs.

We aim to relieve this burden by defining a new type of worksharing construct that generates (explicit) OpenMP tasks to execute a parallel loop. The new construct is designed to be placed virtually anywhere that OpenMP accepts creation of tasks, making the new construct fully composable. The generated tasks are then executed through the existing tasks queues, enabling transparent load balancing and work stealing [3] in the OpenMP runtime system.

The remainder of the paper is organized as follows. Section 2 discusses the rationale and the design principles of the new task-generating worksharing construct. In Section 3, we describe the syntax and semantics of the new construct. We evaluate the performance of the new construct in Section 4. Section 5 presents related work, and Section 6 concludes the paper and outlines future work.

2 Rationale and Design Considerations

OpenMP currently offers loop-based worksharing constructs (`#pragma omp for` for C/C++ and `!$omp do` for Fortran) only to distribute the work of a loop across the worker threads of the current team. When OpenMP 3.0 introduced the notion of task-based programming, the effect of the `parallel` construct was recast to generate so-called implicit tasks that are assigned to run on the threads of the current team. Hence, the existing worksharing constructs now assign loop iterations to these implicit tasks. While this generalizes OpenMP semantics and also simplifies the implementation of an OpenMP compiler and runtime, it still maintains the traditional semantics and restrictions of the worksharing constructs [9].

A worksharing region cannot be closely nested inside another worksharing region. This becomes an issue when not all source code is under control of the programmer, e.g., if the application code calls into a library. Today, the only solution is to employ nested parallelism to create a new team of threads for the nested worksharing construct. However, this approach potentially limits parallelism on the outer levels, while the inner `parallel` regions cannot dynamically balance the load on their level. It also leads to increased synchronization overhead due to the inner barrier. Furthermore, current OpenMP implementations cannot maintain a consistent mapping of OpenMP threads to native threads when nested parallel regions occur, which may lead to bad performance, particularly on systems with a hierarchical memory and cache architecture.

The threading and tasking model in OpenMP is not symmetric for worksharing constructs and tasks. All OpenMP worksharing constructs can arbitrarily

create tasks from their regions. However, the reverse is not permitted: worksharing constructs may not be encountered from the dynamic extent of a task.

OpenMP tasks on the other hand provide an elegant way to describe many different algorithms. Tasks are not restricted to regular algorithms and may be used to describe (almost) arbitrarily irregular algorithms. Unfortunately, OpenMP does not offer an easy-to-use construct to express parallel loops with tasks. Programming languages like Intel® Cilk™ Plus or libraries such as Intel® Threading Building Blocks define keywords or C++ templates to generate tasks from a parallel loop. This is a very convenient approach to express loop parallelism on top of a task-based parallel programming model. In OpenMP this is not possible with the current specification of worksharing constructs.

Today, programmers are forced to use a work-around. Listing 1.1 shows the manual task implementation of a simple loop. The traditional worksharing construct is in the function `daxpy_parallel_for`. The parallel loop with manual tasking is rather cumbersome, since it involves a lot of boilerplate code. In `daxpy_parallel_explicit_tasks`, a `for` loop runs over the individual chunks of the iteration space to create explicit tasks. Programmers are responsible for computing a chunk's lower (`lb`) and upper bound (`ub`). The typical `parallel-single` pattern is used to only have one producer create tasks. The code of function `daxpy_parallel_taskloop` shows how the syntax of the proposed `taskloop` construct eases the implementation and makes the code more concise.

A task-based loop construct for OpenMP can solve these issues with existing worksharing constructs and increase expressiveness. Since tasks in OpenMP are (by definition) nestable, an OpenMP loop construct that generates tasks from a loop is also nestable. Load balancing is performed automatically as tasks are scheduled onto the threads by the runtime system, often by work stealing [3]. Mixing regular tasks and task-generating loop constructs is also possible. The generated tasks of a loop are inserted into the task queue; threads eventually schedule the loop tasks and execute them intermixed with all other tasks created.

3 The Task-Generating Loop Construct

This section describes the syntax and semantics of the proposed `taskloop` construct. We use the same syntax description format as the OpenMP specification.

3.1 Syntax

The syntax of the `taskloop` construct is syntactically similar to the existing worksharing constructs:

`#pragma omp taskloop` *[clause[[,] clause] ...]*
for-loops

where *clause* is one of the following:

- `if(` *scalar-expression* `)`
- `shared(` *list* `)`

```
1  void daxpy_parallel_for(float* x, float* y, float a, int length) {
2  #pragma omp parallel for shared(x,y) firstprivate(a,length)
3    for (int i = 0; i < length; i++) x[i] = a * y[i];
4  }
5
6  void daxpy_parallel_explicit_tasks(float* x, float* y, float a, int length) {
7  #pragma omp parallel shared(x,y) firstprivate(a,length)
8    {
9  #pragma omp single
10     {
11       int lb = 0; // initial loop start
12       for (lb = 0; lb < length; lb += chunksz) {
13         int ub = min(lb +  chunksz, length);
14 #pragma omp task firstprivate(lb,ub)
15         {
16           for (int i = lb; i < ub; i++) x[i] = a * y[i];
17       } }
18 #pragma omp taskwait
19 } } }
20
21 void daxpy_parallel_taskloop(float* x, float* y, float a, int length) {
22 #pragma omp parallel taskloop shared(x,y) firstprivate(a,length)
23   for (int i = 0; i < length; i++) x[i] = a * y[i];
24 }
```

Listing 1.1. Implementation overhead of explicit tasking for a parallel `for` loop

- `private(` *list* `)`
- `firstprivate(` *list* `)`
- `lastprivate(` *list* `)`
- `partition(` *kind[, chunk_size]* `)`
- `collapse(` *n* `)`
- `taskgroup`
- `nowait`

In line with the existing worksharing constructs and for completeness, we also define a combined version of the `parallel` and the `taskloop` construct. The `parallel taskloop` construct is a shortcut for specifying a `parallel` construct containing one `taskloop` construct with its associated loops and no other statements.

`#pragma omp parallel taskloop` *[clause[[,] clause] ...]*
for-loops

The Fortran syntax is similar to C/C++ and the clauses are the same as C/C++:

`!$omp taskloop` *[clause[[,] clause] ...]*
do-loops
[`!$omp end taskloop` *[*`nowait`|`taskgroup`*]]*

`!$omp parallel taskloop` *[clause[[,] clause] ...]*
do-loops
[`!$omp end parallel taskloop` *[*`nowait`|`taskgroup`*]]*

3.2 Semantics

All loops that are supported by the traditional worksharing constructs are also supported by `taskloop`. The `taskloop` construct requires the same restrictions on the `for` and `do` loops as the existing worksharing constructs. Although very similar in syntax to the `do`/`for` worksharing construct, the proposed `taskloop` construct does not follow the definition of an OpenMP worksharing construct, as instead of defining units of work to be executed by threads, it generates tasks. Hence, restrictions on worksharing constructs, such as the requirement to be encountered by all threads of a team, do not apply, nor could the existing `do`/`for` construct be extended to provide this functionality.

When an `if` clause is present on a `taskloop` construct and its expression evaluates to false, the encountering thread must suspend the current task region until the whole loop iteration space is completed.

The `collapse` clause has its well-known OpenMP semantics specifying the number of nested loops that are associated with the `taskloop` construct. The parameter of the `collapse` clause indicates the number of loops which must collapse into a single iteration space.

The data-sharing attributes are slightly reinterpreted for the `taskloop` to fit the notion of task creation. In today's OpenMP semantics, data-sharing clauses are defined in terms of implicit and explicit tasks (and SIMD lanes in OpenMP 4.0 RC2 [10]). For the `taskloop` construct, the `shared` clause declares a list of variables to be shared across the tasks created by the `taskloop` construct. Variables marked as `private`, `firstprivate`, or `lastprivate` are private to the created tasks. The loop index variable is automatically made private.

The `partition` clause defines the way the iteration space is split to generate the tasks from the loop iterations:

- `partition`(*linear*) is the analog to the dynamic schedule of existing loop constructs, i.e., the iteration space is split into (almost) equally sized chunks of the given chunk size.
- `partition`(*binary*) uses a binary splitting approach in which the iteration space is recursively split into chunks. Each chunk is assigned to a new task that continues binary splitting until a minimal chunk size is reached.
- `partition`(*guided*) is the analog of the guided schedule of existing loop constructs, i.e., tasks are generated with continually decreasing amounts of work.

The *chunk_ size* parameter defines the chunk size of the generated tasks:

- If `partition`(*linear*) is specified, then the value determines the exact size of each chunk, as it does in the dynamic schedule of existing loop constructs.
- If `partition`(*binary*) or `partition`(*guided*) is specified, then the value determines the minimal size of a chunk, as it does in the guided schedule of existing loop constructs.
- The default chunk size is 1 (if the *chunk_ size* parameter is not present).

When an implicit or explicit task encounters a `taskloop` construct, it precomputes the iteration count and then starts creating tasks according to the split

policy specified in the `partition` clause. For the `linear` case, the encountering tasks computes the work distribution and creates the tasks to execute based on the distribution computed. For the `binary` case, the encountering task cuts the iteration space into two partitions and creates a child task for each of the partitions. This continues recursively until the threshold is reached and the tasks start to execute loop chunks. The `guided` policy forces the encountering task to create a series of loop tasks of decreasing size.

The default synchronization at the end of the `taskloop` region is an implicit `taskwait` synchronization. Thus, only tasks generated directly by the `taskloop` construct must have been completed at the end of the `taskloop` region. The `taskgroup` clause instead establishes a task group for all the tasks that are generated from the task-generating loop construct and enforces an implicit `taskgroup` synchronization at the end of the `taskloop` region.[1] A `taskgroup` synchronization requires completion of all tasks: not only those tasks generated directly by the `taskloop` construct, but also all descendants of those tasks. The `nowait` clause removes the implicit `taskwait` synchronization at the end of the tasking loop construct. Only one of the `nowait` or `taskgroup` clauses may be specified.

4 Evaluation

In this section, we discuss some parallelization patterns that benefit from the new construct. The main goal for this new construct is to increase the expressiveness of OpenMP, but we present some performance results that demonstrate that increasing such expressiveness can sometimes also improve performance.

4.1 Parallelization Approach

The first benchmark is Cholesky factorization. Cholesky decomposition is a common linear algebra method which is also used to solve systems of linear equations. Our implementation is based on the LAPACK library version and uses four different kernels: *potrf*, *trsm*, *gemm* and *syrk* (Listing 1.2).

A possible parallelization of the algorithm creates a different task for each kernel. In this parallelization we already use task dependences, set to be included in OpenMP 4.0 [10], in order to solve some imbalance problems when using traditional worksharing constructs [6]. Listing 1.2 includes this baseline parallelization.

Holding constant the number of tasks generated (which usually is related to the problem size) while changing the number of threads (which is related to available resources) may greatly impact performance. Some applications can benefit from an extra level of parallelism to alleviate load imbalance. In order to include this extra level of parallelism, we can create a task for each loop iteration (including loop body) but the granularity issue remains: we still have a constant number of tasks and constant task granularity.

[1] Taskgroups are not part of OpenMP 3.1, but have been added to the draft specification of OpenMP 4.0 RC2 [10].

```
void omp_gemm(double *A, double *B, double *C, int ts, int bs) {
  int i, j, k;
  static const char TR = 'T', NT = 'N';
  static double DONE = 1.0, DMONE = -1.0;

  for(k=0; k<ts ;k+=bs)
    for(i=0; i<ts;i+=bs)
      for(j=0; j<ts; j+=bs)
        dgemm_(&NT, &TR, &bs, &bs, &bs, &DMONE, A[k*ts+i],
          &ts, B[k*ts+j], &ts, &DONE, C[j*ts+i], &ts);
}

#pragma omp parallel
#pragma omp single
for (int k = 0; k < nt; k++) {
#pragma omp task depend(inout:Ah[k][k])
  omp_potrf (Ah[k][k], ts);
  for (int i = k + 1; i < nt; i++) {
#pragma omp task depend(in:Ah[k][k],inout:Ah[k][i])
    omp_trsm (Ah[k][k], Ah[k][i], ts);
  }
  for (int i = k + 1; i < nt; i++) {
    for (int j = k + 1; j < i; j++) {
#pragma omp task depend(in:Ah[k][i],Ah[k][j],inout:Ah[j][i])
      omp_gemm (Ah[k][i], Ah[k][j], Ah[j][i], ts, ts/BLOCK_SIZE);
    }
#pragma omp task depend(in:Ah[k][i], inout:C[i][i])
    omp_syrk (Ah[k][i], Ah[i][i], ts);
  }
}
```

Listing 1.2. Cholesky's baseline parallelization code

Following with that solution we can handle the inner loop chunk size and transform this inner loop into a task. If we use the available number of threads as part of the chunk-size computation, we can effectively manage the trade-off between task number and task granularity according to the availability of resources. This approach is demonstrated in function `omp_gemm_tasks()` in Listing 1.3.

The same result can be achieved using a guided policy with the new loop construct (see function `omp_gemm_loop()`, the second function in Listing 1.3). This last solution does not require adding extra code into the user's program, allowing equivalent behavior by including just a single OpenMP pragma directive.

Our second benchmark is the Conjugate Gradient (CG) iterative kernel. The conjugate gradient method is a numerical algorithm to solve systems of linear equations and is commonly used in optimization problems. It is implemented as an iterative method, providing monotonically improving approximations to the exact solution (i.e., the method converges iteration after iteration to the real solution). The algorithm completes after it reaches the required tolerance or after executing some maximum number of iterations. The tolerance and maximum iteration count are fixed as input parameters.

Initial parallelization of this code comprises a sequence of `parallel` regions and a worksharing construct which computes each of the component kernels used in the algorithm. In Listing 1.4, we show only the `matvec` function, the most important kernel in the CG benchmark, but other kernels follow the same

```
1  void omp_gemm_tasks(double *A, double *B, double *C, int ts, int bs) {
2    ...
3    for(k=0; k<ts ;k+=bs) {
4      for(i=0; i<ts;i+=bs) {
5        lower = 0;
6        nthreads = omp_get_num_threads();
7        while ( lower < ts ) {
8          upper = compute_upper( lower, nthreads, bs, ts );
9  #pragma omp task firstprivate(x,k,i,ts,bs) nowait
10         for(j = lower; j < upper; j+=bs)
11           dgemm_(&NT, &TR, &bs, &bs, &bs, &DMONE, A[k*ts+i],
12             &ts, B[k*ts+j], &ts, &DONE, C[j*ts+i], &ts);
13         lower = upper;
14       }
15     }
16 #pragma omp taskwait
17   }
18 }
19
20 void omp_gemm_loop(double *A, double *B, double *C, int ts, int bs) {
21   ...
22   for(k=0; k<ts ;k+=bs) {
23     for(i=0; i<ts;i+=bs) {
24 #pragma omp taskloop partition(guided,1) firstprivate(k,i,ts,bs) nowait
25       for(j=0; j<ts; j+=bs)
26         dgemm_(&NT, &TR, &bs, &bs, &bs, &DMONE, A[k*ts+i],
27           &ts, B[k*ts+j], &ts, &DONE, C[j*ts+i], &ts);
28     }
29 #pragma omp taskwait
30   }
31 }
```

Listing 1.3. Parallelization approaches of the GEMM code

pattern. This approach incurs the overhead costs of creating a `parallel` region to execute each kernel.

Using our proposed loop construct, we only need to create one team using the OpenMP `parallel` construct before starting the iterative computation. We also enclose the `parallel` region (i.e., the user's code associated with the `parallel` construct) with an OpenMP `single` directive. This approach is shown in Listing 1.5. A team of threads is created, but only one thread executes the code inside the `parallel` region due to the closely nested `single` directive. We similarly modify all the other kernels, replacing the existing loop construct with the new loop construct. Although the code still includes a worksharing construct, we eliminate the overhead costs of opening and closing successive `parallel` regions.

4.2 Performance Results

We evaluate all our benchmarks on the MareNostrum III supercomputer, located at the Barcelona Supercomputing Center and on Gothmog, a machine at the Royal Institute of Technology in Stockholm. Due the nature of our evaluation all benchmarks are executed on a single node.

Each MareNostrum node is a 16-core node with two Intel® Xeon® processors E5-2670 (former codename "Sandybridge"), running at 2.6 GHz (turbo mode at 3.3 GHz) and with 20 MB L3 cache. Each node has 32 GB of main memory, which is organized as two NUMA nodes. Gothmog is a 48-core machine with four

```
1  void matvec(Matix *A, double *x, double *y)
2  {
3    ...
4  #pragma omp parallel for private(i,j,is,ie,j0,y0) schedule(static)
5    for (i = 0; i < A->n; i++) {
6      y0 = 0;
7      is = A->ptr[i];
8      ie = A->ptr[i + 1];
9      for (j = is; j < ie; j++) {
10       j0 = index[j];
11       y0 += value[j] * x[j0];
12     }
13     y[i] = y0;
14   }
15   ...
16 }
17
18 for (iter = 0; iter < sc->maxIter; iter++) {
19   precon(A, r, z);
20   vectorDot(r, z, n, &rho);
21   beta = rho / rho_old;
22   xpay(z, beta, n, p);
23   matvec(A, p, q);
24   vectorDot(p, q, n, &dot_pq);
25   alpha = rho / dot_pq;
26   axpy(alpha, p, n, x);
27   axpy(-alpha, q, n, r);
28   sc->residual = sqrt(rho) * bnrm2;
29   if (sc->residual <= sc->tolerance) break;
30   rho_old = rho;
31 }
```

Listing 1.4. CG baseline implementation

12-core AMD Opteron* 6172 processors (codename "Magny-Cours"), running at 2.1 GHz, with 6 MB L2 and 12 MB L3 caches. The machine has 64 GB of main memory organized as eight NUMA nodes. We use the Nanos++ runtime library[2] and Mercurium compiler[3] [2].

In the next subsections, we detail the results obtained for Cholesky and CG.

Cholesky. Figure 4.2 summarizes the results obtained by executing Cholesky on MareNostrum III and Gothmog. We executed three different versions of Cholesky: The first version, labeled *1-level tasks*, uses a single level of parallelism. The second version, labeled *nested*, includes an additional level of nested parallelism with tasks. In the final version, labeled *taskloops*, the nested parallelism is implemented using the `taskloop` construct with guided partitioning.

The MareNostrum results show that adding a new level of parallelism improves the performance when we reach a given number of threads, in this case 16. These nested versions (*nested* and *taskloops*) have a similar behavior on MareNostrum, though we expected some improvement due to `guided` scheduling reducing the overhead of task creation. On Gothmog, the *taskloops* version has better

[2] Based on git repository (`http://pm.bsc.es/git/nanox.git`) revision nanox 0.7a (git master 37f3a0d 2013-02-26 15:14:11 +0100 developer version).

[3] Based on git repository (`http://pm.bsc.es/git/mcxx.git`) revision mcxx 1.99.0 (git b51a11c 2013-04-10 09:27:34 +0200 developer version).

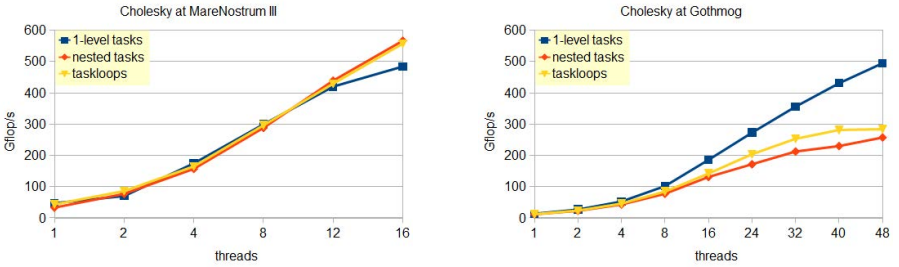
```
1  void matvec(Matix *A, double *x, double *y) {
2    ...
3  #pragma omp taskloop private(i,j,is,ie,j0,y0) partition(linear)
4    for (i = 0; i < A->n; i++) {
5      y0 = 0;
6      is = A->ptr[i];
7      ie = A->ptr[i + 1];
8      for (j = is; j < ie; j++) {
9        j0 = index[j];
10       y0 += value[j] * x[j0];
11     }
12     y[i] = y0;
13   }
14   ...
15 }
16
17 #pragma parallel
18 #pragma single
19 for (iter = 0; iter < sc->maxIter; iter++) {
20   precon(A, r, z);
21   vectorDot(r, z, n, &rho);
22   beta = rho / rho_old;
23   xpay(z, beta, n, p);
24   matvec(A, p, q);
25   vectorDot(p, q, n, &dot_pq);
26   alpha = rho / dot_pq;
27   axpy(alpha, p, n, x);
28   axpy(-alpha, q, n, r);
29   sc->residual = sqrt(rho) * bnrm2;
30   if (sc->residual <= sc->tolerance) break;
31   rho_old = rho;
32 }
```

Listing 1.5. CG implementation based on the `taskloop` construct

performance than *nested* beyond 16 threads. This improvement is a result of decreased overhead of spawning work using taskloops compared to creating independent tasks. In our implementation, a taskloop is represented by a single task descriptor structure that is enqueued using a single enqueue operation rather than using a number of individual task enqueues.

On Gothmog, using the nested versions does not improve the performance compared to the *1-level tasks* version. One explanation of that performance degradation is that although nesting reduces application imbalance (i.e., the ratio between the number of tasks and threads), it degrades data locality. The first level of parallelism distributes large matrix blocks among cores. If we apply a second level of parallelism, we break large matrix blocks into smaller ones that are then spread among all cores, potentially breaking data locality in the NUMA nodes. Since a discussion about NUMA nodes, data locality and nested parallelism is not the main goal of this study, we leave further analysis of this issue to future work.

CG. In order to benchmark the CG kernel we use two matrices of different size. Figure 2(a) shows the results for a small matrix. In an experiment with work of such small granularities, the OpenMP fork/join overhead is noticeable, and the taskloops implementation performs better.



(a) MareNostrum III (b) Gothmog

Fig. 1. Cholesky performance results

Figure 2(b) shows the shape of the larger matrix problem. This is the Fluorem/RM07R Matrix, used in computational fluid dynamics problems. This matrix is publicly available at *The University of Florida Sparse Matrix Collection*[4] web site. Figure 2(c) shows that for this larger problem, in a 16-core node, the behavior of both approaches is almost the same. In this case, the larger data set makes the fork/join overhead comparatively smaller. In larger nodes with higher NUMA memory distances, (e.g., on Gothmog, see Figure 2(d)) the loss of data locality caused by the taskloops approach substantially degrades its performance. Again, this will be a focus of our future work.

5 Related Work

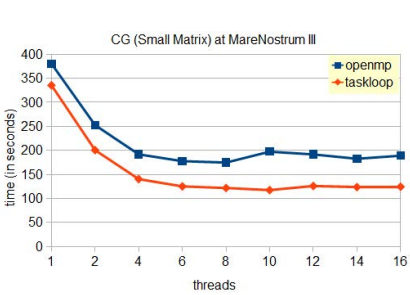
The idea of generating tasks to execute a parallel loop is not new and has been implemented in various other parallel programming languages and libraries.

Intel® Cilk Plus and its predecessor Cilk++ [7] implement their `cilk_for` construct by recursively splitting the iteration space down to a minimum chunk size, generating tasks using `cilk_spawn` at each level of recursion.[5] A similar approach is taken by the `parallel_for` template of the Intel® Threading Building Blocks [11]. The loop is parallelized by splitting the iteration space recursively until the task granularity reaches a threshold. As part of the .NET 4.5 framework, the Task Parallel Library [8] offers task-parallel execution of `for` and `foreach` loops (through `Parallel.For` and `Parallel.ForEach`). All of these approaches only support C and/or C++ and are not applicable for Fortran; they also do not blend well with OpenMP.

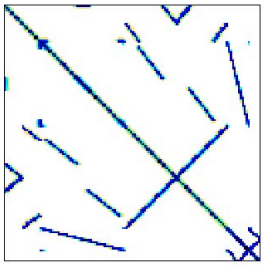
Ferrer et al. [5] show that with minimal compiler assistance, `for` loops containing task parallelism can be successfully unrolled and aggregated for better performance. Ferrer [4] proposes an extension of the `while` loop that generates chunked parallel tasks. Employing this approach "by hand", Terboven et al. [12] found multi-level parallelism with tasks to be more profitable than nested OpenMP for several applications. The proposed construct is a short-cut for the

[4] `http://www.cise.ufl.edu/research/sparse/matrices`

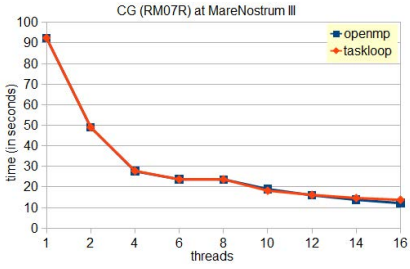
[5] The Cilk Plus run time system is now open source, available at `http://cilkplus.org`



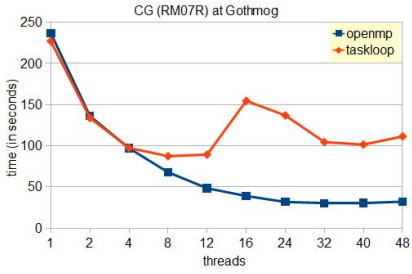
(a) MareNostrum III



(b) RM07R Matrix



(c) MareNostrum III



(d) Gothmog

Fig. 2. CG performance results

programmer to avoid extensive code patterns and to give the compiler more information about the code structure and the intent of the program.

6 Conclusions and Future Work

In this paper, we have introduced the task-generating loop construct, which avoids the limitation on the number of threads reaching a worksharing construct, increasing opportunities for the expression of parallelism. We have also demonstrated the new construct in two different situations. In the first scenario, we exploit a new level of parallelism by using the worksharing construct inside an explicitly created task. Such mechanisms mitigate the imbalance that results from increasing the number of threads and adapt the task granularity to the number of threads. In the second scenario, we use the new construct to avoid creating and closing successive `parallel` regions. With the new approach we create just one `parallel` region with a nested `single` construct, creating the team of threads but allowing only a single thread to execute the enclosed code. To generate the tasks in each kernel, we replace all the inner parallel worksharing constructs with the new task-generating loop construct.

We evaluate both scenarios against baseline implementations of the applications using the Nanos++ run time library and Mercurium compiler infrastructure. The results demonstrate that in addition to improving expressiveness, the new construct improves performance for some, but not all, applications.

As future work we plan to further evaluate our `taskloop` proposal implementation with other benchmarks and on other platforms. We especially seek to explore further the behavior of `taskloop` in the context of NUMA, and analyze more advanced implementation techniques to exploit data locality. We also plan to explore the possibility of nested `taskloop` regions and how these techniques can impact the performance and application load imbalance. Another future topic is the extension of `taskloop` to also support irregular loops such as `while` loops and `for` loops that do not adhere to the restrictions of the current OpenMP worksharing constructs.

Acknowledgments. We would like to acknowledge the support received from the European Comission through the DEEP project (FP7-ICT-287530), and the HiPEAC-3 Network of Excellence (ICT FP7 NoE 287759), from the Spanish Ministry of Education (under contracts TIN2012-34557, TIN2007-60625, CSD2007-00050), and the Generalitat Catalunya (contract 2009-SGR-980).

We thankfully acknowledge the Royal Institute of Technology in Stockholm and the Barcelona Supercomputing Center for the use of their machines (Gothmog, and Marenostrum III).

Sandia is a multiprogram laboratory operated by Sandia Corporation, a Lockheed Martin Company, for the United States Department of Energy's National Nuclear Security Administration under contract DE-AC04-94AL85000.

References

1. Ayguadé, E., Copty, N., Duran, A., Hoeflinger, J., Lin, Y., Massaioli, F., Teruel, X., Unnikrishnan, P., Zhang, G.: The Design of OpenMP Tasks. IEEE Trans. Parallel Distrib. Syst. 20(3), 404–418 (2009)
2. Balart, J., Duran, A., Gonzàlez, M., Martorell, X., Ayguadé, E., Labarta, J.: Nanos Mercurium: a Research Compiler for OpenMP. In: Proc. of the 6th European Workshop on OpenMP (EWOMP 2004), pp. 103–109 (October 2004)
3. Blumofe, R.D., Leiserson, C.E.: Scheduling Multithreaded Computations by Work Stealing. Journal of the ACM 46(5), 720–748 (1999)
4. Ferrer, R.: Task Chunking of Iterative Constructions in OpenMP 3.0. In: Proc. of the 1st Workshop on Execution Environments for Distributed Computing, pp. 49–54 (July 2007)

5. Ferrer, R., Duran, A., Martorell, X., Ayguadé, E.: Unrolling Loops Containing Task Parallelism. In: Gao, G.R., Pollock, L.L., Cavazos, J., Li, X. (eds.) LCPC 2009. LNCS, vol. 5898, pp. 416–423. Springer, Heidelberg (2010)
6. Kurzak, J., Ltaief, H., Dongarra, J.J., Badia, R.M.: Scheduling for Numerical Linear Algebra Library at Scale. In: Proc. of the High Performance Computing Workshop, pp. 3–26 (June 2008)
7. Leiserson, C.E.: The Cilk++ Concurrency Platform. The Journal of Supercomputing 51(3), 244–257 (2010)
8. Microsoft: Task Parallel Library (2013), `http://msdn.microsoft.com/en-us/library/dd460717.aspx` (last accessed June 21, 2013)
9. OpenMP Architecture Review Board: OpenMP Application Program Interface, Version 3.1 (July 2011)
10. OpenMP Architecture Review Board: OpenMP Application Program Interface, Version 4.0: Public Review Release Candidate 2 (March 2013)
11. Reinders, J.: Intel Threading Building Blocks. O'Reilly, Sebastopol (2007)
12. Terboven, C., Schmidl, D., Cramer, T., an Mey, D.: Task-Parallel Programming on NUMA Architectures. In: Kaklamanis, C., Papatheodorou, T., Spirakis, P.G. (eds.) Euro-Par 2012. LNCS, vol. 7484, pp. 638–649. Springer, Heidelberg (2012)

Using OpenMP under Android

Vikas[1], Travis Scott[2], Nasser Giacaman[3], and Oliver Sinnen[4]

The University of Auckland, New Zealand
{vik609,tsco033}@aucklanduni.ac.nz,
{n.giacaman,o.sinnen}@auckland.ac.nz

Abstract. The majority of software authored for the mobile platforms are GUI-based applications. With the advent of multi-core processors for the mobile platforms, these interactive applications need to employ sophisticated programming constructs for parallelism-concurrency, in order to leverage the potential of these platforms. An OpenMP-like, easy to use programming construct, can be an ideal way to add productivity. However, such as environment needs to be adapted to object-oriented needs and should be designed with an awareness of the interactive applications. Also, OpenMP does not provide a binding that target these platforms. This paper presents a compiler-runtime system for Android that presents OpenMP-like directives and GUI-aware enhancements.

Keywords: OpenMP, Android, GUI applications.

1 Introduction

For several years now (since 2006), the number of cores available in mainstream desktops started increasing; only half a decade later (in 2011), this trend started emerging in the mobile space, including smartphones and tablets. All flagship mobile devices are now multi-core, most of which are running Android. Much like the dilemma that faced software developers when mainstream desktops went multi-core [15], the same dilemma now faces mobile application developers; in order to harness the computing power provided by these additional cores, mobile "apps" must be parallelised since traditional sequential code is unable to utilise the increased resources provided by multiple cores.

Java, the language used by Android [3], provides a suite of native constructs for writing concurrent code, in the form of threads, thread pools, handlers, runnables and so forth. However, successfully using these tools can be difficult, and may not be immediately obvious to developers with a sequential programming background. This is especially important as the appeal of application development is being embraced by non-experienced programmers, whereas parallel programming was traditionally the domain of experienced programmers targeting large scientific and engineering problems. But even for experienced users, managing a large number of threading constructs can be time consuming and leads to the introduction of additional bugs [9].

A.P. Rendell et al. (Eds.): IWOMP 2013, LNCS 8122, pp. 15–29, 2013.

1.1 Motivation

Keeping the above in mind, it can be seen that the rapid advancement of multi-core processors for mobile devices necessitate the GUI applications (existing and new ones) to be parallel and concurrent. The mismatch between the pace of hardware advancement and software development can be bridged by using the principles and methodologies of directive-based constructs. The directive-based approach can decrease the general complexities of parallel-concurrent programming. Furthermore, with the ever increasing importance of the mobile domain, and in the absence of any directive-based implementation or binding, an OpenMP-like environment can be helpful to a great extent.

1.2 Contributions

This paper acknowledges OpenMP's expressive power and ease of use in the domain of shared memory parallel programming. However, should we wish to extend OpenMP's coverage to encompass mobile application development, there are concepts vital to the development of object-oriented GUI-based applications that are absent in the OpenMP model. To address these shortcomings, this paper makes the following contributions:

Study of GUI application: We explore the basic nature of GUI-based applications, particularly on mobile platforms. This leads to formalising the general requirements for such applications.

OpenMP-like environment for Android: The most dominant mobile platforms (Android and iOS) use object-oriented languages for the development of interactive applications. This paper presents OpenMP-like[1] directives and runtime support for Android.

GUI aware enhancements: Assisted with the exploration of the nature of GUI-based applications, this paper proposes GUI-aware extensions.

Implementation exploration for Android: Android supports Java as an application development language (and C/C++ through NDK), but it omits some standard Java libraries used in standard GUI application development (such as Swing and AWT). In this light, the paper presents implementation details of the proposed compiler, demonstrating a sample Android-based OpenMP compiler implementation.

Performance evaluation on mobile domain: We evaluate the proposed system using a set of interactive GUI applications for Android.

2 Background

2.1 Distinct Structure of GUI-Based Applications

In many ways, the execution flow in a GUI application is different from that of a conventional batch-type application. Firstly, the interactive applications have

[1] The directives, specially the GUI directives, that we present here are not official directives in the OpenMP standard; they are however designed with the intention of promoting the spirit of OpenMP, and as such, we refer to them as OpenMP-like directives.

their execution flow determined by the various inputs from the user. Secondly, the control flow is largely guided by the framework code of which the program is built on; this distinguishing aspect is known as *inversion of control* [7], when the flow of control is dictated by a framework rather than the application code.

The primary concept is that *events* are generated from within the framework code in response to user actions. The programmer only needs to implement specific routines, known as *event handlers*, in the application code in response to those events. The control returns back to the framework upon completion of the respective event handler, namely to the *event loop*. In most GUI frameworks, this is performed by a dedicated thread known as the GUI thread. In comparison, batch-type programs have the control flow determined by the programmer (i.e. the application code) and deal with serial input-output (even though the processing may be parallelised).

From the requirements perspective, in addition to the generic software requirements, an important criterion for GUI applications is their responsiveness. The application should always remain interactive and responsive to user actions. From the execution semantics, programmers must ensure that the GUI thread minimises its execution in the application code, therefore remaining largely in the event loop to respond to other potential events. In addition to off-loading the main computation away from the GUI thread, regular intermittent progress updates frequently need to be communicated back to the user to be perceived as having a positive user experience.

2.2 Mobile Devices and GUI Application Development

The mobile environment differs from the desktop environment on both the hardware and the software levels. From the hardware aspect, the mobile environment is largely constrained in memory and processing power (the dominant ARM processors focus on efficiency of power consumption). On the software side, the GUI frameworks support high level languages (such as Java in Android, Objective-C in iOS); however, not all the libraries or APIs are supported in a mobile application. Some possible reasons might be that these frameworks cater to the specific needs of the mobile devices and do not need to provide support for general purpose libraries. Also, many libraries and tools have been part of the desktop environment for legacy purposes. In many cases, they were designed and developed for systems with larger memory and faster processing power, and thus do not find a place on mobile platform. Nevertheless, mobile-based GUI applications do follow a similar architecture to desktop-based GUI applications, but the general consideration for memory and processing efficiency is always present.

2.3 Distinctions in Android Application Development Environment

There are some mentionable factors that make Android GUI application development different from that of the desktop environment. First, Android's execution environment runs every application in a separate sandboxed process and only a single application is displayed on the screen at a time (under the hood, Android uses a Linux kernel and every application is treated as an individual user).

This restricts the applications to have shared access to the file system, the application data, and more. The applications are executed in separate instances of the virtual machine, thus each application has a separate instance to itself. This enforces a design requirement over the applications. Moreover, Android's virtual machine, known as the Dalvik Virtual Machine (DVM), is not like a regular Java Virtual Machine (JVM). It is a slimmed down virtual machine that is designed to run on low memory. Importantly, DVM uses its own byte code and runs .dex (Dalvik Executable File) instead of the .class files that the standard JVM runs. In effect, the legacy Java libraries or Java code need a recompilation in order to be used for the Android.

On the framework side, Java is only supported as a language; libraries like Swing, AWT and SWT are not supported [13]. Nevertheless, it provides extra concurrency features, as will be discussed in section 3. Thus, even an experienced Java programmer needs to become acquainted with alternative constructs and the legacy programs cannot be directly ported.

Furthermore, owing to its recent development, Android has incorporated the generic requirements of a GUI-based application into the framework. For example, to counter the responsiveness related application freeze, Android throws an Application Not Responding (ANR) [2] error when an application remains unresponsive for more than a certain amount of time (around 5 seconds for Jelly Bean). In effect, this enforces a rule on the GUI applications to avoid the time consuming computations on the GUI thread and the applications should offload them. Another example is that of the `CalledFromWrongThreadException` exception; like most of the GUI framework, the Android framework is single threaded and the thread safety is maintained by throwing this exception if any non-GUI thread tries to update the GUI. Therefore, the programmer always needs to use specific constructs in concurrent programs for updating the GUI.

3 Related Work

3.1 Android Concurrency

Android supports the prominent concurrency libraries of Java using the `java.util.- concurrent` package, thus supporting the `ExecutorService` framework. The native threading is also supported. Additionally, Android exposes `AsyncTask` and `Handler` [12] for advanced concurrency. `AsyncTask` enables to perform asynchronous processing on background worker threads and enables methods to update the results on the GUI. `Handler` enables the posting and processing of the `Runnables` to the thread's message queue.

3.2 OpenMP for Android

There is no official OpenMP specification for Java, so OpenMP is not supported on Android. The Android Native Development Kit (NDK) supports C/C++ [11], but it does not support an OpenMP distribution. Although there are no Android OpenMP implementations, there are however some respectable Java OpenMP

implementations, namely JOMP [6] and JaMP [8]. While these tools provide important contributions to OpenMP Java bindings, they do not specifically target GUI applications as is the focus of Pyjama. More specifically, these solutions were developed well before the time of Android application development, as is the focus of this research.

4 Android Pyjama Compiler-Runtime

Pyjama [16] is a compiler-runtime system that supports OpenMP-like directives for an object-oriented environment, specifically for Java. Where required, Pyjama adapts the principles and semantics of OpenMP to an object-oriented domain and, in addition, provides the GUI-aware enhancements. The research presented in this paper is based on the preliminary work done on Pyjama, and now extending it to mobile application development.

4.1 Standard Directive Syntax

In the absence of any Android specification for OpenMP, we propose a format that is close to the OpenMP specification. A program line beginning with `//#omp` is treated as a directive by the proposed compiler and ignored as inline comments by the other compilers. Generic syntax is as shown below:

```
//#omp directiveName[clause[[,]clause]..]
```

4.2 Conventional OpenMP Directives and Semantics

The conventional directives [16], are supported in Android Pyjama as well. The system also supports object-oriented semantics within the scope of these conventional directives. For instance, *for-each* loop construct is a way to traverse over a collection of objects in object-oriented programming and parallelising a *for-each* loop is permissible using the `parallel for` construct. Furthermore, the OpenMP synchronisation directives like `barrier`, `critical`, `atomic` and `ordered` are supported with identical semantics as that of OpenMP for C/C++.

4.3 GUI-Aware Extensions

To improve upon the usability of OpenMP for GUI applications, Pyjama on Android introduces the following GUI-aware constructs:

- `freeguithread` directive

Specifies a structured block that is executed asynchronously from the GUI thread, freeing the GUI thread to process events.

```
//#omp freeguithread
structured-block
```

- `gui` directive

Specifies a structured block that is executed by the GUI thread. An implicit barrier is placed after the structured block unless an optional `nowait` clause is used.

```
//#omp gui [nowait]
structured-block
```

`freeguithread` Construct. A prominent limitation of using OpenMP in a GUI application is that OpenMP's fork-join threading model effectively violates the responsiveness of a GUI application. Consider the scenario where the code inside an event handler encounters an OpenMP construct; the master thread (MT) would be the GUI thread and would therefore take part in the processing of the OpenMP region. But in a GUI application, this is a problem; the GUI thread will remain busy processing (the parallel region), effectively blocking the GUI. The GUI-aware thread model addresses this responsiveness related issue; the basic principle is that an application will not have the tendency to become unresponsive, or block, if the GUI thread is free to process user inputs.

To achieve this responsiveness, we need to relieve the GUI thread from the execution of a specified region; hence the `freeguithread` directive. The underlying mechanism determines if the thread encountering `freeguithread` is the GUI thread. If yes, a new thread is created, called the Substitute Thread (ST), which executes this region on behalf of the GUI thread. As a result, the GUI thread is free to return to the event loop and handle incoming events. The structured block of the `freeguithread` directive is executed asynchronously to the GUI thread. When the execution of the region is completed by the ST, the GUI thread is notified and returns to execute the region following the `freeguithread` directive. This approach keeps the OpenMP threading model with its fork-join structure intact. Any `parallel` region that is then encountered by the ST is handled in the usual manner, whereby the ST is the master thread of that region.

`gui` Construct. For GUI applications, there is a need to update the GUI with intermittent results when the application code is still busy in the background processing. This may be related to conveying the partial results of the background processing or it may be a GUI update to convey the completion of background processing. In GUI application development, this is achieved by implementing a way to provide periodic updates to the GUI. Generally, it involves careful synchronisation methods or shared global flags. A programmer needs to introduce major code restructuring to spawn the computational work to other thread(s) and then again to execute GUI code, commonly encapsulated within `Runnable` instances and posted to the GUI thread. These methods have their own limitations and complexities and make it difficult to involve any OpenMP-like programming. It also opposes the the OpenMP philosophy of maintaining the program's original sequential structure when the OpenMP directives are ignored. For an elegant and easy solution of these issues, Pyjama introduces the new `gui` directive. Using it, a program can execute part of the code on the GUI thread from a background-processing region. This eliminates the need to maintain

complexities of synchronisation. The addition of the `freeguithread` and `gui` directives enable programmers to achieve responsive application development by obeying the single-thread rule of most GUI toolkits.

Syntactically, `freeguithread` and `parallel` can be combined in one directive statement. The further combination with the worksharing directives `for` and `sections` is also possible.

4.4 Runtime

The runtime component provides execution environment and timing routines, conforming to OpenMP 2.5 [10]. Additionally, the runtime provides a set of utility methods for the benefit of a programmer.

5 Implementation

5.1 Construction of Compiler

The parser for the compiler was created using Java Compiler Compiler (JavaCC). JavaCC is an open source parser generator for Java and the Java 1.5 grammar is provided as a part of the JavaCC distribution. It should be noted that JavaCC is not a lexical analyser or parser by itself. It needs to be provided with regular expressions and grammar and it generates a lexical analyser and a parser. The Java 1.5 grammar file was used as the base and was augmented with extended Backus Naur form (EBNF) -like[2] grammar notations for the OpenMP-like directives, to generate the lexical analyser and the parser.

5.2 Code Generation

The elementary process involves lexical analysis and parsing of the input code to generate an intermediate representation of the code (in this case, it is an AST); the AST is then traversed, using the visitor design pattern [14], and the directive specific nodes are translated to the respective parallel or concurrent version of the code.

From the software engineering perspective, the implementation is modularised by dividing the generation into two broad passes of *normalisation* and *translation*. Normalisation operations help simplify the process of target code generation. A wider range of directives are supported by actually implementing only the basic directives and normalising other directives to those implemented ones. The next pass is the translation pass, where an explicit parallel and concurrent version of the code is generated.

Translation of Conventional Directives and Clauses. The semantics of multi-threaded and GUI-aware translation adheres to the OpenMP's threading model. While introducing parallelism through the directives, the execution semantics follows the *fork-join* threading model and identifies a distinct master thread, like the OpenMP model.

[2] JavaCC provides the notation for defining grammar in a manner similar to EBNF.

Implementation wise, code outlining is the elementary approach employed by the translation pass. As figure 1 illustrates, the structured block from the `onClick()` method is outlined to form a new method (steps 1 and 2) and then executed using the runtime native queue and the native task-pool (steps 3, 4 and 5). Reflection is used to retrieve and execute the respective user code, thus implementing a *fork* and a barrier is placed after the region, thus implementing the *join*.

Also, the compiler achieves data passing by creating a new class to hold the variables from the encountering thread. Here, based on the data clauses used, an object of this class is instantiated and values from the encountering thread are assigned to it.

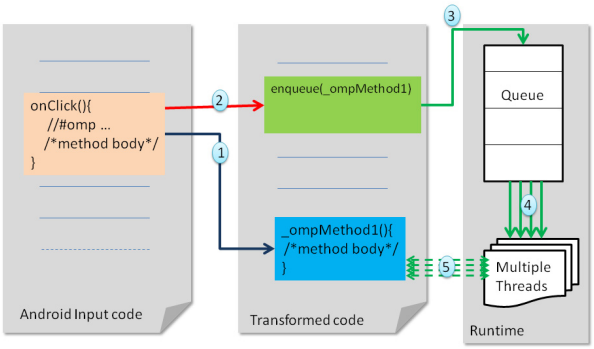


Fig. 1. The code outlining and queuing approach

Translation of GUI Directives. Translation of GUI directives forms an interesting part of this research. Semantically, the `freeguithread` region is treated as a task and moved to a new method which is enqueued and executed in the same way as a conventional directive. Like the semantics for the conventional directive, the enqueuing serves as the *fork*. The exception being that the encountering thread is not assigned as the `master`. Also, no barrier is placed in the original code, but a callback method is created using the code that appears after the `freeguithread` region in the original code. This callback is processed only after the execution of the task. This point serves as a continuation point and effectively as the *join*.

Concerning the specifics of implementation, the Android framework does not have an Event Dispatch Thread (EDT) and that is why the rudimentary implementation uses the application's *looper* instead. Every Android application has a main thread living inside the process in which the application is running. This thread contains a *looper*, which is called the main looper (instance of class `Looper`). For an *activity* or a *service* with a GUI, the main looper handles the GUI events. The main looper in Android is essentially analogous to the EDT in Java. With that knowledge, the GUI-aware directive verifies if the encountering thread is an event loop or not, using the method shown below:

```
private boolean isEventLoop() {
    boolean bELoop = false;
    if(Looper.myLooper()!= null) {
        bELoop = (Looper.myLooper() == Looper.getMainLooper())
    }
    return bELoop;
}
```

The `gui` directive translation searches for the main looper and posts the user code (as a `Runnable`) to it, as illustrated in the following code:

```
pHandler handler = new Handler(Looper.getMainLooper());
    handler.post(new Runnable(() {
        public void run() {
        ....
        }
    });
```

Here, `Handler` is another class that allows sending and processing the runnable objects associated with a thread's message queue. It binds to the threads message queue that created it. Also, as shown in the code, `Looper.myLooper()` returns the looper associated with the current thread, if it has one. If the returned one is not the application's main looper, then it is concluded that the current thread is not the main thread of the application.

6 Evaluations

In this section we present the preliminary evaluations of the conventional and GUI directives for Android. The overall strategy has been to evaluate the system on diverse devices and using diverse applications (non-conventional application, conventional mainstream applications and pure performance measuring applications).

6.1 Evolution Strategy Algorithm

The first application we present is an implementation of the EvoLisa algorithm created by Roger Alsing [1]. It is an evolution strategy algorithm, and is a subset of the broader class of evolutionary algorithms [5]. For this evaluation, we used a Galaxy Nexus 7 tablet, running an ARM Cortex-A9 Nvidia Tegra 1.2 GHz quad core processor with 1GB of RAM.

Strategy. In order to accurately gauge the suitability of the system as a development tool, and also more accurately reflect a real design scenario, an algorithm was chosen that was not already parallelised. Without prior knowledge of the algorithm model it was ensured that selection bias would not result in an algorithm being selected that was naturally suited to a fork-join model. The challenge of parallelising the algorithm also provided the opportunity to explore

how the environment can be used to parallelise non trivial algorithms. In addition to being unknown, the algorithm was also required to be long running and computationally expensive, such that it would be completely impossible to implement purely on the GUI thread. It also needed to provide updates to the user interface and have some level of user interactivity, so that it cannot simply be passed to a background thread to execute in isolation from the rest of the application. In fact, below is the code snippet of this app using Android Pyjama:

```
performImageGeneration(){
initialise starting working set of polygons
....
    //#omp freeguithread parallel
    {
        while(continueToGenerate){
        // fetch portion of polygons for current thread
        ....
        // attempt 100 random mutations and select best one
        ....
            //#omp barrier
            ....
            //#omp single
            {
                //recombine polygons and create new parent
                // create display image
                //#omp gui
                {
                    // update GUI
                }
            }
        }
    }
}
```

This code snippet promotes the harmony of using standard OpenMP for performance and synchronisation (in green), in combination with the GUI-aware constructs of Pyjama (in blue) to adhere to GUI concurrency rules in promoting a responsive Android GUI application. Furthermore, this code demonstrates the elegance of using a directive-based solution (such as OpenMP) to a new class of user interactive applications, not just the batch-type scientific and engineering applications traditionally tackled by OpenMP.

Evaluations. The algorithm was run repeatedly using different numbers of threads on three versions of the application: sequential, multi-threaded using `ExecutorService` and Android Pyjama directives. Tests were all performed on a single boot, with tests interleaved (i.e. single thread test performed, multi-threaded test with 2 threads performed, multi-threaded test with 4 threads performed). To perform the test an image was left to generate for 900 seconds, with

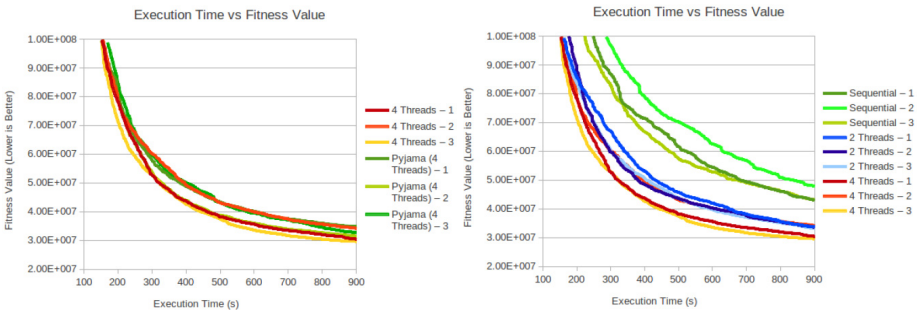


Fig. 2. Execution time vs fitness Values. Left:Pyjama vs native treads. Right:Improvement in fitness value with increase in number of threads.

the execution time and fitness level of the generated image recorded after each evolution cycle.

Results of the testing are shown in figure 2, displaying three runs of each of the manual threading (using `ExecutorService`) and Android Pyjama when each use four threads. It can be observed that the scalability and performance gain achieved is similar in both versions; the directive-base parallel-concurrent version (i.e. Pyjama) performs similar to the manually programmed multi-threaded version (lower fitness value is better). Due to the randomness aspect of the algorithm, there is sometimes a large degree of variation between runs. Figure 2 helps confirm though, that having increased parallelism does in fact speed up the progress of the algorithm on the mobile device, as increasing from 1 to 2 to 4 threads.

From the GUI aspect, intermittent GUI updates (i.e. results from the mutations) is an important part of this application. The code snippet showed how this was effectively handled by the GUI directive, embedded within a standard OpenMP `single` construct to ensure only one update request is made per cycle. In effect, the Android Pyjama implementation exhibited identical responsiveness to the manually threaded implementation using standard Android constructs; however, the Android Pyjama implementation clearly has the advantage of highly resembling the sequential version with minimal recoding. Irrespective of the ongoing background processing, the GUI remained responsive to the user actions. Figure 3, presents a small gallery of screenshots from the application.

6.2 Pattern Rendering Application

We developed another GUI application that creates psychedelic renderings on the screen. We used the Samsung GT-I9300 (Samsung Galaxy S-III) smartphone that has a 1.4 GHz quad-core ARM Cortex-A9 CPU and 1 GB of RAM.

Strategy. We designed the Android application in a way that it can serve as a representation of the major types of applications that are published for mobile

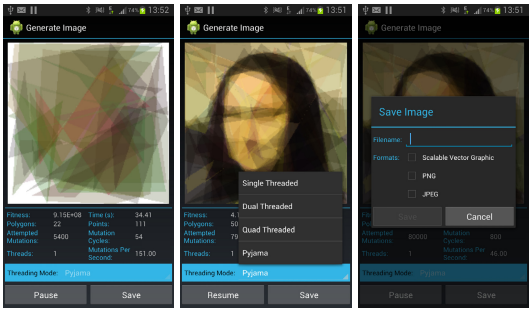


Fig. 3. Responsiveness while reconstruction of images in the EvoLisa Application

devices. Firstly, it creates a standard GUI for information screens and the help-screen using standard components of the Android toolkit, such as *layouts* and *widgets*. Secondly, the rendering screen of the application draws directly to the `canvas`, and controls all the drawings to it directly. In this way, it represents typical gaming and graphical applications, which are more compute intensive and richer in rendering.

The directives add incremental concurrency, performance and responsiveness to the application. Each screen (*activity*) uses the GUI-aware directive (`gui`), to render the screen components (*layouts* and *widgets*). The `parallel for` directive, along with GUI-aware directives (`freeguithread/gui`) are used to migrate the computations to the thread pool and to keep the screen responsive to gestures and touch inputs. For evaluations, we used two versions of the same applications. The first one using the directives and the second one using standard Android constructs, and compared the behaviours. This reflects on the completeness of the system on Android.

Evaluations. Considering the application responsiveness and behaviour, the two versions of the applications performed identically. The GUI display and pattern rendering on the screen is seen to be the same. Looking at the responsiveness, as figure 4 quantifies, the gesture and touch inputs were correctly registered.

6.3 Responsiveness Evaluation with Monkey Tool

The Monkey tool [4], distributed with Android SDK, generates random events (such as touch, motion, key events, clicks and more) which can be fired at the application. In figure 2, we observed that an increase in the number of threads improves the performance; we used the Monkey tool to test the responsiveness in the face of this improved performance. We evaluated different versions of the application by averaging over 20 runs and in each run we fired 5000 UI events at the applications. We measured the number of events that get dropped while the GUI thread is busy processing. We also measured the response time. The lesser the response time, the more responsive the application, and lesser are the chances of an application-freeze. We compared the Android Pyjama

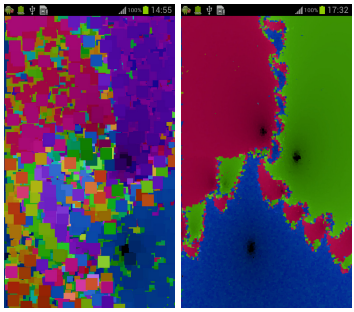


Fig. 4. Snapshots showing response to user's gestures. A "circle" gesture adds more colours to the fractal generation.

version of the application with that of another version developed using the native threads. The non-concurrent (single thread) version remains unresponsive to the events while processing the computationally intensive load and so could not be evaluated (Android throws an exception if workload is processed on the GUI thread). The results are shown in figure 5.

The results provide a fairly quantifiable measurement of the responsiveness in the Android Pyjama version. An overall responsiveness is achieved with both the Android concurrency and Pyjama, with comparable results. Here, it should be noted that Pyjama utilises a native task pool and therefore avoids the thread creation and destruction overheads. Also, as the number of threads increases and the application achieves more computation, the GUI thread gets smaller time slice to process the UI events. But this results in very small loss of responsiveness when compared to the performance gain that the application achieves.

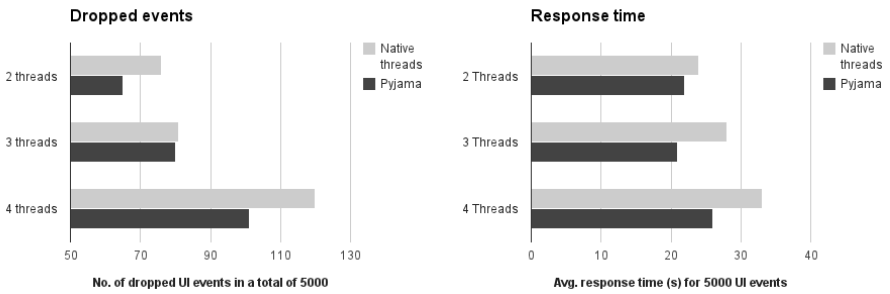


Fig. 5. Results of the responsiveness test with the Monkey tool

6.4 Mandelbrot Application

In order to measure the gross performance gain, by minimising the effects of GUI-based processing, we implemented an Android Mandelbrot application.

Two versions of the application were created for this evaluation; a serial version which does not use any concurrency or parallelism constructs (although a background thread is necessary to be launched for time consuming computations, otherwise the Android operating system throws an exception) and the second one uses the directives for parallelism (the `parallel` construct and the `parallel for` construct). It was observed that Android Pyjama scaled well on the Galaxy S III; for 4 cores, a speedup of 2.7x was observed in comparison to the serial version.

6.5 Productivity Evaluation

Even though in its preliminary stages, the proposed system exhibits good reasons for bringing a directives-based approach to the Android platform. By replacing native threading constructs with compiler directives, we were able to remove much of the fragmentation introduced into the code by these constructs (a 75% reduction was achieved in the EvoLisa demonstration application), resulting in more sequential, readable code. For the GUI applications, the aim is to provide a fluid experience for the user, and achieving this aim is often hindered by a large number of small tasks which introduce slight delays. The proposed system makes it simple for developers to offload these tasks to background threads, and provides the necessary tools to manage the complex interactions between these background threads and the GUI thread, thus removing the burden of manually managing this process.

7 Conclusion

With the mobile domain being so relevant today, we presented a compiler-runtime system to support directives based parallelism for Android promoting the OpenMP philosophy. The evaluations demonstrated positive results using a set of Android applications that focused on the GUI aspect of these applications; here, traditional parallelism in the form of speedup is only one aspect of performance, the other vital measure of performance being that of ensuring a user-perceived positive experience. Code snippets for the used directives also helped illustrate the contribution such a tool can provide for the productivity of mobile application developers.

References

1. Alsing, R.: Genetic Programming: Evolution of Mona Lisa (December 2008)
2. Google Inc. Android. Keep your app responsive (April 2013)
3. Android, Google Inc., `http://developer.android.com/guide/basics/what-is-android.html`
4. Android, Google Inc., `http://developer.android.com/tools/help/monkey.html`
5. Brownlee, J.: Evolution Strategies

6. Bull, J.M., Kambites, M.E.: JOMP—an OpenMP-like interface for Java. In: JAVA 2000: Proceedings of the ACM 2000 Conference on Java Grande, pp. 44–53. ACM, New York (2000)
7. Fayad, M., Schmidt, D.C.: Object-oriented application framework. Communications of the ACM 40(10), 32–38 (1997)
8. Klemm, M., Bezold, M., Veldema, R., Philippsen, M.: JaMP: an implementation of OpenMP for a Java DSM. Concurrency & Computation: Practice & Experience 19(18), 2333–2352 (2007)
9. Lee, E.A.: The Problem With Threads. IEEE Computer 39(5), 33–42 (2006)
10. OpenMP Architecture Review Board. OpenMP Application Program Interface Version 2.5 (2005)
11. Ratabouil, S.: Android NDK: discover the native side of Android and inject the power of C/C++ is your applications: begineer's guide. Packt Pub., Birmingham (2012)
12. Satya, K., Dave, M., Franchomme, E.: Pro Android 4. Apress, New York (2012)
13. Satya, K., Dave, M., Sayed, H.Y.: Pro Android 3. Apress, New York (2011)
14. Schordan, M.: The language of the visitor design pattern. Journal of Universal Computer Science 12(7), 849–867 (2006)
15. Sutter, H.: A fundamental turn toward concurrency in software. Dr. Dobb's Journal 30(3) (February 2005)
16. Vikas, Giacaman, N., Sinnen, O.: Pyjama: OpenMP-like implementation for Java, with GUI extensions. In: International Workshop on Programming Models and Applications for Multicores and Manycores (PMAM) Held in Conjunction with 18th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP 2013 (2013)

Expressing DOACROSS Loop Dependences in OpenMP

Jun Shirako[1], Priya Unnikrishnan[2], Sanjay Chatterjee[1], Kelvin Li[2], and Vivek Sarkar[1]

[1] Department of Computer Science, Rice University
[2] IBM Toronto Laboratory

Abstract. OpenMP is a widely used programming standard for a broad range of parallel systems. In the OpenMP programming model, synchronization points are specified by implicit or explicit barrier operations within a parallel region. However, certain classes of computations, such as stencil algorithms, can be supported with better synchronization efficiency and data locality when using *doacross parallelism* with point-to-point synchronization than *wavefront parallelism* with barrier synchronization. In this paper, we propose new synchronization constructs to enable doacross parallelism in the context of the OpenMP programming model. Experimental results on a 32-core IBM Power7 system using four benchmark programs show performance improvements of the proposed doacross approach over OpenMP barriers by factors of 1.4× to 5.2× when using all 32 cores.

1 Introduction

Multicore and manycore processors are now becoming mainstream in the computer industry. Instead of using processors with faster clock speeds, all computers— embedded, mainstream, and high-end systems — are being built using chips with an increasing number of processor cores with little or no increase in clock speed per core. This trend has forced the need for improved productivity in parallel programming models. A major obstacle to productivity lies in the programmability and performance challenges related to coordinating and synchronizing parallel tasks. Effective use of barrier and point-to-point synchronization are major sources of complexity in that regard. In the OpenMP programming model [1, 2], synchronization points are specified by implicit or explicit barrier operations, which force all parallel threads in the current parallel region to synchronize with each other[1]. However, certain classes of computations such as stencil algorithms require to specify synchronization only among particular iterations so as to support *doacross parallelism* [3] with better synchronization efficiency and data locality than *wavefront parallelism* using all-to-all barriers.

In this paper, we propose new synchronization constructs to express cross-iteration dependences of a parallelized loop and enable doacross parallelism in

[1] This paper focuses on extensions to OpenMP synchronization constructs for parallel loops rather than parallel tasks.

A.P. Rendell et al. (Eds.): IWOMP 2013, LNCS 8122, pp. 30–44, 2013.

the context of the OpenMP programming model. Note that the proposed constructs aim to express ordering constraints among iterations and we do not distinguish among flow, anti and output dependences. Experimental results on a 32-core IBM Power7 system using numerical applications show performance improvements of the proposed doacross approach over OpenMP barriers by the factors of 1.4–5.2 when using all 32 cores.

The rest of the paper is organized as follows. Section 2 provides background on OpenMP and discusses current limitations in expressing iteration-level dependences. This section also includes examples of low-level hand-coded doacross synchronization in current OpenMP programs, thereby providing additional motivation for our proposed doacross extensions. Section 3 introduces the proposed extensions to support cross-iteration dependence in OpenMP. Section 4 discusses the interaction of the proposed doacross extensions with existing OpenMP constructs. Section 5 describes compiler optimizations to reduce synchronization overhead and runtime implementations to support efficient cross-iteration synchronizations. Section 6 presents our experimental results on a 32-core IBM Power7 platform. Related work is discussed in Section 7, and we conclude in Section 8.

2 Background

2.1 OpenMP

In this section, we give a brief summary of the OpenMP constructs [2] that are most relevant to this paper. The `parallel` construct supports the functionality to start parallel execution by creating parallel threads. The number of threads created is determined by the environment variable `OMP_NUM_THREADS`, runtime function `omp_set_num_threads` or `num_threads` clause specified on the `parallel` construct. The `barrier` construct specifies an all-to-all barrier operation among threads in the current `parallel` region[2]. Therefore, each `barrier` region must be encountered by all threads or by none at all. The loop constructs, `for` construct in C/C++ and `do` construct in Fortran, are work-sharing constructs to specify that the iterations of the loop will be executed in parallel. An implicit barrier is performed immediately after the loop region. The implicit barrier may be omitted if a `nowait` clause is specified on the loop directive. Further, a `barrier` is not allowed inside a loop region. The `collapse` clause on a loop directive collapses multiple perfectly nested rectangular loops into a singly nested loop with an equivalent size of iteration space. The `ordered` construct specifies a structured block in a loop region that will be executed in the order of the loop iterations. This sequentializes and orders the code within an `ordered` region while allowing code outside the region to run in parallel. Note that an `ordered` clause must be specified on the loop directive, and the `ordered` region must be executed only once per iteration of the loop.

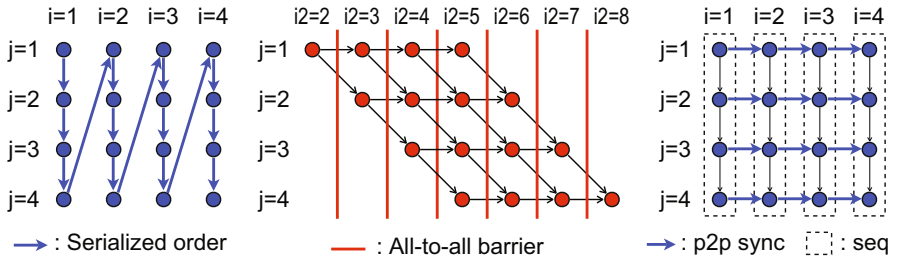
[2] A region may be thought of as the dynamic or runtime extent of construct - i.e., region includes any code in called routines while a construct does not.

```
1 #pragma omp parallel for collapse(2) ordered
2 for (i = 1; i < n-1; i++) {
3     for (j = 1; j < m-1; j++) {
4         #pragma omp ordered
5         A[i][j] = stencil(A[i][j], A[i][j-1], A[i][j+1],
6                           A[i-1][j], A[i+1][j]);
7 }   }
8                     (a) Ordered construct
9
10 #pragma omp parallel private(i2)
11 {
12     for (i2 = 2; i2 < n+m-3; i2++) {   /* Loop skewing */
13         #pragma omp for
14         for (j = max(1,i2-n+2); j < min(m-1,i2); j++) {
15             int i = i2 - j;
16             A[i][j] = stencil(A[i][j], A[i][j-1], A[i][j+1],
17                               A[i-1][j], A[i+1][j]);
18 }   }   }
19                 (b) Doall with implicit barrier
```

Fig. 1. 2-D Stencil using existing OpenMP constructs: (a) serialized execution using ordered construct, (b) doall with all-to-all barrier after loop skewing

2.2 Expressiveness of Loop Dependences in OpenMP

As mentioned earlier, a `barrier` construct is not allowed within a loop region, and `ordered` is the only synchronization construct that expresses cross-iteration loop dependences among `for`/`do` loop iterations. However, the expressiveness of loop dependence by `ordered` construct is limited to sequential order and does not cover general loop dependence expressions such as dependence distance vectors. Figure 1a shows an example code for 2-D stencil computation using `ordered`. The `collapse` clause on the `for` directive converts the doubly nested loops into a single nest. This clause is used to ensure that the `ordered` region is executed only once per iteration of the parallel loop, as required by the specifications. Although the dependence distance vectors of the doubly nested loop are (1,0) and (0,1) and hence it has doacross parallelism, the `ordered` construct serializes the execution and no parallelism is available as shown in Figure 2a. An alternative way to exploit parallelism is to apply loop skewing so as to convert doacross parallelism into doall in wavefront fashion. Figure 1b shows the code after loop skewing and parallelizing the inner j-loop using a `for` construct, which is followed by an implicit barrier. As shown in Figure 2b, the major performance drawback of this approach is using all-to-all barrier synchronizations, which are generally more expensive than point-to-point synchronizations used for doacross. Further, this requires programmer expertise in loop restructuring techniques - i.e., selecting correct loop skewing factor and providing skewed loop boundaries.



(a) Ordered construct **(b)** Doall with implicit barrier **(c)** Doacross extensions

Fig. 2. Synchronization Pattern for 2-D Stencil

2.3 Examples of Hand-Coded Doacross Synchronization in Current OpenMP Programs

Some expert users provide customized barriers/point-to-point synchronizations based on busy-wait local spinning implementations [4] using additional volatile variables for handling synchronization. Although such customized implementations can bring fully optimized performance, they require solid knowledge of parallel programming and the underlying system and are easy to introduce error and/or potential deadlock.

Examples can be found in the OpenMP version of the NAS Parallel Benchmark [5, 6], which is a widely used HPC benchmark suite since 1992. E.g., in NPB3.3.1/NPB3.3-OMP/LU, the pipelining of the SSOR algorithm is achieved by point-to-point synchronizations through extra synchronization variables, busy-waiting, and `flush` directive for memory consistency. The code utilizes OpenMP library functions, `threadprivate` directive in addition to the loop construct with `schedule(static)` and `nowait` clauses. The end result is code that is non-intuitive and unduly complicated.

Further, the LU implementation in NPB reveals a non-compliant usage of `nowait`. Although it makes an assumption that a `nowait` is always enforced, the OpenMP standard states that *"... If a* `nowait` *clause is present, an implementation* **may** *omit the barrier at the end of the worksharing region."*. Because of "may", implementations are allowed to ignore the `nowait` and introduce a barrier. However, if this happens, then the LU implementation will deadlock.

Our proposal aims to address all those issues regarding complexity and deadlock avoidance while keeping the synchronization efficiency via point-to-point synchronizations.

3 New Pragmas for Doacross Parallelization

This section introduces our proposed OpenMP extensions to express general cross-iteration loop dependences and thereby support doacross parallelization. Due to space limitations, we focus on the pragma syntax for C/C++ in this paper, although our proposal is naturally applicable to both C/C++ and Fortran.

The proposed doacross annotations consist of `nest` clause to specify target loops of doacross parallelization and `post`/`await` constructs to express source/sink of cross-iteration dependences. Ideally, an `await` construct works as a blocking operation that waits for `post` constructs in specific loop iterations. The `post` construct serves as the corresponding unblocking operation. According to the OpenMP terminology, loops that are affected by a loop directive are called *associated loops*. For ease of presentation, we will call the associated loops that are the target of doacross parallelization as a *doacross loop nest*. According to the OpenMP specification [2], all the associated loops must be perfectly nested and have canonical form.

- **nest clause:**
  The "`nest`(n)" clause, which appears on a loop directive, specifies the nest-level of a doacross loop nest. Although `nest` clause defines associated loops as with `collapse` clause, there are two semantical differences from `collapse` clause: 1) `nest` clause is an informational clause and does not necessarily imply any loop restructuring, and 2) `nest` clause permits triangular/trapezoidal loops, whereas `collapse` clause is restricted to rectangular loops.
- **await construct:**
  The "`await depend`(*vect*)[[,] `depend`(*vect*)...]" construct specifies the source iteration vectors of the cross-iteration dependences. There must be at least one `depend` clause on an `await` directive. The current iteration is blocked until all the source iterations specified by *vect* of `depend` clauses[3] execute their `post` constructs. Based on OpenMP's default sequential semantics, the loop dependence vector defined by `depend` clause must be lexicographically positive, and we also require that the dependence vector is constant at compile-time so as to simplify the legality check. Therefore, we restrict the form of *vect* to $(x_1 - d_1, x_2 - d_2, ..., x_n - d_n)$, where n is the dimension specified by the `nest` clause, x_i denotes the loop index of i-th nested loop, and d_i is a constant integer for all $1 \leq i \leq n$ [4]. The dependence distance vector is simply defined as $(d_1, d_2, ..., d_n)$. If the *vect* indicates an invalid iteration (i.e., if vector $(d_1, \ldots, d_n)$ is lexicographically non-positive) then the `depend` clause is ignored implying that there is no real cross-iteration dependence. The `await` is a stand-alone directive without associated executable user codes and designates the location where the blocking operation is invoked. Note that at most one `await` construct can exist in the lexical scope[5] of the loop body of a doacross loop nest.
- **post construct:**
  The "`post`" construct indicates the termination of the computation that causes loop dependences from the current iteration. This stand-alone

[3] The `depend` clause is also under consideration for expressing "inter-task" dependences for `task` construct in OpenMP 4.0.

[4] It is a simple matter to also permit + operators in this syntax since $x_i + d$ is the same as $x_i - (-d)$.

[5] This means `await` and `post` cannot be dynamically nested inside a function invoked from the loop body because they are closely associated with the induction variables of the doacross loop nest.

```
1 #pragma omp parallel for nest(2)
2 for (i = 1; i < n-1; i++) {
3     for (j = 1; j < m-1; j++) {
4         #pragma omp await depend(i-1,j) depend(i,j-1)
5         A[i][j] = stencil(A[i][j], A[i][j-1], A[i][j+1],
6                           A[i-1][j], A[i+1][j]);
7 }   }
8   (a) Iteration-level dependences: explicit await at top and
        implicit post at bottom
9
10 #pragma omp parallel for nest(2)
11 for (i = 1; i < n-1; i++) {
12     for (j = 1; j < m-1; j++) {
13         int tmp = foo(A[i][j]);
14         #pragma omp await depend(i-1,j) depend(i,j-1)
15         A[i][j] = stencil(tmp, A[i][j-1], A[i][j+1],
16                           A[i-1][j], A[i+1][j]);
17         #pragma omp post
18         B[i][j] = bar(A[i][j]);
19 }   }
20   (b) Statement-level dependences: explicit await before line
        15 and explicit post after line 16
```

Fig. 3. 2-D Stencil with the doacross extensions

directive designates the location to invoke the unblocking operation. Analogous to `await` construct, the location must be in the loop body of the doacross loop nest. The difference from `await` construct is that there is an implicit `post` at the end of the loop body. Note that the parameter in the `nest` clause determines the location of the implicit `post`. Due to the presence of the implicit `post`, it is legal to have no `post` constructs inserted by users while the explicit `post` is allowed at most once. The implicit `post` becomes no-op when the invocation of the explicit `post` per loop body is detected at runtime. Finally, it is possible for an explicit `post` to be invoked before an `await` in the loop body.

Figure 3 contains two example codes for the 2-D stencil with the doacross extensions that specify cross-iteration dependences $(1, 0)$ and $(0, 1)$. As shown in Figure 3a, programmers can specify iteration-level dependences very simply by placing an `await` construct at the start of the loop body and relying on the implicit `post` construct at the end of the loop body. On the other hand, Figure 3b shows a case in which `post` and `await` are optimally placed around lines 15 and 16 to optimize statement-level dependences and minimize the critical path length of the doacross loop nest. Note that functions foo and bar at lines 13 and 18 do not contribute to the cross-iteration dependences; foo can be executed before the `await` and bar can execute after the `post`. The `post`/`await` constructs semantically specify the source/sink of cross-iteration dependences and allow flexibility on how to parallelize the doacross loop nest.

4 Interaction with Other OpenMP Constructs

This section discusses the interaction of the proposed doacross extensions with the existing OpenMP constructs. We classify the existing constructs into three categories: 1) constructs that cannot be legally used in conjunction with the doacross extensions, 2) constructs that can be safely used with the doacross extensions, and 3) constructs that require careful consideration when used with the doacross extensions.

4.1 Illegal Usage with Doacross

The `nest`, `await` and `post` constructs make sense only in the context of loops. So the usage of these constructs along with the `sections`, `single` and `master` constructs are illegal. Similarly using them with a `parallel` region without an associate loop construct is illegal, e.g., `#pragma omp parallel for nest()` is legitimate but `#pragma omp parallel nest()` is not.

4.2 Safe Usage with Doacross

The `nest`, `await` and `post` constructs are to be used in conjunction with loops only. The doacross extension can be safely used with the following clauses.

- **private/firstprivate/lastprivate clauses:**
  These are data handling clauses that appear on a loop construct. Because they are only concerned with the data and have not affect on loop scheduling nor synchronization, it is always safe to use with the doacross extensions.
- **reduction clause:**
  This clause appears on a loop directive and specifies a reduction operator, e.g., `+` and `*`, and target variable(s). Analogous to `private` constructs, `reduction` clause can be safely combined with the doacross constructs.
- **ordered construct:**
  The `ordered` clause to appear on a loop directive serializes loop iterations as demonstrated in Section 2. The `ordered` construct specifies a statement or structured block to be serialized in the loop body. Because loop dependences specified by `await` constructs are lexicographically positive, any doacross dependence does not go against with the sequential order by the `ordered` construct, and hence the combination of `ordered` and `await` constructs creates no theoretical deadlock cycle.
- **collapse clause:**
  The `collapse` clause attached on a loop directive is to specify how many loops are associated with the loop construct, and the iterations of all associated loops are collapsed into one iteration space with equivalent size. We allow the combination of `collapse` clause and `nest` clause in the following manner. When `collapse(`m`)` and `nest(`n`)` clauses are specified on a loop nest whose nest-level is l, the loop transformation due to `collapse` clause is first processed and the original l-level loop nest is converted into

a $(l - m + 1)$-level loop nest. Then, the `nest(`n`)` clause and corresponding `post`/`await` constructs are applied to the resulting loop nest. Note that $l \geq m + n - 1$, otherwise it results in a compile-timer error.

- **schedule clause:**
  The `schedule` clause attached on a loop directive is to specify how the parallel loop iterations are divided into chunks, which are contiguous non-empty subsets, and how these chunks are distributed among threads. The `schedule` clause supports several loop scheduling kinds: `static`, `dynamic`, `guided`, `auto` and `runtime`, and allows users to specify the chunk size. As discussed in Section 5, any scheduling kind and chunk size are safe to use with the doacross extensions.
- **task construct:**
  The `task` construct to define an explicit task is available within a loop region. Analogous to `ordered` construct, we prohibit a `post` construct from being used within a `task` region since such a `post` will have a race condition with the implicit `post` at the loop body end.
- **atomic/critical constructs:**
  The `atomic` and `critical` constructs to support atomicity and mutual exclusion respectively can be used within a loop region. In order to avoid deadlock, we disallow a `critical` region to contain an `await` construct as with `ordered` construct.
- **simd construct:**
  The `simd` construct, which will be introduced in the OpenMP 4.0, can be applied to a loop to indicate that the loop can be transformed into a SIMD loop (that is, multiple iterations of the loop can be executed concurrently using SIMD instructions). We disallow `simd` clause and `nest` clause from appearing on the same loop construct due to conflict in semantics, i.e., `nest` clause implies loop dependence while `simd` mentions SIMD parallelism. Instead, we allow SIMD loop(s) to be nested inside a doacross loop nest.

4.3 Constructs Requiring Careful Consideration

The lock routines supported in the OpenMP library can cause a deadlock when interacted with the `await` construct, especially under the following situation: 1) an `await` construct is located between lock and unlock operations and 2) a `post` construct is located after the lock operation. It is user's responsibility to avoid such deadlock situations due to the interaction.

5 Implementation

This section describes a simple and efficient implementation approach for the proposed doacross extensions to OpenMP; however, other implementation approaches are possible as well. Our approach only parallelizes the outermost loop of the doacross loop nest and keeps inner loops as sequential to be processed by a single thread. The loop dependences that cross the parallelized iterations

are enforced via runtime point-to-point synchronizations. Figure 2c shows the synchronization pattern of this approach for the doacross loop in Figure 3, where the outer i-loop is parallelized and its cross-iteration dependence, $(1, 0)$, is preserved by the point-to-point synchronizations. In order to avoid deadlock due to the point-to-point synchronization, the runtime loop scheduling must satisfy a condition that an earlier iteration of the parallel loop is scheduled to a thread before a later iteration. According to the OpenMP Specification (lines 19–21 in page 49 for 4.0 RC2) [2], this condition is satisfied by any OpenMP loop scheduling policy. Note that the same condition is necessary for `ordered` clause to avoid deadlock. Further, any chunk size can be used without causing deadlock. However, chunk sizes greater than the dependence distance of the parallel loop significantly reduce doacross parallelism. Therefore, we assume that the default chunk size for doacross loops is 1. Further, the default loop schedule is `static` so as to enable the lightweight version of synchronization runtime as discussed in Section 5.2.

5.1 Compiler Supports for Doacross Extension

The major task for compilers is to check the legality of the annotated information via the `nest` and `post/await` constructs, and convert the information into runtime calls to POST/WAIT operations. Further, we introduce a compile-time optimization called *dependence folding* [7], which integrates the specified cross-iteration dependences into a conservative dependence vector so as to reduce the runtime synchronizations [7].

Legality Check and Parsing for Doacross Annotations: As with `collapse` clause, the loop nest specified by `nest` clause must be perfectly nested and have canonical loop form [2]. To verify `nest` clauses, we can reuse the same check as `collapse` clause. The legality check for `await`, `depend` and `post` constructs ensure 1) at most one `post/await` directive exists at the nest-level specified by the `nest` clause, 2) the dependence vector of a `depend` clause is lexicographically positive and constant, and 3) the dimension of the dependence vector is same as the parameter of the `nest` clause.

After all checks are passed and the optimization described in next paragraph is applied, the doacross information is converted into the POST and WAIT runtime calls. The locations for these calls are same as the `post/await` constructs. The argument for the POST call is the current iteration vector $(x_1, x_2, ..., x_n)$, while the argument for the WAIT call is defined as $(x_1 - c_1, x_2 - c_2, ..., x_n - c_n)$ by using the conservative dependence vector discussed below.

Dependence Folding: In order to reduce the runtime synchronization overhead, we employ dependence folding [7] that integrates the multiple cross-iteration dependences specified by the `await` construct into a single conservative dependence. First, we ignore dependence vectors whose first dimension - i.e., the dependence distance of the outermost loop - is zero because such dependences are always preserved by the single thread execution. The following discussion

assumes that any dependence vector has a positive value in the first dimension in addition to the guarantee of constant dependence vectors.

For an n-th nested doacross loop with k dependence distance vectors, let $\boldsymbol{D^i} = (d_1^i, d_2^i, ..., d_n^i)$ denote the i-th dependence vector ($1 \leq i \leq k$). We define the conservative dependence vector $\boldsymbol{C} = (c_1, c_2, ..., c_n)$ of all k dependences as follow.

$$\boldsymbol{C} = \begin{pmatrix} C[1] : (c_1) \\ C[2..n] : (c_2, c_3, ..., c_n) \end{pmatrix} = \begin{pmatrix} gcd(d_1^1, d_1^2, ..., d_1^k) \\ min\_vect(D^1[2..n], D^2[2..n], ..., D^k[2..n]) \end{pmatrix}$$

Because the outermost loop is parallelized, the first dimension of $\boldsymbol{D^i}$, d_1^i, denotes the stride of dependence across parallel iterations. Therefore, the first dimension of $\boldsymbol{C}$ should correspond to the GCD value of d_1^1, d_1^2, ..., d_1^k. The remaining dimensions, $\boldsymbol{C}[2..n]$, can be computed as the lexicographical minimum vector of $D^1[2..n]$, $D^2[2..n]$, ..., $D^k[2..n]$ because such a minimum vector and the sequential execution of inner loops should preserve all other dependence vectors. After dependence folding, the conservative dependence $\boldsymbol{C}$ is used for the POST call as described in the previous paragraph.

5.2 Runtime Supports for POST/WAIT Synchronizations

This section briefly introduces the runtime algorithms of the POST/WAIT operations. When the loop schedule kind is specified as `static`, which is the default for doacross loops, we employ the algorithms introduced in our previous work [7]. For other schedule kinds such as `dynamic`, `guided`, `auto`, and `runtime`, we use the following simple extensions. Figure 4 shows the pseudo codes for the extended POST/WAIT operations. To trace the POST operations, we provides a 2-dimensional synchronization field $sync\_vec[lw : up][1 : n]$, where lw/up is the lower/upper bound of the outermost loop and n is the nest-level of the doacross loop nest. Because the iteration space of the outermost loop is parallelized and scheduled to arbitrary threads at runtime, we need to trace all the parallel iterations separately while the status of an iteration i ($lw \leq i \leq up$) is represented by $sync\_vec[i][1 : n]$. During the execution of inner loops by a single thread, the thread keeps updating the $sync\_vec[i]$ via the POST operation with the current iteration vector $pvec$, while the WAIT operation is implemented as a local-spinning until the POST operation corresponding to the current WAIT is done - i.e., $sync\_vec[i]$ becomes greater or equal to the dependence source iteration vector $wvec$. As shown at Line 9 of Figure 4, the WAIT operation becomes no-op if the $wvec$ is outside the legal loop boundaries.

6 Experimental Results

In this section, we present the experimental results for the proposed doacross extensions in OpenMP. The experiments were performed on a Power7 system with 32-core 3.55GHz processors running Red Hat Enterprise Linux release 5.4.

```
1  volatile int sync_vec[lw:up][1:n];
2
3  void post(int pvec[1:n]) {
4      int i = pvec[1];            /* Outermost loop index value */
5      for (int j = n; j > 0; j--) sync_vec[i][j] = pvec[j];
6  }
7
8  void wait(int wvec[1:n]) {
9      if (outside_loop_bounds(wvec)) return; /* invalid await */
10     int i = wvec[1];            /* Outermost loop index value */
11     while (vector_compare(sync_vec[i], wvec]) < 0) sleep();
12 }
```

Fig. 4. Pseudo codes for POST and WAIT operations

The measurements were done using a development version of the XL Fortran 13.1 for Linux, which supports automatic doacross loop parallelization in addition to doall parallelization. Although we use the Fortran compiler and benchmarks for our experiments, essential functionalities to support the doacross parallelization are also common for any C compilers. We used 4 benchmark programs for our evaluation: SOR and Jacobi, which are variants of the 2-dimensional stencil computation, Poisson computation, and 2-dimensional LU from the NAS Parallel Benchmarks Suite (Version 3.2). All these benchmarks are excellent candidates for doacross parallelization. All benchmarks were compiled with option "-O5" for the sequential baseline, and "-O5 -qsmp" for the parallel executions. a) omp doacross is the speedup where the doacross parallelism is enabled by the proposed doacross extensions (right), b) omp existing is the speedup where the same doacross parallelism is converted into doall parallelism via manual loop skewing and parallelized by the OpenMP loop construct (center), and c) auto par represents the speedup where the automatic parallelization for doall and doacross loops by the XL compiler is enabled (left). As shown below, auto par does not always find the same doacross parallelism as omp doacross. We used default schedule kind and chunk size, i.e., `static` with chunk size = 1 for omp doacross and auto par, and `static` with no chunk size specified for omp existing.

6.1 SOR and Jacobi

Figure 5 shows the kernel computation of SOR, which repeats `mjmax`×`mimax` 2-D stencil by `nstep` times. We selected `nstep` = 10000, `mjmax` = 10000 and `mimax` = 100 so as to highlight the existing OpenMP performance with loop skewing. Jacobi has a very similar computation to SOR and both have the same pattern of cross-iteration dependences. For the proposed doacross extensions, we specified the outermost `l`-loop and middle `j`-loop as doubly nested doacross loops with cross-iteration dependences (1,-1) and (0,1). For the existing OpenMP parallelization, we converted the same doacross parallelism into doall via loop skewing.

```
1 !$omp parallel do nest(2)
2       do 10 l = 1, nstep
3        do 10 j = 2, mjmax
4 !$omp await(l-1,j+1) await(l,j-1)
5          do 10 i = 2, mimax
6           p(i,j)=(p(i,j)+p(i+1,j)+p(i-1,j)+p(i,j+1)+p(i,j-1))/5
7  10    continue
8 !$omp end parallel do
```

Fig. 5. SOR Kernel

Figure 6 shows the speedups of the three versions listed above when compared to the sequential execution. As shown in the Figure 6a and 6b, omp doacross has better scalability than omp existing for both SOR and Jacobi despite of the same degree of parallelism. This is mainly because the doacross version enables point-to-point synchronizations between neighboring threads, which is more efficient than the all-to-all barrier operations by the existing approach. The version of auto par applied doacross parallelization to the middle `j`-loop and innermost `i`-loop; the scalability is worse than the manual approaches due to the finer-grained parallelism. Note that the outermost `l`-loop is time dimension and difficult for compilers to automatically compute dependence distance vectors. The enhanced dependence analysis in the XL compiler should be addressed in future work

6.2 Poisson

The kernel loop of Poisson is also a triply nested doacross loop with size of 400×400×400. As with SOR and Jacobi, omp doacross and omp existing use the doubly nested doacross parallelism of the outermost and middle loops, and the innermost loop is executed without any synchronization. Figure 6c shows the doacross version has better scalability due to the point-to-point synchronizations. On the other hand, although auto par exactly detected all the dependence distance vectors and applied doacross parallelization at the outermost loop level, it parallelized the loop nest as a triply nested doacross and inserted the POST/WAIT synchronizations at the innermost loop body. Note that auto par applies compile-time and runtime granularity controls (loop unrolling and POST canceling, respectively) based on the cost estimation [7]. However, selecting the doacross nest-level as 2 brought more efficiency for the manual versions as shown in Figure 6c. The automatic selection of doacross nest-level is another important future work for the doacross parallelization by the XL compiler.

6.3 LU

LU has 2 doacross loop nests in subroutines `blts` and `buts`, which are 160×160 doubly nested doacross loops and account for about 40% of the sequential execution time. As observed for other benchmarks, omp doacross has better scalability than omp existing due to the synchronization efficiency. For the case of LU, both omp doacross and auto par use the same doacross parallelism. A difference is that

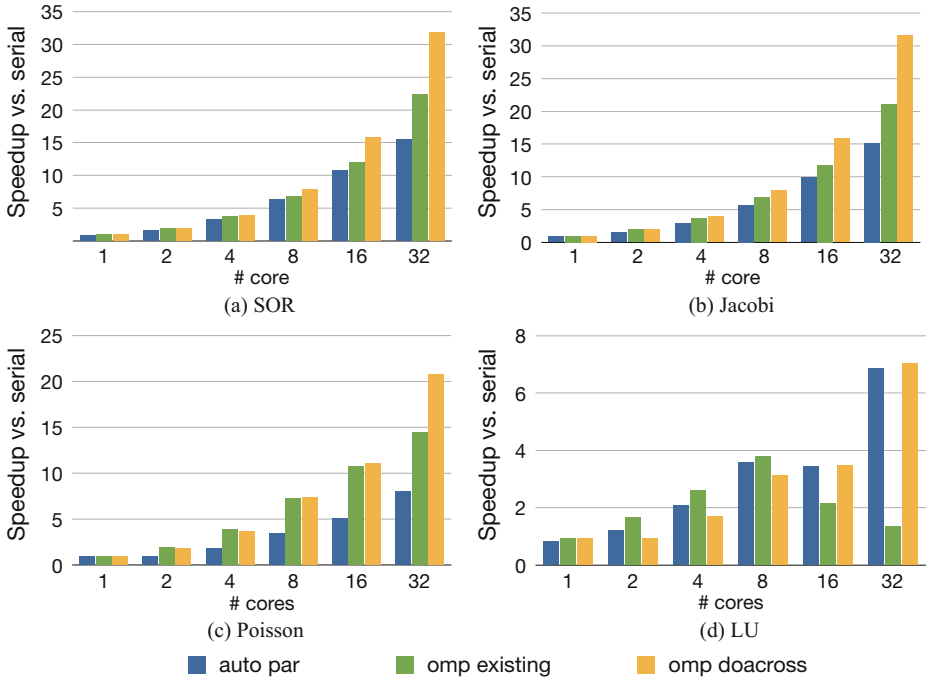


Fig. 6. Speedup related to sequential run on Power7

the granularity control is disabled for omp doacross since we should not assume such optimizations in the language specification. However, the execution cost for the doacross loop bodies in `blts` and `buts` is not small; disabling granularity control did not result in large performance degradation with up to 8 cores, and even better/same performance was shown with 32/16 cores because increasing granularity can also affect the amount of parallelism.

7 Related Work

There is an extensive body of literature on doacross parallelization and point-to-point synchronization. In this section, we focus on past contributions that are most closely related to this paper.

Some of the seminal work in synchronization mechanism was done by Padua and Midkiff [8, 9], where they focused on synchronization techniques for single-nested doacross loops using synchronization variable per loop dependence. MPI supports several functions for point-to-point synchronization and communication among threads, such as `MPI_send` and `MPI_recv`. As a recent research outcome, phasers in the Habanero project [10] and `java.util.concurrent.Phaser` from Java 7, which was influenced by Habanero phasers [11], support point-to-point synchronizations among dynamically created tasks. Further, a large amount of existing work on handling non-uniform cross-iteration dependences at runtime [12–15] have been proposed.

There is also a long history on doacross loop scheduling and granularity control [3, 16–18] and compile-time/runtime optimizations for synchronizations [19–21]. These techniques are applicable to the proposed doacross extensions.

8 Conclusions

This paper proposed new synchronization constructs to express cross-iteration dependences of a parallelized loop and enable doacross parallelism in the context of OpenMP programming model. We introduced the proposed API designs and detailed semantics, and discussed the interaction with the existing OpenMP constructs. Further, we described the fundamental implementations for compilers and runtime libraries to support the proposed doacross extensions. Experimental results on a 32-core IBM Power7 system using numerical applications show performance improvements of the proposed doacross approach over existing OpenMP approach with additional loop restructuring by factors of 1.4–5.2 when using all 32 cores. Opportunities for future research include performance experiments with different program sizes and platforms, explorations for the combination with other OpenMP features, e.g., `simd` and `task` constructs, and generalization of point-to-point synchronization aiming for the support of task dependence in OpenMP 4.0.

References

1. Dagum, L., Menon, R.: OpenMP: An industry standard API for shared memory programming. IEEE Computational Science & Engineering (1998)
2. OpenMP specifications, `http://openmp.org/wp/openmp-specifications`
3. Cytron, R.: Doacross: Beyond vectorization for multiprocessors. In: Proceedings of the 1986 International Conference for Parallel Processing, pp. 836–844 (1986)
4. Mellor-Crummey, J., Scott, M.: Algorithms for Scalable Synchronization on Shared Memory Multiprocessors. ACM Transactions on Computer Systems 9(1), 21–65 (1991)
5. N. A. S. Division, NAS Parallel Benchmarks Changes, `http://www.nas.nasa.gov/publications/npb_changes.html#url`
6. Jin, H., Frumkin, M., Yan, J.: The openmp implementation of nas parallel benchmarks and its performance. Tech. Rep. (1999)
7. Unnikrishnan, P., Shirako, J., Barton, K., Chatterjee, S., Silvera, R., Sarkar, V.: A practical approach to doacross parallelization. In: International European Conference on Parallel and Distributed Computing, Euro-Par (2012)
8. Padua, D.A.: Multiprocessors: Discussion of sometheoretical and practical problems. PhD thesis, Department of Computer Science, University of Illinois, Urbana, Illinois (October 1979)
9. Midkiff, S.P., Padua, D.A.: Compiler algorithms for synchronization. IEEE Transactions on Computers C-36, 1485–1495 (1987)
10. Shirako, J., et al.: Phasers: a unified deadlock-free construct for collective and point-to-point synchronization. In: ICS 2008: Proceedings of the 22nd Annual International Conference on Supercomputing, pp. 277–288. ACM, New York (2008)

11. Miller, A.: Set your Java 7 Phasers to stun (2008), http://tech.puredanger.com/2008/07/08/java7-phasers/
12. Su, H.-M., Yew, P.-C.: On data synchronization for multiprocessors. In: Proc. of the 16th Annual International Symposium on Computer Architecture, Jerusalem, Israel, pp. 416–423 (April 1989)
13. Tang, C.Z.P., Yew, P.: Compiler techniques for data synchronization in nested parallel loop. In: Proc. of 1990 ACM Intl. Conf. on Supercomputing, Amsterdam, Amsterdam, pp. 177–186 (June 1990)
14. Li, Z.: Compiler algorithms for event variable synchronization. In: Proceedings of the 5th International Conference on Supercomputing, Cologne, West, Germany, pp. 85–95 (June 1991)
15. Ding-Kai Chen, P.-C.Y., Torrellas, J.: An efficient algorithm for the run-time parallelization of doacross loops. In: Proc. Supercomputing 1994, pp. 518–527 (1994)
16. Lowenthal, D.K.: Accurately selecting block size at run time in pipelined parallel programs. International Journal of Parallel Programming 28(3), 245–274 (2000)
17. Manjikian, N., Abdelrahman, T.S.: Exploiting wavefront parallelism on large-scale shared-memory multiprocessors. IEEE Transactions on Parallel and Distributed Systems 12(3), 259–271 (2001)
18. Pan, Z., Armstrong, B., Bae, H., Eigenmann, R.: On the interaction of tiling and automatic parallelization. In: Mueller, M.S., Chapman, B.M., de Supinski, B.R., Malony, A.D., Voss, M. (eds.) IWOMP 2005 and IWOMP 2006. LNCS, vol. 4315, pp. 24–35. Springer, Heidelberg (2008)
19. Krothapalli, P.S.V.P.: Removal of redundant dependences in doacross loops with constant dependences. IEEE Transactions on Parallel and Distributed Systems, 281–289 (July 1991)
20. Chen, D.-K.: Compiler optimizations for parallel loops with fine-grained synchronization. PhD Thesis (1994)
21. Rajamony, A.L.C.R.: Optimally synchronizing doacross loops on shared memory multiprocessors. In: Proc. of Intl. Conf. on Parallel Architectures and Compilation Techniques (November 1997)

Manycore Parallelism through OpenMP
High-Performance Scientific Computing with Xeon Phi

James Barker[1] and Josh Bowden[2]

[1] Application Support, CSIRO Advanced Scientific Computing (IM&T)
james.barker@csiro.au
[2] Novel Technologies, CSIRO Advanced Scientific Computing (IM&T)
josh.bowden@csiro.au

Abstract. Intel's Xeon Phi coprocessor presents a manycore architecture that is superficially similar to a standard multicore SMP. Xeon Phi can be programmed using the OpenMP standard for shared-memory parallelism. We investigate the performance and optimisation of two real-world scientific codes, parallelised with OpenMP and accelerated on Xeon Phi, and compare with a conventional CPU architecture. We conclude that Xeon Phi offers the potential of significant speedup compared to conventional CPU architectures, much of which is attainable through the use of OpenMP.

1 Introduction

A defining trend of recent high-performance scientific computing has been the movement towards specialised architectures. Although transistor density gains and reductions in lithographic feature size have continued to occur at the pace predicted by Moore's Law, this is unlikely to continue indefinitely [1]; the rises in clock speed and general-purpose single-core performance that drove computing in the 1990s have ceased. Modern architecture design instead focuses on parallelism, demonstrated in the development of manycore *accelerator architectures*. The most notable of these is the graphics programming unit (GPU).

Since their uptake in high-performance computing in the mid-2000s, there has been a flood of research into the capabilities of these devices and their potential application to algorithms in various fields. The results have often been positive: GPUs have been used to achieve significant speedup over conventional CPU architectures in scientific applications such as the modelling of ion channels over the walls of the heart [2]. Nevertheless, despite advances in both architecture design and development tools, GPUs remain difficult to program effectively, and challenging to integrate into existing workflows.

Intel's Xeon Phi coprocessor architecture is the scion of an experimental line that traces back to the Larrabee prototype [3]. At first glance, Xeon Phi's manycore design resembles a conventional multicore chip, writ large. The cores on a Xeon Phi chip support both standard x86 instructions and a comprehensive set of SIMD instructions, and possess fully-coherent L1 and L2 caches (32K and 512K respectively).

A.P. Rendell et al. (Eds.): IWOMP 2013, LNCS 8122, pp. 45–57, 2013.

While GPUs have been programmed through specialised languages such as Nvidia's CUDA and Khronos' OpenCL, Xeon Phi can be programmed under several models, including OpenMP. The ability to run existing OpenMP codes on Xeon Phi, potentially without modification, is attractive. However, it is not clear whether performance benefits over conventional hardware are achievable through the use of OpenMP alone.

Here, we present a brief overview of the Xeon Phi architecture, as well as the software development environment that surrounds it. We then measure and analyse the performance of two scientific codes, parallelised for conventional CPU architectures with OpenMP and ported to Xeon Phi.

2 The Xeon Phi Architecture

The Xeon Phi architecture can be described as an homogenous manycore co-processor. It comprises up to 61 in-order dual-issue processing cores, each x86-capable and supplemented by a 512-bit vector processing unit (VPU). The cores are arranged around a bidirectional ring interconnect. Cores are clocked at up to 1.1GHz, and can fetch and decode instructions from four hardware thread contexts. No single thread context can issue back-to-back instructions in consecutive cycles; therefore, at least two threads must be resident per core for peak computational throughput. (All hardware and software specifications are drawn from [4] and [5] unless otherwise cited.)

The VPU attached to each core executes vector instructions drawn from a new 512-bit SIMD instruction set extending the Intel 64 ISA. The VPUs are compliant with the IEEE 754 2008 floating-point standard, for both single- and double-precision packed vectors. Intel emphasises that extracting peak application performance from Xeon Phi requires use of the vector unit.

Each Xeon Phi coprocessor is equipped with up to 16GB of off-chip GDDR5 RAM. The coprocessor maintains a separate 64-bit address space to the host system; as coprocessors are currently only available in a PCIe 2.0 form-factor, data must be transferred between the host and the device across the PCIe bus, which is a potential application performance bottleneck. Each core possesses a 32KB L1 instruction cache, a 32KB L1 data cache, and a shared 512KB L2 instruction/data cache. Cache lines are 64B; caches are fully coherent with main device memory. A core holding the requested data in cache may service a miss generated by any other core.

The theoretical peak performance of a Xeon Phi coprocessor is approximately 1.1 TFLOP/s in double-precision. Although it remains unclear how closely this can be approached by real-world application code, Intel has reported speeds up to 883 GFLOP/s running the DGEMM BLAS routine from its Math Kernel Library (MKL) [6].

3 The Xeon Phi Software Development Environment

A Xeon Phi coprocessor presents as a separate Symmetric Multi-Processing (SMP) device, loosely coupled to a host system over PCIe. Each device runs

a stripped-down Linux OS, and maintains an in-memory file-system. The Intel Composer XE compiler suite (for both C/C++ and FORTRAN) can cross-compile code on the host for execution on the device; Intel also provides tools for profiling and debugging. Intel's Math Kernel Library (MKL) provides numerical routines (including BLAS and LAPACK functions) optimised for the device.

The Intel compilers support the full set of directives and functionality defined in the OpenMP 3.1 standard; however, limitations and features of the device architecture mean that performance and scaling characteristics may differ from conventional architectures. Applications access the coprocessor through two main programming models: *native execution* and *offload execution*.

Natively-executed code is cross-compiled for the Xeon Phi architecture, and is executed exclusively on the device. OpenMP directives may be applied as usual. Codes with a high ratio of parallel to serial work are most likely to benefit from native execution, because serial sections of natively-executing code are limited to a single hardware thread context; a single device core is underpowered compared to a modern general-purpose CPU core, especially when restricted to an effective clock speed no higher than 550MHz.

For offload execution, the compiler generates "fat" binaries containing codes for both the host and the device. These are run simultaneously, with data and execution flow transitioning between the two. This allows serial sections to benefit from a modern CPU, and sufficiently-parallel problems to be accelerated on the coprocessor. Offloaded execution is either *automatic* or *manual*.

Under automatic offload mode, calls made to supported MKL functions are flagged by the compiler, and selectively offloaded to available devices at runtime. Although it requires no modification of code, automatic offload implies data movement to and from the device for each MKL call, which can be inefficient for smaller computations. Applications using MKL and exhibiting a high ratio of computation to data size stand to benefit the most from automatic offload.

Manual offloads are specified through *offload sections* – blocks of code, marked with the `offload` pragma, to be executed on a coprocessor[1]. Upon encountering an offload section on the host, execution flow transfers to the coprocessor, through the offloaded block, and then back to the host. Pragma clauses are used to specify variables and flat data-structures which must move to and/or from the device; data can persist on the device between offload sections. Directives exist for specifying asynchronous data movement and offload execution. OpenMP directives can be inserted into offload sections, and control parallel behaviour on the device only.

OpenMP thread/core affinity can be adjusted in both native and automatic offload modes using the `KMP_AFFINITY` environment variable, as made visible on the device. Three affinity presets are available: the familiar `compact` and `scatter`, and a new preset, `balanced`, which distributes threads as evenly as possible between cores, with contiguously-numbered threads as close as possible to each other under this constraint.

[1] Jeffers and Reinders note in [5] that specifications for OpenMP 4.0 may include `offload`-style semantics, as set forth in [7].

Optimisation for Xeon Phi requires attention to three key elements: parallel efficiency, cache/memory behaviour, and effective use of SIMD capabilities. These are also crucial to extracting peak performance out of modern multicore SMP systems, so the developer is in the unusual position of being able to optimise both host and device code simultaneously. An optimisation intended to target a performance characteristic on the device may produce similar or better performance improvements on the host, and vice versa.

4 Optimised Data-Parallelism in SOMA

The self-organising map (SOM) is a tool for unsupervised learning, introduced by Kohonen as a clustering and visualisation method for high-dimensional data [8]. A self-organising map embeds a two-dimensional mesh of nodes on a logical grid into an n-dimensional data space, and iteratively trains the mesh towards the set of sample data points. The trained mesh can be considered an approximation to a nonlinear manifold representing the topological distribution of the sample data, from which a number of desirable visualisations and metrics may be obtained. SOMs have been applied to many problems in high-dimensional statistics and machine learning, and several extensions have been made to the underlying algorithm [9]. In [10], Paini et al. investigate the exposure of the contiguous United States to invasive insect pest species. They use a SOM approach to cluster high-dimensional biotic presence/absence datasets, allowing calculation of the likelihood that pests will establish in a given state.

Most modern implementations of SOMs use the *batch-training algorithm*. Some number N of sample data points drawn from $\mathbb{R}^d$ are stored as rows $\boldsymbol{x}_i$ of an $N \times d$ data matrix $\boldsymbol{X}$. A SOM mesh geometry is determined, and a codebook matrix $\boldsymbol{C}$ of size $M \times d$ is generated, where M is the number of nodes to be placed on the SOM mesh and $\boldsymbol{c}_i^t$ is the embedded location, or *weight vector*, of the ith mesh node at timestep t. The number of nodes M is determined according to the desired purpose of the SOM, but is usually an order of magnitude smaller than N. The node weights are linearly initialised in data space on a grid swept out by the first two principal components of $\boldsymbol{X}$, with side lengths determined by the magnitude of the respective principal components.

The batch-training process itself is repeated for T training iterations, with two operations performed in each. First, the best-matching units (BMUs) of the various data points $\boldsymbol{x}_i$ are calculated. The BMU of a data point is the node weight that is "closest" to that point in data space at timestep t, and is defined as

$$\mathrm{BMU}(\boldsymbol{x}_i, t) = \boldsymbol{c}_j^t, \text{ such that } d(\boldsymbol{x}_i, \boldsymbol{c}_j^t) = \underset{k}{\operatorname{argmin}} \left\{ d(\boldsymbol{x}_i, \boldsymbol{c}_k^t) \right\}, \tag{1}$$

where $d(\cdot, \cdot)$ is an appropriate metric on $\mathbb{R}^d$, usually Euclidean distance. Once all BMUs have been calculated, the codebook vectors are updated to a weighted mean of the data points $\boldsymbol{x}_i$:

$$\boldsymbol{c}_i^{t+1} = \frac{\sum_{j=1}^{N} h_t(i, \mathrm{BMU}(\boldsymbol{x}_j, t)) \cdot \boldsymbol{x}_j}{\sum_{j=1}^{N} h_t(i, \mathrm{BMU}(\boldsymbol{x}_j, t))}, \tag{2}$$

where $h_t(i, j)$ is a *neighbourhood function* (usually Gaussian) between two map units i and j at iteration t, representing a sympathetic force between mesh nodes on the logical grid topology. The radius of the neighbourhood function decreases over time, aiding fine-scale convergence.

Paini et al. performed analyses for [10] using the SOM Toolbox package for the MATLAB numerical computing environment [11]. The SOM Toolbox allows initialisation, batch-training, and numerical and graphical investigation of SOMs. However, it was discovered during further work by Paini that the SOM Toolbox's initialisation and training algorithms scaled poorly to extremely large datasets. To address this, the SOMA (Self-Organising Maps, Accelerated) package was developed.

SOMA implements the batch-training algorithm in C99, with particular care paid to efficient and scalable memory usage. SOMA is intended as a companion tool for the SOM Toolbox, and uses data formats compatible with that package. SOMA offers two efficiency improvements over the SOM Toolbox. Firstly, the first two principal components of the data matrix that are used during initialisation are computed through the Non-linear Iterative Partial Least Squares (NIPALS) algorithm [12]. This algorithm, implemented using BLAS and LAPACK functions drawn from MKL, allows the efficient calculation of only the first two principal components; SOM Toolbox uses a standard eigensystem routine to calculate *all* principal components for the data, an approach which scales poorly.

Secondly, SOMA exposes data parallelism explicit in the SOM algorithm using OpenMP. This parallelism is obvious, and relatively coarse-grained: the BMUs for every data point can be calculated in parallel, as can updates to the rows of the codebook matrix. An abridged version of the main SOMA computation loop can be found in Fig. 1, demonstrating the basic OpenMP parallelisation scheme applied. Several other data-parallel loops external to the main loop were also parallelised using OpenMP.

As the majority of time during execution is spent in the main loop, SOMA represents an opportunity to evaluate the performance of Xeon Phi in native-execution mode. SOMA was initially recompiled for Xeon Phi, with no modification required to the code and the addition of only a single compiler flag to the makefile. However, profiling showed the potential for further optimisation. To remove the risk of false sharing and improve compiler optimisation, all dynamically-allocated memory was aligned to cache-line size, and matrix row dimensions were padded to cache-line boundaries. Additionally, a loop-tiling scheme was introduced to block data access to fit within cache, reducing the impact of cache misses on performance. These optimisations were also applicable, and were applied, to the original CPU version of the code.

SOMA uses a relatively simple nested loop-tiling scheme, applied separately to each of the parallelised sections in the main loop. Selecting the optimal tile sizes for a given dataset and map size proved non-trivial. Inner loops behaved best when blocked to half of cache size; CPU code performed best when inner loops were tiled for L1 cache (32KB on Sandy Bridge), while Xeon Phi inner

```
for (step = 0; step < nTimeSteps; step++) {

  // Calculate BMUs.
#pragma omp parallel for
  for (i = 0; i < nDataPoints; i++) {
    for (j = 0; j < nMapNodes; j++) {
      ...
    }
  }

  // Update map codebook vectors.
#pragma omp parallel for
  for (i = 0; i < nMapNodes; i++) {
    for (j = 0; j < nDataPoints; j++) {
      ...
    }
  }
}
```

Fig. 1. Main computation loop of SOMA, displaying the two inner sections and demonstrating the use of OpenMP for data parallelism over their outer loops. Pragma clauses are omitted for reasons of space and clarity.

loops were optimal when tiled for L2 cache (512KB). Outer loops were harder to tile, as the choice of tile size impacts on parallel decomposition. If an outer-loop tile size does not evenly divide the number of work items, some threads will be allocated one fewer tile than others. This load imbalance can have a substantial impact on parallel efficiency, especially when tile sizes are large and there are fewer tiles to distribute between threads (of which there may be hundreds). An heuristic method was used to select candidate tilings; these candidates were then trialled for a small number of iterations, and the best-performing adopted for the main training run.

Although the compiler's attempts at automatic vectorisation for Xeon Phi are reasonable, closer inspection indicated the emission of peel and remainder loops during the vectorisation of the two innermost loops. (That is, the calculation of Euclidean distance in (1), and accumulation into the weight vector $\boldsymbol{c}_i^{t+1}$ in (2).) Given the alignment and padding of rows in the codebook and data matrices, these constructions are unnecessary, and are a potential source of inefficiency. To ensure optimal use of the VPU, these two loops were manually vectorised for Xeon Phi using intrinsic functions. Manual vectorisation reduces the ability of the compiler to automatically perform some optimisations; in particular, manually-vectorised loops required unrolling by hand, and prefetching behaviour needed aggressive tuning.

5 Mixed Data- and Task-Parallelism in FDTD-GPR

The Finite Difference in the Time Domain (FDTD) method was first described in a seminal paper by Yee [13]. It provides a second-order method for discretising Maxwell's equations for the time-dependent electromagnetic field, through the adoption of a finite difference scheme over a spatially-staggered pair of grids representing the electric and magnetic fields on a bounded domain, with a leapfrog

method used for integration in time. FDTD is a well-understood and widely-studied algorithm that has been applied to many problems in electromagnetics.

An important application of the FDTD method is the modelling of electromagnetic transmission behaviour in non-dispersive media, such as the behaviour of ground-penetrating radar (GPR) signals. The most common form of GPR transmits an electromagnetic pulse into the ground, which is reflected at interfaces between layers with contrasting dielectric properties and then received on the surface. The received signal can be used to image and evaluate subsurface material properties and geometry; uses of GPR include archaeological surveying, buried landmine detection, and snow thickness evaluation [14].

In [14], Strange describes a system for the application of low-powered GPR to the estimation of coal seam geometry, using a pattern-recognition approach to extract feature vectors from and thereby classify segmented GPR signal data. The pattern-recognition system is trained using synthetic data generated by the FDTD-GPR (FDTD for Ground-Penetrating Radar) software, a two-dimensional implementation of the FDTD method. FDTD-GPR models signal propagation through specified subsurface geometries, according to the tranverse-magnetic form of the FDTD update equations. Mur's absorbing boundary conditions [15] are applied to the edges of the grid. Pulse output from the GPR transmitter is modelled using a wavelet function, perturbed according to timestep resolution.

FDTD-GPR was originally implemented in the MATLAB numerical computing environment, and has since been ported to C++ to enable greater performance. An optimised OpenCL kernel has also been developed, employing mixed-precision arithmetic to enable effective use of texture caches on GPU devices. The main FDTD computational loop is given in Fig. 2. The `CalculateInternal(...)` and `CalculateMagnetic(...)` subroutines of the main loop implement the FDTD update equations across the interior of the electric field domain and the whole of the magnetic field domain respectively; the computational costs of these routines scale as $O(mn)$, where m and n are the height and width of the discretised grid. OpenMP `parallel for` directives expose data parallelism in these subroutines. Although the $O(mn)$ routines demonstrate good parallel performance, the surrounding serial code (which includes the expensive allocation and population of 2D pointer arrays) in the main computational loop remains a barrier to overall speedup, in accordance with Amdahl's Law.

Since model initialisation requires a large amount of serial work, and typical use of the software often involves many separate model calculations, FDTD-GPR is a poor candidate for Xeon Phi native execution. Instead, the main computational loop was marked as an offload section. As the device maintains a separate memory space to the host, pointer values used to access data structures were recalculated inside the offload section. Although the $O(mn)$ routines in the main loop exhibited adequate parallel behaviour, the lower-order routines surrounding them consumed a disproportionately large amount of runtime, even considering their negative impact on conventional architectures.

```
for (t = 0; t < nTimeSteps; t++) {

  // Store information from time (t-1).
  // O(1) and O(m + n) respectively.
  SaveCorners(...);
  SaveWalls(...);

  // Update the electric field inside domain boundaries. O(mn).
  CalculateInternal(...);

  // Perturb the source wavelet. O(1).
  UpdateSource(...);

  // Update the electric field on domain boundaries, using
  // absorbing boundary conditions. O(m + n) and O(1).
  CalculateWalls(...);
  CalculateCorners(...);

  // Update the magnetic field. O(mn).
  CalculateMagnetics(...);
}
```

Fig. 2. The main computational loop of the FDTD-GPR system, annotated to display the computational cost of the various subroutines. m and n are grid height and grid width respectively.

To address memory allocation inefficiencies in main loop subroutines, an inlined version of the main loop was developed. As well as the existing OpenMP `parallel for` directives, a task-based approach was used to minimise the serial span of the main loop, and thereby increase overall main loop parallel efficiency. Small logically-independent tasks within the various lower-order routines were exposed through the use of the OpenMP `omp task` directive; for example, update calculations on domain boundaries can be decomposed into eight separate subtasks: one for each of four walls, and one for each of four corners. Correctness was enforced using the `omp taskwait` directive.

6 Results

We measured the performance of the SOMA and FDTD-GPR codes, both on Xeon Phi and on a conventional CPU platform. All Xeon Phi performance results were obtained using pre-production hardware[2]. All CPU-specific results were obtained using dual Intel Xeon E5-2650 processors on a dual-socket cluster compute node, for a total of 16 Sandy Bridge cores clocked at 2.0GHz. Although reference against a single CPU core is interesting, it is also important to consider the performance of the compute node as a whole; production runs of shared-memory parallel scientific software generally make use of as many threads as are usefully available on a node, and an accelerator architecture is of particular interest if it can outperform their collective efforts.

[2] Precise Xeon Phi specifications are as follows. Silicon: B0. SW level: gold. Cores: 61 @ 1090909kHz. Memory: 7936MB @ 2750000kHz. Power consumption: 300W, active cooling.

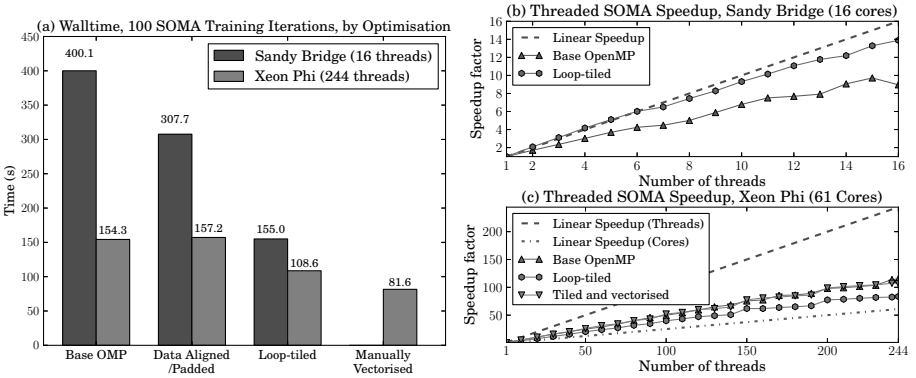


Fig. 3. Performance figures for SOMA

SOMA's serial initialisation routines performed roughly an order of magnitude slower on the device than on the compute node. The MKL routines in the NIPALS initialisation component made good use of the device, delivering comparable performance to the compute node (operating 16 threads). Figure 3(a) displays the walltime taken to execute 100 iterations of SOMA over a representative test case by both compute node and device, under the optimisations described in Sect. 4: the base OpenMP SOMA code, fitted with, cumulatively, data alignment and padding, loop-tiling, and (in the case of Xeon Phi) manual vectorisation. Manually-vectorised code for Xeon Phi was unable to be executed unmodified on Sandy Bridge; preliminary experiments with manually-vectorised code targeting Sandy Bridge have failed to produce performance benefits, suggesting that automatic vectorisation is more effective on Sandy Bridge. We found that compact thread affinity consistently produced the best results at all stages of optimisation, for both Xeon Phi and Sandy Bridge, as it maximises cache-sharing between contiguously-numbered threads. The use of balanced thread affinity produced results that were comparable with or slightly worse than compact affinity, depending on the number of threads used; the use of scatter affinity produced a significant drop in performance for all numbers of threads.

Although all SOMA optimisations were targeted at Xeon Phi, the relative benefit to CPU code was sometimes greater. Most notably, the introduction of data alignment/padding, which is cited by Intel as a critical optimisation, produced no benefit on Xeon Phi, while driving a 30% increase in performance on Sandy Bridge. (Effort spent on data-alignment did, however, allow further optimisation through manual vectorisation.) Loop-tiling was similar: Sandy Bridge obtains a 50% improvement through loop-tiling compared to the data-aligned version, while Xeon Phi experiences only 29% improvement. The manually-vectorised version of SOMA offers the best performance, outperforming the Sandy Bridge blade by 1.9x, which translates to a speedup of 27x versus a single Sandy Bridge CPU core running loop-tiled SOMA. Achieving this performance gain requires substantial additional code: compared to the original OpenMP code, the size of the SOMA main loop increased from around 40 LOC to around 800 LOC.

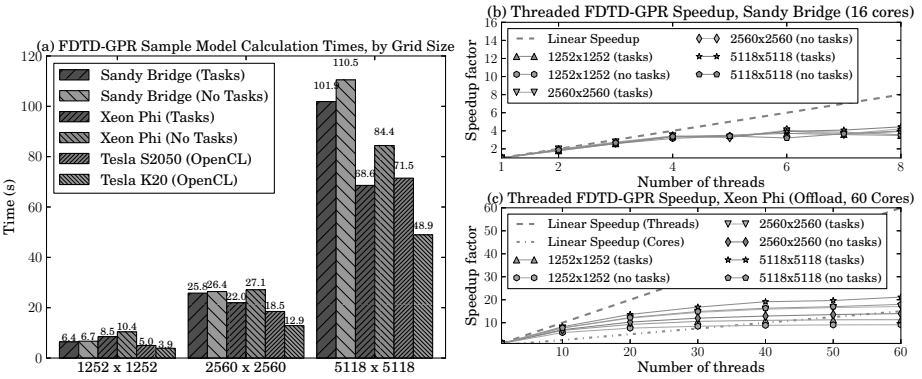


Fig. 4. Performance figures for FDTD-GPR

Figures 3(b) and 3(c) display the observed speedup of both the original OpenMP, loop-tiled, and loop-tiled and vectorised (Xeon Phi only) versions of SOMA operating different numbers of threads, on Sandy Bridge and Xeon Phi respectively. Figure 3(c) includes reference lines for linear speedup measured both in the number of threads, and the number of cores (calculated as number of threads divided by four). The original OpenMP code scales adequately on Sandy Bridge, but is improved substantially by the addition of loop-tiling, approaching linear speedup. The introduction of loop-tiling on Xeon Phi appears to negatively impact the scaling characteristics of the code; however, scaling behaviour with manual vectorisation is near-identical to that of the unoptimised OpenMP code. This illustrates a challenge of measuring speedup on Xeon Phi, namely the difficulty of obtaining a reference single-thread timing; different optimisations may help or hinder the single-thread case, which can potentially skew speedup results. As manual vectorisation only impacts the speed of calculation of inner loops, we have no reason to expect it to scale so differently to the automatically-vectorised loop-tiling code. Nevertheless, Fig. 3(c) demonstrates that all versions of the code scale roughly linearly in the number of Xeon Phi cores used; this implies that while optimisations such as manual vectorisation can decrease walltime, good parallel scaling behaviour is available through the use of OpenMP only.

Parallel code that exhibits poor scaling and performance characteristics on the compute node generally behaves similarly, or worse, on the device. Figures measuring the performance and scaling of the main computation loop of FDTD-GPR over three sample grid sizes are provided in Fig. 4(a); reference timings are also provided for the calculation of these test cases using an optimised mixed-precision OpenCL kernel executing on two Nvidia GPUs, a Tesla S2050 (Fermi generation) and a Tesla K20 (Kepler generation) respectively. As the OpenCL kernel has been aggressively tuned for GPU hardware, its application to Xeon Phi was not considered. We found that balanced core affinity produced best results. Xeon Phi achieves speedup of 1.49x over the Sandy Bridge node for

the 5118 x 5118 case, comparable with the Fermi-generation GPU; we note in passing that the effort required to port and optimise the FDTD algorithm for OpenCL was significantly greater than for Xeon Phi.

However, Xeon Phi and the full Sandy Bridge node execute the main FDTD-GPR computation loop only 7x and 4.4x faster than a single Sandy Bridge core respectively, limited by serial sections in the main loop, as noted in Sect. 5. This inefficiency is obvious in the threaded speedup graphs for Sandy Bridge and Xeon Phi, given in Figs. 4(b) and (c); the unoptimised OpenMP version scales poorly in the number of threads on the compute node, and this remains true when the main computation loop is offloaded to the device. (Speedup was tested up to 16 threads on Sandy Bridge, and up to 240 threads on Xeon Phi; however, as the speedup behaviour remains flat at higher number of threads, Figs. 4(b) and (c) are clipped at 8 and 60 threads respectively in the interest of readability.) Nevertheless, the device does offer improved performance, particularly for the largest grid; larger problem sizes expose more work to Xeon Phi threads, improving the performance of the two `parallel for` regions relative to the serial sections. We anticipate that problems with even larger grid sizes will see increased benefit from Xeon Phi.

The tasking approach used to minimise serialisation inside the loop provides performance benefit, although it does not substantially adjust the parallel scaling properties of the code (cf. Figs. 4(b) and (c)). This is particularly true on Xeon Phi, where the application of tasking reduces runtime by 19% for the 5118 x 5118 case (cf. Fig. 4(a)). This suggests that algorithms that contain serial sections may be able to maximise acceleration from the device, by ensuring that the serial span of the algorithm is minimised through task-based parallelism.

7 Conclusions

Intel's Xeon Phi coprocessor offers a compelling proposition: a powerful and relatively general-purpose accelerator architecture that can execute existing OpenMP codes with minimal or no modification. This stands in contrast to other accelerator architectures, such as GPUs, which offer comparable power to Xeon Phi but are more challenging to port existing codes to. Although further optimisation is required to extract maximum benefit from Xeon Phi, the optimisation process is relatively straightforward, and will be familiar to those experienced with conventional SMP architectures. Standard OpenMP parallel directives for data- and task-parallelism expose much of the power available to Xeon Phi, and (given a sufficiently-large problem size) scale as well or better than on a conventional CPU architecture.

We ported two real-world scientific codes to Xeon Phi, using the offload and native execution programming models; both displayed significant performance benefits relative to execution on a conventional CPU system. The codes were simultaneously optimised for both Xeon Phi and a conventional CPU architecture, which allows us to be confident that this performance is not available to the CPU. We conclude that Xeon Phi offers the potential for significant speedup,

relative to both a single CPU core and a modern multicore SMP compute node, attainable through the effective use of OpenMP. We anticipate that further research into high-performance computing with Xeon Phi will clarify the benefit of particular optimisation techniques for the device.

Acknowledgements. The SOMA and FDTD-GPR packages are the intellectual properties of Dr. Dean Paini (CSIRO Ecosystem Sciences) and Dr. Andrew Strange (CSIRO Earth Science and Resource Engineering) respectively. Funding for development and optimisation of SOMA and FDTD-GPR was drawn from the CSIRO Computational and Simulation Sciences platform. Xeon Phi hardware was provided by Intel; technical support was provided by Sam Moskwa (Novel Technologies), and Brian Davis and Steve McMahon (Cluster Services), CSIRO Advanced Scientific Computing (IM&T).

References

1. Mack, C.A.: Fifty Years of Moore's Law. IEEE Transactions on Semiconductor Manufacturing 24(2), 202–207 (2011)
2. Sadrieh, A., Mann, S.A., Subbiah, R.N., Domanski, L., Taylor, J.A., Vandenberg, J.I., Hill, A.: Quantifying the Origins of Population Variability in Cardiac Electrical Activity through Sensitivity Analysis of the Electrocardiogram. J. Physiol. (April 2013); Epub ahead of print
3. Seiler, L., Carmean, D., Sprangle, E., Forsyth, T., Abrash, M., Dubey, P., Junkins, S., Lake, A., Sugerman, J., Cavin, R., Espasa, R., Grochowski, E., Juan, T., Hanrahan, P.: Larrabee: a Many-Core x86 Architecture for Visual Computing. In: ACM SIGGRAPH 2008 papers. SIGGRAPH 2008, pp. 18:1–18:15. ACM, New York (2008)
4. Intel Corporation: Intel Xeon Phi Coprocessor System Software Developers Guide (April 2013)
5. Jeffers, J., Reinders, J.: Intel Xeon Phi Coprocessor High-Performance Programming. Elsevier Inc. (2013)
6. Huck, S.: Intel Xeon Phi Product Family Performance (April 2013)
7. Stotzer, E., Beyer, J., Das, D., Jost, G., Raghavendra, P., Leidel, J., Duran, A., Narayanaswamy, R., Tian, X., Hernandez, O., Terboven, C., Wienke, S., Koesterke, L., Milfeld, K., Jayaraj, A., Dietrich, R.: OpenMP Technical Report 1 on Directives for Attached Accelerators. Technical report, OpenMP Architecture Review Board (November 2012)
8. Kohonen, T.: Self-Organized Formation of Topologically Correct Feature Maps. Biological Cybernetics 43(1), 59–69 (1982)
9. Kohonen, T.: Self-Organizing Maps, 3rd edn. Springer Series in Information Sciences. Springer (2001)
10. Paini, D.R., Worner, S.P., Cook, D.C., De Barro, P.J., Thomas, M.B.: Threat of Invasive Pests From Within National Borders. Nat. Commun. 1(115) (2010)
11. Vesanto, J., Himberg, J., Alhoniemi, E., Parhankangas, J.: Self-Organizing Map in MATLAB: the SOM Toolbox. In: Proceedings of the MATLAB DSP Conference, vol. 99, pp. 16–17 (1999)

12. Wold, S., Esbensen, K., Geladi, P.: Principal Component Analysis. Chemometrics and Intelligent Laboratory Systems 2(1-3), 37–52 (1987); Proceedings of the Multivariate Statistical Workshop for Geologists and Geochemists
13. Yee, K.: Numerical Solution of Initial Boundary Value Problems involving Maxwell's Equations in Isotropic Media. IEEE Transactions on Antennas and Propagation 14(3), 302–307 (1966)
14. Strange, A.: Robust Thin Layer Coal Thickness Estimation Using Ground Penetrating Radar. PhD thesis, School of Engineering Systems, University of Queensland (March 2007)
15. Mur, G.: Absorbing Boundary Conditions for the Finite-Difference Approximation of the Time-Domain Electromagnetic-Field Equations. IEEE Transactions on Electromagnetic Compatibility EMC-23(4), 377–382 (1981)

Performance Characteristics of Large SMP Machines

Dirk Schmidl[1,3], Dieter an Mey[1,3], and Matthias S. Müller[1,2,3]

[1] Center for Computing and Communication, RWTH Aachen University, D - 52074 Aachen
[2] Chair for High Performance Computing, RWTH Aachen University, D - 52074 Aachen
[3] JARA High-Performance Computing, Schinkelstraße 2, D 52062 Aachen
{schmidl,anmey,mueller}@rz.rwth-aachen.de

Abstract. Different types of shared memory machines with large core counts exist today. Standard x86-based servers are build with up to eight sockets per machine. To obtain larger machines, some companies, like SGI or Bull, invented special interconnects to couple a bunch of small servers into one larger SMP, Scalemp uses a special software layer on top of a standard cluster for the same purpose. There is also a trend to couple many small and simple cores into one chip, like in the Intel Xeon Phi chip. In this work we want to highlight different performance attributes of these machine types. Therefor we use some kernel benchmarks to look at basic performance characteristics and we compare the performance for real application codes. We will show different scaling behaviors for the applications which we explain with the use of the kernel benchmarks used before.

1 Introduction

OpenMP is probably the most widely used paradigm for shared memory parallelization in high performance computing. It is often said to be easy to use and in fact it is often easy to get a first loop parallel version of an application, but getting good performance is often more difficult due to the complex design of large shared memory machines. Especially for a larger number of threads good scaling can in many cases only be achieved if the underlying hardware is taken into account.

Hardware vendors have established different types of shared memory machines. Standard servers, based on x86 processors exist with up to 8 sockets in a single machine. E.g. SGI and Bull invented a special interconnect to combine smaller two or four socket machines into one larger shared memory machines. Scalemp provides a software layer, called vSMP foundation, to couple several servers with a standard Infiniband network into one machine running a single OS instance. A different approach to provide the ability to run hundreds of threads is used in Intel's new Many Integrated Core architecture. Here, a lot of small and simple cores are combined in one single chip. The first product in this architecture line is the Intel Xeon Phi coprocessor. The Xeon Phi chip resides in a PCIe extension card and can on the one hand be used as an accelerator to speedup applications and on the other hand it is able to execute standalone executables, since it runs a full operating system instance.

Given this variety of different machines, it is hard for a programmer to choose the appropriate machine for his application. It is also difficult for computing centers to decide

A.P. Rendell et al. (Eds.): IWOMP 2013, LNCS 8122, pp. 58–70, 2013.

on a machine type when a new machine is going to be purchased. Since, it is often hard to get excess to all these machine types to try out the performance on a given application, we want to present a comparison of the basic differences of these machine types. We investigate performance attributes on a standard 8 socket HP server, a SGI Altix ultraviolet, a Bull BCS machine, which both use a special network to couple smaller machines into a larger single system, a Scalemp machine which uses a software for the same purpose and a Xeon Phi extension card which can run a lot of threads on one single chip. We will look at basic characteristics of the memory subsystem, investigate the influence of memory allocation and initialization and run several applications on all platforms to highlight relevant differences between these machines.

The rest of this paper is structured as follows: First, we present related work in section 2 and describe the systems used in section 3. Then, we look at basic performance relevant attributes by means of kernel benchmarks in section 4, before we compare the performance of applications in section 5. After all, we draw our final conclusions in section 6.

2 Related Work

Comparing the performance of shared memory machines is subject to investigations for many years. Many benchmarks or benchmark suites exist to investigate attributes of standard machines. The Stream benchmark [7] for example is widely used to measure the memory bandwidth of a machine and the LMbench benchmark suite [8] offers kernels to measure certain operating system and machine properties. Other benchmarks like the EPCC benchmark [4] concentrate on the performance of the OpenMP runtime, which of course is essential for the scalability of OpenMP applications.

Besides these very basic benchmarks which can give a good indication on specific machine or software attributes, also application benchmarks exist. The SPEC OMP Benchmark Suite [1] delivers a collection of representative OpenMP applications. Other benchmark suites like the Barcelona OpenMP Task Suite [5] or the NAS parallel benchmarks [2], contain relevant application kernels. These benchmarks can be used to compare the performance of different architectures for representative application codes.

These projects focus on the benchmark techniques, also they provide reference results for several architectures. In this work we will focus more on the differences of the architecture, also we reuse some of the ideas provided in the benchmarks mentioned above.

3 Architecture Description

3.1 HP ProLiant

The HP ProLiant DL980 G7 server used for our experiments is a single server equipped with eight Intel Xeon X6550 processors. All processors are clocked at 2 GHz and connected to each other through the Intel Quick Path interconnect. Every processor contains a memory controller attached to 32 GB of main memory, making this server a ccNUMA machine with a total of 256 GB of memory.

3.2 SGI Altix UltraViolet

The SGI Altix UV system consists of several two socket boards, each equipped with two Intel Xeon E7- 4870 10-core processors clocked at 2.4 GHz. All of these boards are connected with SGIs NUMALink interconnect into a single shared memory machine. Since on one board the cache-coherence is established directly over the QPI, whereas the NUMALink network is needed for different board, this machine is a hierarchical NUMA machine, with different cache-coherency mechanisms on different hierarchical levels. The total machine used in our tests has 2080 cores and about 2 TB of main memory. All of our tests were done on up to 16 processors during batch operation of the system. For a better comparison with the 8-core processors used in the other systems, all tests were done using only eight of the ten available cores on each socket.

3.3 BCS

The BCS system consists of four bullx s6010 boards. Each board is equipped with four Intel Xeon X7550 processors and 64 GB of main memory. The Intel Quick Path Interconnect combines the four sockets to a single system and the Bull Coherence Switch (BCS) technology is used to extents the QPI to combine four of those boards into one SMP machine with 128 cores and 256 GB of main memory. So, this system is also a hierarchical NUMA system.

3.4 Scalemp

The Scalemp machine investigated here consists of 16 boards, each board is also equipped with four Intel Xeon X7550 processors clocked at 2 GHz and 256 GB of main memory. The boards are connected via a 4x QDR InfiniBand network, where every board is connected via two HCAs. Thus, from a hardware point of view this is an ordinary (small) cluster. The innovative part of the machine is the vSMP software of the company Scalemp, which runs below the operating system and creates a Single System Image on top of the described hardware. The virtualization layer of the processors and the InfiniBand network is used by the vSMP software to create cache-coherency on a per page basis and to allow remote memory access between all the boards. A partition of the main memory is reserved by the vSMP software to run different caching and prefetching mechanisms automatically in the background, as well as a page-based memory migration mechanism. These mechanisms do not only move pages on access, they can also adjust the home node of memory pages if frequently used on a remote node. This is a notable difference to standard x86-based non-uniform memory architectures (NUMA), like the Altix or BCS machine, where page migration needs to be done by the user, if possible at all. From a user point of view the machine looks like a single Linux machine with 64 eight-core processors and about 3.7 TB of main memory. About 300 GB of the available memory are used by the vSMP software internally for caching. Linux sees 64 NUMA nodes, each containing about 64 GB of main memory.

3.5 Intel Xeon Phi

The Intel Xeon Phi coprocessor is based on the concepts of the Intel Architecture (IA) and provides a shared-memory many-core CPU that is packed on a PCI Express

extension card. The version used here has 60 cores clocked at 1.053 GHz and offers full cache coherency across all cores with 8 GB of GDDR5 memory. A ring network connects all cores with each other and with memory and I/O devices. Every core supports 4-way Hyperthreading, which allows the system to run up to 240 threads in parallel. The comparably small amount of main memory is attached to the Xeon Phi Chip as one NUMA node. Thus, the system is a 240-way parallel system with a uniform memory architecture, which is another difference to the other machines with are large NUMA systems.

The Xeon Phi card used in out experiments was plugged into a host system with two Intel Xeon E5 processors. For all of our experiments we used the host system only to cross compile the executables, which were copied and executed stand-alone on the Xeon Phi. This procedure gives us insight in the performance attributes of the chip, independent from the programming model used. Of course comparing one extension card with complete systems is an uneven comparison, but for sure we will see standalone systems with hundreds of cores in the near future and the Phi might give evidence on the behavior of such systems.

4 Performance Characteristics

Before we look into the performance of application codes, it is useful to understand in detail the differences of the investigated architectures. We use several benchmarks to highlight certain properties of the machines.

4.1 Serial Memory Bandwidth

Many scientific applications are limited by the memory bottleneck in modern systems. Therefore, the memory bandwidth is the most important factor for these applications.

We measured the read and write bandwidth with one thread on these machines. Since the trends for read and write bandwidth were nearly identical with the only difference that the write bandwidth was slightly slower on all machines, we concentrate on the write bandwidth during the following discussion. The simple benchmark writes a single value to an array of a given size several times and calculates the reached bandwidth. On the NUMA machines, i.e. HP, Altix, BCS and Scalemp, we measured the bandwidth to NUMA nodes on all NUMA levels (local, on the same board and on a remote board). We used the `numactl` tool provided by Linux to influence the used core and NUMA node. On the Xeon Phi machine only one NUMA level exists, so the placement was not varied here.

Figure 1 illustrates the reached write bandwidths for increasing memory footprints. On all machines we can see a typical cache behavior. For very small data sizes the bandwidth is poor, since complete cache lines need to be written. Then the bandwidth rises for memory sizes that fit into the caches. On the Xeon Phi a peak of 4 GB/s is reach, on the other systems a bandwidth of about 14-18 GB/s is achieved. When the memory sizes exceed the capacity of the last level caches, the memory bandwidth drops down significantly. On the Xeon Phi system the `Standard` curve displays the bandwidth reached, when the code was just cross compiled for the system. The Intel Compiler allows to provide a switch to insert software prefetch instructions into the binary.

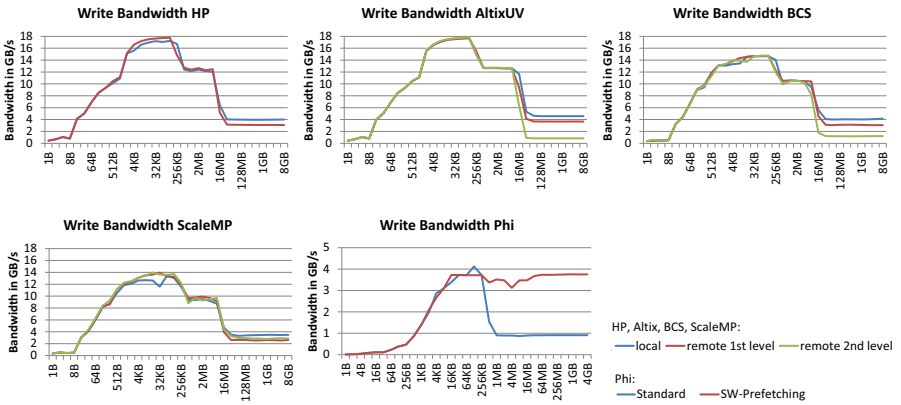


Fig. 1. Write Bandwidth in GB/s of the HP, Altix, BCS, Scalemp and Xeon Phi system for different memory footprints

The downside of this approach is, that the user needs to specify exactly the range that should be prefetched. When we instruct the Compiler to prefetch 64 elements ahead for the L2 cache and 8 for the L1 cache, we can improve the reached bandwidth for writing as depicted by the `SW-prefetching` line. These results show, that no temporal store operations are used by the compiler and that all cachelines are first read before they are written, so prefetching can have a positive effect. Here, nearly the full L2 cache bandwidth can be reached for arbitrary data sizes, as long as they fit into the 8 GB of memory.

On the other systems these switches did not work. On these systems we observe a bandwidth reduction to about 4-5 GB/s if the memory is located in the local NUMA node and to about 3 GB/s if a NUMA node on the same board is used. If the memory is located on a different board, we can observe an interesting difference between the Altix and BCS machine on the one hand and the Scalemp machine on the other hand. The bandwidth on Altix and BCS descents to 0.5 GB/s whereas it stays at 3 GB/s on the Scalemp machine. The reason therefor is, the caching inside of the vSMP software. The memory pages can be kept in a board local cache and thus only data on the local board is affected by the benchmark. This mechanism can help to achieve a good data locality, even if the data is spread across the system and no memory migration is performed by the user.

4.2 Distance Matrix

Of course the remote bandwidth to different sockets depends on the distance of the sockets and the used interconnect. In [10] we described a way to measure and present the distance between sockets in a distance matrix. Basically, we measure the bandwidth between all sockets and scale the matrix in a way that the upper left element has the value 10 and the others are scaled in a way that decreasing bandwidth between two sockets results in increasing distance. Figure 2 shows the distance matrices for the HP

and BCS machine. For the ScaleMP and Altix machine the tests were not possible, since we could not get the machine exclusively and on the Phi machine the measurements are useless, since only one socket exists.

Socket	0	1	2	3	4	5	6	7
0	10	10	17	13	18	18	18	18
1	10	10	17	13	18	18	18	18
2	17	17	10	11	18	18	18	18
3	17	17	10	11	19	19	18	18
4	18	18	18	18	10	10	17	17
5	18	18	18	18	10	10	17	17
6	18	18	18	18	17	17	10	10
7	18	19	18	18	17	17	10	9

Socket	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	10	13	13	13	57	57	57	57	59	59	59	59	57	57	57	
1	13	10	13	13	56	55	56	56	56	56	56	55	55	56	56	55
2	14	13	10	13	58	58	58	58	56	56	56	56	58	58	58	58
3	13	13	13	10	56	55	56	55	56	56	56	55	56	55	56	55
4	56	56	56	56	10	13	13	13	56	56	56	57	58	58	58	58
5	55	55	55	55	13	10	13	13	55	55	55	55	56	56	56	55
6	58	58	58	59	13	13	10	13	58	58	58	58	56	56	56	57
7	56	55	56	55	13	13	13	10	56	56	56	56	56	56	56	56
8	58	58	58	58	56	57	56	56	10	13	13	13	56	56	56	56
9	56	56	55	55	55	55	55	55	13	10	13	13	55	55	56	55
10	56	56	56	56	58	58	58	58	13	13	10	13	58	58	58	58
11	56	56	56	55	56	56	56	55	13	13	13	10	56	56	56	56
12	56	56	56	56	58	58	58	58	56	57	56	56	10	13	13	13
13	55	55	55	55	56	56	55	55	56	55	55	55	13	10	13	13
14	58	58	58	58	56	56	56	56	58	58	58	58	13	13	10	13
15	56	56	56	56	56	56	56	56	56	56	56	56	13	13	13	10

Fig. 2. Distance matrix of the HP (left) and the BCS system (right). The matrix is scaled such that the upper left value is always ten, larger numbers indicate higher distances.

Obviously, there are differences between both systems. On the HP board there are always two sockets which seem to be connected very fast, whereas the other sockets have distances between 17 - 19. On the BCS system all accesses on one board are between 10 and 13, which is faster than most of the connections on the one HP board. All connections using the BCS chip, are significantly slower, here a distance of 55 - 59 is reached. So, the BCS machine can provide cache coherence over a larger number of cores, but some performance regressions comes along with this.

4.3 Parallel Memory Bandwidth

Parallel applications of course use more than one core at a time. Thus, the total bandwidth for a parallel application is important. We modified the benchmark from section 4.1 to work with several threads on an array and measure the read and write bandwidth. We use a compact thread binding on all of the machines. So, we first fill up cores and sockets with the maximum number of threads, before we use the next core or socket.

Figure 3 shows the bandwidth for an increasing number of threads on the different platforms for a memory footprint of 16 MB per thread. On the Intel Xeon Phi machine the maximum bandwidth of about 130 GB/s for reading and 60 GB/s for writing can be achieved with about 120 threads. Beyond this, the bandwidth stagnates. On the NUMA systems the bandwidth rises with the number of sockets used and does not stagnate at all. Here, the read bandwidth is higher than the write bandwidth available for all systems. Noticeable is, that the BCS machine reaches an 5-10 % higher maximum bandwidth compared to the Scalemp machine, also the same underlying hardware is used. The reason therefor is the vSMP software layer which inserts a slight overhead for memory management and leads to a small reduction in the available bandwidth.

4.4 Memory_Go_Around

The parallel bandwidth measured before is the optimal bandwidth that is reached, when all threads work on their own data. In many algorithms, some data sharing is required

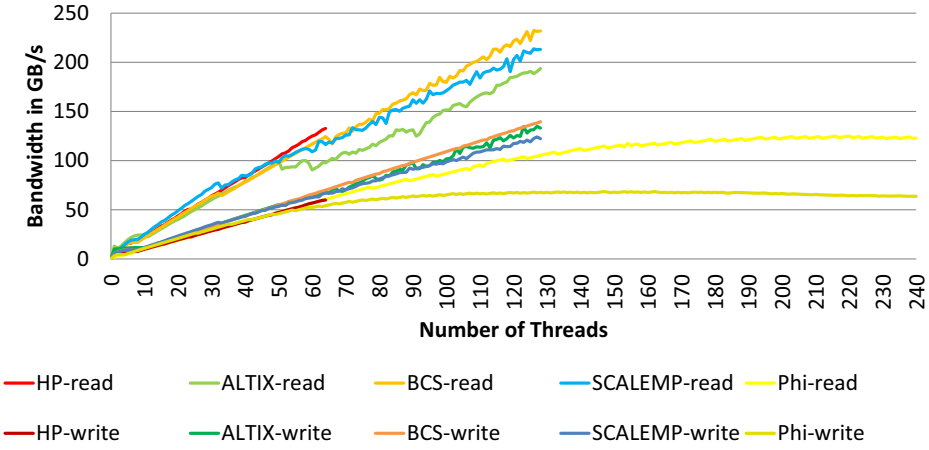


Fig. 3. Parallel read and write bandwidth on the HP, Altix, BCS, Scalemp and Xeon Phi systems for an increasing number of threads

which leads to a certain amount of necessary remote accesses. To investigate the drop down in the reached bandwidth with remote accesses, we modified the bandwidth benchmark in a way, that it no longer works only on local data.

The modified benchmark, we call it `memory_go_around`, works in $n+1$ steps for n threads. In step zero, every thread initializes its own data and measures the memory bandwidth to access it. In step one, all threads work on the data of their right neighbor, so thread t works on memory initialized by thread $(t+1)\ mod\ (n-1)$, as exemplified in figure 4. As before, we place threads in a way that neighboring threads are as close as possible to each other, meaning that there is a high chance that they run on the same NUMA node or board. Hence, the number of remote memory accesses rises for the first $n/2$ steps. Then the number shrinks again. In step $n-1$ every thread works on the memory of the left neighbor and in step n again only local accesses occur.

Figure 5 shows the result for 64 Threads on the HP, 120 threads on the Xeon Phi and for 128 threads on the Altix, BCS and Scalemp machine. Since the Xeon Phi machine has only one NUMA node, the data is always stored on this NUMA node and the reached bandwidth is the same in all steps, about 130 GB/s. On the other machine the bandwidth declines for the first half of the steps and then rises up again. Of course this is related to the increasing number of remote accesses and the increasing distance between these accesses. On the HP system, the performance drops down from about 120 to 60 GB/s. So, with maximum traffic over the QPI bus, still 50% of the peak bandwidth can be reached. On the Altix and BCS machine the drop down is from about 250 to 18 or 8 GB/s which is 6% or 3% of the available maximum bandwidth. On the Scalemp the drop down is even higher, here only 0.5 GB/s are reached, which means roughly 0.2% of the maximum bandwidth. Overall, the bandwidth goes down on all hierarchical NUMA machines (Altix, BCS and Scalemp) significantly, this is one of the biggest differences in comparison to single systems with one level of NUMAness, like the HP machine. However, if an application does not require a lot of data sharing between the threads, proper data placement can avoid these problems.

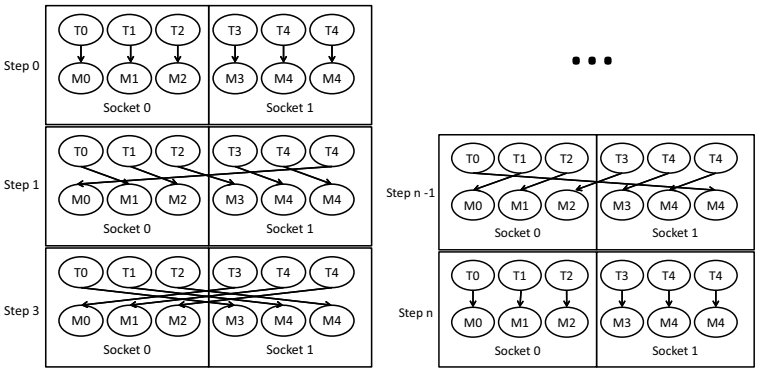


Fig. 4. The memory_go_around benchmark forks in n+1 steps. In the first step the memory of the right neighbour is used to measure the bandwidth, in the next step the memory of the next neighbour and so on. This increases the distance between thread and memory in every step, until half of the steps are done, then the distance decreases until it reaches zero in the last step.

4.5 Synchronization Overhead

Besides the memory bandwidth, locks are often critical for the performance of an application on larger shared memory systems. Locks can occur explicitly, like calls to `omp_set_lock`, or they can happen inside of other calls, e.g. a call to `malloc` requires synchronization. We measured the overhead of these two mentioned routines. The results are shown in the left part of table 1 for all investigated systems. For memory allocation the Intel OpenMP runtime provides a version of `malloc` which is optimized for multithreading, called `kmp_malloc`. We investigated this version as well, the results are shown in the bottom right part of the table.

To measure the overhead of explicit locks we used the `syncbench` out of the EPCC microbenchmarks [4]. Our `malloc` test calls `malloc` or `kmp_malloc` on all threads simultaneously 1000 times and computes the average duration of one call. The overhead of OpenMP Locks increases with the number of threads involved on all platforms. On the HP, Altix, BCS and Scalemp machine it is nearly the same, as long as only a small number of threads is involved, but for 64 and 128 threads the overhead rises more drastically on the Scalemp machine, where the maximum is about 35 microseconds, what is high compared to the 1 - 3 microseconds on the other machine. On the Xeon Phi system, the overhead for 120 threads is nearly the same on the other hardware based systems. However, the overhead with one thread is higher, due to the slower serial threads. Overall, the ratio between serial and parallel execution is advantageous on the Phi system. The overhead for a lock goes up for a factor of 5 for 120 threads (0.4 to 2 microseconds) whereas for example it goes up by a factor of about 27 on the BCS system (0.06 to 1.64). This means, that the scaling of lock based applications on the Phi is better than on the other system.

For the `malloc` calls, the overhead rises much faster with the number of threads on all systems, than for the OpenMP locks. E.g on the Altix machine it starts at 3 microseconds and goes up to about 20,000. On the other machines the behavior is nearly the same for a large number of threads.

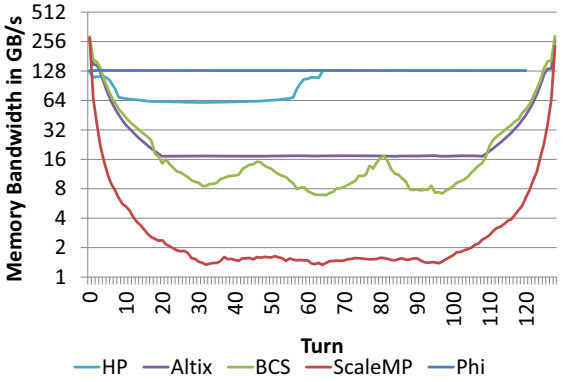


Fig. 5. Bandwidth measured with the memory_go_around benchmark for n+1 steps with n threads on the HP, Altix, BCS, Scalemp and Xeon Phi system

The `kmp_malloc` calls introduce less overhead on most of the machines. On the Scalemp machine the overhead is smaller for a medium number of threads, but higher for 128 threads compared to ordinary `malloc` calls. On the Xeon Phi machine the overhead is significantly higher for all numbers of threads, e.g. with 175,641 microseconds compared to 26,603 for 120 threads.

Another point where synchronization might be needed, which is not always obvious to the programmer is, when data is initialized. The operating system needs to establish a unique mapping between virtual and physical addresses which might require lock. Therefor we implemented a test which initializes 2 GB of data and measures the bandwidth reached during initialization. Table 1 in the upper right part shows the reached bandwidth on all systems. Noticeable is, that the bandwidth rises on all systems for a small numbers of threads. Of course this is due to the higher memory bandwidth which seems to be the bottleneck for only a few threads. On the HP, Altix and BCS machine the bandwidth goes up until the end of 64 / 128 threads is reached. However, on the Scalemp machine the bandwidth drops down as soon as more than 32 threads are involved, meaning as soon as more than one physical board is used. The overhead of the vSMP software seems to slow down the initialization of memory whereas the hardware based solution in the HP, Altix and BCS system does not. On the Phi system, the bandwidth goes slightly down for 120 threads. However, we have seen before, that the total memory bandwidth also goes down at the end, so this is not a surprise here.

Overall, the overhead for locking seems to work better on the hardware based solutions. The vSMP software seems to introduce a significant overhead, so it is much more important to avoid extensive synchronization, e.g. though many `malloc` calls, on the Scalemp machine.

5 Application Case Studies

Finally, we want to look at the performance of two applications from the RWTH Aachen University and compare the results on the different systems.

Table 1. Overhead of OpenMP locks, calls to malloc and calls to kmp_malloc on the investigated machines were measured in microseconds and initialization times to initialize 2 GB of data was measured in GB/s. 30, 60 and 120 threads were used on the Xeon Phi, 32,46 and 128 on the other machines.

	OMP Locks					**Initialization**				
#Threads	HP	ALTIX	BCS	SCMP	PHI	HP	ALTIX	BCS	SCMP	PHI
1	0.03	0.05	0.06	0.07	0.40	1.42	1.38	1.31	1.15	0.67
32/30	0.97	3.29	0.62	0.99	1.77	16.70	18.30	18.36	15.07	17.73
64/60	1.07	3.72	1.04	24.36	1.94	32.93	33.70	34.24	4.86	23.12
128/120		2.99	1.64	35.78	2.01		72.10	67.98	3.63	19.32
	malloc					**kmp_malloc**				
#Threads	HP	ALTIX	BCS	SCMP	PHI	HP	ALTIX	BCS	SCMP	PHI
1	3.61	3.46	4.30	63.64	15.27	3.04	2.60	3.63	45	981
32/30	6075	5146	4902	6887	11023	411	558	530	546	37211
64/60	12470	10473	14007	13882	19807	1476	2646	2786	12260	88717
128/120		21030	29552	28702	26603		11742	12958	54959	175641

5.1 NestedCP

NestedCP [6] is developed at the Virtual Reality Group of the RWTH Aachen University and is used to extract critical points in unsteady flow field datasets. Critical points are essential parts of the velocity field topologies and extracting them helps to interactively visualize the data in virtual environments.

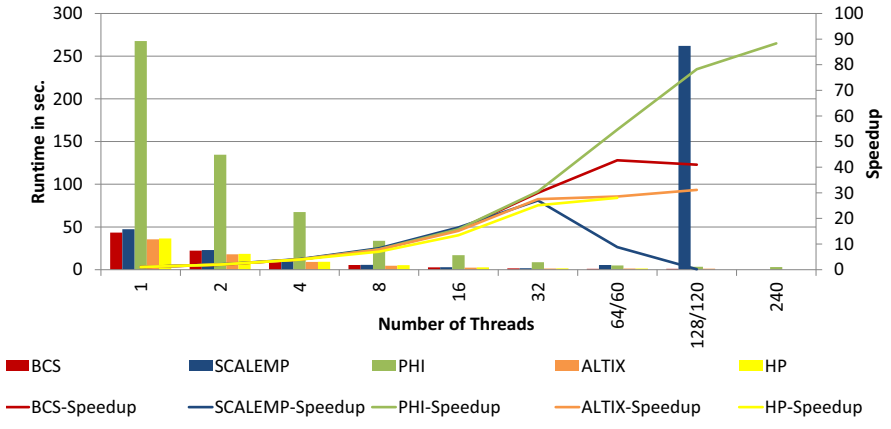


Fig. 6. Peformance and Speedup of the NestedCP code

Figure 6 shows the runtime and speedup of NestedCP on all platforms. We used a code version parallelized with OpenMP tasks for our experiments. Apparently the runtime of the Code on the Xeon Phi is significantly slower than on the other systems. This is due to the slower clockrate and the simple structure of the cores. However, the

scalability is best on the Xeon Phi system. With 240 threads a speedup of about 90 can be reached. On the BCS system, a speedup of 45 can be observed for 64 threads. On the HP system and the Altix machine, a slightly worse speedup can be observed. This is due to the slightly better single threaded runtime on these systems. But, on the HP and Altix machine, the code scales until 60 or 128 threads are used, whereas the performance slightly drops down on the BCS machine at the end. On the Scalemp machine the scalability is much worse. Here, the performance rises as long as only one board is used and drops down significantly with 64 and 128 threads.

This behavior is exactly what we have seen for the synchronization mechanisms on the mentioned machines. The Xeon Phi system has the worst serial performance but a better scaling behavior than the other machines. The HP system provides a good scaling behavior. On the Altix and BCS system the performance goes down, when more than one board is used, but not as significant as on the Scalemp machine. Our assumption is, that the NestedCP code is limited by locking routines and therefore the mentioned scaling behavior is observed.

5.2 TrajSearch

The second code investigated here is TrajSearch. TrajSearch is a code to investigate turbulences which occur during combustion. It is a post-processing code for dissipation element analysis developed by Peters and Wang [9] from the Institute for Combustion Technology at the RWTH Aachen University. It decomposes a highly resolved 3D turbulent flow field obtained by Direct Numerical Simulation (DNS) into non-arbitrary, space-filling and non-overlapping geometrical elements called 'dissipation elements'. Starting from every grid point in the direction of ascending and descending gradient of an underlaying diffusion controlled scalar field, the local maximum and minimum point are found. A dissipation element is defined as a volume from which all trajectories reach the same minimum and maximum point.

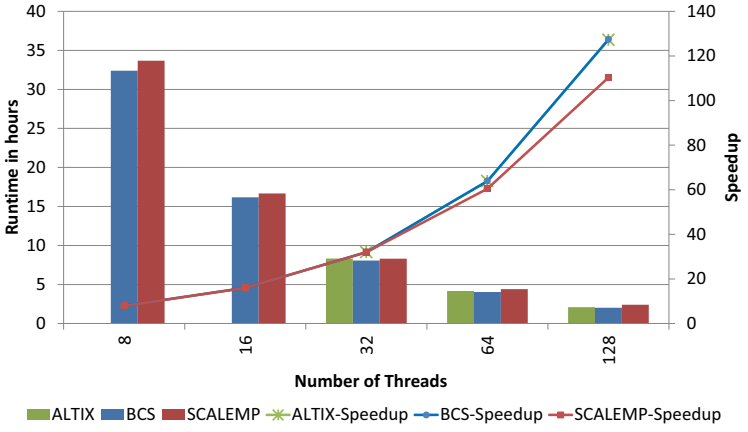


Fig. 7. Runtime and scaleability of the TrajSearch code on the Altix, BCS and Scalemp machine

We reduced locking and optimized the data placement of this code to gain good performance on the Scalemp machine, see [3] for details. Figure 7 shows the runtime and performance of TrajSearch on the Altix, BCS and Scalemp machine. The memory available for the Xeon Phi did not suffice to store the dataset, so the Phi system is not taken into account for this comparison and the HP machine was not available for a time slot large enough to do this tests. Although the code was optimized for the Scalemp system, it scales slightly better on the Altix and BCS machine. It reaches a speedup of about 127 on the Altix and BCS and about 110 on the Scalemp machine for 128 threads. This indicates, that the tuning steps done for the Scalemp machine were also useful for the Altix and BCS machine. However, the code reaches a noticeable speedup, even on the Scalemp machine, which makes all three machines suitable for execution.

6 Conclusion

We investigated performance attributes of several large SMP systems. One standard 8-socket server from HP, an SGI Altix UV, a system based on the Bull Coherence Switch, a Scalemp system and the Intel Xeon Phi chip. We showed, differences in the memory bandwidth on all systems, e.g. when the vSMP software cache has positive influence. Furthermore we investigated the performance influence of remote accesses with the help of the `memory_go_around` benchmark. The benchmark showed, that the negative influence is significantly higher on the Altix, BCS and Scalemp machine than on the HP server and that there is no influence on the Xeon Phi system. Synchronization and locks had a bad influence on all systems, when the number of threads rises. On the Xeon Phi the best ratio of locking overhead between serial and parallel execution could be observed.

This behavior was also reflected in the NestedCP code, where we observed a good scaling on the HP and Xeon Phi, a slightly worse scaling on the Altix and BCS and a bad scaling on the Scalemp machine. However, the TrajSearch code investigated at the end scaled well on all investigated systems, the Altix, the BCS and the Scalemp system.

Overall, if a code is optimized and does not need many locking routines, it can perform well on all investigated systems. If locking or remote accesses cannot be avoided, there is a high chance that a code scales best on the Phi system, followed by the one level NUMA machine and then followed by the hierarchical NUMA machines. However, non-hierarchical NUMA machines are typically limited to 8 sockets or less, whereas the hierarchical machines allow the use of 16 sockets in case of the BCS machine and several hundred sockets for the Altix UV and the ScaleMP machine. So, if the application scales well hierarchical NUMA machines offer an tremendous amount of shared memory compute resources.

Of course the scaling behavior of a system is important for the application performance, but the single core performance is equally important for the overall runtime. All investigated systems besides the Xeon Phi system use comparable Xeon Processors which deliver nearly the same single thread performance, so the differences in scaling directly reflect the overall performance. On the Xeon Phi the single thread performance is worse compared to the Xeon based systems, so the overall runtime of an application might still be worse on this system, even if it scales well.

Acknowledgement. Some of the Tests were performed with computing resources granted by JARA-HPC from RWTH Aachen University under project jara0001. Parts of this work were funded by the German Federal Ministry of Research and Education (BMBF) under Grant No. 01IH11006. The authors also like to thank the LRZ in Munich for the provided compute time on the SGI Altix system.

References

1. Aslot, V., Domeika, M., Eigenmann, R., Gaertner, G., Jones, W.B., Parady, B.: SPEComp: A New Benchmark Suite for Measuring Parallel Computer Performance. In: Eigenmann, R., Voss, M.J. (eds.) WOMPAT 2001. LNCS, vol. 2104, pp. 1–10. Springer, Heidelberg (2001)
2. Bailey, D.H., Barszcz, E., Barton, J.T., Browning, D.S., Carter, R.L., Fatoohi, R.A., Frederickson, P.O., Lasinski, T.A., Simon, H.D., Venkatakrishnan, V., Weeratunga, S.K.: The NAS parallel benchmarks. Technical report, NASA Ames Research Center (1991)
3. Berr, N., Schmidl, D., Göbbert, J.H., Lankes, S., Mey, D., Bemmerl, T., Bischof, C.: Trajectory-Search on ScaleMP's vSMP Architecture. In: Applications, Tools and Techniques on the Road to Exascale Computing: Proceedings of the 14th Biennial ParCo Conference, ParCo 2011, Advances in Parallel Computing, Ghent, Belgium, vol. 22. IOS Press, New York (2012)
4. Bull, J.M.: Measuring Synchronisation and Scheduling Overheads in OpenMP. In: Proceedings of First European Workshop on OpenMP, pp. 99–105 (1999)
5. Duran, A., Teruel, X., Ferrer, R., Martorell, X., Ayguade, E.: Barcelona OpenMP Tasks Suite: A Set of Benchmarks Targeting the Exploitation of Task Parallelism in OpenMP. In: International Conference on Parallel Processing, ICPP 2009, pp. 124–131 (2009)
6. Gerndt, A., Sarholz, S., Wolter, M., Mey, D.A., Bischof, C., Kuhlen, T.: Nested OpenMP for Efficient Computation of 3D Critical Points in Multi-Block CFD Datasets. In: SC 2006 Conference, Proceedings of the ACM/IEEE, p. 46 (November 2006)
7. McCalpin, J.: STREAM: Sustainable Memory Bandwidth in High Performance Computers (1999), `http://www.cs.virginia.edu/stream` (accessed March 29, 2012)
8. McVoy, L., Staelin, C.: lmbench: Portable Tools for Performance Analysis. In: Proceedings of the 1996 Annual Conference on USENIX Annual Technical Conference, ATEC 1996, Berkeley, CA, USA, p. 23. USENIX Association (1996)
9. Peters, N., Wang, L.: Dissipation element analysis of scalar fields in turbulence. C. R. Mechanique 334, 493–506 (2006)
10. Schmidl, D., Terboven, C., an Mey, D.: Towards NUMA Support with Distance Information. In: Chapman, B.M., Gropp, W.D., Kumaran, K., Müller, M.S. (eds.) IWOMP 2011. LNCS, vol. 6665, pp. 69–79. Springer, Heidelberg (2011)

Evaluating OpenMP Tasking at Scale for the Computation of Graph Hyperbolicity*

Aaron B. Adcock[1], Blair D. Sullivan[2], Oscar R. Hernandez[2], and Michael W. Mahoney[1]

[1] Department of Mathematics
Stanford University, Stanford, CA 94305
aadcock@stanford.edu,mmahoney@cs.stanford.edu
[2] Computer Science and Mathematics Division,
Oak Ridge National Laboratory, Oak Ridge, TN 37831
{sullivanb,oscar}@ornl.gov

Abstract. We describe using OpenMP to compute δ-hyperbolicity, a quantity of interest in social and information network analysis, at a scale that uses up to 1000 threads. By considering both OpenMP workshare and tasking models to parallelize the computations, we find that multiple task levels permits finer grained tasks at runtime and results in better performance at scale than worksharing constructs. We also characterize effects of task inflation, load balancing, and scheduling overhead in this application, using both GNU and Intel compilers. Finally, we show how OpenMP 3.1 tasking clauses can be used to mitigate overheads at scale.

1 Introduction

Many graph analytics problems present challenges for thread-centric computing paradigms because the dynamic algorithms involve irregular loops, where special attention is needed to satisfy data dependencies. Perhaps better suited is a tasking model, where independent units of work can be parceled out and scheduled at runtime. OpenMP, the de facto standard in shared memory programming, originally targeted worksharing constructs to coordinate distribution of computation between threads. In the OpenMP 3.0 specification, this model was extended to include tasks, and additional tasking features, such as `mergeable` and `final`, were added in 3.1. The task-based model allows asynchronous completion of user-specified blocks of work, which are scheduled to the threads at runtime to achieve good load balance. The tasking model of OpenMP also solves the problem of dealing with multiple levels of parallelism in the application. For example, tasks may spawn child tasks in complex nested loops that cannot be parallelized with OpenMP worksharing constructs. OpenMP 3.1 enables the programmer to control task overhead via the `task final` and `if` clauses, and to reduce the

* This manuscript has been authored by a contractor of the U.S. Government under Contract No. DE-AC05-00OR22725. Accordingly, the U.S. Government retains a non-exclusive, royalty-free license to publish or reproduce the published form of this contribution, or allow others to do so, for U.S. Government purposes.

A.P. Rendell et al. (Eds.): IWOMP 2013, LNCS 8122, pp. 71–83, 2013.

data environment size with the `mergeable` clause. These features operate by managing the overhead of creating tasks at runtime and can easily be used to control the parallelism of the applications. Also, in OpenMP, the programmer is responsible for laying out and placing the memory correctly for shared data structures to achieve good data locality and avoid task inflation [1] overheads.

Clearly, it is of continuing interest to evaluate OpenMP tasking at scale in the context of challenging real-world applications where loop-level parallelism creates significant load-imbalances between threads. In this paper, we work with one such application, the calculation of the δ-hyperbolicity of a graph. The δ-hyperbolicity is a number that captures how "tree-like" a graph is in terms of its metric structure; and thus it is of interest in internet routing, complex network analysis, and other hyperbolic graph embedding applications [2–5]. The usual algorithm to compute δ involves looping over all quadruplets of nodes; its $\Theta(n^4)$ running time presents scalability challenges, and its looping structure creates serious load balancing problems.

Our main contribution is to describe challenges we encountered while using OpenMP 3.1 to calculate exactly, on a large shared-memory machine, the δ-hyperbolicity of real-world networks. The networks have thousands of nodes, and the experiments used up to 1015 threads. We evaluate both worksharing and tasking implementations of the algorithm, demonstrating improved performance using multilevel tasking over OpenMP worksharing clauses. We also evaluate and compare the performance of GNU and Intel compilers with regards to task scheduling and load balance at scale. Finally, we show that performance gains can be made at very large scale by improving data structures, adding tasking levels, and using the `task final, if`, and `mergeable` clauses to manage overheads.

2 Background and Preliminaries

2.1 Gromov δ-hyperbolicity

The concept of δ-hyperbolicity was introduced by Gromov in the context of geometric group theory [6], and has received attention in the analysis of networks and informatics graphs. We refer the reader to [2–5, 7, 8], and references therein, for details on the motivating network applications; but we note that our interest arose as part of a project to characterize and exploit tree-like structure in large informatics graphs [8]. Due to the $\Theta(n^4)$ running time of the usual algorithm for computing δ, previous work resorted to computing δ only for very small networks (with up to hundreds of nodes [7]) or involved sampling triangles in networks (of up to 10,000 nodes [4]). In our application, we needed to compute δ exactly for networks that were as large as possible.

Let $G = (V, E)$ be a *graph* with vertex set V and unordered edge set $E \subseteq V \times V$, and assume G has no *self-loops*, i.e., if $u \in V$, $(u, u) \notin E$. We refer to $|V|$ as the *size* of the graph. A *path* of length l is an alternating sequence of vertices and edges $v_1 e_1 v_2 \ldots e_l v_{l+1}$ such that $e_k = (v_k, v_{k+1})$ and no vertex is repeated. A graph is *connected* (which we will always assume) if there exists a path between all vertices. We define a function $l : V \times V \to \mathbb{Z}^+$ that equals the length of the

shortest path between $u, v \in V$. This function defines a metric on G, creating a metric space (G, l), and a *geodesic* is a shortest path in G. A *geodesic triangle* is composed of three vertices and a geodesic between each vertex pair.

There are several characterizations of δ-hyperbolic spaces, all of which are equivalent up to a constant factor [6]. We tested the computation of three such definitions as candidates for parallelization: δ-slim triangles [6], δ-fat triangles [7], and the 4-point condition [6]. Except for a brief discussion in Section 4.1 of other notions of δ, in this paper we will only consider the following definition.

Definition 1. *Let (X, l) be a metric space, and let $0 \leq \delta < \infty$. (X, l) is called* 4-point δ-hyperbolic *if and only if for all $x, y, u, v \in X$, ordered such that $l(x, y) + l(u, v) \geq l(x, u) + l(y, v) \geq l(x, v) + l(y, u)$, the following condition holds:*

$$(l(x, y) + l(u, v)) - (l(x, u) + l(y, v)) \leq 2\delta.$$

Thus, the 4-point condition requires sets of *four* points (called *quadruplets*) to have certain properties, and these can be checked by looping over all quadruplets.

2.2 OpenMP and Parallel Computations

There are several task parallel languages and runtime libraries that have been used to parallelize graph applications [9]. OpenMP task parallelism is a profitable approach for dynamic applications because it provides a mechanism to express parallelism on irregular regions of code where dependencies can be satisfied at runtime. Studies [10–12] have shown that OpenMP tasks are often more efficient for parallelizing graph-based applications than thread-level parallelism because it is easier to express the parallelism on unstructured regions while leaving the task scheduling decisions to the runtime. However, such studies do not include applications with large numbers of threads on production codes. Additional work has shown that load imbalances, scheduling overheads and work inflation (due to data locality) can adversely affect the efficiency of task parallelism at scale [1]. These sources of overhead need to be mitigated carefully in applications, especially at large scale. OpenMP 3.1 provides mechanisms to manage some of these overheads by allowing work stealing with the `untied` clause to improve load balance, reducing the memory overheads by merging the data environment of tasks with the `mergeable` clause, and by reducing the task overhead with the specification of undeferred and included tasks via the `if` and `final` clauses.

In dynamic and irregular applications, it is difficult to know the total number of tasks and granularity generated at runtime and how this affects synchronization points and overheads. Controlling task granularity is important to reduce runtime overhead and improve load balance — e.g. if the tasks generated are too fine grained, the application will lose parallel efficiency due to runtime overheads. Few studies [13] have evaluated the use of the `final` and `mergeable` clauses to manage runtime overheads on large graph-based applications running on hundreds of threads. These can further be combined with the task `cutoff` technique: when the `cutoff` threshold is exceeded, newly generated tasks are serialized.

Different techniques have been explored [12], including the use of adaptive cut-off schemes [14] and iterative chunking [15].

3 Algorithm for Computing δ and Its Implementation

3.1 The Four-Point Algorithm

To describe the algorithm for computing δ on a graph $G = (V, E)$ of size n, we represent V as a set of integers, i.e., $V = \{0, 1, 2, \ldots, n-1\}$. We precompute the distance matrix l using a breadth first search and store it in memory; the graph itself is not needed after l is constructed. We then let $\delta(i, j, k, p)$ represent the hyperbolicity of a quadruplet and Δ be a vector where $\Delta[\delta]$ is the number of quadruplets with hyperbolicity δ. Given an ordered tuple of vertices (i, j, k, p), we let ϕ be a function re-labelling them as (x, y, u, v) so that $l(x, y) + l(u, v) \geq l(x, u) + l(y, v) \geq l(x, v) + l(y, u)$. Then, to calculate the 4-point δ-hyperbolicity of G, we use a set of nested `for` loops and loop over all vertices satisfying $0 \leq i < j < k < p < n$ to find

$$\delta(i, j, k, p) = (l(x, y) + l(u, v)) - (l(x, u) + l(y, v)) \text{ s.t. } (x, y, u, v) = \phi(i, j, k, p). \quad (1)$$

These quantities are recorded by incrementing $\Delta[\delta(i, j, k, p)]$.

Clearly, this algorithm is naturally parallelizable, since for each set of four vertices, the δ calculation (which occurs in the inner-most loop) depends only on the distances between the nodes (and not on the calculated δ of any other quadruplet). One must be slightly careful to avoid conflicts or contention when storing values in the Δ vector, but this can be alleviated by allocating thread-local storage for Δ and summing on completion to achieve the final distribution. It is important to note that we require $0 \leq i < j < k < p < n$ to reduce total work by a factor of 24 (since δ of a quadruplet is independent of the ordering). This, however, has a significant effect on the load balancing of the loops. The number of iterations of each `for` loop is dependent on the index in the previous loop, and decreases as we progress through the calculation. With four levels of nested loops, this effect becomes very pronounced for later iterations.

3.2 OpenMP Implementations

We implemented two versions of this algorithm in OpenMP, both using the Boost Graph Library to store the graph as an adjacency list. The first approach (Code 1.1) makes use of the `omp for` workshare construct on the outer loop. The innermost loop consists of a straightforward retrieval of the distances between the six different pairings of each quadruplet and the calculation of Eqn (1). Due to the load balancing issues described previously, we obtain a significant speedup using dynamic (instead of static) scheduling, especially with smaller chunksizes (see Table 1(b)). After the loop, we use a short critical region to collate the local Δ vectors into a single master Δ.

The second approach (Code 1.2) implements parallelization using multiple levels of tasking to split the computations into smaller chunks (with the intent of balancing the load given to each processor). We determined two levels of

tasking was optimal—three or more resulted in massive overheads for generating/maintaining the tasks, increasing time by an order of magnitude. Figure 1 shows the task graph associated with this approach, when processing a network with n nodes, and it illustrates why load balancing is such a challenge. Each task is labelled with the vertices it sweeps over in the network and, e.g., the 1st level task $(1, *, *, *)$ (on the left) has $n-1$ child tasks, which in total has $O(n^3)$ iterations of computation, but its sibling task $(n-3, *, *, *)$ (on the far right) generates only a single child which has a single iteration of work.

```
/*Distance matrix precalculated*/
#pragma omp parallel shared(Delta[])
{
    /* Variable initializations */
#pragma omp for schedule(dynamic,1)
    for(size_t i=0; i<size; ++i)
       for(size_t j=i+1; j<size; ++j)
          for(size_t k=j+1; k<size; ++k)
             for(size_t p=k+1; p<size; ++p)
                /*calculate delta(i,j,k,p) as in Eq. (1)*/
#pragma omp critical
      /* Collate local Deltas */
}
```

Code 1.1. Parallelization using the `for` construct

The critical region in these implementations may seem to be a bottleneck for the computation, but because of the complexity of the main loop, the small size of the δ vectors (on the order of graph diameter), and the linear nature of the collation, the runtime of this region is small relative to the total runtime. Our empirical results support this analysis—e.g., the critical region took less than one second on all runs using 1015 threads.

```
/*Distance matrix precalculated*/
#pragma omp parallel shared(Delta,Delta_ptr)
{
   /* Variable initializations */
   vector<double> Delta_loc(diam_of_network,0);
   int thread_id = omp_get_thread_num();
   Delta_ptr[thread_id] = &Delta_loc;
#pragma omp single
   {
      for(size_t i=0; i<size; ++i) //Task level 1
#pragma omp task shared(Delta_ptr,distance_matrix)
         for(size_t j=i+1; j<size; ++j) //Task level 2
#pragma omp task shared(Delta_ptr,distance_matrix)
            for(size_t k=j+1; k<size; ++k)
               for(size_t p=k+1; p<size; ++p)
                  /*Get local Delta vector*/
                  int tn = omp_get_thread_num();
                  vector<double> &loc_Delta = *Delta_ptr[tn];
                  /*calculate delta(i,j,k,p) as in Eq. (1)*/
   }
#pragma omp critical
      /* Collate local Deltas */
}
```

Code 1.2. Parallelization using two levels of tasking

In the worksharing case, the details of writing/collating the Δ vectors are straightforward. With tasking, the situation is more complex, as the thread that

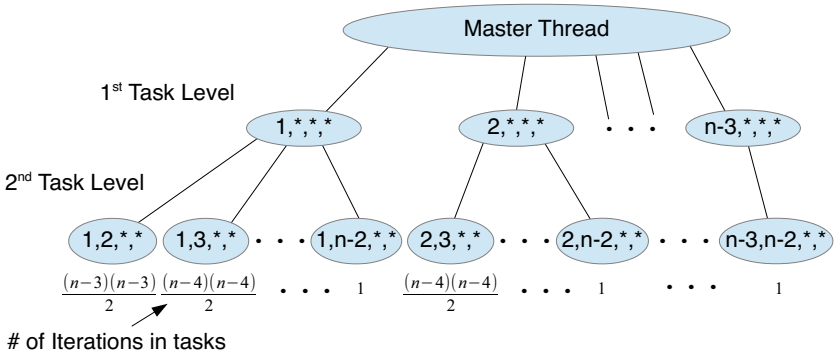


Fig. 1. The task graph of a network with n nodes

generates the first task may not be the same thread that executes the subsequent levels of tasking. As each lower level task takes on the memory space of the task above it, we would have threads writing to the local Δ vector of other threads (i.e., the threads that generated the upper level tasks). Related to this locality issue, multiple threads could be writing to the same local Δ vector, depending on how the tasks are passed to the threads. To avoid this, we create a shared array of N pointers, where N is the number of threads and pointer i points to a local copy of Δ for thread i. Then, when we write out δ values, we first check which thread is executing the task and use the shared pointer array to find the appropriate Δ to update.

4 Empirical Evaluation and Main Results

In this section, we describe the results of our implementation of the algorithms of Section 3. We considered four networks (Polblogs, CA-GrQc, as20000102, and Gnutella09; the last three are from `http://snap.stanford.edu`, and the first is from [16]) of interest in social network analysis. These networks were chosen to represent a range of sizes (1222 to 8104 vertices) where $\Theta(n^4)$ is feasible in a parallel environment, but too large for serial codes.

Our computations were performed using Nautilus, an SGI Altix UV 1000 system at the National Institute for Computational Science (NICS) consisting of 1024 Intel Nehalem EX processor cores and 4 terabytes of shared memory. Each core has a speed of 2.0 GHz and the machine's peak performance is 8.2 Teraflops. As eight of the cores are reserved for system operations, only 1016 cores can be used for a single job. We performed our experiments up to 1015 threads, leaving one core for helper threads or the operating system. The system runs on SUSE 11.1 and Propack 7SP1. The Altix **dplace** command was used to bind threads to cores. We used Intel 11.1 and GNU 4.6.3 to evaluate task scalability, and the newer GNU 4.7.3 to evaluate the new OpenMP 3.1 tasking features at the end of Section 4.3. Newer versions of the Intel compiler (12.1 and 13.x) experienced a massive runtime slowdown which prevented completion

Table 1. Representative Running Times (in seconds)

(a) Timing of Three δ Definitions

Definition of δ	Time (96 threads)
δ-slim	2910
δ-fat	1187
4-point δ	111

(b) Timing of Scheduling Policies

	Dynamic		Static	
Chunksize	1	10	1	10
128 threads	23851	19901	34854	46384
256 threads	17359	20705	22456	25546

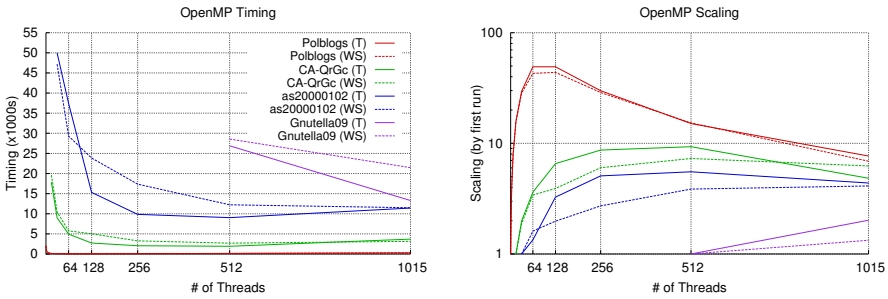
of jobs using even the smallest of our test networks, which we believe can be attributed to calls to the Boost Graph Library.

4.1 An Aside: Comparison of Different δ Definitions

Both the δ-slim and δ-fat triangle-based definitions of δ (see [6, 7] for precise definitions) restrict computations to triplets of nodes, but they require us to compute, store, and then check the distances between nodes on each side of a geodesic triangle. Representative timings for computations based on all three definitions using straightforward worksharing implementations are presented in Table 1(a). The more than an order of magnitude improvement for computations based on the 4-point condition are largely because the data structures needed to track all of the shortest paths between points, as required by the triangle-based definitions of δ, are not needed for the 4-point condition.

4.2 Comparison of Tasking versus Worksharing

Our initial evaluation of the tasking feature of OpenMP pitted it against the worksharing approach on the GNU compiler. For each of the four networks, we ran the algorithms in Codes 1.1 and 1.2 repeatedly, starting with a single thread, repeatedly doubling the number of threads until we reached the hardware limit. The results are presented in Figure 2 and Table 2, where missing values are due to a wall-clock limit of 24 hours on Nautilus, preventing completion of jobs. Since the single thread job did not complete for all networks, in Figure 2, we present scaling relative to the "first run," meaning the timing of the execution with the smallest number of threads which completed in under 24 hours (e.g., Gnutella is relative to a 512 thread run). Smaller numbers in the table correspond to faster timings, and these results clearly indicate that—as a general rule, e.g., aside from a performance degradation on the smallest network when using the largest number of threads—tasking is better than worksharing. In addition, for the worksharing implementation, we tested the impact of choosing static versus dynamic scheduling with the `omp for` directive, again varying the chunksize. Our timings, a representative sample of which are presented in Table 1(b), indicate that the best results are generally achieved using the dynamic clause with a chunksize of one. Our profiling data indicate this is most likely caused by the increased imbalances in the amount of work associated with each chunk as the chunksize increases.



(a) Timing versus number of threads (b) Scaling versus number of threads

Fig. 2. Comparison of tasking (T) and workshare (WS) implementations

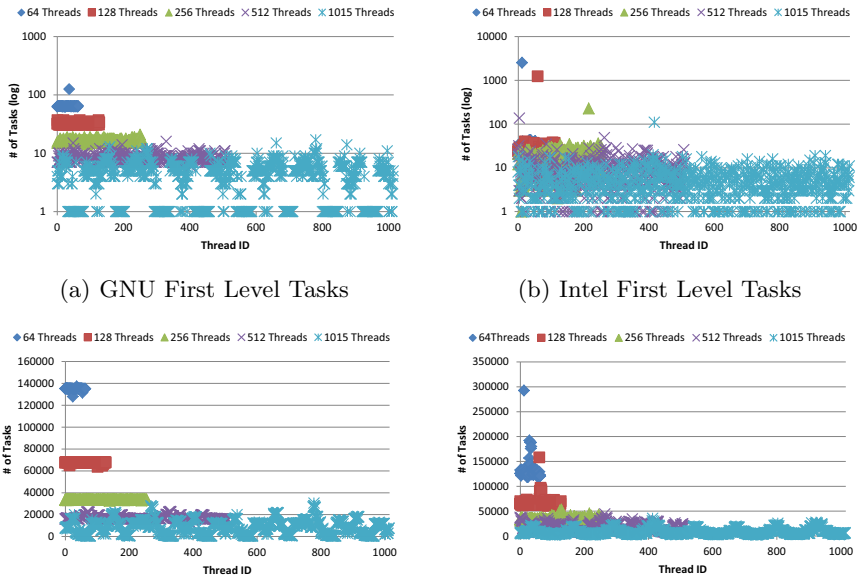
Table 2. Computation time (in seconds) of tasking versus workshare

Network	n		Number of CPUs 32	64	128	256	512	1015
Polblogs	1222	tasking	70	42	42	69	137	269
		workshare	71	47	46	70	132	292
CA-GrQc	4158	tasking	8989	4933	2723	2053	1916	3691
		workshare	10433	5749	5012	3260	2688	3136
as20000102	6474	tasking	50002	37417	15308	9851	9039	11419
		workshare	47197	29309	23851	17359	12231	11491
Gnutella09	8104	tasking	-	-	-	-	26888	13295
		workshare	-	-	-	-	28564	21456

4.3 Comparison of Tasking Performance on Different Compilers

Next, we compare the task scheduling and load balancing strategies of the GNU and Intel compilers. In doing so, we illustrate differences in challenges encountered on problems with small versus large numbers of threads and tasks. Profiling runs that calculate the δ-hyperbolicity of CA-GrQc (4158 nodes) allow us to evaluate the number of tasks per thread, amount of task switching, and load balancing up to 1015 threads. Comparison with runs on Polblogs (1222 nodes) provides perspective on the scaling behavior of each compiler's scheduler.

The first characteristic considered is the number of tasks (at each level) that are executed per thread. At the first level, each task is primarily concerned with spawning its child tasks (distributing the work of task creation). As shown in Figures 3(a) and 3(b), the GNU compiler has a relatively equitable distribution on first level tasks, but the Intel compiler has "outlier" threads that execute an order of magnitude more first level tasks than the other threads. Further investigation revealed that, when using Intel, the thread creating the first level of tasks (in the `single` directive region) schedules more first level tasks to itself. For both compilers, the distribution becomes more imbalanced at higher thread counts—we suspect this is due to either the variability in the numbers of children spawned by each first level task (recall Figure 1) or the imbalance in the amount of computation (i.e., number of iterations) in each of these children. For second



(a) GNU First Level Tasks

(b) Intel First Level Tasks

(c) GNU Second Level Tasks

(d) Intel Second Level Tasks

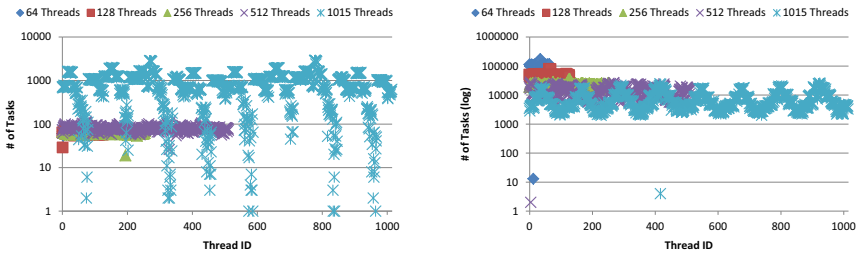
Fig. 3. Number of first and second level tasks executed per thread in CA-GrQc

level tasks, Figures 3(c) and 3(d) illustrate a relatively uniform distribution for the GNU compiler, with imbalances appearing at 1015 thread count, as well as an outlier thread with the Intel compiler, but only at small thread counts.

Given these data, a natural question to ask is how many tasks are "switching" threads between creation and execution. Figure 4 shows this count for second level tasks, and it clearly highlights a difference in task scheduling strategy between the compilers—which differ by two orders of magnitude at all thread counts. In particular, Intel's runtime scheduler seems to do more aggressive load balancing, leading to higher switch counts. Also note the order of magnitude increase in switching for the GNU compiler when we reach 1015 threads which starts to do more aggressive load balancing at this scale. Analysis of the data reveals the number of tasks switching is not uniformly distributed across threads, and we suspect load imbalances are occurring at this size scale.

The remaining evaluations use both CA-GrQc and Polblogs, whose sizes differ by a factor of approximately 4. Figure 5 shows the time spent executing tasks by each thread, sorted in decreasing order to illustrate more clearly the load imbalances.[1] For CA-GrQc, the work is well-balanced among the threads on both compilers when using 64 and 128 threads. Figure 5(a) shows that, with the

[1] While Figure 5 shows that the Intel compiler has a higher average execution time than the GNU compiler, note that this task inflation is due to the way that Intel load balancing is affecting locality and the way it is optimizing calls to the C++ Boost library.



(a) GNU (b) Intel

Fig. 4. Number of tasks created by one thread and executed by another one

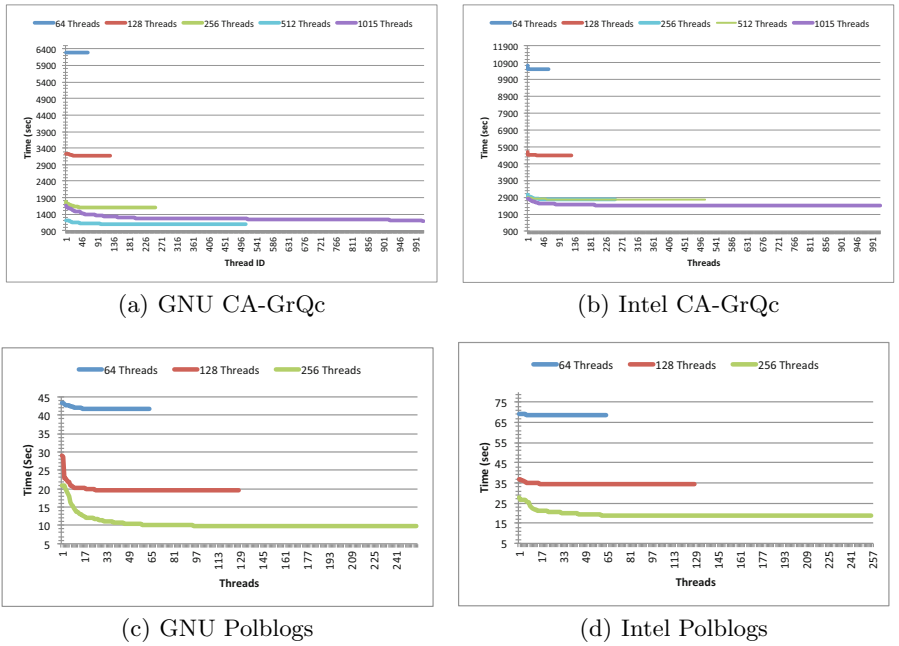
Table 3. Performance ratio (original runtime / optimized runtime) on GNU-compiled code

Dataset	Polblogs			CA-GrQc		
Threads	64	128	256	256	512	1015
PSD	0.990	0.694	0.926	0.709	0.813	1.410
PSD-CM	1.010	0.840	0.800	0.794	0.862	1.370
PSD-CMF	0.500	0.758	0.746	0.775	0.952	1.450

GNU compiler, by the time one reaches 512 threads, the variability has grown so that 10% of the task region overhead is attributable to load imbalance (at 1015 threads, this balloons to 27%). In contrast, Figure 5(b) shows that the Intel compiler limits this overhead contribution to 3% and 9% for 512 and 1015 threads, respectively. This is unlikely to be independent of the increased Intel task switching seen in Figure 4(b). Figures 5(a) and 5(b) also show that there is a significant performance loss due to task inflation at higher thread counts. When one decreases the size of the network by a factor of 4 (and thus the number of tasks by approximately 2^8), load imbalance occurs at a lower thread count, but the effects of task inflation are limited because data locality impacts the performance less at smaller thread counts. Figure 5(c) and 5(d) suggest that, independent of task inflation, Intel may have a better OpenMP load balancing strategy than GNU at this scale.

Finally, in Figure 6, we break down the overhead of the task region into that attributable to load imbalance and that caused by task creation and scheduling. For the smaller Polblogs network, most of the overhead is due to load imbalance, although at 256 threads we see the balance begin to shift for the GNU compiler. When considering the larger CA-GrQc network, large thread counts correspond to significant load imbalance with the GNU compiler. In contrast, the Intel compiler maintains a low load imbalance, but at the expense of a drastically higher task creation and scheduling overhead for large numbers of threads.

Table 3 gives the running time performance ratio under various optimizations (larger numbers are better) using the new OpenMP 3.1 tasking features; we note that statistics are given only for the GNU compiler, as in some cases the Intel-compiled code slows down to the point of timing out under similar optimizations (even on the Polblogs network). First, to mitigate the cost of task inflation

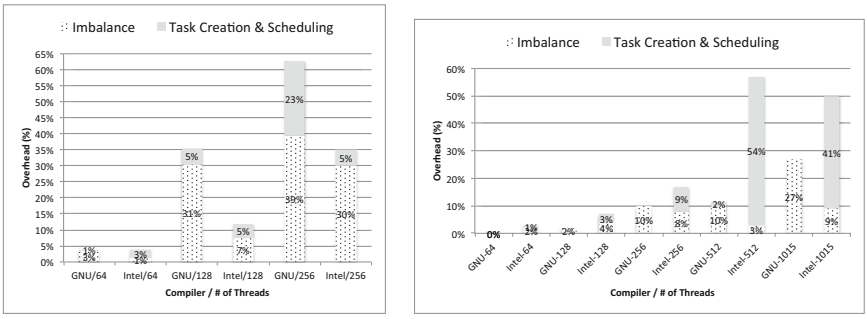


(a) GNU CA-GrQc (b) Intel CA-GrQc

(c) GNU Polblogs (d) Intel Polblogs

Fig. 5. Time spent executing tasks per thread

seen in CA-GrQc, we padded the task-shared data structures (PSD) with an extra dimension of the size of a memory page (4096 bytes). Once shared data structures were restructured, we tried to address load imbalance seen at high thread counts by adding an additional level of tasking with a `cutoff` at .8*size of the k iteration space, and using the `mergeable` clause to merge the data environment with that of the second level task (denoted as PSD-CM). Finally, we inserted a `final` clause that applies to the last two iterations of the second level task (denoted PSD-CMF). In most cases, these optimizations did not reduce the running time until the number of threads became very large, which highlights the importance of testing these OpenMP features at scale. In particular, padding shared data structures might make them less cache friendly when page migration costs are not as expensive at small thread counts, but it drastically improves the locality when significant task switching occurs for load balance at high thread counts. Furthermore, adding the third level of tasking in PSD-CM allowed better load balancing at high thread counts, but it could not overcome the increased overhead of task creation, without the additional control exerted by the `final` clause. We will still need to investigate why the mergeable clause only decreases the performance of the application at scale, but one possible reason is that sharing the data environment among tasks may stress the memory interconnect when two tasks are executed on different cores. When applying the final clause, this issue may be resolved because the second and third level of tasks become un-deferred and may execute on the same core. This would allow the tasks to benefit from the data locality of the merged data environment.



(a) Polblogs (b) Ca-GrQc

Fig. 6. Overhead of the task region

5 Conclusions

We have found that algorithms with multiple levels of tasking give improved performance over the OpenMP `workshare` construct since they allow us to parallelize irregular loops by splitting the work into smaller chunks, and enable better load balancing among threads. We have also used performance tools to analyze and compare the GCC 4.6.3 and Intel 11.1 compilers, finding that the two compilers use different task scheduling and load balancing strategies whose differences emerge when performing moderately large-scale versus very large-scale computations; and we have used new tasking features in OpenMP 3.1 to mitigate the cost of task creation and scheduling overheads. We expect that our conclusions will be useful in other applications that require hundreds or thousands of threads.

Acknowledgments. This work was funded by the Defense Advanced Research Projects Agency (DARPA), and it was supported by an allocation of advanced computing resources provided by the National Science Foundation. The computations were performed on Nautilus at the National Institute for Computational Sciences. This work was also funded by the Office of Advanced Scientific Computing Research, Office of Science, U.S. Department of Energy under Contract No. DE-AC05-00OR22725 with UT-Battelle, LLC.

References

1. Olivier, S.L., de Supinski, B.R., Schulz, M., Prins, J.F.: Characterizing and mitigating work time inflation in task parallel programs. In: Proc. of the Intl. Conf. on High Performance Computing, Networking, Storage and Analysis (SC 2012), pp. 65:1–65:12 (2012)
2. Kleinberg, R.: Geographic routing using hyperbolic space. In: Proc. of the 26th IEEE Intl. Conf. on Computer Communications (INFOCOM), pp. 1902–1909 (2007)
3. Shavitt, Y., Tankel, T.: Hyperbolic embedding of Internet graph for distance estimation and overlay construction. IEEE/ACM Trans. Netw. 16, 25–36 (2008)

4. Narayan, O., Saniee, I.: Large-scale curvature of networks. Phys. Rev. E 84, 066108 (2011)
5. Chen, W., Fang, W., Hu, G., Mahoney, M.W.: On the hyperbolicity of small-world and tree-like random graphs. In: Proc. of the 23rd ISAAC, pp. 278–288 (2012)
6. Bridson, M.R., Häfliger, A.: Metric Spaces of Non-Positive Curvature. Springer (1999)
7. Jonckheere, E., Lohsoonthorn, P., Bonahon, F.: Scaled Gromov hyperbolic graphs. J. of Graph Theory 57(2), 157–180 (2008)
8. Adcock, A.B., Sullivan, B.D., Mahoney, M.W. In preparation: Tree-like structure in large social and information networks (2013)
9. Khaldi, D., Jouvelot, P., Ancourt, C., Irigoin, F.: Task parallelism and synchronization: An overview of explicit parallel programming languages. Technical Report CRI/A-486, MINES ParisTech (2012)
10. Olivier, S.L., Prins, J.F.: Evaluating OpenMP 3.0 run time systems on unbalanced task graphs. In: Müller, M.S., de Supinski, B.R., Chapman, B.M. (eds.) IWOMP 2009. LNCS, vol. 5568, pp. 63–78. Springer, Heidelberg (2009)
11. Terboven, C., Schmidl, D., Cramer, T., an Mey, D.: Assessing OpenMP tasking implementations on NUMA architectures. In: Chapman, B.M., Massaioli, F., Müller, M.S., Rorro, M. (eds.) IWOMP 2012. LNCS, vol. 7312, pp. 182–195. Springer, Heidelberg (2012)
12. Duran, A., Teruel, X., Ferrer, R., Martorell, X., Ayguadé, E.: Barcelona OpenMP tasks suite: A set of benchmarks targeting the exploitation of task parallelism in OpenMP. In: Proc. of the 2009 Intl. Conf. on Parallel Processing (ICPP 2009), pp. 124–131 (2009)
13. Ayguadé, E., Beyer, J., Duran, A., Ferrer, R., Haab, G., Li, K., Massaioli, F.: An extension to improve OpenMP tasking control. In: Sato, M., Hanawa, T., Müller, M.S., Chapman, B.M., de Supinski, B.R. (eds.) IWOMP 2010. LNCS, vol. 6132, pp. 56–69. Springer, Heidelberg (2010)
14. Duran, A., Corbalán, J., Ayguadé, E.: An adaptive cut-off for task parallelism. In: Proceedings of the 2008 ACM/IEEE Conference on Supercomputing (SC 2008), 36:1–36:11 (2008)
15. Ibanez, R.F.: Task chunking of iterative constructions in OpenMP 3.0. In: First Workshop on Execution Environments for Distributed Computing, pp. 49–54 (2007)
16. Adamic, L.A., Glance, N.: The political blogosphere and the 2004 U.S. election: divided they blog. In: Proc. of the 3rd Intl. Workshop on Link Discovery (LinkKDD 2005), pp. 36–43 (2005)

Early Experiences with the OpenMP Accelerator Model*

Chunhua Liao[1], Yonghong Yan[2], Bronis R. de Supinski[1], Daniel J. Quinlan[1], and Barbara Chapman[2]

[1] Center for Applied Scientific Computing, Lawrence Livermore National Laboratory
{liao6,dquinlan,desupinski1}@llnl.gov
[2] Department of Computer Science, University of Houston
{yanyh,chapman}@cs.uh.edu

Abstract. A recent trend in mainstream computer nodes is the combined use of general-purpose multicore processors and specialized accelerators such as GPUs and DSPs in order to achieve better performance and to reduce power consumption. To support this trend, the OpenMP Language Committee has approved a set of extensions to OpenMP (referred to as the OpenMP accelerator model). The initial version is the subject of Technical Report 1 (TR1) while OpenMP 4.0 Release Candidate 2 (RC2) further refines the extensions.

In this paper, we examine the newly released accelerator directives and create an initial reference implementation, referred to as HOMP (Heterogeneous OpenMP). Focused on targeting NVIDIA GPUs, our work is based on an existing OpenMP implementation in the ROSE source-to-source compiler infrastructure. HOMP includes extensions to parse the new constructs and to represent them in the AST and other compiler translation details. Further we provide initial runtime support. For our evaluation, we have adapted a few existing OpenMP codes to use the accelerator model directives and present preliminary performance results. Finally, we critique the accelerator model in terms of its impact on developers and compiler writers and suggest possible improvements.

1 Introduction

Heterogeneous computer architectures that combine general-purpose multicore CPUs with specialized accelerators have become a viable solution to build high performance supercomputers, as demonstrated by Titan at ORNL (NVIDIA GPGPUs) and Stampede at TACC (Intel Xeon Phi) in the recent top500 list. Multicore CPUs are good at processing coarse-grained, irregular tasks; while accelerators excel in certain workloads such as large-scale data parallel and finer-grained vector processing. However, to exploit their computation capabilities

* LLNL-CONF-636479. This work was performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract DE-AC52-07NA27344. This work was also supported by the National Science Foundations Computer Research Infrastructure program under Award No. CNS-1205708.

A.P. Rendell et al. (Eds.): IWOMP 2013, LNCS 8122, pp. 84–98, 2013.

efficiently has required significant programmer effort to optimize an application program with respect to the specific hardware features of each type of accelerator.

Programming models such as OpenCL, CUDA and Brook provide mechanisms for an application to exploit the hardware capabilities of accelerators. High-level programming models, such as OpenACC [1], aim to provide an easier migration option from a sequential or parallel CPU version to the use of accelerators, typically GPGPUs. However, using these programming models to exploit their capabilities completely still poses significant challenges, even for expert programmers. Using multiple programming models in one application, as is likely with models that provide accelerator support that is distinct from CPU models, increases code complexity and decreases its portability. Mixing multiple programming models also complicates the compiler and runtime support due to the language complexity and to support runtime interoperability.

OpenMP has proven to be a productive solution for parallel programming with CPUs in shared memory systems. Recently, the OpenMP Language Committee has been working toward a single specification that supports heterogeneous computation nodes using both CPUs and accelerators. The committee has developed a set of extensions that they released first as a dedicated Technical Report 1 (TR1) and then as part of OpenMP 4.0 Release Candidate 2 (RC2) [2]. The extensions in this OpenMP accelerator model build on existing OpenMP concepts and constructs to provide a unified model for GPUs and CPUs. This model relies on compiler analysis and transformations to generate code that can execute on accelerators for specified source code regions, as well as runtime support to provide data movement and other support for hybrid execution.

In this paper, we review the OpenMP accelerator model and share our experiences of creating an initial implementation, the Heterogeneous OpenMP (HOMP) compiler. We have two goals: to provide early feedback on the usability of the OpenMP accelerator model and its impact on compiler writers; and to create a reference implementation for the extensions that the research community can leverage to explore further extensions.

The rest of the paper is organized as follow. Section 2 reviews the major accelerator extensions to OpenMP. Section 3 describes our initial implementation of those extensions. We present our preliminary results in Section 4 and critique the current model in Section 5. Section 6 presents related work. Section 7 concludes the paper and discusses our future work.

2 The OpenMP Accelerator Model

OpenMP 4.0 Release Candidate 2 [2] extends the execution model of the specification to support accelerators with device constructs. The OpenMP accelerator model assumes that a computation node has a host device connected with one or multiple accelerators as target devices. It uses a host-centric model in which a host device "offloads" code regions and data to accelerators for execution, specified using the `target` construct. This construct causes data and an executable to be copied (offloaded) to the accelerator before computation.

The OpenMP memory model is extended so that the code region has its own data environment. A device appears to have an independent shared memory, although copies cannot be assumed. Data-mapping attributes, specified using the map clause, define how variables are handled for the device data environments, including allocation, initialization and assignment to the host variables at the end of a target [data] region.

A device, which can be any logical execution engine defined by an implementation, has threads that behave almost the same as threads on the host device. Initially, only a single thread starts on a device to run an implicit task region. This single thread can fork more threads later when it encounters parallel constructs. It can also generate tasks as can its CPU counterpart. RC2 also introduced "thread teams" for organizing device threads in a structured way, which we will discuss in more detail in later sections.

2.1 Directives for Data and Computation Offloading

The target directive is introduced to offload data and computation to a device. It can have clauses to indicate a target device (device), data-mapping attributes (map), and an if condition to control the use of offloading at runtime.

The device data environment reflects the data-mapping attributes specified by the map clauses and the existing device data environment, which may have previously mapped variables due to target data constructs. Data mapping attributes include alloc, to, from, and tofrom, which determine how the list item is allocated, initialized and copied (handled) at region completion. The map clause can apply to "array sections", which designate a subset of an array, building on standard Fortran syntax or syntax added to OpenMP to support the concept for C and C++ for native arrays and pointer-based arrays.

Figure 1 shows a Jacobi iteration kernel written using the OpenMP 4.0 RC2 specification. One directive (line 6 and 7 of Figure 1) converts the existing host OpenMP code to device code. Since a target region can run on a host device whenever an implementation chooses, programmers should generally write a host version before adding accelerator-specific directives.

To avoid repetitive creation and cancellation of device data environments, the target data directive defines a device data region, in which multiple target regions can share the same device data. As shown at lines 1 and 2 in Figure 1, a device data region is defined before the while loop that contains the kernel. So each kernel launch within the while loop can reuse the enclosing data region. However, the map type of a data item in a map clause of a target construct can change if it is enclosed in a data region. For example, the map type for uold is to (copy the new values generated to the device) at line 6 while it has an alloc type at line 2. The reason is that outside of the while loop, uold is neither live-in nor live-out. Users must use care for their choice of map type depending on where they define data regions.

Another new directive, target update, can have motion clauses (to and from) and a device clause. According to the motion clauses, this construct makes a set of variables in the device data environment consistent with their original

```
1  #pragma omp target data map(to:n, m, omega, ax, ay, b, f[0:n][0:m]) \
2            map(tofrom:u[0:n][0:m]) map(alloc:uold[0:n][0:m])
3  while ((k<=mits)&&(error>tol))
4  {
5  // a "target + parallel for" loop copying u to uold is omitted ...
6  #pragma omp target map(to:n, m, omega, ax, ay, b, f[0:n][0:m], \
7                      uold[0:n][0:m]) map(tofrom:u[0:n][0:m])
8  #pragma omp parallel for private(resid,j,i) reduction(+:error)
9  for (i=1;i<(n-1);i++)
10   for (j=1;j<(m-1);j++)
11   {
12     resid = (ax*(uold[i-1][j] + uold[i+1][j])\
13         + ay*(uold[i][j-1] + uold[i][j+1])+ b * uold[i][j] - f[i][j])/b;
14     u[i][j] = uold[i][j] - omega * resid;
15     error = error + resid*resid ;
16   } // the rest code omitted  ...
17 }
```

Fig. 1. Jacobi kernel using accelerator directives

list items. With it, programmers can selectively update data values between the host and device data environments. Another directive, declare target, specifies that variables, functions (C, C++ and Fortran), and subroutines (Fortran) are mapped to (compiled for) a device. This directive generates device binaries for code that is not in the lexical scope of the target region, including the use of OpenMP constructs.

2.2 Directives for Thread Hierarchy

Accelerators are often massively parallel architecture devices that support hundreds or even thousands of concurrent threads with a hierarchical organization. Language constructs that allow users to manage the thread hierarchy are often needed. For example, CUDA provides the hierarchy of threads in blocks and grids. RC2 provides the teams and distribute constructs to manage a two-level thread hierarchy. Previously, OpenMP included the concept of a thread team, a group of synchronizable threads, to support nested parallelism. The teams construct creates a league or group of these thread teams. Initially each team in the league has one thread; subsequent parallel regions can create more threads in that team. The distribute construct specifies that the iterations of an associated loop are distributed across the master threads of all teams that execute the teams region to which the distribute region binds. Figure 2 gives a simple example of calculating the sum of an integer array using these constructs. The complex semantics lead to less intuitive code than existing OpenMP constructs. Without combined constructs, users must manually split a single loop into two loops in order to schedule the original loop at two levels of threads. The resulting code may be only useful for certain accelerators types such as NVIDIA GPGPUs.

3 HOMP: A Prototype Implementation

We are building a prototype implementation (referred to as HOMP, short for Heterogeneous OpenMP) for the OpenMP accelerator model. The current

```
1  int sum = 0;
2  int A[1000];
3  ...
4  #pragma omp target map(to: A[0:1000])
5  #pragma omp teams num_teams(2) num_threads(100) reduction(+:sum)
6  #pragma omp distribute
7    for (i_0 = 0; i_0 < 1000; i_0 += 500)
8  #pragma omp parallel for reduction(+:sum)
9    for (i = i_0; i < i_0 + 500; i++)
10     sum += A[i];
```

Fig. 2. Calculating sum explicitly using multiple contention teams

focus is to generate CUDA code because of the popularity of NVIDIA GPUs for high performance computing. Built upon ROSE's OpenMP implementation [3], HOMP is designed as an open implementation that the community can leverage to explore the design space of OpenMP extensions for accelerators. In particular, we have extended ROSE's pragma parsing to parse the new directives and clauses. We added new node types to ROSE's intermediate representation for the new directives and clauses related to accelerators, including the target and target data regions and the map clause. We similarly extended OpenMP lowering and runtime support. We give more details about the fundamental OpenMP implementation and our additional work for device constructs below.

3.1 ROSE and HOMP

HOMP is built on ROSE [4], a source-to-source compiler infrastructure developed at Lawrence Livermore National Laboratory to build compilers or program transformation and analysis tools for large-scale C/C++ and Fortran applications. Essentially, ROSE provides an object-oriented abstract syntax tree (AST) with a set of parsing, unparsing, analysis and transformation interfaces allowing users to build translators, analyzers, optimizers, and specialized tools quickly.

The existing ROSE OpenMP implementation [3] supports OpenMP 3.0 directives for C, C++ and a subset of Fortran. Internally, ROSE's OpenMP support works through the following steps: 1) AST generation of input code. 2) OpenMP pragma parsing since the frontends used by ROSE do not recognize OpenMP. 3) AST patching for adding new nodes and edges representing OpenMP directives and clauses. These new OpenMP-specific AST nodes are created to represent the semantics of OpenMP intuitively. For example, a node named (SgOmpParallelStatement) with a body statement block represents an **omp parallel** region. 4) OpenMP lowering to generate multithreaded code calling runtime functions. 5) Generate (unparse) transformed source code from the AST. A backend compiler will be transparently invoked to generate object code from the output code. 6) Link with runtime support to generate the executable. ROSE defines a generic runtime layer (XOMP) that abstracts common runtime support for OpenMP implementations and insulates the compiler translation from minor changes to runtime libraries. As a result, ROSE is unique in that a single set of OpenMP translations can work with multiple OpenMP runtime libraries.

3.2 Implementing the Accelerator Model

Target Regions. A target region starts a sequential execution of the initial implicit task on a target device. Using the latest CUDA 5.0 environment and GPUs with Compute Capability 3.5 or beyond, a target region can be implemented with a kernel launch configured with a single thread block with a single thread. When a parallel or teams region is encountered, dynamic parallelism can be used to launch another CUDA kernel configured with the requested number of thread blocks and threads per block.

We have at least two choices for CUDA environments that lack support for dynamic parallelism. The first one is to launch enough thread blocks and threads per block when the first sequential region of a target region is encountered despite actually using only a single thread. However, accurate estimation of the thread and block counts is difficult since later parallel regions may occur in functions and may dynamically change the counts. This choice may also waste energy if the sequential region has a long duration. The other choice translates each sequential portion and parallel portion into an independent kernel launch, with unnecessary synchronization after each launch. We consider this the better choice.

For a target region immediately followed by a parallel region, directly launching a multiple-thread execution kernel without an initial sequential part is the best choice. This choice more intuitively fits the semantics that users often express for GPUs. Thus, combined omp target parallel or omp target teams parallel are more useful and more intuitive than their separate forms.

Parallel Regions and Teams. With this target region implementation, each enclosed parallel region can be implemented as a separate kernel launch. However, with CUDA, only threads within the same thread block can (easily) synchronize. Unless developers explicitly use teams with parallel, an implementation cannot blindly spawn threads across multiple blocks since the parallel region may have synchronization points in the middle of its execution.

When the programmer does not specify the teams construct, compilers can limit the spawned threads to be those belonging to a single thread block, without leveraging all available GPU threads. Alternatively, they can use analysis to rule out synchronization points in the middle of the parallel region and then freely spawn threads across thread blocks as needed. This optimization requires a scan of the parallel region for synchronization constructs in the middle, such as barrier and atomic, and any unresolvable function calls that might contain such constructs. This alternative allows more parallelism in exchange for increased compiler complexity. Another solution would introduce a new clause such as no-middle-sync for a parallel region to indicate explicitly that the region does not contain any synchronization points.

For example, we outline the source code of the parallel region in Figure 1 so it can be transformed into the CUDA kernel in Figure 3. We insert CUDA execution configuration and kernel launch statements at lines 19 to 27 in Figure 4. Two runtime functions xomp_get_maxThreadsPerBlock() and xomp_get_max1DBlock() obtain the default execution configuration based on the hardware information and

```
1  __global__ void OUT__1__10117__(int n, int m, float omega, float ax, \
2                      float ay, float b, float *_dev_per_block_error, \
3                      float *_dev_u, float *_dev_f, float *_dev_uold)
4  {
5    /* local variables for  loop , reduction, etc */
6    int _p_j; float _p_error; _p_error = 0; float _p_resid;
7    int _dev_i, _dev_lower, _dev_upper;
8
9    /* Obtain loop bounds for current thread of current block */
10   XOMP_accelerator_loop_default (1, n-2, 1, &_dev_lower, &_dev_upper);
11   for (_dev_i = _dev_lower; _dev_i<= _dev_upper; _dev_i ++) {
12     for (_p_j = 1; _p_j < (m - 1); _p_j++) {
13 /*   replace original variables with device variables
14      linearize 2-D array accesses */
15       _p_resid = (((((  ax * (_dev_uold[(_dev_i - 1) * 512 + _p_j]  \
16                           + _dev_uold[(_dev_i + 1) * 512 + _p_j])) \
17                       + (ay * (_dev_uold[_dev_i * 512 + (_p_j - 1)] \
18                       + _dev_uold[_dev_i * 512 + (_p_j + 1)]))))     \
19                       + (b * _dev_uold[_dev_i * 512 + _p_j]))       \
20                       - _dev_f[_dev_i * 512 + _p_j]) / b);
21       _dev_u[_dev_i * 512 + _p_j] = (_dev_uold[_dev_i * 512 + _p_j] \
22                                    - (omega * _p_resid));
23       _p_error = (_p_error + (_p_resid * _p_resid));
24     }
25   }
26   /* thread block level reduction for float type*/
27   xomp_inner_block_reduction_float(_p_error,_dev_per_block_error,6);
28 }
```

Fig. 3. Generated CUDA kernel

the number of iterations. Our current strategy uses the full number of supported hardware threads within a thread block before using more blocks.

Data Handling. Based on the specified map types, the map clause guides the translation of device variable declarations, memory allocation, value copying between CPU memory and GPU memory, and deallocation. Since a variable in a nested map clause may already exist in an enclosing data environment (e.g., array u shown at both lines 2 and 7 in Figure 1), an implementation must track active device data environments to reuse the versions in enclosing data environments when they exist.

We track the data environments that target data and target constructs create in a stack and add runtime functions for that purpose. First, xomp_DDE_Enter() (line 2 in Figure 4; DDE stands for deviceDataEnvironment) initializes a data structure for each new data environment and pushes it onto the stack. The data structure stores information about variables allocated within the current data environment. Second, xomp_DDE_GetInheritedVariable() (line 6) checks if a variable in a map clause already exists in enclosing environments. Third, xomp_DDE_AddVariable (line 13) registers a newly mapped variable with its original address, device address, size, and a copy back flag. Finally, based on stored information for mapped variables, xomp_DDE_Exit() (line 35) transparently copies data back to the host and deallocates device memory deallocation before it is popped from the stack.

We linearize the storage of array variables with two or more dimensions. Accordingly, we replace all references to the original array elements with

```
1  /* Initialize a new data environment, push it to a stack */
2  xomp_deviceDataEnvironmentEnter();
3
4  int _dev_u_size = sizeof(float ) * (n - 0) * (m - 0);
5  /* Try to grab a mapped variable from enclosing data environments */
6  float *_dev_u=(float*)xomp_deviceDataEnvironmentGetInheritedVariable \
7                 ((void*)u, _dev_u_size);
8  /* If not inheritable, allocate and register the mapped variable */
9  if (_dev_u == NULL)
10 {
11   _dev_u = ((float *)(xomp_deviceMalloc(_dev_u_size)));
12 /* Register CPU address, device address, size, and a copy-back flag */
13   xomp_deviceDataEnvironmentAddVariable ((void*)u, _dev_u_size, \
14                                          (void*) _dev_u, true);
15   // data copy from Host to Device also here if specified
16 }
17 ... // handling of other variables is omitted
18
19 /* Execution configuration: threads per block and total block numbers*/
20 int _threads_per_block_ = xomp_get_maxThreadsPerBlock();
21 int _num_blocks_ = xomp_get_max1DBlock((n - 1) - 1);
22 float *_dev_per_block_error = (float *)(xomp_deviceMalloc( \
23                                         _num_blocks_ * sizeof(float)));
24 /* Launch the CUDA kernel ... */
25 OUT__1__10117__<<<_num_blocks_,_threads_per_block_, \
26                   (_threads_per_block_ * sizeof(float))>>> \
27     (n,m,omega,ax,ay,b,_dev_per_block_error,_dev_u,_dev_f,_dev_uold);
28 /*  Beyond thread block reduction */
29 error = xomp_beyond_block_reduction_float(_dev_per_block_error, \
30                                           _num_blocks_,6);
31 /*  Data deallocation, copy-back, etc.  */
32 xomp_freeDevice(_dev_per_block_error);
33 ...
34 /*Copy back and deallocate variables within this environment, pop stack */
35 xomp_deviceDataEnvironmentExit();
```

Fig. 4. Generated kernel configuration and launch code

references that use the device variable with a linear address calculation (e.g., _dev_u[_dev_i*512 + _p_j] at line 21 in Figure 3).

For simplicity, we use a two-level algorithm to implement reductions that leverage GPUs. One level is within each CUDA thread block and the other is across multiple thread blocks on the host side. We provide a set of runtime functions (e.g., xomp_inner_block_reduction_*() and xomp_beyond_block_reduction_*()) to support these two-level reductions. Figure 3 shows example code for the GPU-based inner-block reduction (line 27). The CPU side's across-block reduction is shown at line 29 of Figure 4.

Loop and Distribute Directives. By default, only the outermost loop is affected by the loop constructs unless a collapse clause is specified to allow an implementation to combine multiple loops into a larger iteration space. Three choices to schedule loop iterations among GPU threads are available: 1) use only master threads of multiple thread blocks when distribute is used right before the loop; 2) use threads from a single thread block; or 3) use a combination of multiple blocks and multiple threads per block, when applicable. Figure 3 shows an example translation. The translation calls XOMP_accelerator_loop_default() at line 10 to obtain the bounds for the current thread within the current thread block. No loop splitting is needed even for scheduling loops across teams.

On a final note, all runtime functions are designed to have a C binding so that they can interoperate easily with multiple programming languages including C, C++ and Fortran. The function interfaces are designed to be similar to their counterparts in C libraries. The bodies of the functions can be conditionally implemented through CUDA or OpenCL so that the same compiler translation can be reused across different lower-level accelerator APIs.

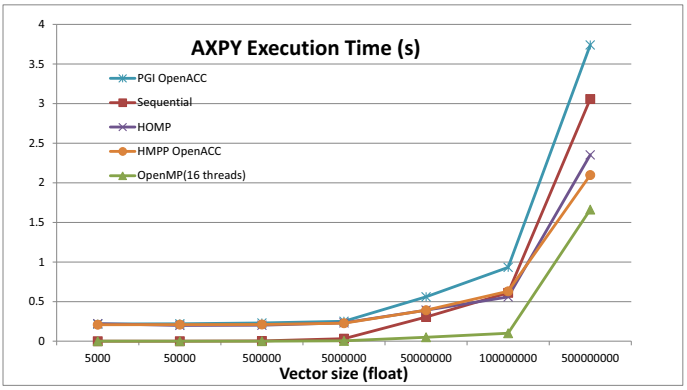
4 Preliminary Results

We have chosen three scientific kernels, including AXPY, Jacobi and matrix mulplication to evaluate our initial implementation. We also use the PGI [5] and HMPP [6] OpenACC compilers for comparison. All execution time measurements include the data transfer time between CPU memory and GPU memory. In addition, both sequential and OpenMP versions' performance results on CPUs are provided as a baseline.

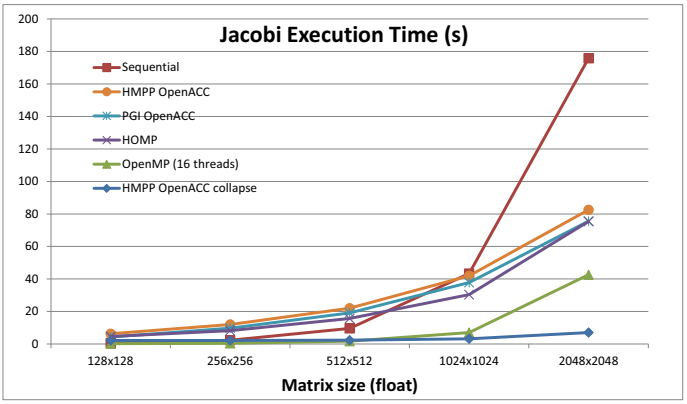
The machine used for this evaluation has 4 quad-core Intel Xeon processors (16 cores in total) running at 2.27GHz with 32GB DRAM. An NVIDIA Tesla K20c of the Kepler architecture is installed on the machine, with CUDA version 5.0/5.0 driver as its software environment. The PGI OpenACC compiler used is version 13.4 with the command line options *pgcc -acc -ta=nvidia -Minfo=accel -mp -O3*; and the HMPP OpenACC compiler used is version 3.3.3 with command line options as *hmpp gcc -fopenmp -O3*. As source-to-source compilers, both the HOMP compiler and the HMPP compiler use GCC 4.4.7 and the CUDA 5.0 compiler as backend compilers.

Figure 5(a) shows the performance results for AXPY. The performance for the HOMP and HMPP versions are close. However, the actual computation of AXPY is very small compared to the data transfer cost, which accounts for 99% of the total execution time. Therefore, the OpenMP version of AXPY outperformed all three GPU versions when the vector size is large. The performance of using the PGI OpenACC compiler is relatively poor for large input data. We were able to look at the intermediate files generated by the PGI compiler, and have observed that the PGI compiler performs aggressive loop unrolling, which introduces a large number of branch instructions. Those instructions create divergence during thread executions, which can hurt GPU performance.

Figure 5(b) shows the performance results of Jacobi, which is computation intensive. More than 95% of the total GPU-related execution time is spent on kernel execution. While we are still working to implement collapse in HOMP, we tried to test the OpenACC version with the collapse clause, which is supported by the HMPP compiler. The PGI compiler could not compile the code when collapse is used with reduction. Without the loop collapse, the difference in performance between the three compilers is small. The use of collapse significantly improves performance on the GPU since iterations of both loops are exposed to exploit the abundant GPU threads. According to the generated CUDA code, HMPP translates the collapse clause by mapping the associated two-level loop nest to a 2-D grid, instead of linearizing the loops.



(a) AXPY

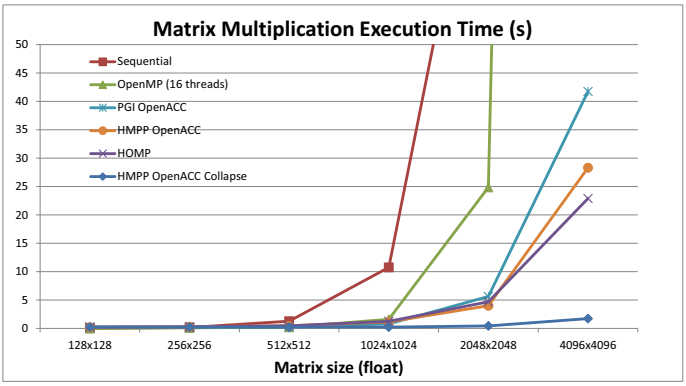


(b) Jacobi

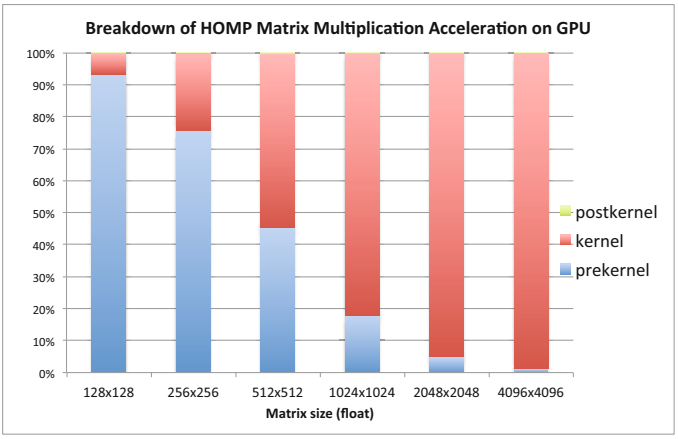
Fig. 5. Performance results for AXPY and Jacobi

Figure 6(a) shows results for matrix multiplication, which has significant computation for each element. Using the GPU for larger data sets easily outperforms the corresponding OpenMP version. The kernel execution time begins to dominate the total acceleration time when the data size is larger, as shown in Figure 6(b). Again, we tried to add the collapse clause for the OpenACC version of the kernel. The HMPP compiler could efficiently exploit this addition and generated much more efficient code as shown in the figure. The PGI compiler did not generate any better performance so we do not show those results.

We further compared the performance of the best generated CUDA code so far with a handwritten CUDA SDK versions and the CUBLAS version. The CUDA SDK and CUBLAS versions use completely different algorithms and apply agressive optimizations to the algorithms [7], including the use of shared memory within blocks and apply tiling to the algorithms. The performance difference (ratio between execution times) can be large (3.21 X-9931.29 X) as shown in Table 1, although the difference decreases for larger inputs. Generation of those



(a) Execution time



(b) HOMP execution time breakdown

Fig. 6. Performance results for matrix multiplication

highly-optimized codes using a compiler is challenging without introducing new language constructs.

5 Discussion

Unifying programming for both CPUs and accelerators in a single high-level programming interface is an important and challenging effort. Based on our early experiences shown above, the current OpenMP extensions for accelerators are a useful step that can lead to a complete solution. The extensions are mostly intuitive for users and straightforward for compiler developers to implement. Nevertheless, the following refinements and additions would improve the usability of the OpenMP accelerator model.

Multiple Device Support. Specifying a device ID in the device() clause may not be portable. The current design may require manual code assignment and data

Table 1. Compare performance results of matrix multiplication (in seconds)

Version/Size	128x128	256x256	512x512	1024x1024	2048x2048	4096x4096
HMPP collapse	0.238351	0.222798	0.226316	0.238678	0.444121	1.728459
CUDA SDK	0.000034	0.000141	0.001069	0.008441	0.067459	0.538174
CUBLAS	0.000024	0.000054	0.000207	0.001092	0.007229	0.052283
Ratio(HMPP/SDK)	7010.32	1580.13	211.71	28.28	6.58	3.21
Ratio(HMPP/BLAS)	9931.29	4125.89	1093.31	218.57	61.44	33.06

decomposition for each device ID if multiple devices are used. New clauses such as device_type(), num_devices() and data_distribute() would support automatic code assignment and data distribution by the compiler across multiple devices.

Combined Constructs. Separate target and parallel constructs do not intuitively express what users often want: immediate parallelism on accelerators without any sequential execution. Combined constructs such as target parallel (or target teams parallel) would conveniently meet user needs and simplify compiler implementation. Similarly, a combined teams distribute parallel for construct could be allowed so that a compiler could automatically schedule an affected loop over multiple threads from multiple teams without loop splitting in the source code.

No-Middle-Sync Clause. Compilers may not have sufficient analysis to determine if a parallel region within a target region will have synchronization points during its execution. An implementation may have to execute the parallel region conservatively within a single CUDA thread block, which may severely underutilize abundant GPU threads. Manually adding teams and distribute by users is often cumbersome and may not be portable. We suggest to introduce new clauses such as no-middle-sync or ignore-middle-sync to facilitate an implementation to leverage threads across multiple thread blocks. no-middle-sync expresses the semantics of no middle synchronization points while ignore-middle-sync is used to tell an implementation to ignore any possible middle synchronization points.

Array Sections. Some may feel that RC2's different array section notations for C/C++ ([*lower-bound* : *length*]) and Fortran (built-in triplet format) are confusing. Although RC2's notation may prove more natural to C/C++ programmers, a consistent notation for both languages would fit well with many compilers that have a single IR shared by multiple languages.

Global Barrier. The current device constructs do not support specifying synchronizations across multiple thread teams, or a league, which may often be needed. Instead, multiple target regions can be used effectively to provide barriers within a single target region. A clause (such as league or team) in the barrier directive could explicitly set the synchronization scope.

Mapping Nested Loops. RC2 keeps the original collapse semantics, which combines multiple associated loops into one large iteration space. OpenACC has a

similar clause, but does not restrict only to linearization. For example, the associated loops could be mapped to multi-dimensional grids and thread blocks when using NVIDIA GPUs. OpenMP should explicitly allow similar flexibility since linearization may not always lead to optimal performance.

Mapped Data Reuse. In RC2, reusing mapped data relies on looking up enclosing target data regions and the data declared in the global scope using the declare target directive. Passing mapped variable pointers across a function scope may become tricky and inconvenient. A possible solution is to have explicit *liveness* attributes of mapped variables (keep, present, and final) in the map clause, a similar approach adopted in OpenACC. Making data reuse explicit can also simplify the implementation so that less runtime support is needed.

6 Related Work

Several previous studies [1, 6, 8–11] have explored directive-based language extensions and compiler techniques to exploit parallelism using NVIDIA GPUs. We briefly mention a few of them in this section.

OpenACC [1] is a standard for programming accelerators in conjunction with a host CPU, which could be a multicore platform. Similar to OpenMP, OpenACC programmers annotate a sequential program written in either C/C++ or Fortran with OpenACC constructs so compilers can transform the annotated program region to be executed on accelerator devices. OpenACC supports both implicit (using acc kernels) and explicit (using acc parallel) parallelism. The current OpenACC standard has limited expressivity for hybrid parallelism between CPU and GPU tasks and most compiler supports do not yet address multiple accelerators. Similarly, PGI Fortran & C accelerator extensions [8] define pragma-based directives, such as acc region and acc data region, for programmers to specify regions of computation and data to be offloaded to GPUs. An accelerator loop directive (acc for) is also provided to allow programmers to specify more explicit information for parallelizing loops. For loops without explicit scheduling clauses, its implementation relies on sophisticated compiler analysis to choose a better mapping among a few choices [5, 8].

Lee and Eigenmann [10] presented an approach of directly translating OpenMP CPU code to GPU code without using language extensions. Compiler analysis finds synchronization points in each parallel region, which can then be split into multiple subregions as necessary for generating multiple CUDA kernels. Mint [11] is a domain-specific language extension specialized in stencil kernels. Based on ROSE, Mint translates annotated C code into CUDA code. OmpSs [12, 13] is another interesting effort that allows users to define data dependences among tasks. The solution includes a powerful runtime that manages data and schedules tasks among different types of hardware devices, thus requiring little compiler support. More recently, Lee et. al. [14] compared six different directive-based GPU programming models.

Compared to previous work, our work examines OpenMP accelerator extensions and creates a prototype implementation for them. We are interested in

accelerator language extensions that are compatible with existing OpenMP execution and memory models. Consequently, the implementation techniques that we explore are based on an existing open-source OpenMP compiler.

7 Conclusions and Future Work

In this paper, we have examined the newly introduced accelerator model in OpenMP 4.0 (RC2) and shared our experiences of creating a prototype implementation for it. Our implementation has already been released under a BSD license as part of the ROSE compiler framework.

The OpenMP accelerator extensions represent a major enhancement for OpenMP to meet the increasing demands to support accelerators and heterogeneous architectures. For developers, most extensions are intuitive and fit well with OpenMP's existing execution model and memory model. Complexity arises from the use of `teams` and `distribute` constructs to organize the thread teams and hierarchy. Combined constructs are needed. For compiler developers, creating a working implementation that leverages an existing OpenMP compiler framework is straightforward based on our early experience, though aggressive compiler analysis and optimization techniques further enhance the performance of generated codes. It also requires efficient runtime support to manage the data mapping and to coordinate the executions of CPU tasks and accelerator kernels.

Our future work includes the following research directions. We will target more hardware architectures such as the Intel Many Integrated Core Architecture (MIC). We also will generate OpenCL in addition to CUDA. We will investigate techniques to aid users in choosing between CPU threads, accelerators and vectorization; We will explore a peer-to-peer execution model that can express code and data offload without always involving a host device.

References

1. OpenACC: Directives for Accelerators, `http://www.openacc-standard.org/`
2. OpenMP Architecture Review Board, The OpenMP API Specification for Parallel Programming, `http://www.openmp.org/`
3. Liao, C., Quinlan, D.J., Panas, T., de Supinski, B.R.: A ROSE-Based OpenMP 3.0 Research Compiler Supporting Multiple Runtime Libraries. In: Sato, M., Hanawa, T., Müller, M.S., Chapman, B.M., de Supinski, B.R. (eds.) IWOMP 2010. LNCS, vol. 6132, pp. 15–28. Springer, Heidelberg (2010)
4. Quinlan, D., et al.: ROSE Compiler Infrastructure, `http://www.rosecompiler.org/`
5. Wolfe, M.: Implementing the PGI Accelerator Model. In: Proceedings of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units, GPGPU 2010, pp. 43–50. ACM, New York (2010)
6. Dolbeau, R., Bihan, S., Bodin, F.: HMPP: A Hybrid Multicore Parallel Programming Environment (2007)
7. Volkov, V., Demmel, J.W.: Benchmarking GPUs to Tune Dense Linear Algebra. In: Proceedings of the 2008 ACM/IEEE Conference on Supercomputing, SC 2008, pp. 31:1–31:11. IEEE Press, Piscataway (2008)

8. The Portland Group, "PGI Fortran & C Accelerator Compilers and Programming Model," Tech. Rep. (November 2008)
9. Han, T.D., Abdelrahman, T.S.: hiCUDA: A High-Level Directive-Based Language for GPU Programming. In: Proceedings of 2nd Workshop on General Purpose Processing on Graphics Processing Units, GPGPU-2, pp. 52–61. ACM, New York (2009)
10. Lee, S., Eigenmann, R.: OpenMPC: Extended OpenMP Programming and Tuning for GPUs. In: Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis, SC 2010, pp. 1–11. IEEE Computer Society, Washington, DC (2010)
11. Unat, D., Cai, X., Baden, S.B.: Mint: Realizing CUDA Performance in 3D Stencil Methods with Annotated C. In: Proceedings of the International Conference on Supercomputing, ICS 2011, pp. 214–224. ACM, New York (2011)
12. Duran, A., Ayguade, E., Badia, R.M., Labarta, J., Martinell, L., Martorell, X., Planas, J.: OmpSs: A Proposal for Programming Heterogeneous Multi-core Architectures. Parallel Processing Letters 21(02), 173–193 (2011)
13. Bueno, J., Planas, J., Duran, A., Badia, R.M., Martorell, X., Ayguade, E., Labarta, J.: Productive Programming of GPU Clusters with OmpSs. In: 2012 IEEE 26th International on Parallel & Distributed Processing Symposium (IPDPS), pp. 557–568. IEEE (2012)
14. Lee, S., Vetter, J.S.: Early Evaluation of Directive-Based GPU Programming Models for Productive Exascale Computing. In: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis, SC 2012, pp. 23:1–23:11. IEEE Computer Society Press, Los Alamitos (2012)

An OpenMP* Barrier Using SIMD Instructions for Intel® Xeon Phi™ Coprocessor

Diego Caballero[1,2], Alejandro Duran[3], and Xavier Martorell[1,2]

[1] Barcelona Supercomputing Center
diego.caballero@bsc.es
[2] Universitat Politecnica de Catalunya
xavim@ac.upc.edu
[3] Intel Corporation
alejandro.duran@intel.com

Abstract. Barrier synchronisation is a widely-studied topic since the supercomputer era due to its significant impact on the overall performance of parallel applications. With the current shift to many-core architectures, such as the Intel® Many Integrated Core Architecture, software barriers need to be revisited from an on-chip point of view to exploit their new specific resources. In this paper, we propose a tree-based barrier that takes advantage of SIMD instructions and the inter-thread cache locality provided by the 4-way SMT of the Intel® Xeon Phi™ coprocessor. Our SIMD approach shows a speed-up of up to 2.84x over the default Intel OpenMP* barrier in the EPCC barrier microbenchmark. It also improves by up to 60% and 21% the Livermore Loop kernel number six and the NAS MG benchmark, respectively.

Keywords: Barrier, SIMD, synchronisation primitives, combining tree, many-cores, Intel® Xeon Phi™ coprocessor, Intel® Many Integrated Core Architecture, OpenMP*.

1 Introduction and Motivation

Barrier synchronisation primitives became especially relevant for large-scale supercomputers where communication among hundreds of threads is known to be very expensive in terms of time. The traditional software approaches imply an important number of memory operations through very high-latency interconnection networks [24]. According to this, barriers turns into a potentially critical bottleneck in terms of performance [17]. As a result, in an attempt to improve this issue, expensive hardware approaches have been adopted in the design of supercomputers [9] [22] [5].

This time-consuming component has gone unnoticed for small general-purpose multi-core CMPs. However, the continuous improvements in core integration rise this topic into prominence again for many-core CMPs [19], such as the Intel® Many Integrated Core Architecture (Intel MIC Architecture) [2]. Because of the inherently high number of threads in many-core architectures, the synchronisation process acquires greater importance at the same time as computational work is now distributed between a larger number of *workers*. Therefore, this *finer* distribution decreases the weight of computation time and increases the weight of synchronisation constructs, such as it occurs exploiting fine-grained parallelism [19].

A.P. Rendell et al. (Eds.): IWOMP 2013, LNCS 8122, pp. 99–113, 2013.

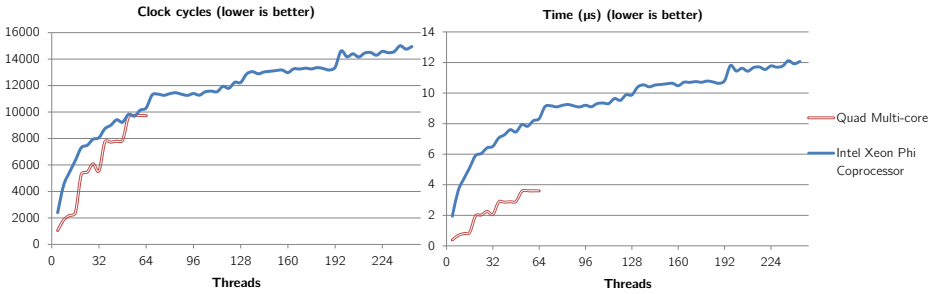


Fig. 1. Clock cycles (on the left) and execution time (on the right) of the default Intel OpenMP barrier (EPCC barrier). Quad-socket Intel Xeon processor E5-4650 at 2.7 GHz (32 cores / 64 threads) and Intel Xeon Phi coprocessor 7120P at 1.238 GHz (61 cores / 244 threads). Mean of 10 executions. 4 threads step.

Fig. 1 shows the clock cycles (on the left) and the execution time (on the right) of a single barrier in an Intel® Xeon® processor (quad-socket) and an Intel® Xeon Phi™ coprocessor. From the single-chip point of view, a single-socket of the Intel Xeon processor E5-4650 (16 threads) takes 2,393 clock cycles whereas the Intel Xeon Phi coprocessor (244 threads) spends 15,880 cycles. Even in the quad-socket scenario (4 multi-cores, 64 threads), a single barrier consumes only 9,721 clock ticks. Despite the fact that synchronising 4 multi-core processors requires a more expensive off-chip communication, the performance is still notably better than in the single-chip Intel Xeon Phi coprocessor. Such significant difference in the synchronisation process is mainly due to the aforementioned increase in the number of threads from the multi-core to the many-core architecture.

Furthermore, many-core architectures may suffer from harder frequency constrains than multi-core architectures [13]. The Intel Xeon Phi coprocessor used in our experiments runs at a considerably slower frequency (1.238 GHz) than the Intel Xeon processor E5-4650 (2.7 GHz). Thus, as far as a single barrier execution time is concerned, the Intel Xeon processor E5-4650 farther outperforms the Intel Xeon Phi coprocessor.

Apart from software approaches, much hardware research has been done on barrier synchronisation for CMPs [19] [20]. Nevertheless, such approaches are prohibitive in terms of on-chip area since a hardware implementation would reduce the resource available for computational units.

All these facts reveal that software barriers need to be revisited from the many-core architectures point of view, where the interconnection network is essential but also the resources of the cores must be taken into account.

Focusing on the Intel MIC Architecture, with 4-way Symmetric Multi-Threading (SMT) and 512-bit SIMD instructions, we propose a reconfigurable and multi-degree combining tree barrier algorithm that uses groups of distributed counters orchestrated by SIMD instructions. It is aimed at exploiting the 4-way inter-thread cache locality in the synchronisation process and optimising the acquisition and releasing phase with SIMD instructions. SIMD operations prevent the algorithm from having to iterate in a scalar fashion on every single counter to release a group of threads. This fact reduces the well-known ping-pong effect that occurs when several threads share a cache line.

We compare our SIMD barrier against the production-optimised barrier of the Intel OpenMP runtime using three different benchmarks and two thread binding policies. Our algorithm outperforms the implementation of the Intel OpenMP barrier in our experiments.

2 Related Work

Barrier synchronisation came into prominence with the advent of first supercomputers. Initially, threads busy-waited on a shared variable for the synchronisation process to be completed. The memory contention problem caused by many threads accessing to the same memory module at once is a well-known issue from that time [18]. Soon, the relevance of this issue brought expensive hardware-specific solutions, such as dedicated interconnection networks [9], that still have influence on more recent systems [22] [5].

To tackle barrier synchronisation constructs from a cheaper and more flexible perspective, many software barrier algorithms have been proposed. Mellor-Crummey and Scott [15] described several barrier approaches already available and improved the combining tree and tournament barriers spinning on separate locally-accessible flags. Nanjegowda et al. [16] evaluated these algorithms in OpenMP, concluding that the best algorithm is dependent on the number of threads used, the architecture and the application. Hoefler et al. [12] summarised nine barrier algorithms and analysed them in the context of the InfiniBand* interconnection network. Zhang et al. [25] introduced a barrier algorithm based on distributed counters with locally-separated waiting flags, which considerably reduced overhead on IBM* POWER3 and IBM POWER4 systems.

Focusing on tree-based barrier algorithms similar to ours, Yew et al. [24] presented the combining tree structure for the first time to palliate memory contention on shared variables. They used a lock-based centralised counter per tree node and determined an optimal uniform group size of four (fan-in) for each node in the tree. This is far from our heterogeneous lock-free distributed approach. Mellor-Crummey and Scott [15] described several barrier approaches and investigated on a new lock-free combining tree barrier with scalar flags where two tree structures (arrival and wakeup) are used. Threads are statically linked to a unique node in both trees, not only to leaf nodes but also to nodes of any level. Only one thread is attached to non-leaf nodes and a group of threads are attached to leaf nodes. Unlike us, they use a uniform group size of four (fan-in) in the arrival tree and two (fan-out) in the wakeup tree. Their implementation is not reconfigurable and is bound to these fixed fan-in and fan-out parameters. On the contrary, we use a traditional combining tree structure that allows us to set a different tree degree per level. Furthermore, nodes contain reconfigurable distributed counters operated by SIMD instructions which allow checking and releasing a larger group of threads at the same time. In addition, all threads start from leaf nodes but some threads are designated to continue to the following levels. There, threads are also placed in groups.

Gupta and Hill [10] presented and adaptive combining tree with lock-based centralised counters which reshapes itself to move late threads towards the root of the tree. Scott [21] later improved this idea, and Eichenberger [8] also evaluated it from the load imbalance point of view, denoting that the optimal degree of combining trees could reach up to 128 in the presence of load imbalance.

Gupta and Hill [10] presented and adaptive combining tree with lock-based centralised counters which reshapes itself to move late threads towards the root of the tree. A *fuzzy* [11] barrier version is also proposed and later improved by Scott and Mellor-Crummey [21]. Eichenberger and Abraham [8] revisited the idea of placing slow threads close to the root but based on the tree-based barrier structure published by Mellor-Crummey and Scott [15]. They pointed out that the optimal degree of combining trees could reach up to 128 in the presence of load imbalance.

Regarding current multi- and many-core architectures, research trends on barriers are dominated by hardware proposals. Abellán et al. [4] developed a hardware-based barrier mechanism by means of an independent interconnection network based on global interconnection lines. Sampson and González [19] remarked the inefficiency of software barriers in the presence of fine-grained parallelism. They also proposed a new cache-based barrier scheme that only requires additional hardware in the shared memory subsystem. Sartori and Kumar [20] evaluated three hybrid hardware-software approaches that take advantage of the on-chip network with slight modifications. They achieve performance comparable with dedicated synchronisation networks. Villa et al. [23] studied four hardware and software algorithms and evaluated them in a Network-on-Chip architecture. It was demostrated that, in many cases, simple networks can be more efficient than highly connected topologies.

In spite of the fact that an extensive variety of hardware proposals have been developed for CMPs, they have not been widely adopted by industry because on-die hardware implementations would decrease the resources available for computational units.

3 The Intel Xeon Phi Coprocessor

The Intel Xeon Phi coprocessor [2] is the first product based on the Intel MIC architecture. It features a large number (at least 50) of cores.

Each core is based on the x86 architecture but are in-order cores that run on a much lower frequency than the standard Intel Xeon processors. Each core can issue up to two instructions per cycle from different thread contexts. In addition, it comes with a specially designed Vector Processing Unit (VPU) that provides the architecture with 512-bit SIMD instructions, denominated Intel® Initial Many Core Instructions (Intel® IMCI). This SIMD instruction set also supports masked operations, gathers, scatters, and fused multiply-and-add operations. As a result, the coprocessor can yield a performance of 1.2 TFLOPS/s in double precision on a 300W TDP package.

Each core has four hardware threads that are scheduled in a round-robin fashion. The L1 cache (32KB for instructions, 32KB for data) is shared among the hardware threads. The L2 cache is distributed across the different cores (with a 512KB slice in each core for both data and instructions) and it is also shared among the threads of the same core. Both L1 and L2 caches are fully coherent. The coherence protocol is implemented by means of a distributed tag directory (DTD) which keeps the coherence information of each cache line.

Cores and the L2 slices are connected through a double ring (one ring in each direction). The ring also connects the memory controllers and the I/O interface that allows the coprocessor to communicate with the host through the PCIe bus. The memory controllers has up to 16 memory channels available to access the on-board GDDR memory.

Table 1. Characteristics of the Intel Xeon Phi coprocessor 7120P

Number of chips	1	SIMD size	512 bits
Cores / chip	61	Memory size	16 GB
Hardware stepping	C0	Memory bandwidth	352 GB/s
Threads / core	4	ECC mode	Supported
Frequency	1.238 GHz	Peak performance (DP)	1.2 TFLOPS/s
L1 size / core	32+32 KB	Power consumption (TDP)	300 W
L2 size / core	512 KB	Software stack	Gold

The coprocessor supports a standard software stack with a Linux operating system and programming models such as OpenMP*, OpenCL*, MPI, or Intel® Threading Building Blocks. In this sense, applications written in one of these paradigms are readily available to run on the Intel Xeon Phi coprocessor. Therefore, the optimization of these programming models for the Intel MIC Architecture is of great importance.

In this work, we use the Intel Xeon Phi coprocessor described in Table 1.

4 Multi-Degree SIMD Combining Tree Barrier Algorithm

To profit SIMD resources in the barrier synchronisation primitive, we propose a vector barrier algorithm that exploits SIMD instructions to perform memory operations on a large set of data (vector length) at once. The shortest way of describing this barrier is as a reconfigurable multi-degree combining tree barrier with lock-free distributed SIMD counters. It is a *combining tree* barrier because it uses a traditional combining tree data structure. It is *reconfigurable* so as the internal structure of the barrier can be reconfigured depending on the number of threads and how we want to distribute them across this structure. Because of its *multi-degree* denomination, each level of the tree may have a different degree of children. Furthermore, it has *lock-free distributed SIMD counters* as each node of the tree contains lock-free distributed counters partially orchestrated by SIMD memory operations.

4.1 Barrier Design

The barrier algorithm deploys a combining tree data structure which is walked from leaves to root in the acquisition phase and from root to leaves in the releasing phase. The degree of each level of the tree depends on the total number of threads that executes the barrier and the configuration of the nodes of that level. Each tree node in the same level groups a prefixed number of threads. This number of threads is the same for every node within that level whilst it may be different for each particular level of the tree. If the number of threads reaching a level is not divisible by the group size, the last group of that level will contain the remaining threads of the division. Fig. 2 shows the scheme of the barrier for the synchronisation of 21 threads with group sizes of 6, 2 and 2 for levels 1, 2 and 3 of the tree, respectively.

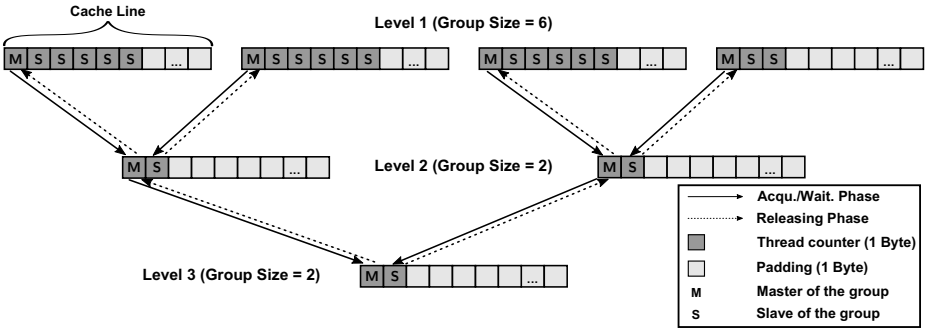


Fig. 2. SIMD Combining tree barrier scheme for 21 threads. First, second and third level group size = 6, 2 and 2, respectively. Seven nodes/*distributed SIMD counters* in total.

In each particular tree node only one thread is designated to play the master role (M) of the group while the remaining threads assume the slave role (S). These roles will affect their duties in the different phases of the barrier, described in Section 4.2 and Section 4.3.

Those threads assigned to the same tree node constitute an independent group of synchronisation. Inside this group, each thread has an exclusive one-byte counter available for taking part into the synchronisation process. All counters of the group are allocated contiguously in memory and satisfying the alignment constraints of the underlying SIMD instruction set, since they will be handled with SIMD memory operations by the master thread. To prevent false-sharing, only one group of counters is placed per cache line and the remaining memory in the line is padded. In Fig. 2 one-byte counters are depicted in dark grey and padding in light grey.

This particular tree-based design with distributed counters allows exploiting SIMD resources and the inter-thread cache locality in cores with simetric multithreading. This inter-thread locality may be useful to carry out a first intra-core syncronisation step. In addition, the tree structure also offers the possibility of reshaping the flavour of the barrier from a multi-level combining tree structure to a lock-free totally-centralised barrier. Hence, it will be possible to take advantage of two utterly different barrier algorithms with only one implementation. It will be able to choose the most appropriate one depending on the number of threads and the characteristics of the system.

4.2 Acquisition and Waiting Phase

In the acquisition phase, the intra-group synchronisation is carried out through the already mentioned distributed counters. On the one hand, slave threads signal their arrival to the barrier changing the value of their exclusive counters. It is important to note that just in case that several slaves from different cores in the same group achieve the barrier at the same time, their stores could be partially serialised as several cores will be using the same cache line. Nevertheless, groups with only slave threads of the same core will not suffer from this risk of serialisation.

On the other hand, the master thread waits for all its slaves to reach the group. To do that, the master thread busy-waits on the slave counters using SIMD instructions. In this way, just one SIMD instruction allows checking at the same time as many slave counters as bytes the vector length has. Hence, we prevent the master thread from iterating in a scalar fashion on each single counter.

Once all threads of a particular group have reached the barrier, the master thread might compute the partial reductions of its group or designate some slave to do it. Afterwards, it continues to the next level of the tree (continuous arrow in Fig. 2). In the meantime, slaves wait on their counters for the master to release them at its return. In the next level, several master threads from different groups converge at a same group, and new master and slave roles are reassigned as in the previous step.

This process is repeated until the last level (tree root) is reached. At that point, it is guaranteed that all threads have arrived at the barrier and the releasing phase starts.

4.3 Releasing Phase

In the releasing phase, the tree structure is traversed from root to leaves until all threads have been freed. In this case, starting from the root node, the master thread carries out a SIMD store on all the slave counters of the level that releases them at the same time. Then, both master and slaves move back to their respective groups in the previous level and retake their former master role (dashed arrow in Fig. 2).

As in the previous phase, these steps are applied to each level until the first level is revisited again and each master thread releases to all its slaves. It is at that point when the released slave threads and their masters are allowed to leave the barrier and the synchronisation is completed.

The biggest benefit of performing only one SIMD store is that master threads avoid the intensive time-consuming ping-pong phenomenon. This would occur if masters wrote each counter in a scalar way as slaves, in between, were requesting the same cache line for reading their counters. Thus, per each scalar store, the master could have to reclaim the ownership of the cache line since it could have been spread to some slaves.

5 Implementation

Similarly to the vast majority of the previous paper on this topic [16] [15], we opted for a combined implementation scheme of the differentiated phases of the barrier. However, an implementation scheme with several independent functions representing each phase of the barrier is feasible.

We studied several recursive and iterative approaches. Since the barrier synchronisation is a process dominated by memory transactions, we did not observe any significant variation in performance. Hence, we selected a recursive approach that uses less amount of memory at some minimal extra computation sacrifice.

```
1 void tree_barrier(int level, int level_threads, int level_thread_id)
2 {
3     if (my_level_group_offset == 0) {
4         // Master SIMD busy-waits for all its slaves in its group
5         while(!check_slaves(simd_load(&(my_level_tree_counters[my_group_id]))));
6
7         // Master computes the partial reductions of its group
8         compute_group_partial_reductions();
9
10        if (level != last_level) // Master goes to next tree level
11            tree_barrier(level+1,
12                         level_threads/group_size[level],
13                         level_thread_id/group_size[level]);
14
15        // Master releases to its slaves with only one SIMD instruction
16        simd_store(&(my_level_tree_counters[my_group_id]), init_value[level]);
17    }
18    else{
19        // Slave signals its arrival on its one-byte counter
20        my_level_tree_counters[my_level_char_id] = 0x00;
21
22        // Slave busy-wait on its scalar counter
23        while(my_level_tree_counters[my_level_char_id] == 0);
24    }
25 }
```

Fig. 3. Basic SIMD combining tree barrier algorithm (pseudocode). For simplicity, it is not shown how to deal with odd groups when threads in a particular level are not multiple of the group size.

5.1 Generic Implementation

The generic implementation of the SIMD barrier is outlined in Fig. 3. Function `tree_barrier` receives three parameters: `level` is the current level of the tree (initially 0), `level_threads` is the number of threads that reaches that level and `level_thread_id` is the thread id in that level. The `char` pointer `my_level_tree_counters` points to the group counters of the current level, initialized to the value `0xFF`. Padding is set to `0x00`.

The master thread in a group is the first thread of the group (line 3). When all its slaves have reached the barrier, the master thread computes the partial reductions of the group. After that, it proceeds to the next level calling recursively to the `tree_barrier` function (line 11). Once it returns from the recursive call, it carries out the SIMD release of its slaves (line 16) using `init_value[level]`, that corresponds to the initial value of the counters.

Slave threads signal on their one-byte local counter (line 20) and then they busy-wait on them (line 23) until their release.

5.2 SIMD Implementation for Intel MIC Architecture

Fig. 4 shows the changes from Fig. 3 that are required for incorporating the 64-bytes SIMD instructions of the Intel MIC Architecture. It is important to note that this SIMD instruction set does not support working directly on 64 one-byte `char` data types. To overcome this issue, we load and compare 64 one-byte counters using instructions for 16 four-byte integer.

```
1       // Master SIMD busy-waits for all its slaves in its group
2       while(_mm512_cmpneq_epi32_mask(
3               _mm512_load_epi32((void *) &(((volatile __m512i *)
4                       my_level_tree_counters)[my_group_id])),
5               _mm512_setzero_epi32()));
6 ...
7       // Master releases to its slaves with only one SIMD instruction
8       _mm512_storenrngo_ps((void *)&(((__m512i *)
9               my_level_tree_counters)[my_group_id]), (__m512) init_value[level]);
10
11        // Memory fence operation
12        __asm__ __volatile__ ("lock addq $0x0,(%rsp)");
```

Fig. 4. Basic tree-based SIMD barrier algorithm for the Intel MIC Architecture. Only differences from Fig. 3 are shown (Master busy-waiting and Master releasing).

Firstly, in the SIMD busy-wait of the master thread (line 2), intrinsic `__mm512_cmpneq_epi32_mask` compares if two registers, one with the group counters and another set to all-zero, are not equal. This intrinsic returns a special 16-bit integer where each bit represents the comparison between every pair of four-byte integers of the two source registers. Only when all counters are zero, the SIMD busy-wait will be done.

Secondly, to release the slave threads in a SIMD way, the master thread carries out a SIMD store (line 8). `init_value[level]` is a `__m512i` vector that contains the SIMD values to reset the counters to their initial state and release the slaves. In order to optimise this SIMD store, we use the intrinsic `_mm512_storenrngo_ps` (stream store [2]) which performs a 64-byte store with a no-read hint. This hint avoids reading the original content of the entire 64-bytes cache line since we are completely overwriting it. Moreover, it relaxes the memory consistency model in the way that these stores are not globally-ordered [3]. As a consequence of this, a memory fence-like operation (line 12) is necessary to make this store available to slave threads as soon as possible.

So as to optimise the exhaustive busy-waits of the algorithm, both the master's and the slaves', we use the intrinsic `_mm_delay_32` that stops the issue of instructions of the thread for a parameterised number of cycles. Hence, this reduces the intensity of the busy-wait and leaves more execution cycles available for those threads that are outside of the waiting region.

6 Evaluation

To measure the performance of our barrier algorithm on the Intel Xeon Phi coprocessor, we conducted some experiments on OpenMP benchmarks where barrier synchronisation primitives have different impact on the overall performance.

6.1 Benchmarks

The selection of benchmarks was particularly difficult since we needed benchmarks where the barrier strongly impacts on the overall execution time as well as it provides the coprocessor with enough parallelism. In addition, we decided to choose benchmarks

Table 2. Benchmarks setup

Benchmark	Number of Threads	Size	Executions (mean)
EPCC barrier (1st Expl.)	8,16,24,32,61,122,183,244	1.000.000	5
EPCC barrier (Best Conf.)		4.000.000	20
Livermore Loop	8,16,32,61,122,183,244	244 (≈234KB) 2440 (≈23MB) 24400 (≈2.21GB)	1000 * 20 100 * 20 10 * 20
NAS MG		S, W, A, B, C	20

that do not perform OpenMP reductions since they could significantly alter the final performance, overtaking the impact of the barrier algorithm.

Our final selection of benchmarks embraces three varied scopes. It comprises an OpenMP-target microbenchmark focused on barriers, another kernel benchmark that operates on a big amount of data with load imbalance, and one benchmark closer to a real application.

EPCC Microbenchmarks is a suite of benchmarks aimed at measuring the impact of different OpenMP parallel services [7]. For our purpose, we choose only the EPCC *barrier* where the `omp barrier` directive is intensively tested.

Livermore Loops. [14] is a set of kernels that has long been used to evaluate the optimisation capabilities of compilers. To exploit fine-grained parallelism, we choose kernel 6 and modify it in a similar way to how is done in [19]. We also add an outermost loop to run the kernel multiple times (1000, 100 and 10 in Table 2) which requires the output array to be reset to its initial values. We do that with a loop annotated with an `omp for` directive. This kernel introduces an incremental load imbalance since the number of threads that does not have work to do increases proportionally to the progress of the iterations of the outer loop.

MG Benchmark belongs to the NAS Parallel Benchmarks suite [6] and it represents a full application-like algorithm that computes the approximate solution to a scalar poisson problem using multigrid data structures. The original benchmark has been modified adding a `collape(2)` clause to the `omp for` directives in order to increase the parallelism and reduce its grain. The innermost loop is not collapsed, so each thread continues exploiting the original cache locality pattern.

6.2 Testing Environment and Methodology

We run our experiments natively in an Intel Xeon Phi coprocessor, model 7120P with C0 silicon and ECC memory mode enabled. Further details of the architecture are described in Section 3. We also set up the device with MPSS v2.1.6720-13, driver v6720-13, flash v2.1.02.0386 and device OS v2.6.38.8-g5f2543d.

The coprocessor is hosted in a system with a dual-socket Intel Xeon processor E5-2670 at 2.60GHz, with 64 GB of main memory. We use the Intel® Composer XE 2013 Update 4 C/C++ compiler and its bundled OpenMP runtime library to run all the experiments with the default Intel barrier and our SIMD approach. We set the environment

Table 3. EPCC *barrier*. Clock cycles of the default Intel OpenMP barrier and the best two configurations of the SIMD barrier on the Intel Xeon Phi coprocessor. *Some cores have less threads.

Num Thrds	Compact Binding				Balanced Binding			
	Cores (Thrds/Core)	SIMD Tree Barrier		Intel Barrier	Cores (Thrds/Core)	SIMD Tree Barrier		Intel Barrier
		Local Group Sizes	Cycles	Cycles		Local Group Sizes	Cycles	Cycles
8	2 (4)	8 4, 2	1608.67 1815.94	4565.40	8 (1)	2, 4 2, 2, 2	3654.13 4080.09	5414.75
16	4 (4)	16 4, 4	2382.03 2558.29	6434.82	16 (1)	2, 4, 2 2, 3, 3	4985.75 5029.98	7583.84
24	6 (4)	24 4, 3,	3435.15 3541.54	7336.66	24 (1)	3, 8 2, 4, 3	5918.51 5953.99	8416.42
32	8 (4)	4, 2, 4 4, 3, 3	3989.14 4012.54	8088.15	32 (1)	2, 4, 4 2, 6, 3	6320.07 6474.41	9337.46
61	16 (4*)	4, 4, 4 8, 5, 2	5170.41 5405.65	10109.74	61 (1)	4, 4, 4 3, 3, 5, 2	7299.77 8219.90	12308.85
122	31 (4*)	8, 4, 4 4, 3, 4, 3	6194.33 6351.53	12150.87	61 (2)	2, 3, 6, 4 2, 3, 5, 5	7647.77 8125.92	13306.19
183	46 (4*)	12, 4, 4 12, 3, 6	6680.26 7012.77	13278.33	61 (3)	3, 3, 6, 4 3, 3, 4, 6	7733.00 7823.09	14675.34
244	61 (4)	16, 4, 4 12, 3, 7	7583.07 7682.86	14733.50	61 (4)	16, 4, 4 12, 3, 7	7583.07 7682.86	14733.50

variables `KMP_LIBRARY` to *turnaround* and `KMP_AFFINITY` to *`binding`, granularity=thread*, where `binding` is set to *compact* (threads within the same core are used before threads of different cores) and *balanced* (threads of different cores are used before threads within the same core)[1]. We keep `KMP_BLOCKTIME` to its default value since we did not perceive any different in performance in our tests.

To run experiments with our SIMD barrier, we intercept the call to the Intel runtime function `kmp_barrier`, and replace it with a call to our barrier algorithm previously initialised. We set the value of `_mm_delay_32` of our busy-wait regions to 100 cycles and run the benchmarks with the parameters summarised in Table 2. Firstly, we explore our SIMD barrier running an exhaustive set of experiments with EPCC barrier. We repeat the executions of the 20 best configurations per threads with a larger number of barriers and executions. For the remaining benchmarks, we set up our SIMD barrier with the best configuration per thread achieved in EPCC barrier (shown in Table 3).

6.3 Results

Table 3 shows the two best configurations for each number of threads using *compact* and *balanced* thread binding policies. In general terms, as the number of threads increases, it is interesting to observe how the best configurations for the `compact` policy tend to shape a shallower tree with a larger first group. The main reason is that this binding policy allows the four threads sharing a core to carry out a first intra-core synchronisation through the local L1 cache. However, when many threads take part in the barrier, an even larger first group that includes several cores (up to 16 threads / 4 cores) comes up as the best configuration. Such configurations considerably reduce the total number of master threads which may be beneficial since the number of in-flight stores is limited in the architecture. SIMD instructions become indispensable to achieve the best performance using these large groups.

On the other hand, a single cache-line totally-centralised barrier is worth as the best configuration for a number of threads lower or equal than 24 (i.e. up to 6 cores with 4

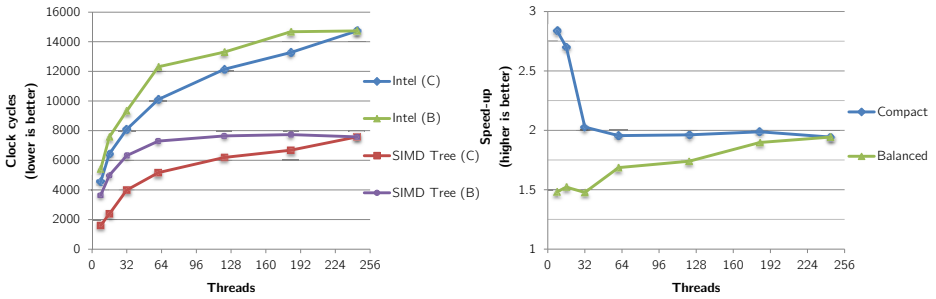


Fig. 5. EPCC *barrier*. Execution time (left) and speed-up (right) of the best SIMD barrier configuration and the default Intel barrier (speed-up baseline). *Compact* (C) and *balanced* (B) policies.

threads per core). This centralised approach makes sense when the number of threads and cores are not enough to congest the memory hierarchy. Again, SIMD instructions are crucial to achieve the best performance with these configurations.

With regard to the `balanced` policy, threads are placed farther away from each other and memory traffic increases. As a result, the best configurations deploy a deeper tree with smaller groups. The main reason is that a first intra-core synchronisation step is not possible when only one thread is bound to each core (scenarios with 61 total threads and lower). Furthermore, when a core hosts only a single thread, not all its resources can be exploited and no other threads within the core are asking for the same cache line (there is no prefetching effect). As long as the number of threads increases and cores host several threads, the first level group allows exploiting a first intra-core synchronisation step and the best configurations converge to the `compact` policy ones.

Fig. 5 plots the scalability (left chart) and the speed-up (right chart) of the best configuration of our SIMD barrier (shown in Table 3) and the default Intel OpenMP barrier. As depicted, both barriers are slower for *balanced* policy than for *compact* policy. Furthermore, both `compact` approaches scale in a logarithmic way, though the slope of the scalability curve is slightly smaller in our barrier than in Intel's.

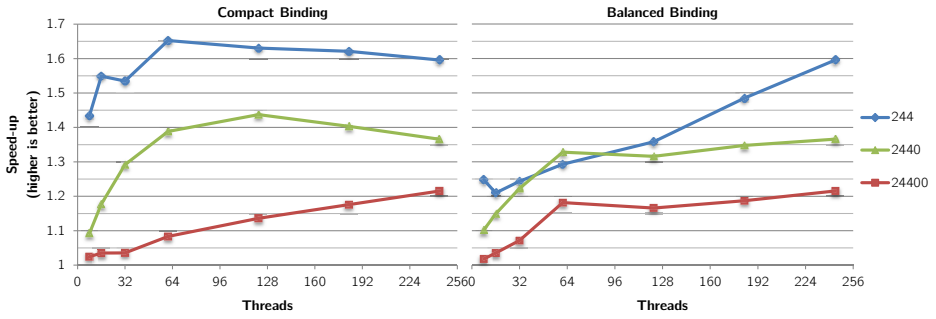


Fig. 6. 6th Livermore Loop speed-up. Best SIMD configuration versus the default Intel barrier (baseline). Benchmark size: 244, 2440 and 24400 elements.

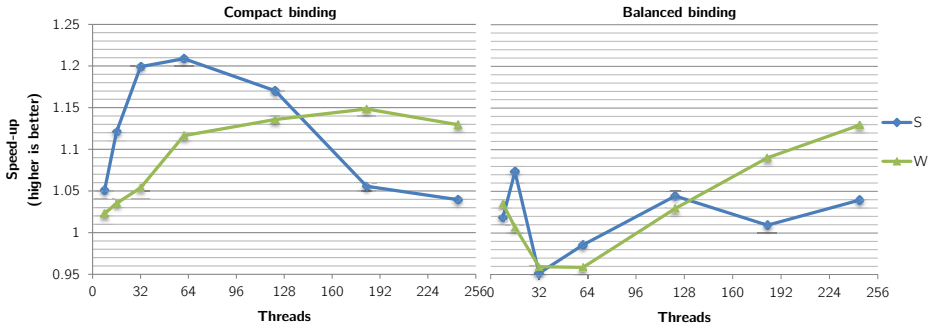


Fig. 7. MG benchmark speed-up (*compact* and *balanced* binding polices). Best SIMD configuration vs default Intel barrier (baseline). Speed-up of A, B and C sizes $\approx 1 \pm 0.05$ (not shown).

Concerning the speed-up, on the one hand the SIMD approach keeps a sustained factor of two over the Intel barrier for a large number of threads in *compact* policy. It peaks at 2.84 when this number is small. On the other hand, our barrier gains at least 48% of speed-up over Intel's with *balanced* policy, even when a first intra-core synchronisation step is not possible. However, the speed-up converges towards the *compact* policy curve as threads increases and the binding policies tend to be similar.

The Livermore Loop (Fig. 6) runs between 43% and 65% (*compact* policy), and 20% and 57% (*balanced* policy) faster using our SIMD barrier for the size of 244 elements. Here, each thread computes a small number of elements and also load imbalance is present from the beginning of the execution. In the two other cases, the speed-up is also very significant. Even for the largest size, where the barrier weight is reduced in favour of computation, our SIMD barrier gains up to a 21% of speed-up in both policies.

Fig. 7 depicts the speed-up of the SIMD barrier for the MG benchmark. Because of the high computational density of this benchmark, barriers only has relevant impact on the overall execution time of the two smallest sizes of the problem (S and W). For S size in *compact* policy, the speed-up reaches a peak for 61 threads (21%) and drops as the number of threads increases. Since the data layout of the application is distributed among a higher number of threads, the downswing may be due to the increase of the false-sharing on the application data. Thus, the weight of the barrier falls off. Regarding the *balanced* policy, the speed-up is not significant enough except from the 7% achieved with 32 threads. For W size, the speed-up mainly increases with the number of threads, reaching up to a 15% in *compact* policy and 13% in *binding* policy. With this very small input sizes, cache misses have a great impact on the overall performance, which justifies the curve fluctuations. For bigger sizes, such as A, B and C, the speed-up is sustained around 1 ± 0.05 (not shown for simplicity), since the barrier weight becomes

7 Conclusions and Future Work

In this paper we propose a reconfigurable lock-free combining tree algorithm that makes use of SIMD instructions to implement a barrier synchronisation construct. Our approach shows an outstanding improvement over the default Intel OpenMP barrier. This

is especially remarkable for the *compact* binding policy, where the algorithm better takes advantage of the inter-thread data locality to carry out a first intra-core synchronisation step.

On the Intel Xeon Phi coprocessor, we achieve a sustained 2x of speed-up with peaks of up to 2.84x in the EPCC barrier. Furthermore, our barrier improves a memory-intensive benchmark (Livermore loop) by 60% and a computation-intensive benchmark (NAS MG) by 21%.

Our exhaustive experiments show that the best barrier configuration may significantly vary for different number of threads, such as a totally centralised group of 24 threads, a 3-level tree with groups of 16, 4 and 4 threads and a 4-level tree with groups of 3, 3, 6 and 4. Therefore, a flexible and reconfigurable barrier algorithm is required. Hence, either with a lower or a larger number of threads, our results strongly support the use of SIMD to achieve the best barrier configuration for the Intel MIC architecture in our barrier scheme.

As a future work, we will deeply examine the cache coherency protocol of the Intel Xeon Phi coprocessor so that our SIMD barrier takes even more advantage of the sophisticated memory hierarchy of this architecture. We will also extend our idea of exploiting SIMD resources to other barrier algorithms.

Acknowledgments. Intel, Xeon, Xeon Phi and Many Integrated Core are trademarks or registered trademarks of Intel Corporation or its subsidiaries in the United States and other countries.

This work was supported by AGAUR through the grant FI-DGR 2012 (FI_B 00295), the European Commission through the DEEP project (FP7-ICT-287530), the Spanish Ministry of Education (contracts TIN2012-34557, TIN2007-60625, CSD2007-00050), and the Generalitat de Catalunya (contract 2009-SGR-980).

References

1. Balanced affinity type. Intel® C++ Compiler XE 13.1 User and Reference Guides, `http://software.intel.com/sites/products/documentation/doclib/iss/2013/compiler/cpp-lin/` (accessed: May 09,2013)
2. Intel® Xeon Phi ™ Coprocessor - The Architecture, `http://software.intel.com/en-us/articles/intel-xeon-phi-coprocessor-codename-knights-corner` (accessed: May 09, 2013)
3. Intel® Xeon Phi ™ Coprocessor Instruction Set Architecture Reference Manual (2012)
4. Abellán, J.L., Fernández, J., Acacio, M.E.: Efficient and scalable barrier synchronization for many-core CMPs. In: Proceedings of the 7th ACM International Conference on Computing Frontiers, CF 2010, pp. 73–74 (2010)
5. Almási, G., Heidelberger, P., Archer, C.J., Martorell, X., Erway, C.C., Moreira, J.E., Steinmacher-Burow, B., Zheng, Y.: Optimization of MPI collective communication on BlueGene/L systems. In: Proc. of the 19th Int. Conf. on Supercomp., ICS 2005 (2005)

6. Bailey, D.H., Barszcz, E., Barton, J.T., Browning, D.S., Carter, R.L., Dagum, L., Fatoohi, R.A., Frederickson, P.O., Lasinski, T.A., Schreiber, R.S., Simon, H.D., Venkatakrishnan, V., Weeratunga, S.K.: The NAS parallel benchmarks - summary and preliminary results. In: Proc. of the 1991 ACM/IEEE Conf. on Supercomp., SC 1991, pp. 158–165 (1991)
7. Bull, J.M., Reid, F., McDonnell, N.: A microbenchmark suite for openMP tasks. In: Chapman, B.M., Massaioli, F., Müller, M.S., Rorro, M. (eds.) IWOMP 2012. LNCS, vol. 7312, pp. 271–274. Springer, Heidelberg (2012)
8. Eichenberger, A.E., Abraham, S.G.: Impact of load imbalance on the design of software barriers. In: Proc. of the 1995 Int. Conf. on Parallel Processing, pp. 63–72 (1995)
9. Gottlieb, A., Grishman, R., Kruskal, C.P., McAuliffe, K.P., Rudolph, L., Snir, M.: The NYU ultracomputer. designing an MIMD shared memory parallel computer. IEEE Transactions on Computers C-32(2), 175–189 (1983)
10. Gupta, R., Hill, C.R.: A scalable implementation of barrier synchronization using an adaptive combining tree. Internat. Journal of Parallel Programming 18(3), 161–180 (1989)
11. Gupta, R.: The fuzzy barrier: a mechanism for high speed synchronization of processors. SIGARCH Comput. Archit. News 17(2), 54–63 (1989)
12. Hoefler, T., Mehlan, T., Mietke, F., Rehm, W.: A survey of barrier algorithms for coarse grained supercomputers chemnitzer informatik berichte (2004)
13. Huang, W., Stant, M.R., Sankaranarayanan, K., Ribando, R.J., Skadron, K.: Many-core design from a thermal perspective. In: Proceed. of the 45th Annual Design Automation Conference, DAC 2008, pp. 746–749. ACM, New York (2008)
14. McMahon, F.H.: The Livermore Fortran kernels: A computer test of the numerical performance range (1986)
15. Mellor-Crummey, J.M., Scott, M.L.: Algorithms for scalable synchronization on shared-memory multiprocessors. ACM Trans. Comput. Syst. 9(1), 21–65 (1991)
16. Nanjegowda, R., Hernandez, O., Chapman, B., Jin, H.H.: Scalability evaluation of barrier algorithms for openMP. In: Müller, M.S., de Supinski, B.R., Chapman, B.M. (eds.) IWOMP 2009. LNCS, vol. 5568, pp. 42–52. Springer, Heidelberg (2009)
17. Petrini, F., Kerbyson, D.J., Pakin, S.: The case of the missing supercomputer performance: Achieving optimal performance on the 8,192 processors of ASCI Q. In: Proceedings of the 2003 ACM/IEEE Conference on Supercomputing, SC 2003, p. 55 (2003)
18. Pfister, G.F., Norton, V.A.: Hot-spot contention and combining in multistage interconnection networks. IEEE Transactions on Computers C-34(10), 943–948 (1985)
19. Sampson, J., Gonzalez, R., Collard, J., Jouppi, N.P., Schlansker, M., Calder, B.: Exploiting fine-grained data parallelism with chip multiprocessors and fast barriers. In: Proc. of the 39th Annual IEEE/ACM Int. Symp. on Microarchitecture, MICRO 39, pp. 235–246 (2006)
20. Sartori, J., Kumar, R.: Low-overhead, high-speed multi-core barrier synchronization. In: Patt, Y.N., Foglia, P., Duesterwald, E., Faraboschi, P., Martorell, X. (eds.) HiPEAC 2010. LNCS, vol. 5952, pp. 18–34. Springer, Heidelberg (2010)
21. Scott, M.L., Mellor-Crummey, J.M.: Fast, contention-free combining tree barriers for shared-memory multiprocessors. Int. Journal of Parallel Prog. 22(4), 449–481 (1994)
22. Scott, S.L.: Synchronization and communication in the T3E multiprocessor. SIGPLAN Not. 31(9), 26–36 (1996)
23. Villa, O., Palermo, G., Silvano, C.: Efficiency and scalability of barrier synchronization on NoC based many-core architectures. In: Proceedings of the 2008 International Conference on Compilers, Architectures and Synthesis for Embedded Systems, pp. 81–90 (2008)
24. Yew, P., Tzeng, N., Lawrie, D.H.: Distributing hot-spot addressing in large-scale multiprocessors. IEEE Transactions on Computers C-36(4), 388–395 (1987)
25. Zhang, G., Martínez, F., Tal, A., Blainey, B.: Busy-wait barrier synchronization using distributed counters with local sensor. In: Proc. of the WOMPAT, pp. 84–98 (2003)

OpenMP on the Low-Power TI Keystone II ARM/DSP System-on-Chip

Eric Stotzer[1], Ajay Jayaraj[1], Murtaza Ali[1], Arnon Friedmann[1], Gaurav Mitra[2], Alistair P. Rendell[2], and Ian Lintault[3]

[1] Texas Instruments, Dallas TX, USA
{estotzer,ajayj,mali,arnon}@ti.com
[2] Australian National University, Canberra ACT, Australia
{gaurav.mitra,alistair.rendell}@anu.edu.au
[3] nCore HPC, USA
ian@ncorehpc.com

Abstract. The Texas Instrument (TI) Keystone II architecture integrates an octa-core C66X DSP with a quad-core ARM Cortex A15 MPCore processor in a non-cache coherent shared memory environment. This System-on-a-Chip (SoC) offers very high Floating Point Operations per second (FLOPS) per Watt, if used efficiently. This paper reports an initial attempt at developing a bare-metal OpenMP runtime for the C66X multi-core DSP using the Open Event Machine RTOS. It also outlines an extension to OpenMP that allows code to run across both the ARM and the DSP cores simultaneously. Preliminary performance data for OpenMP constructs running on the ARM and DSP parts of the SoC are given and compared with other current processors.

1 Introduction

High performance computing has evolved to use specialized accelerators such as Graphics Processing Units (GPUs) for a variety of problems. However, such accelerators suffer from two main issues of excessive power consumption and insufficient device memory. Low-power SoCs with on-chip accelerators sharing the same address space and physical memory are increasingly being considered as alternatives. In recent work on using ARM NEON Floating Point Units (FPU) to accelerate application codes [1], low-power ARM based SoCs demonstrated comparable performance speedups to Intel processors using SSE2.

It has also been demonstrated that the TI Keystone I C66X Multi-core DSP provides higher GFLOPS/Watt (with 57% utilization of resources) for SGEMM matrix multiplication using OpenMP than Intel Core i7-960, IBM Cell Broadband Engine, Stratix IV FPGA and NVIDIA GTX480, GTX280 systems [2]. Increase in net utilization of the C66x DSP for other applications would result in higher GFLOPS/Watt. The TI Keystone II architecture integrates this C66X octa-core DSP with a quad-core ARM Cortex A15 MPCore processor. This combination of ARM and DSP processors on the same SoC promises excellent energy efficient performance if used efficiently.

A.P. Rendell et al. (Eds.): IWOMP 2013, LNCS 8122, pp. 114–127, 2013.

Performance of OpenMP based applications depends heavily on the runtime library implementation. The runtime in [2] used the underlying SYS/BIOS RTOS. In this paper, we demonstrate a lighter weight OpenMP runtime implementation for the C66X multi-core DSP using the Open Event Machine RTOS. For key OpenMP directives, overheads are 2.5× lower than the previous implementation using SYS/BIOS. EPCC v3 micro-benchmarks [3] are provided for Keystone II and other Intel, ARM processors for comparison.

The rest of the paper is organized as follows. Section 2 provides a concise overview of the TI Keystone architecture. Description of our new bare-metal implementation of the OpenMP runtime for C66X DSP is given in Section 3. A brief introduction to our OpenMP accelerator dispatch prototype is outlined in Section 4. Micro-benchmarks of CPU cycle overheads for OpenMP constructs are discussed in Section 5. Related work, conclusions and future work, and acknowledgements follow in Sections 6, 7 and 7.

2 TI Keystone Overview

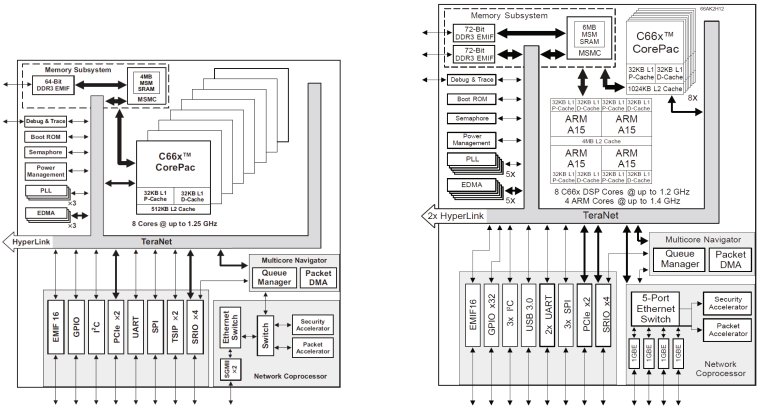
The Keystone architecture from Texas Instruments is an innovative platform integrating RISC and DSP cores along with application-specific co-processors and input/output peripherals. This high performance structure includes adequate internal bandwidth for non-blocking access to all processing cores, peripherals, co-processors and I/O. Figure 1 shows two instantiations of the Keystone architecture applicable to high performance compute applications.

2.1 C66x DSP Core

The main compute core inside the Keystone architecture is the C66x DSP from Texas Instruments [4]. This is based on a Very Long Instruction Word (VLIW) architecture. The core has two data-paths, each capable of executing four instructions per cycle on four functional units named M, D, L and S. The M unit primarily performs multiplication operations, the D-unit performs load/store and address calculations, and the L and S units perform addition and logical operations. Overall the two data-paths appear as an 8-way VLIW machine capable of executing up to eight instructions in each cycle. The instruction set also includes Single Instruction Multiple Data (SIMD) instructions allowing vector processing on up to 128-bit vectors. For example, the M unit can perform four single precision multiplies per cycles, whereas each L and S unit can each perform two single precision additions per cycle. Together the two data-paths can issue 16 single FLOP per cycle. The double precision capability is about one-fourth of single precision FLOPs.

2.2 C6678 'Shannon' System-on-Chip

The C6678 System-on-Chip (SoC) is the highest performance Keystone I device that includes only DSP cores [5]. Figure 1(a) shows the block diagram of



(a) C6678 SoC based on Keystone I Architecture

(b) 66AK2H12 SoC based on Keystone II Architecture

Fig. 1. Texas Instruments Keystone Architectures

this device. It has eight C66x cores and a three-level memory system. The cores can run at 1.25 GHz, thereby providing a peak performance of 160 single precision GFLOPS and 40 double precision GFLOPS. The memory system is a Non-Uniform Memory Architecture (NUMA) [6]. A C66x subsystem can access different memory regions, with accesses to memories that are physically closer to a processor being faster. The memory regions and access times are as follows:

- Level-1 program (L1P) and data (L1D): 32KB, 1-cycle access time, configurable as mapped RAM, cache, or a combination of mapped and cached.
- Local-L2: 512KB, 2-cycle access time, configurable as mapped RAM, cache, or a combination of mapped and cached, and shared between the L1D and L1P caches.
- Shared-L2: 4MB, 2-cycle access time, shared memory on-chip.
- Shared-L3: multiple megabytes of off-chip memory with greater than 60-cycle access time. One DDR3 controller in this device.

The device comes with a rich set of standard interfaces like PCI express, Serial Rapid I/O (SRIO), multiple Gigabit Ethernet ports as well as a proprietary interface known as the Hyperlink that provides a 50 Gbps point-to-point connectivity.

2.3 66AK2H12 'Hawking' System-on-Chip

The 66AK2H12 SoC is the highest performance Keystone II device architecture that includes an ARM RISC processor in addition to the compute-efficient C66x DSP cores [7]. This particular device (Figure 1(b)) integrates a Cortex-A15 quad-core cluster and a C66x DSP octa-core cluster. The Cortex-A15 quad cores are fully cache coherent, although as on the C6678 the DSP cores do not maintain cache coherency. External memory bandwidth is doubled with dual DDR3 controllers. An additional Hyperlink interface is also included. Compared to the

Keystone I C6678 SoC the memory sizes are also increased. On the DSP, the L1D and L1P cache sizes remain at 32KB per core, but the L2 cache size is increased to 1024 KB per core. On the ARM side, there is 32 KB of L1D and 32 KB of L1P cache per core, and a coherent 4 MB L2 cache. The level 2 shared memory is increased to 6 MB and is accessible by all ARM and DSP cores. This SoC brings the ARM-based processor and DSP accelerator together in the same memory address space, along with infrastructure class I/O peripherals. It therefore provides an attractive low-power alternative for HPC applications.

3 Bare-Metal Implementation of OpenMP on C66X DSP

Most compilers translate OpenMP into multi-threaded code with calls to a custom runtime library, either via outlining [8] or inlining [9]. Because many execution details are often unknown in advance, much of the actual work of assigning computations must be performed dynamically. Part of the implementation complexity is in ensuring that the presence of OpenMP constructs does not impede sequential optimization in the compiler. An efficient runtime library to support thread management and scheduling, and shared memory and fine-grained synchronization execution is essential.

The basic hardware and operating environment of the DSP cores on the Keystone I and II systems presents some special challenges when seeking to support the OpenMP programming model. Notably the shared memory controller in Keystone devices does not maintain coherency between the C66X subsystems, and it is the responsibility of the running program to use synchronization mechanisms and cache control operations to maintain coherent views of the memories. (Coherency within a given C6X subsystem for all levels of memory is maintained by the hardware).

Traditionally, application codes executing across multiple C6X subsystems are required to explicitly manage thread synchronization and cache coherence, and communicate via the shared-L2 and shared-L3 memories. A processor can transfer a data buffer to the local-L2 via a direct memory access (DMA) controller. The hardware maintains L1D cache coherency with the local-L2 for DMA accesses. Also, the DMA transfer completion event can be used as a synchronization event between the data producer and data consumer. There is no virtual memory management unit (MMU), but a memory protection mechanism protects some shared memory from being accessed by a non-authorized processor.

TI provides a light-weight multi-core task dispatch API called Open Event Machine (OpenEM). OpenEM is designed to require minimal memory and CPU cycles [10]. OpenEM is implemented to leverage the C66X SoC's Navigator hardware queues. Various types of interactions between cores, such as blocking, communication and synchronization, are implemented by OpenEM. OpenEM also provides a fast, shared, thread-safe memory management system that is used to allocate/deallocate memory in the runtime.

An understanding of the memory model used by OpenMP is fundamental to its implementation on the Keystone I/II systems. In this respect, OpenMP specifies a *relaxed consistency* memory model that is close to weak consistency [11].

In this model threads execute in parallel with a temporary view of shared memory until they reach memory synchronization or *flush* points in the execution flow. At a flush point, threads are required to write back and invalidate their temporary view of memory. After the memory synchronization point, threads again have a temporary view of memory.

Although the C66x provides a shared memory, its consistency is not automatically maintained by the hardware. It is the responsibility of the OpenMP runtime library to perform the appropriate cache control operations to maintain the consistency of the shared memory when required.

3.1 Memory Model

OpenMP has both shared and private variables. Each thread has its own copy of a private variable that the other threads cannot access. There is only one copy of a shared variable, and all threads can access it. Private variables are located on the stack of each thread of execution. The stack can be placed in any of on-chip local, on-chip shared or off-chip shared memory.

OpenMP requires that threads synchronize their local view of shared variables with the global view at a set of implicit and explicit flush points defined in the OpenMP specification. The runtime performs this synchronization in software. The synchronization steps depend on whether the shared variable is placed in on-chip local memory (L2SRAM) or on-chip/off-chip shared memory (MSMCSRAM/DDR) as follows:

- Shared variables in on-chip "local" scratch memory(L2SRAM)
  1. L2SRAM on a core is accessible to external DSP cores via a global address space
  2. Any updates to L2 scratch by external DSP cores are kept coherent by the memory subsystem
  3. The runtime performs a write-back invalidate of L1 at all flush points.
- Shared variables in on-chip/off-chip shared memory (MSMCSRAM/DDR)
  1. Shared memory regions are marked write-through
  2. The runtime performs cache invalidate operations at all flush points. Since write-through is enabled shared memory has already been updated and there is no need to write-back data.

3.2 Parallel Regions

The essential parts of the OpenMP runtime library are implemented using the OpenEM API. For each parallel region, the OpenMP compiler divides the workload into chunks that are assigned to OpenEM tasks (micro-tasks) at runtime. One of the DSP cores is treated as a master core and the other cores are worker cores. The master core runs the main thread of execution. It is responsible for initializing the OpenMP runtime and starts executing the OpenMP program (main). The worker cores wait in a dispatch loop for OpenEM tasks to show up in a queue.

A parallel region's fork-join mechanism is implemented by the following steps:

1. After initialization, worker threads wait in a dispatch loop and check a task queue for micro-task execution notification.
2. The master thread assigns micro-tasks to worker threads by posting the micro-tasks to an OpenEM queue. The micro-task description includes a function pointer and a data pointer. It also initializes a shared counter to the number of micro-tasks generated.
3. Worker cores pull micro-tasks out of the OpenEM queue. Upon receipt of the micro-task, each worker thread executes the micro-task specified by the function pointer. The data pointer is passed as an argument to the micro-task.
4. Upon completion of a micro-task, the worker core that executed the micro-task decrements the shared micro-task counter.
5. After the master completes the execution of its own chunk, it waits for the shared micro-task counter to reach 0, indicating that all workers have completed their micro-tasks.

3.3 Synchronization

The runtime implements three methods of synchronization depending on what is being synchronized:

1. To synchronize master and worker cores during runtime initialization, a fast synchronization mechanism is implemented using coherent shared memory to store a vector. Each core independently sets or clears an element in the vector. Every core can concurrently query the entire vector by using a single 64-bit memory access. As shown in Figure 2, this mechanism is based on Lamport's Bakery algorithm [12]. The buffers are stored in non-cacheable shared memory. The message queue is used at the start of a parallel region, but all other synchronizations are performed using this new mechanism.

```
1   void sync(char buf0[8], char buf1[8])
    {
3       int core_id = get_core_id();

5       buf1[core_id] = 1;
        buf0[core_id] = 0;
7       /* wait until all threads have cleared buf0 */
        while (*(volatile long long *)buf0 != 0) ;
9
        buf1[core_id] = 0;
11      /* wait until all threads have cleared buf1 */
        while (*(volatile long long *)buf1 != 0) ;
13
        /* reset buf0 */
15      buf0[core_id] = 1;
    }
```

Fig. 2. Fast synchronization mechanism using coherent shared memory

2. To synchronize master and worker cores at the end of a parallel region, a shared counter, as described in Section 3.2 is used.
3. For implicit and explicit OpenMP barriers, the sense reversing barrier shown in Figure 3 is used. This has a counter that keeps track of the number of cores participating in the barrier and a sense flag to allow the barrier to be re-used. To avoid coherency overheads, the barrier variable is placed in non-cached memory.

```
void sense_reversing_barrier(Barrier *barrier)
{
    /* To allow re-use, the barrier contains a sense variation */
    char mysense = !barrier->sense;

    if (atomic_decrement(barrier->count) == 1)
    {
        /* Last thread resets the sense and count */
        barrier->count = barrier->size;
        barrier->sense = !(barrier->sense);
    }
    else
    {
        /* Modification of sense represents end of the barrier */
        while (mysense != barrier->sense);
    }
}
```

Fig. 3. Sense reversing barrier

4 ARM to DSP OpenMP Dispatch

We have implemented an early prototype of the OpenMP 4.0 accelerator extension [13,14] *target* construct. Our prototype implementation uses the *dispatch* keyword along with memory *copyin* and *copyout*. A host program uses the dispatch construct in the following way:

```
void foo(int *in1, int *in2, int *out1, int count)
{
    #pragma omp dispatch copyin  (in1[0:count-1], in2[0:count-1], count) \
                         copyout (out1[0:count-1])
    {
        #pragma omp parallel shared(in1, in2, out1)
        {
            int i;
            #pragma omp for
            for (i = 0; i < count; i++)
                out1[i] = in1[i] + in2[i];
        }
    }
}
```

Fig. 4. Usage of dispatch construct in host program

A source-to-source translator is used to transform the initial source file with code outlined in Figure 4 to produce the ARM host side and DSP target side annotated code, as shown in Figure 5. The ARM host annotations include data movement calls. In Keystone II, these calls resort to memory maps using the UNIX *mmap* system call to leverage physical shared memory between the ARM and DSP. Therefore copyin/copyout operations have minimal overhead. The DSP source annotations made by the translator include standard OpenMP pragma additions. These two separate source files are then compiled with the gcc toolchain on the ARM side, and the TI Code Generation Tools OpenMP compiler toolchain and our OpenMP DSP runtime library on the DSP side to produce a fat executable with the DSP binary embedded inside the ARM executable. This is launched from the ARM using the TI Multi-Process Manager which loads and runs the DSP binary and the ARM host executable. Shared memory regions are set up, synchronization messages are exchanged between the ARM and DSP and the required functions are then run on DSP, which writes back the result to shared memory. An initial implementation of the dispatch construct was tested on the Appleton EVM which has a TCI6614 SoC [15] with ARM Cortex-A8 and quad-core C66X DSP on-chip. Porting the dispatch construct to utilize our current implementation of the OpenMP runtime on Keystone II is in progress.

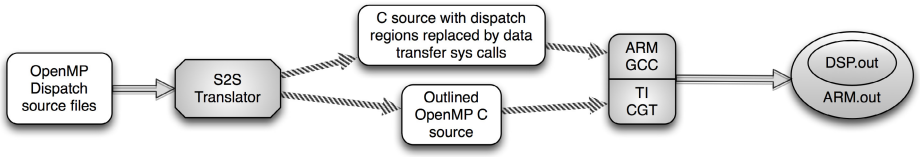


Fig. 5. Dispatch executable compilation strategy

5 Evaluation Using Micro-benchmarks

In order to evaluate the performance of our DSP runtime implementation, we used the EPCC v3 benchmark [3]. For comparison we also collected data across other contemporary ARM and Intel platforms. The EPCC suite of micro-benchmarks measure the time overheads associated with invoking the different OpenMP constructs. For example, the cost of *parallel* to create a parallel region or *barrier* to synchronize threads. Here we report the overheads associated with some of the most widely used constructs.

Table 1 lists the systems considered and their main characteristics. Two different Intel platforms with at least four physical cores were considered. The Core2 Q9400 Yorkfield processor, has four cores running at 2.66 Ghz. The hexa-core Xeon X5650 Westmere processor is part of a dual-socket system and runs at 2.66 Ghz. The ARM platforms considered, include the Keystone II Hawking EVM's ARM Cortex A15 quad-core processor running at 625 Mhz and a Samsung Exynos 4412 prime SoC with quad-core ARM Cortex-A9 processors running at 2 Ghz. The ARM Cortex A15 is referred to as Hawking-A15 and the A9 as Exynos-A9. The octa-core C66X DSP processor in Keystone II Hawking EVM ran at 983 Mhz and is referred to as Hawking-DSP.

Table 1. Platforms Used in Benchmarks

PROCESSOR	CODENAME	Threads/Cores/Ghz	Memory
Intel Core2 Q9400	YorkField	4/4/2.67	8GB DDR3
Intel Xeon X5650	Westmere	12/6/2.67	24GB DDR3
Samsung Exynos 4412 (ARM)	Odroid-X2	4/4.Cortex-A9/2.0	2GB LPDDR2
TI Keystone II (ARM)	Hawking	4/4.Cortex-A15/0.625	2GB DDR3
TI Keystone II (DSP)	Hawking	8/8.C6678 DSP/0.983	2GB DDR3

5.1 Compilers and Tools

For the Intel Westmere platform we used GCC 4.6.4 and ICC 13.1.1 (compatible with GCC 4.6) to separately compile and run the benchmarks. These versions are denoted as X5650-GCC and X5650-ICC. They were linked against libgomp and libiomp5 respectively. On the Intel Yorkfield and ARM platforms GCC 4.7.3 with libgomp was used. The compiler option *-mcpu=cortex-a9* was used on the Exynos and *-mcpu=cortex-a15* on the Hawking. In addition both ARM platforms used the *-mfpu=neon,mfloat-abi=hard* compiler flag. TI Code Generation

Tools 7.4.2, XDC Tools 3.24.05.48, OpenEM 1.2.0.1, PDK Keystone2 1.00.00.09, PDK C6678 1.1.2.6 along with our current version of the OpenMP runtime were used to create the executable for the C6678 DSP. All platforms except the Hawking-A15 and the DSPs were running Ubuntu Linux with kernel version greater than 3.0. The Hawking ARM cores used a custom distribution of Linux, called Arago, built specifically for the Hawking EVM. It includes the 3.8.4 Linux kernel. On the Linux hosts, the `OMP_PROC_BIND` and `GOMP_CPU_AFFINITY` environment variables were set to bind threads to processor cores and to prevent thread migration between cores. For timing measurements on the Intel and ARM platforms, the EPCC v3 timer function `getclock()` which uses `omp_get_wtime()` remained unchanged and reported times in microseconds. Measurements on the DSP required modifications to the timer function. A native time-stamp counter was used to measure the exact CPU cycles elapsed as shown in Figure 6. For direct comparison of all platforms, all time measurements were normalized w.r.t CPU clock speed and reported in CPU cycles using the equation, $cpu\_cycles = overhead\_time(\mu s) * mhz$.

```
/* Wall cycles using TSC_read */
void wcycles(unsigned long long *c)
{
    static int first = 1;
    extern void TSC_enable(void);
    extern unsigned long long TSC_read(void);
    if (first)
    {
        TSC_enable();
        first = 0;
    }
    *c = TSC_read();
}

/* TSC_enable Assembly Code */
.global TSC_enable

TSC_enable:

RETNOP  B3, 4
MVC     B4, TSCL  ; writing any value enables timer

/* TSC_read Assembly Code*/
.global TSC_read

TSC_read:

RETNOP  B3, 2
DINT
MVC TSCH,   B5    ; Read the snapshot of the high half
MV  B5,  A5
```

Fig. 6. Measuring cpu cycles on the DSP

5.2 Discussion

A crucial difference between the multi-core DSP in the Keystone architectures and other processors evaluated in this study is cache coherence. While the Intel and ARM multi-core processors have hardware managed cache coherence protocols, programs running on the multi-core DSP have to ensure cache coherence in software. As explained in Section 3, the Keystone shared memory controller does not ensure this memory consistency. As a result we perform *flush* operations at implicit and explicit synchronization points in our OpenMP runtime, which invalidate L1 and L2 caches and write them back to main memory. In our

Table 2. Cost of software managed cache coherency operation for DSP (cycles)

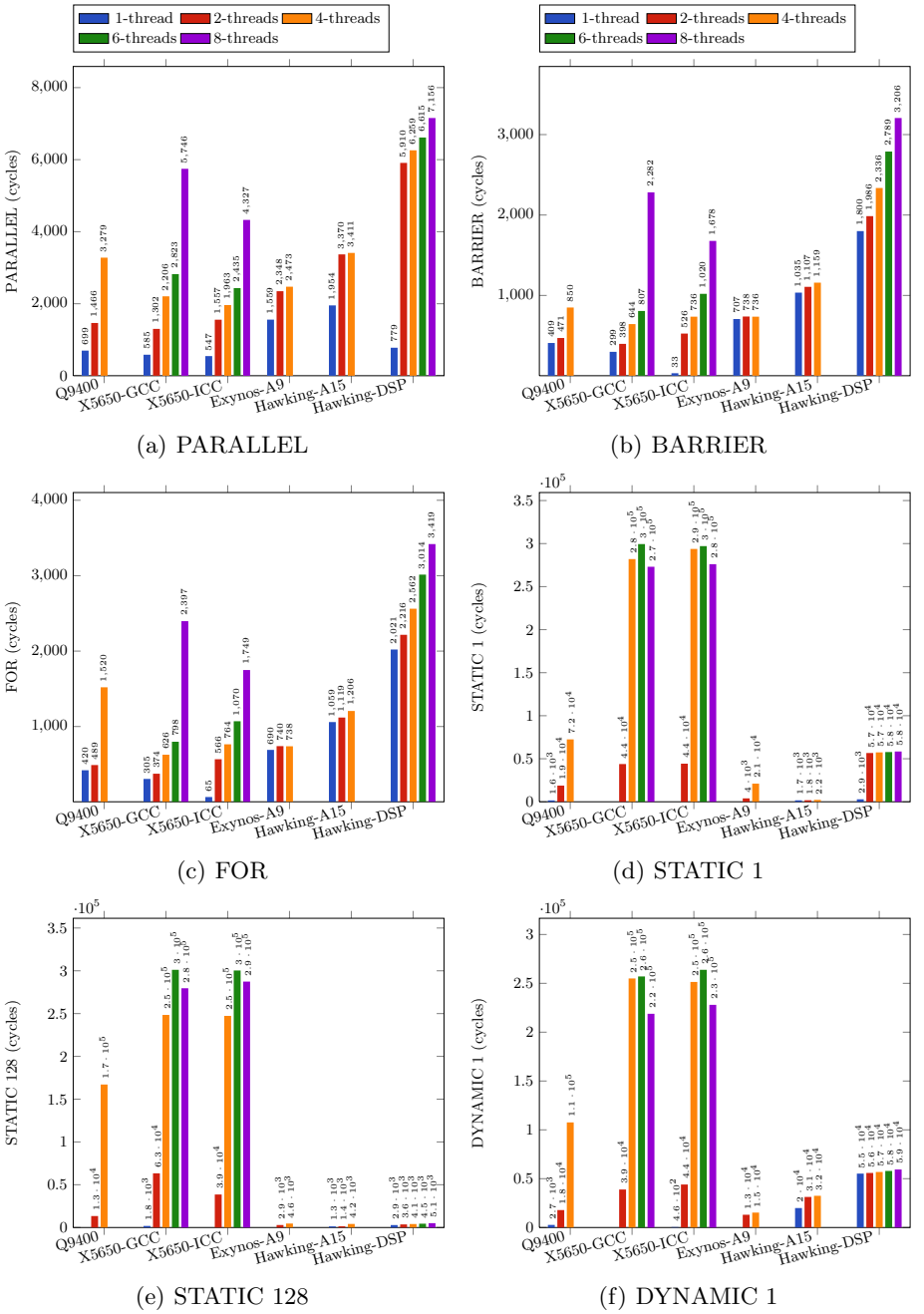
DSP (L2 = 0)	1 Thread	2 Threads	4 Threads	6 Threads	8 Threads
Hawking-DSP	1350	1355	1357	1353	1364

evaluation, we set DSP L2 cache to be 0K to minimize flush overhead. Table 2 presents operation cycle counts on Hawking DSPs averaged over 200 iterations. This shows the cost to be roughly 1350 cycles regardless of thread count.

Results for the EPCC benchmarks are given in Figure 7. In each bar-graph the cpu cycle overhead values are given for each platform using 1-8 OpenMP threads. For platforms with 4 cores, only results with up to 4 threads are given.The PARALLEL construct is most fundamental, specifying the creation of an OpenMP parallel region and spawning a team of threads. Each of the platforms in Figure 7(a) demonstrates the expected behaviour of increasing cycles overhead with increasing number of threads. The X5650-ICC results show the least 1-thread overhead time of 585 cycles. Not surprisingly a sharp increase is seen on the X5650 with both the GCC and ICC compilers in going from 6-thread to 8-threads as the last 2 threads are spawned on a different socket. The Exynos and Hawking ARM processors show comparable overheads to the Intel platforms. Overheads for more than 1-threads observed on the DSP are higher than Intel and ARM platforms. However, each parallel region incorporates implicit cache-coherence flush operations. Discounting the cost of these flush operations, performance of the OpenMP DSP runtime is comparable to Intel and ARM processors.

The BARRIER construct is used to specify an explicit synchronization point inside a parallel region which all threads must reach for any to progress beyond that point. As shown in Figure 7(b), X5650-ICC performs the best among all platforms across all thread configurations. Similar to PARALLEL overheads, the ARM platforms have comparable times to Intel. The Hawking DSP has cycle overheads of between 1800 and 3206. Subtracting the cost of 1 flush operation from these yields overheads of between 450 and 1842. The latter are almost on a par with the X5650-GCC values. The FOR construct is used to split for loops between thread and data in Figure 7(c) and show similar patterns to those observed for PARALLEL and BARRIER.

The STATIC constructs are used to specify compile-time scheduling of loop iterations between threads. STATIC 1 indicates that each thread gets 1 loop iteration to process at a time, while STATIC 128 gives 128 loop iterations at a time. Figure 7(d) and 7(e) show that the DSP platforms perform significantly better than the Intel platforms, while the ARM platforms perform the best overall. This suggests that the chunk sizes of 1 and 128 are not ideal for the memory hierarchy and architecture of the Intel platforms, but are more suited for the ARM and DSP platforms. It also shows that the DSP OpenMP runtime performs effective static scheduling for these chunk sizes. The DYNAMIC construct is similar to STATIC in that it partitions the scheduling of loop iterations between threads. However, in contrast to STATIC the loop iterations are now partitioned dynamically with the next available thread executing the next loop iteration.



(a) PARALLEL (b) BARRIER

(c) FOR (d) STATIC 1

(e) STATIC 128 (f) DYNAMIC 1

Fig. 7. Cost comparison of OpenMP constructs in CPU cycles

This permits load balancing when iterations give rise to variable work. Interestingly the results for DYNAMIC 1 scheduling in Figure 7(f) are very similar to those for STATIC 1.

6 Related Work

Use of TI C66X DSPs for HPC is demonstrated in [2,16,17,18]. Directive based programming for GPU accelerators has most recently been standardized by OpenACC [19] after previous implementations such as hiCUDA [20] and PGI Accelerator Model [21]. IBM provides an OpenMP compiler [22] and runtime library for the Cell Broadband Engine. Extensions to OpenMP to support accelerators were introduced in [23,24,25,14]. Various RTOSs such as SYS/BIOS [26] and DSP/BIOS [10] have been used on the C6678 DSP. [27] provides an OpenMP runtime using DSP/BIOS for the TI C64x DSP. A bare-metal implementation of an OpenMP runtime for the Cradle CT3400 multi-core DSP is outlined in [28].

7 Conclusions and Future Work

We have presented our initial experiences with implementing a bare-metal OpenMP runtime for the Keystone II C66X multi-core DSP. We addressed various challenges such as lack of memory management units and cache coherence as part of our implementation process. CPU cycle overheads for various OpenMP synchronization and scheduling constructs were measured using the EPCC micro-benchmarks on the C66X DSP and other contemporary Intel and ARM processors. The results demonstrated that our DSP runtime performed at par or better than Intel and GCC OpenMP runtimes for most OpenMP constructs. Software managed cache coherence was acknowledged to be a limiting factor for DSP runtime performance. Porting and evaluating the efficiency and performance of the OpenMP accelerator dispatch construct to Keystone II is a future work item. Evaluation of OpenMP task performance on C66X DSP and implementation of task dispatch from ARM to DSP is of interest. Measurement of energy efficiency and performance of various application codes using OpenMP is also of interest.

Acknowledgements. This work is supported in part by the Australian Research Council Discovery Project DP0987773. We thank Andrew Tridgell for providing us with an the ODROID-X2 development platform.

References

1. Mitra, G., Johnston, B., Rendell, A.P., McCreath, E., Zhou, J.: Use of SIMD vector operations to accelerate application code performance on low-powered ARM and Intel platforms. In: Parallel and Distributed Processing Symposium Workshops & PhD Forum (IPDPSW). IEEE (2013)

2. Igual, F.D., Ali, M., Friedmann, A., Stotzer, E., Wentz, T., van de Geijn, R.A.: Unleashing the high-performance and low-power of multi-core dsps for general-purpose hpc. In: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis, vol. 26. IEEE Computer Society Press (2012)
3. Bull, J.M., Reid, F., McDonnell, N.: A microbenchmark suite for openMP tasks. In: Chapman, B.M., et al. (eds.) IWOMP 2012. LNCS, vol. 7312, pp. 271–274. Springer, Heidelberg (2012)
4. Texas Instruments Literature: SPRUGH7: TMS320C66x DSP CPU and Instruction Set Reference Guide
5. Texas Instruments Literature: SPRS691C: TMS320C6678 Multicore Fixed and Floating-Point Digital Signal Processor
6. Hennessy, J.L., Patterson, D.A.: Computer Architecture: A Quantitative Approach. Morgan Kaufmann Publishers Inc., San Francisco (2003)
7. Texas Instruments Literature: SPRS866: 66AK2H12/06 Multicore DSP+ARM Keystone II System-on-Chip (SoC)
8. Brunschen, C., Brorsson, M.: OdinMP/CCp - a portable implementation of OpenMP for C. Concurrency - Practice and Experience 12(12), 1193–1203 (2000)
9. Liao, C., Hernandez, O., Chapman, B., Chen, W., Zheng, W.: OpenUH: An optimizing, portable OpenMP compiler. In: Concurrency and Computation: Practice and Experience, Special Issueon CPC 2006 selected papers (2006) (accepted)
10. Texas Instruments Literature: SPRU423D: DSP/BIOS user's guide
11. Hoeflinger, J.P., de Supinski, B.R.: The openmp memory model. In: Mueller, M.S., Chapman, B.M., de Supinski, B.R., Malony, A.D., Voss, M. (eds.) IWOMP 2005/2006. LNCS, vol. 4315, pp. 167–177. Springer, Heidelberg (2008)
12. Lamport, L.: The parallel execution of do loops. Commun. ACM 17(2), 83–93 (1974)
13. OpenMP, A.: Openmp application program interface, v. 4.0 - rc 2 (2013)
14. Beyer, J.C., Stotzer, E.J., Hart, A., de Supinski, B.R.: OpenMP for accelerators. In: Chapman, B.M., Gropp, W.D., Kumaran, K., Müller, M.S. (eds.) IWOMP 2011. LNCS, vol. 6665, pp. 108–121. Springer, Heidelberg (2011)
15. Texas Instruments Literature: SPRT610: TMS320TCI6612/14 High Performance comes to small cell base stations
16. Ali, M., Stotzer, E., Igual, F.D., van de Geijn, R.A.: Level-3 blas on the ti c6678 multi-core dsp. In: 2012 IEEE 24th International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD), pp. 179–186. IEEE (2012)
17. Ahmad, A., Ali, M., South, F., Monroy, G.L., Adie, S.G., Shemonski, N., Carney, P.S., Boppart, S.A.: Interferometric synthetic aperture microscopy implementation on a floating point multi-core digital signal processer. In: SPIE BiOS, International Society for Optics and Photonics, p. 857134 (2013)
18. Note, F.W., Van Zee, F.G., Smith, T., Igual, F.D., Smelyanskiy, M., Zhang, X., Kistler, M., Austel, V., Gunnels, J., Low, T.M., et al.: Implementing level-3 blas with blis: Early experience (2013)
19. Reyes, R., Lopez, I., Fumero, J.J., de Sande, F.: Directive-based programming for gpus: A comparative study. In: 2012 IEEE 14th International Conference on High Performance Computing and Communication & 2012 IEEE 9th International Conference on Embedded Software and Systems (HPCC-ICESS), pp. 410–417. IEEE (2012)

20. Han, T.D., Abdelrahman, T.S.: hi cuda: a high-level directive-based language for gpu programming. In: Proceedings of 2nd Workshop on General Purpose Processing on Graphics Processing Units, pp. 52–61. ACM (2009)
21. Wolfe, M.: Implementing the pgi accelerator model. In: Proceedings of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units, pp. 43–50. ACM (2010)
22. Eichenberger, A.E., O'Brien, J.K., O'Brien, K.M., Wu, P., Chen, T., Oden, P.H., Prener, D.A., Shepherd, J.C., So, B., Sura, Z., et al.: Using advanced compiler technology to exploit the performance of the cell broadband engine architecture. IBM Systems Journal 45(1), 59–84 (2006)
23. Ayguade, E., Badia, R.M., Cabrera, D., Duran, A., Gonzalez, M., Igual, F., Jimenez, D., Labarta, J., Martorell, X., Mayo, R., Perez, J.M., Quintana-Ortí, E.S.: A proposal to extend the openMP tasking model for heterogeneous architectures. In: Müller, M.S., de Supinski, B.R., Chapman, B.M. (eds.) IWOMP 2009. LNCS, vol. 5568, pp. 154–167. Springer, Heidelberg (2009)
24. Cabrera, D., Martorell, X., Gaydadjiev, G., Ayguade, E., Jiménez-González, D.: Openmp extensions for fpga accelerators. In: International Symposium on Systems, Architectures, Modeling, and Simulation, SAMOS 2009, pp. 17–24. IEEE (2009)
25. Ayguadé, E., Badia, R.M., Bellens, P., Cabrera, D., Duran, A., Ferrer, R., Gonzàlez, M., Igual, F., Jiménez-González, D., Labarta, J., et al.: Extending openmp to survive the heterogeneous multi-core era. International Journal of Parallel Programming 38(5-6), 440–459 (2010)
26. Texas Instruments Literature: SPRUGO6A: SYS/BIOS inter-processor communication (IPC) and I/O user's guide
27. Chapman, B., Huang, L., Biscondi, E., Stotzer, E., Shrivastava, A., Gatherer, A.: Implementing openmp on a high performance embedded multicore mpsoc. In: IEEE International Symposium on Parallel & Distributed Processing, IPDPS 2009, pp. 1–8. IEEE (2009)
28. Jeun, W.C., Ha, S.: Effective openmp implementation and translation for multiprocessor system-on-chip without using os. In: Proceedings of the 2007 Asia and South Pacific Design Automation Conference, pp. 44–49. IEEE Computer Society (2007)

A Prototype Implementation of OpenMP Task Dependency Support

Priyanka Ghosh, Yonghong Yan, Deepak Eachempati, and Barbara Chapman

Department of Computer Science, University of Houston
{pghosh06,yanyh,dreachem,chapman}@cs.uh.edu

Abstract. OpenMP 3.0 introduced the concept of asynchronous *tasks*, independent units of work that may be dynamically created and scheduled. Task synchronization is accomplished via the insertion of *taskwait* and *barrier* constructs. However, the inappropriate use of these constructs may incur significant overhead owing to global synchronizations for specific algorithms on large platforms. The performance of such algorithms may benefit substantially if a mechanism of specifying finer gained point-to-point synchronization between tasks is available. In this paper we present extensions to the current OpenMP *task* directive to enable the specification of dependencies among tasks. A task waits only until the explicit dependencies as specified by the programmer are satisfied, thereby enabling support for a dataflow model within OpenMP. We evaluate the extensions implemented in the OpenUH OpenMP compiler using LU decomposition and Smith-Waterman algorithms. By applying the extensions to the two algorithms, we demonstrate significant performance improvement over the standard tasking versions. When comparing our results with those obtained using related dataflow models - OmpSs and QUARK, we observed that the versions using our task extensions delivered an average speedup of 2-6x.

1 Introduction

To improve its expressivity with respect to unstructured parallelism, OpenMP 3.0 introduced the concept of asynchronous *tasks*, independent units of work that may be dynamically created and scheduled. Task synchronization is accomplished via the insertion of *taskwait* and *barrier* constructs. However, the inappropriate use of those constructs and the runtime overhead incurred during the synchronization between concurrent tasks could typically present an obstacle in obtaining high performance and scalability on parallel systems [11]. For example, the inability to express dependence relationships among specific tasks in some algorithms forces the programmer to utilize either a *taskwait* or *barrier*, which dictates global synchronizations, including all tasks created before these constructs. This prevents such parallel algorithms from fully exploiting their maximum concurrency that is theoretically achievable.

Thus, the need for point-to-point synchronization among specific tasks is particularly important when dealing with applications that can be expressed through a task graph, exhibit pipeline parallelism, or require wavefront synchronization. Computations in these types of problems exhibit a data dependency

A.P. Rendell et al. (Eds.): IWOMP 2013, LNCS 8122, pp. 128–140, 2013.

pattern within certain periods of its execution, and stand to benefit from the exploitation of fine-grained parallelism.

In this paper, we present extensions to the current OpenMP task directive to enable the specification of dependencies between tasks sharing the same parent. A task waits only until the explicit dependencies as specified by the programmer are satisfied, thereby enabling support for a dataflow model within OpenMP. We apply the extensions on two algorithms, LU decomposition [6] and Smith-Waterman [7] by replacing the *taskwait* barrier-type synchronizations with our extensions, and demonstrate the performance improvement obtained over the standard tasking versions. When comparing our results with those obtained using other dataflow models - OmpSs [8] and QUARK [16], we observed that the versions using our task extensions delivered an average speedup of 2-6x. Our approach is similar to the *depend* clause of *task* directive that is currently under discussion in the OpenMP 4.0 release candidate [2]. We hope our experience could provide a proof of concept of this feature, and also the solution to implementing this feature in our compiler could be helpful for other implementers.

In the rest of the paper, Section 2 describes the motivation for supporting task dependency. Section 3 discusses two relevant efforts that support the specification of task dependencies and introduces our approach. Section 4 provides a brief description of the implementation in OpenUH runtime library. Section 5 presents the performance results. In Section 6, we briefly discuss the related work, and Section 7 contains our conclusion and future work.

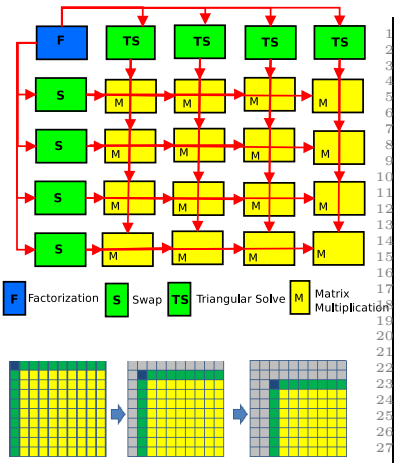
2 Motivation

In this section we introduce two algorithms and evaluate how their performance may be improved when using point-to-point inter-task synchronizations.

2.1 LU Decomposition

LU decomposition is a widely used algorithm in solving a system of linear equations, matrix inversion or computing the determinant of a matrix. A typical parallel implementation uses a blocking algorithm, which is illustrated in Figure 1. Each block, a submatrix, is being processed by an explicit task, and the data dependencies of those tasks are denoted by red arrow lines. In every iteration, the outermost diagonal block is factorized. After the *factorization*(F) of that block, the execution of the *swap*(S) and *triangular solve*(TS) operations on all blocks in the same column and row can be started. These blocks are updated in parallel since there are no dependencies among them. Blocks of the inner blocks can be updated with a *matrix multiplication*(M) operation as soon as their dependencies are solved, and each of these inner M blocks is only dependent on their corresponding S and TS blocks. In the absence of means of specifying explicit dependencies between tasks, synchronization between the M and S/TS blocks requires the use of *taskwait* constructs to maintain the dependence relationships, as shown in Listing 1.1. Such an approach constrains the execution order of those

tasks, that is, each of the M blocks has to wait for the completion of all the S and TS blocks. Thus the algorithmic parallelism will not be fully exploited due to the lack of this particular language feature.




```
#pragma omp parallel
{
 #pragma omp master
 {
  for ( i=0; i < matrix_size; i++ ) {
  /** Processing Diagonal block **/
  ProcessDiagonalBlock(.....);

  for (j=1;j<M;j++) {
  /** Processing block on column **/
  #pragma omp task
     ProcessBlockOnColumn(.......);
  /** Processing block on row **/
  #pragma omp task
     ProcessBlockOnRow(........);
   }
  #pragma omp taskwait /* global synchronization point */
/* Processing remaining inner block */
  for (i=1;i<M;i++)
     for (j=1;j<M;j++){
  #pragma omp task
  ProcessInnerBlock(............);
    }
  #pragma omp taskwait
  } // end of outer for loop
 } // end of master region
} // end of parallel region
```

Listing 1.1. Tasking implementation for LU decomposition

Fig. 1. Left: Diagram showing the existing data dependencies in a single iteration of the LU kernel. Right:Pseudocode of LU decomposition algorithm implemented using OpenMP tasks.

2.2 Smith-Waterman Algorithm

The Smith-Waterman algorithm is used primarily in the field of DNA and protein sequencing to determine similarities between biomolecule sequences according to a scoring system defined by a substitution matrix and gap penalty function. It employs a dynamic computational matrix technique that makes the algorithm more computationally intense, especially in the presence of data dependencies which restrict it from scaling well for parallel applications. Figure 2 represents the algorithm using 2D wavefront operations. Updating an element in the matrix requires updating the previous neighboring elements, resulting in a computation that resembles a diagonal sweep across the elements in the logical plane. Each element in the scoring matrix has three explicit dependencies on: a) its immediate north neighbor, b) its immediate west neighbor, and c) its immediate north-west neighbor.

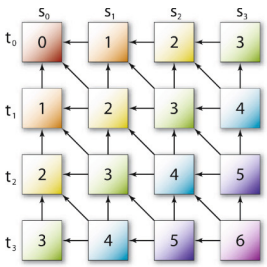


Fig. 2. 2D wavefront of smith-waterman algorithm [7]

The computations start at the extreme upper-left corner of the matrix and a gradual sweep moves along the diagonal down to the opposite corner. The diagonal represents the number of elements that could be executed in parallel. Hence the individual elements on each diagonal are mutually independent of each other and depend only on the respective neighboring elements from the previous two diagonals. In a chunked (blocked) tasking implementation of the algorithm, the presence of a *taskwait* introduces complexity in accessing elements belonging to multiple diagonals at a given time. This means elements of any given diagonal have to wait until all the elements of its previous diagonal have completed execution, even after their respective dependencies in the aforementioned diagonal have been satisfied. This prevents the code from exploiting the maximum amount of concurrency that is achievable, even in the presence of available resources.

3 Approaches to Handling *Task* Dependencies

OpenMP 4.0 release candidate 2 [2] introduced the *depends* clause for *task* directive for the purpose of specifying dependencies of asynchronous tasks, inspired by several previous efforts. While these efforts all aim to achieve the same goal, there are differences in both the language syntax and implementation details.

3.1 The OmpSs Programming Model

OmpSs (OpenMP SuperScalar) language and StarSs programming model developed by Barcelona Supercomputing Center [8] are the early efforts of defining additions to the OpenMP standard to enable a dataflow representation in C and C++ programs. It makes use of pragmas that define tasks with a set of *input*, *output*, and *inout* parameters. With the addresses for expressions provided in these clauses as arguments, the dependence information is evaluated at task creation time. Additionally, OmpSs currently supports array sections which may completely overlap. OmpSs embodies the dataflow principles of execution with the implementation of a task dependency graph at runtime, where tasks are scheduled for execution as soon as all their predecessors in the graph have finished (which does not mean they are executed immediately) or at creation if they have no predecessors.

3.2 QUARK Runtime API

QUARK (QUeuing And Runtime for Kernels) [16] developed primarily at the University of Tennessee, provides a runtime environment which enables the dynamic execution of tasks with data dependencies in a multi-core, multi-socket, shared-memory environment. QUARK infers data dependencies and precedence constraints among tasks from the way that the data is being used, and then executes the tasks in an asynchronous, dynamic fashion in order to achieve a high utilization of the available resources. Even though the main focus for the development of the API was catered to support basic linear algebraic algorithms

(BLAS) for the PLASMA [3] library, it is capable of supporting other data-driven applications that can be decomposed into tasks with data dependencies. Parallelization using QUARK relies on two steps: transforming function calls to task definitions and replacing function calls with task queuing constructs. QUARK was designed to embody the principle of a dataflow model in an easy-to-use interface, and scheduling is based on data dependencies between tasks in a task graph. It also enables built-in optimizations to obtain comparable performance with respect to the static scheduler employed by the PLASMA library.

3.3 Extensions Implemented in OpenUH Compiler

The extensions we proposed in our OpenUH OpenMP compiler are syntactically similar to other related efforts. Three clauses for the OpenMP *task* construct, described in greater detail in [9], were introduced to allow the programmer to express dependencies among sibling tasks of the same parent in an OpenMP program. We apply the notion of "tags", or task synchronizers among sibling tasks. These "tags" may be ideally expressed as a list of integral expressions (as simple as a unique constant). If the programmer's intent is to synchronize a variable access, then this identifier may uniquely identify that variable (e.g. an address). Listing 1.2 explains very briefly the functionality of the extensions:

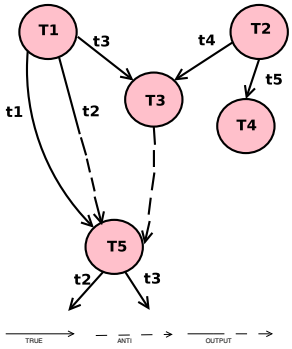
- *#pragma omp task* ***out*** *[t_1,t_2,....,t_n]* A generated task is denoted to have "*out*" dependence if it compute's variables required by succeeding tasks.
- *#pragma omp task* ***in*** *[t_1,t_2,....,t_n]* A generated task is denoted to have "*in*" dependence if it requires variables that have been computed previously.
- *#pragma omp task* ***inout*** *[t_1,t_2,....,t_n]* Entails a task having an *in* and *out* dependence on the same *tag*.

t_1,t_2,....,t_n are arguments of integer expressions (which could be constants, expressions, variables, and addresses) denoting the *tags* as specified by the programmer. Using these extensions, programmers are able to specify dependencies of either true(RaW), Output(WaW) or Anti(WaR) as seen in Figure 3.

In Figure 3, tasks T_1 and T_2 can be scheduled in parallel since they have no prior unresolved dependencies. Tasks T_3 and T_4 have true dependencies on tasks T_1 and T_2 (referenced by *tags* t3, t4 and t5 respectively) and can only be scheduled after T_1 and T_2 have completed. Similarly, task T_5 has a true and output dependence based on *tags* t1 and t2 (with respect to task T_1) respectively, as well as an anti dependence on *tag* t3. Hence T_5 shall be scheduled last. We also observe in this instance that tasks 3 and 4 can be scheduled in parallel. Details of the implementation of the extensions within the OpenUH OpenMP runtime library are described in section 4.

3.4 Comparison

There are several differences of the above three approaches. Firstly, in OpenUH, the actual specification of the dependencies among tasks (by the programmer in the argument list associated with the extension) comprises of integer expressions as compared to blocks of contiguous memory locations proposed by OmpSs.




```
1  #pragma omp parallel
2   {
3   #pragma omp master
4    {
5     #pragma omp task out(t1) out(t2) out(t3) /* task 1 created
            */
6        x = f1();
7     #pragma omp task out(t4) out(t5)  /* task 2 created */
8        y = f2();
9     #pragma omp task in(t3) in(t4)  /* task 3 created */
10       v = f3();
11    #pragma omp task in(t5)  /* task 4 created */
12       z = f4(v,y);
13    #pragma omp task in(t1) out(t2) out(t3) /* task 5 created
            */
14       w = f5(x,z)
15   }
16  }
```

Listing 1.2. sample code quoting the extensions to the *task* construct.

Fig. 3. Left: Task graph of Listing 1.2. Right: example quoting the extensions described in 3.3

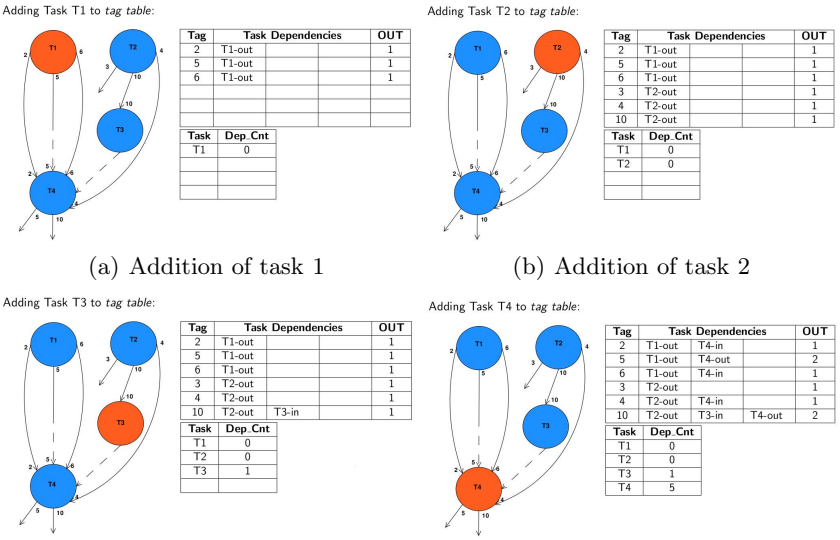
Secondly, OmpSs constructs a task dependency graph (DAG) and maintains a table data structure to store and manage the data dependencies among tasks, whereas OpenUH employs only a *tag table* (unordered hash map) to account for the same. Dependencies among tasks in OmpSs are expressed based on arbitrary accesses made to regions in memory which may pose challenges in terms of implementation. The dynamic detection of overlapping dependent regions on the fly and the resolution of such dependencies, at task level granularity may introduce significant overheads at runtime. OmpSs offers expressivity in terms of application of their proposed extensions, allowing programmers to specify dependencies among tasks at program level with ease. However the specified array sections must overlap in order to maintain the dependence. OpenUH alternatively supports a lower level abstraction declaring dependencies more directly with the use of virtual variables in the form of task *tags*. This simplifies the detection of dependent tasks by matching their respective *tags*, thereby promoting an easier implementation at runtime and incurring lesser overhead. In QUARK however, the master thread is solely devoted to inserting the tasks, determining their dependencies and queuing the tasks within the DAG and therefore, does not participate in the actual execution of the tasks. All these dataflow model implementations support a ready pool of tasks, containing tasks with resolved or no dependencies, allowing worker threads to steal and execute the work.

4 Implementation of Extensions in OpenUH

The OpenUH compiler [4] is a branch of the Open64 compiler suite for C, C++, Fortran 2003. OpenUH includes support for OpenMP 3.x tasks. This consists of front-end support ported from the GNU C/C++ compiler, a back-end translation implemented jointly with Tsinghua University, and an efficient runtime task scheduling infrastructure which holds support for improved nested parallelism. Its implementation supports a configurable task pool framework that allows the

user to adapt the runtime environment based on the needs of the applications. For instance, for greater control over task scheduling, the programmer can choose at runtime an appropriate task queue organization as well as control the order in which tasks are added/removed from a task queue.

The efficiency of an OpenMP runtime implementation will heavily impact the performance of application's using tasks. An ideal task scheduler will schedule tasks for execution in a way that maximizes concurrency and, therefore, performance. We introduce a parent table (unordered hashmap) called the "tag" table in the runtime which holds the dependencies associated among the tasks. If the programmer does not specify any dependencies, the OpenUH runtime places a newly created task immediately into a task pool, allowing it to be executed by the next available thread.



(a) Addition of task 1 (b) Addition of task 2

(c) Addition of task 3 (d) Addition of task 4

Fig. 4. Addition of tasks to tag table at the time of task creation

However if dependencies are enforced, a task will be placed in the pool only after its corresponding *Dep_Count* is equivalent to zero. The *Dep_Count* counter is a parameter that indicates the number dependencies associated with that task. The task is put on hold (in a WAIT state) until all its previous requisite dependencies are resolved. Figure 4 illustrates the process of addition of *tags* to the tag table at the time of task creation.

The dependencies specified by the programmer are individually associated with a "*tag*". Each *tag* holds a unique entry in the parent table as a hash key. Its corresponding hash value is a linked list representation of all dependent tasks sharing the same *tag*. At any time when a task's *Dep_Counter* reaches zero, it has satisfied all its related dependencies and can now be placed in the task pool

for it to be scheduled. However, if the *Dep_Counter* is not zero, it implies that it has to wait for the subsequent dependent tasks to finish execution before it could be placed on the task pool. Once a task is placed in the task pool it is available for execution. Similarly tasks after execution need to be removed from the *tag table* at the time of task deletion. Before the tasks are removed from the table, it is essential that the *Dep_Counter* of its subsequent dependent tasks are updated accordingly, so that they may be scheduled for execution after their dependencies has been resolved.

In the implementation of our extensions we avoid the frequent use of global locks thereby eliminating waiting time for tasks created at runtime to access the parent table. With the use of *compare* and *swap* operations we ensure atomic access to shared resources at the time of task creation and task exit. We omit further details of the implementation owing to space limitation and refer the reader to [9].

5 Experimental Results

In this section we evaluate, in terms of performance, scalability and overhead incurred, the extensions implemented in the OpenUH compiler for two algorithms, LU decomposition and Smith-Waterman. The comparison was performed with respect to a) the standard tasking versions on some commercial and open source compilers and b) related dataflow model implementations, QUARK and OmpSs. Performance results for the standard tasking version have been acquired on the following compilers: GNU version 4.7.1, Intel version 12.0, OpenUH version 3.0.27, PGI version 11.9, Sun/Oracle version 12.3, and OmpSs programming model - Mercurium compiler with Nanos++ runtime system (1.3.5.8), with the default scheduling policy. The version of QUARK we have used is 0.9.0.

5.1 Performance Analysis for LU Decomposition Algorithm

Our testbed for the following set of experiments comprises of an AMD Opteron Processor 6174 with 48 cores - 63GB of main memory, L1 cache - 64KB, L2 cache - 512KB and last level cache, L3 of size 10MB.

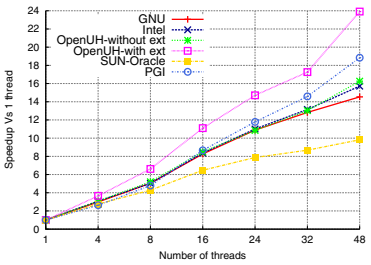
Listing 1.3 below represents the pseudo-code for implementing the LU decomposition algorithm using the OpenUH *task* extensions thereby eliminating the need to apply the first *taskwait*, depicted in Line 17 of Listing 1.1. The flexibility of task execution in the hands of the programmer warrants the reduction of overheads normally encountered (in the absence of the extensions), while having to wait until all the tasks executing *ProcessBlockOnColumn* and *ProcessBlockOnRow* have concluded, prior to the execution of *ProcessInnerBlock*.

Figure 5(a) measures the speedup obtained by varying number of cores, with 16 blocks per dimension on a matrix of size 4096 X 4096. We observe that the version implemented with the OpenUH task extensions scales well up to 24x on a single core. Figure 5(b) measures the performance obtained by varying the number of blocks per dimension (8,16,32 and 48), on 48 threads, wherein

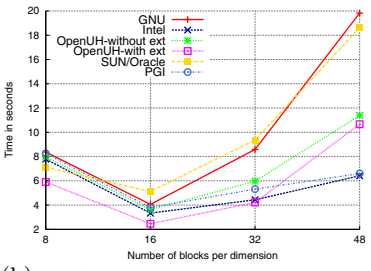
the matrices are partitioned into smaller blocks so that they could fit in memory registers and cache. This allows spatial locality to improve by increasing reuse of the data in cache memory and can be acknowledged as a significant memory optimization uplifting code performance. We observe improvement in performance as we gradually increase the number of blocks per dimension. However, implementing 32 blocks per dimension degrades performance mainly due to the creation of excessive tasks, adding additional overhead attributed to task creation/deletion as well as task synchronization. Additionally by gradually decreasing the size of blocks we reduce the chances of achieving higher data reuse causing the performance to suffer.

```
#pragma omp parallel
{
 #pragma omp master
 {
   for ( i=0; i<matrix_size; i++ ) {
/* Processing Diagonal block */
   ProcessDiagonalBlock(.......);
   for (j=1;j<M;j++){
/* Processing block on column */
   #pragma omp task out(2*j)
     ProcessBlockOnColumn(........);
/* Processing block on row */
   #pragma omp task out(2*j+1)
     ProcessBlockOnRow
          (...................);
     }
/* Processing remaining inner block
   */
   for (i=1;i<M;i++)
     for (j=1;j<M;j++){
   #pragma omp task in(2*i) in(2*j+1)
   ProcessInnerBlock(..............)
        ;
     }
   #pragma omp taskwait
     }
  }
}
```

Listing 1.3. LU decomposition Algorithm using OpenMP tasks with extensions implemented in OpenUH [15]



(a) Speedup for matrix size 4096 X 4096 with -O3 optimization

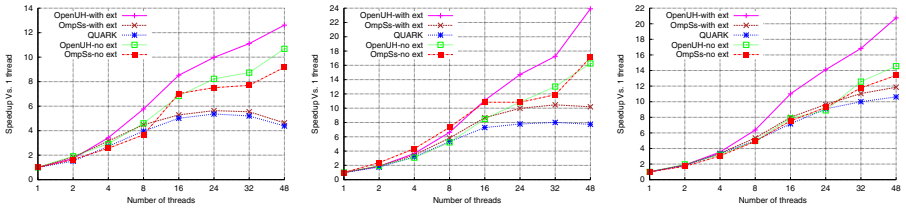


(b) Performance varying blocks per dimension-matrix size 4096 with -O3 optimization with 48 threads

Fig. 5. Results obtained for LU decomposition on OpenUH with and without the use of *task* extensions and with respect to the results obtained from standard tasking versions on other compilers

Figure 6 allows us to make an assessment on how well OmpSs and QUARK scale in comparison to OpenUH, with varying matrix sizes. With the increase in size of input data, both OmpSs and QUARK show consistent improvement in terms of scalability. In the case of OpenUH, where, with the application of the extensions, we account for an average performance improvement of 20% (compared to the version excluding the extensions), there is a subsequent degradation

in performance estimated close to an average 30%, observed for OmpSs. This may be attributed to that fact that the master thread invests significant amount of time in maintaining the task graph at runtime along with the overhead encountered by worker threads waiting for tasks to populate the ready pool [8]. We observe that QUARK scales poorly as well for smaller data sizes and the drop in scalability for large number of cores can be attributed to overhead incurred due to use of broader thread mutex locks resulting in contention.

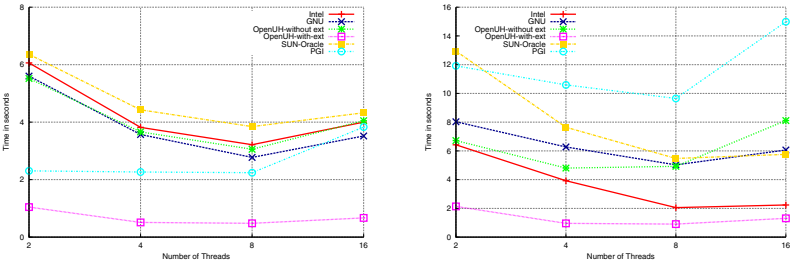


(a) matrix size 2048 X 2048 (b) matrix size 4096 X 4096 (c) matrix size 6144 X 6144

Fig. 6. Scalability (measure of speedup Vs 1 thread) obtained across the three dataflow model implementations by varying the number of threads for various matrix sizes, with 16 blocks per dimension

5.2 Performance Analysis for Smith-Waterman Algorithm

Our testbed for the following set of experiments comprises of a Dual Intel Nehalem - E5520 processor with 16 cores. 32GB of total memory capacity, L1 cache 32K, L2 cache 256K and an L3 cache of 8MB.

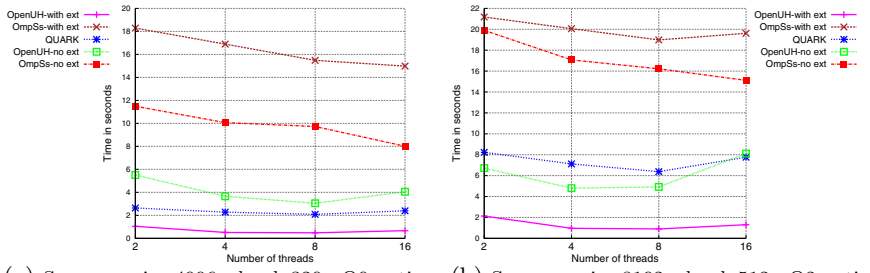


(a) Performance for sequence size 4096 with -O0 optimization (b) Performance for sequence size 8192 with -O2 optimization

Fig. 7. Performance results (in seconds) for Smith-Waterman kernel with varying number of threads. We compare the results obtained on OpenUH with and without the *task* extensions and with respect to the results obtained from standard tasking versions on other compilers.

Figures 7(a) and 7(b) present the performance obtained for sequences of size 4096 and 8192, with -O0 and -O2 levels of optimization respectively, for a standard tasking version of the Smith-Waterman kernel where tasks have been created as chunks of 320 and 512 per diagonal. The version with the OpenUH task

extensions outperforms the version without the use of the extensions by a factor of 6x. This improvement is attributed to the use of less constrained synchronization, allowing tasks to execute as soon as their respective dependencies have been satisfied. This performance is consistent even when tested with -O0 optimization validating that the improvement observed is due to the elimination of the *taskwait*, and without the interference of any other optimizations. Another observation suggests none of the compilers produce scalable results beyond 8 threads. This is attributed to the memory bound nature of the application, wherein tasks being very fine grained add additional overhead of task scheduling.



(a) Sequence size 4096, chunk 320, -O0 optimization

(b) Sequence size 8192, chunk 512, -O2 optimization

Fig. 8. Performance comparison amongst the dataflow models (in seconds)

Figures 8(a) and 8(b) perform a comparative analysis based on the performance obtained from the three dataflow model implementations, across varying number of threads for sequences of size 4096 and 8192 tested with task chunk sizes 320 and 512 with -O0 and -O2 optimizations respectively. For both test cases, we observe that OpenUH has the overall better performance. We argue that avoiding the frequent use of global lock operations, eliminates waiting time for tasks created at runtime attempting to access the *tag* table, thereby achieving higher scalability. The overhead incurred by OmpSs may be attributed to difficulty encountered by the runtime, in computing task dependencies and attending to finished tasks fast enough, (owing to the fine grained nature of the tasks) in order to keep all cores busy [5]. The drop in performance for QUARK is attributed to the fact that the master thread, which does not participate in computation, spends cycles tracking dependencies among fine grained tasks in a memory bound kernel. QUARK being sensitive to task size generates overhead at the runtime when tasks are too fine grained and the switching time between ready tasks is very short [10].

6 Related Work

Other than the OmpSs and QUARK presented in Section 2, data-driven tasks (DDT) have been widely advocated by researchers to replace the use of potentially expensive global barriers [13]. Some of these efforts rely on compiler transformation to decompose data parallel computations into tasks with dependencies, and to achieve higher degree of overlap and concurrency between these

tasks [14]. Other extension to task parallelism, described as data-driven futures (DDFs) in [12], allows users to create write-once tags as input and output events that could trigger other tasks. The write-once restriction, same as in the Intel Concurrent Collection [1] programming model for data-flow parallelism, simplify the programming logic and algorithms reasoning, as well as the runtime implementations, but may introduce overheads when handling a large number of tags requiring multiple read/write.

The standardization of the task dependency model in OpenMP represents a major step toward its adoption from research community to applications in real world. We believe our work and other related efforts have demonstrated its effectiveness and usability toward this direction.

7 Conclusions and Future Work

In this paper we highlighted extensions implemented in the OpenUH OpenMP compiler and runtime library to support a dependence-based synchronization which resembles a dataflow model of execution. These extensions enable simple and intuitive expression of data driven algorithms that rely especially on patterns such as pipeline processing and wavefront propagation. Experiments conducted on the LU decomposition and Smith-Waterman algorithms, exhibited an improvement in performance by a average factor of 2x and 6x respectively, in comparison to the the standard tasking versions. On comparing the performance against related dataflow models, OmpSs and QUARK, we observed OpenUH achieved better performance. This is attributed to the reduction in task synchronization and scheduling overheads when dealing with larger input data sizes. Additionally with limited global lock operations it incurs minimal overhead at runtime thereby providing scope for achieving higher scalability.

As future work we would like to extend support for the specification of the extensions in Fortran. We also wish to extend our implementation of such data driven algorithms in the direction of a distributed memory environment.

Acknowledgments. This work was supported in part by the National Science Foundations Computer Systems Research program under Award No. CCF-0833201 and Department of Energy under Award Agreement No. DE-FC02-12ER26099. The evaluation platform used for for this work was supported by the National Science Foundation's Computer Systems Research program under Award No. CNS-0833201 and CRI-0958464. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation or the Department of Energy. We would also like to thank Sayan Ghosh for reviewing our paper. Our appreciations also went to Elkin Garcia and Professor Guang R. Gao who provided us the sequential version of the LU program.

References

1. Intel Concurrent Collections,
`http://software.intel.com/en-us/articles/intel-concurrent-collections-for-cc/`

2. OpenMP 4.0 release candidate 2, http://www.openmp.org/mp-documents/OpenMP_4.0_RC2.pdf/
3. Agullo, E., Demmel, J., Dongarra, J., Hadri, B., Kurzak, J., Langou, J., Ltaief, H., Luszczek, P., Tomov, S.: Numerical linear algebra on emerging architectures: The plasma and magma projects. In: Journal of Physics: Conference Series. vol. 180, p. 012037. IOP Publishing (2009)
4. Chapman, B., Eachempati, D., Hernandez, O.: Experiences developing the openuh compiler and runtime infrastructure. International Journal of Parallel Programming, 1–30 (2012)
5. Dallou, T., Juurlink, B.: Hardware-based task dependency resolution for the starss programming model. In: 2012 41st International Conference on Parallel Processing Workshops (ICPPW), pp. 367–374. IEEE (2012)
6. Desprez, F., Domas, S., Tourancheau, B.: Optimization of the scalapack lu factorization routine using communication/computation overlap. In: Euro-Par 1996 Parallel Processing, pp. 1–10. Springer (1996)
7. Dios, A.J., Asenjo, R., Navarro, A., Corbera, F., Zapata, E.L.: Evaluation of the task programming model in the parallelization of wavefront problems. In: 2010 12th IEEE International Conference on High Performance Computing and Communications (HPCC), pp. 257–264. IEEE (2010)
8. Duran, A., Perez, J.M., Ayguadé, E., Badia, R.M., Labarta, J.: Extending the openMP tasking model to allow dependent tasks. In: Eigenmann, R., de Supinski, B.R. (eds.) IWOMP 2008. LNCS, vol. 5004, pp. 111–122. Springer, Heidelberg (2008)
9. Ghosh, P., Yan, Y., Chapman, B.: Support for dependency driven executions among openmp tasks. In: Workshop on Data-Flow Execution Models for Extreme Scale Computing (DFM 2012) in conjunction with PACT (September 2012)
10. Haidar, A., Ltaief, H., Luszczek, P., Dongarra, J.: A comprehensive study of task coalescing for selecting parallelism granularity in a two-stage bidiagonal reduction. In: 2012 IEEE 26th International Parallel & Distributed Processing Symposium (IPDPS), pp. 25–35. IEEE (2012)
11. Olivier, S.L., de Supinski, B.R., Schulz, M., Prins, J.F.: Characterizing and mitigating work time inflation in task parallel programs. In: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis, SC 2012, pp. 65:1–65:12. IEEE Computer Society Press, Los Alamitos (2012)
12. Taşırlar, S., Sarkar, V.: Data-Driven Tasks and their Implementation. In: Proceedings of the International Conference on Parallel Processing (September 2011)
13. Vajracharya, S., Karmesin, S., Beckman, P., Crotinger, J., Malony, A., Shende, S., Oldehoeft, R., Smith, S.: Smarts: Exploiting temporal locality and parallelism through vertical execution. In: Proceedings of the 13th International Conference on Supercomputing, pp. 302–310. ACM (1999)
14. Weng, T.H.: Translation of OpenMP to Dataflow Execution Model for Data locality and Efficient Parallel Execution. PhD thesis, Department of Computer Science, University of Houston (2003)
15. Yan, Y., Chatterjee, S., Orozco, D.A., Garcia, E., Budimlić, Z., Shirako, J., Pavel, R.S., Gao, G.R., Sarkar, V.: Hardware and software tradeoffs for task synchronization on manycore architectures. In: Jeannot, E., Namyst, R., Roman, J. (eds.) Euro-Par 2011, Part II. LNCS, vol. 6853, pp. 112–123. Springer, Heidelberg (2011)
16. YarKhan, A., Kurzak, J., Dongarra, J.: Quark users guide: Queueing and runtime for kernels. University of Tennessee Innovative Computing Laboratory Technical Report ICL-UT-11-02 (2011)

An Efficient OpenMP Loop Scheduler for Irregular Applications on Large-Scale NUMA Machines

Marie Durand[1], François Broquedis[2], Thierry Gautier[1], and Bruno Raffin[1]

[1] INRIA
[2] Grenoble Institute of Technology
MOAIS Team, Computer Science Laboratory of Grenoble, France
marie.durand@inria.fr, françois.broquedis@imag.fr,
thierry.gautier@inrialpes.fr, bruno.raffin@inria.fr

Abstract. Nowadays shared memory HPC platforms expose a large number of cores organized in a hierarchical way. Parallel application programmers struggle to express more and more fine-grain parallelism and to ensure locality on such NUMA platforms. Independent loops stand as a natural source of parallelism. Parallel environments like OpenMP provide ways of parallelizing them efficiently, but the achieved performance is closely related to the choice of parameters like the granularity of work or the loop scheduler. Considering that both can depend on the target computer, the input data and the loop workload, the application programmer most of the time fails at designing both portable and efficient implementations. We propose in this paper a new OpenMP loop scheduler, called *adaptive*, that dynamically adapts the granularity of work considering the underlying system state. Our scheduler is able to perform dynamic load balancing while taking memory affinity into account on NUMA architectures. Results show that *adaptive* outperforms state-of-the-art OpenMP loop schedulers on memory-bound irregular applications, while obtaining performance comparable to *static* on parallel loops with a regular workload.

Keywords: OpenMP, NUMA, loop scheduling, runtime systems.

1 Introduction

Large-scale multicore platforms are commonly used by the HPC community. They expose a constantly-growing number of cores organized in a hierarchical way, leading to large-scale NUMA machines. To exploit them at their full potential, the application programmer needs to express massive fine-grain parallelism while taking memory affinity into account. Applications exposing irregular workloads are really difficult to execute efficiently on such platforms, as they require to deal with both load balancing and memory locality.

OpenMP [17], the *de-facto* standard for shared memory parallel programming, provides the programmer with high-level constructs to ease the parallelization of serial applications. The parallel loop certainly remains the most widely used OpenMP construct, allowing to easily parallelize loops with independant iterations. The end-user can control the way loop iterations are assigned to OpenMP threads by invoking OpenMP loop schedulers. Choosing the best scheduler for a specific parallel loop can be a difficult

A.P. Rendell et al. (Eds.): IWOMP 2013, LNCS 8122, pp. 141–155, 2013.

task to perform in a portable way [1]. The application programmer is also responsible for defining the granularity of work within the loop, using the `chunk_size` clause.

While being designed to tackle loops with irregular workloads, the OpenMP `dynamic` scheduler suffers from two main issues on large-scale NUMA machines.

First, defining a chunk size from the application level that achieves both high and portable performance has never been so difficult. Indeed, parallel loops with big chunks may suffer from load imbalance, while the ones with smaller chunks are more sensitive to runtime-related overheads which are getting more and more noticeable as the number of cores per NUMA node increases.

Second, traditional techniques to increase the performance of memory bound applications, like guiding the data allocation on the different NUMA nodes of the platform, are useless using dynamic scheduling, as the assignment of loop iterations to OpenMP threads is non-deterministic.

We introduce in this paper a new loop scheduler, called *adaptive*, that outperforms state-of-the-art loop schedulers when executing memory bound irregular applications. In particular, our loop scheduler:

1. dynamically adapts the granularity of work within parallel loops according to the machine resources utilization,
2. relies on data placement information to guide load balancing on NUMA platforms.

The remainder of the paper is organized as follows. We first describe the related work on loop scheduling over NUMA architectures in section 2. Then we introduce the *adaptive* loop scheduler and the way we implemented it inside the LIBGOMP library in sections 3 and 4. Eventually, we report the performance we obtained on several benchmarks and applications in section 5 before concluding.

2 Related Work

Many research projects have been carried out to improve execution of OpenMP applications on NUMA machines.

The HPCTools group at the University of Houston has been working in this area for a long time, proposing compile-time techniques that can help improving memory affinity on hierarchical architectures like distributed shared memory platforms [13]. Huang et al. [10] proposed OpenMP extensions to deal with memory affinity on NUMA machines, like ways of explicitly aligning tasks and data inside logical partitions of the architecture called *locations*. While the proposed extension is interesting to deal with regular memory-bound applications, it does not tackle the problems induced by irregular workloads.

Olivier et al. [16,15] introduced node-level queues of OpenMP tasks, called *locality domains*, to ensure tasks and data locality on NUMA systems. The runtime system does not maintain affinity information between tasks and data during execution. Data placement is implicitly obtained considering that the tasks access memory pages that were allocated using the *first-touch* allocation policy. The authors thus ensure locality by always scheduling a task on the same locality domain, preventing application programmers to experiment with other memory bindings.

The INRIA Runtime group at the University of Bordeaux proposed the ForestGOMP runtime system [2] that comes with an API to express affinities between OpenMP parallel regions and dynamically allocated data. ForestGOMP implements load balancing of nested OpenMP parallel regions by moving branches of the corresponding tree of user-level threads on a hierarchical way. Memory affinity information is gathered at runtime and can be taken into account when performing load balancing. Our work extends this approach to deal with parallel loops while ensuring load balancing in a different way.

Mahéo et al. [12] used similar techniques to speed up hybrid MPI/OpenMP synchronizations on hierarchical architectures, including NUMA machines. Both our work and theirs build upon the same concepts and can be stated as complementary.

Subramanian and Eager [18] introduce an affinity loop scheduler for unbalanced workload. They study iterative applications involving series of parallel loops in which *"the execution times of any particular iteration do not vary widely from one execution of the loop to the next"*[18]. Their proposition is based upon a two-phase algorithm: first, the iterations are equally divided between the processors; then the scheduler dynamically adapts the workload by making idle processors steal a constant fraction $(1/P)$ of the remaining iterations from occupied ones. In [21], Yong *et al.* extends the work of Subramanian and Eager by providing new ways of adapting the workload considering an history of previous executions of a particular parallel loop.

Taking advantage of the temporal coherency of the simulation is a very interesting idea but cannot be used in all situations. For instance, it would not be effective on applications involving several parallel loops with varying workloads, like the PMA application we used to evaluate our *adaptive* loop scheduler. The first phase of our scheduling strategy is similar to the one introduced by Subramanian *et al.*, as *adaptive* equally pre-distributes the loop iterations over the processors, which is a compromise between balancing the workload of the loop and preserving the affinity across several executions of the same loop. However, the second phase of our algorithm is different from the one implemented by Subramanian's adaptive loop scheduler. Indeed, *adaptive* relies on a work-stealing algorithm [8] in which idle processors steal half of the remaining iterations from a randomly-selected victim. With such an approach, our *adaptive* loop scheduler does not require to maintain a global view of the workloads associated to each processor, unlike proposition [18].

3 Introducing the Adaptive Loop Scheduler

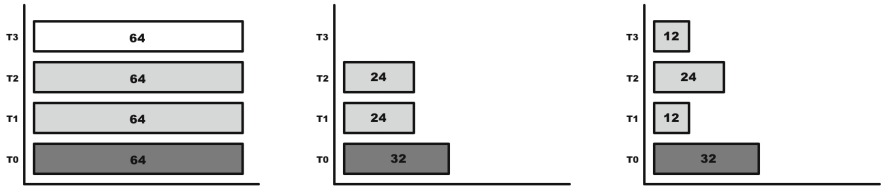
The OpenMP programmer can rely on two main loop schedulers to specify the way loops iterations should be assigned to threads. The first one, called *static*, statically assigns fixed portions of work in a round-robin fashion. This scheduler behaves well on loops with a regular workload and is often used in the context of NUMA architectures, along with the *first-touch* allocation policy, to maximize memory locality. The second one, called *dynamic*, makes OpenMP threads steal fixed portions of work from a centralized queue. This scheduler behaves better than *static* on loops with an irregular workload. However, *dynamic* is seldom used on NUMA architectures because of its non-deterministic behavior, preventing the programmer from controlling memory locality.

The loop scheduler we propose goes beyond those two, providing ways of balancing the load of irregular loops while respecting memory locality. This section first introduces the main scheduling algorithm provided by our *adaptive* loop scheduler before presenting the way we extended it to deal with memory locality on NUMA machines.

3.1 Designing an OpenMP Loop Scheduler with Adaptive Granularity

Dealing with irregular parallel applications requires efficient runtime-level functionalities to perform dynamic load balancing with the lowest overhead possible. OpenMP application programmers can rely on the *dynamic* loop scheduler to execute loops with irregular workload, as long as they manage to specify a chunk size that achieves good performance. Indeed, a too small chunk size will increase the time spent inside the runtime system, while a too coarse chunk size will limit the potential parallelism and the ability to balance the work load.

We adopted a different approach implementing our *adaptive* scheduler. We relieve the application programmer from statically deciding the granularity of work which can lead to non-portable solutions. Instead, we consider work-stealing as an oblivious technique to dynamically balance the load on the threads of the corresponding OpenMP parallel region.



(a) Initial distribution of the iteration space.

(b) T3 has completed its range and starts stealing.

(c) T3 stole half of the remaining iterations of T1.

Fig. 1. Illustration of adaptive loop scheduling on a 256-iterations loop with irregular workload executed on 4 threads. Darker color means higher workload.

We broke the vision of centralized work used by the state-of-the-art OpenMP loop scheduler to introduce a per-thread data structure describing the range of iterations assigned to each thread. Figure 1 illustrates the behavior of our *adaptive* scheduler on a synthetic example. Considering a loop of **imax** iterations executed by **nthr** threads, the scheduler first assigns **imax / nthr** iterations to each thread, like presented on figure 1a. This initial behavior allows our scheduler to achieve performance that is comparable to *static* on loops with regular workload. Even if every thread has the same number of iterations to execute here, the associated workload is different: the 64 iterations of thread **T3** have shorter execution times than the ones assigned to **T2** for example. This leads to load imbalance: at some point of the execution of the loop, thread **T3** will be starving like showed on figure 1b. **T3** will then trigger *adaptive*'s work-stealing algorithm which steals half of the remaining iterations of a loaded thread.

```
iter_adaptive_next:
1  if (try_local_work (&begin, &end) == true)
2    return (begin, end);
3
4  /* We've completed our previously-assigned range,
5  let's try to steal some new work! */
6  while (!loop_is_finished()) {
7    victim = pick_random_victim ();
8    if (steal_from_victim (victim, &begin, &end) == true)
9      return (begin, end);
10 }

steal_from_victim:                          try_local_work:
11 if ((victim->end-victim->begin)>0){  27 begin = own->begin +1;
12  lock (victim);                      28 own->begin = begin;
13  chunk_size                          29 if (begin < own->end) {
     =(victim->end - victim->begin)/2;  30   end = begin;
14  end = victim->end - chunk_size;     31   begin = end - 1;
15  victim->end = end;                  32   return true;
16  if (end <= victim->begin) {         33 }
      /* rollback and abort */          34 /*conflict detected: rollback and lock*/
17    victim->end = end + chunk_size;   35 own->begin = begin-1;
18    unlock (victim)                   36 lock (self)
19    return false;                     37 begin = own->begin;
20 }                                     38 if (begin < own->end)
21  begin = end;                        39   end = own->begin = begin + 1;
22  end = begin+chunk_size;             39 unlock (self);
23  unlock (victim);                    40 if (begin < own->begin) return true;
24  return true;                        41 return false;
25 }
26 return false;
```

Fig. 2. Pseudo-code of the adaptive loop scheduler. The implementation extends the THE protocol by stealing more than one item at each steal operation.

The algorithm[1] called to select the next chunk of iterations to execute is summarized on figure 2. Most importantly, our approach deals with dynamic per-thread chunk sizes, as shown on line 13 of this pseudo-code. The amount of work a thread will steal depends on the amount of work its victim still has to execute, unlike the *dynamic* scheduler in which the *chunk_size* is statically defined and cannot change during execution.

Unlike [18,21] in which a constant fraction of the work $(1/P)$ is removed from the most loaded processor, the random selection of the work-stealing algorithm does not suffer from maintaining the global state of the workloads associated to processors. Moreover, it is possible to derive theoretical performance guarantee of scheduling parallel loop with work stealing [20]. Frigo *et al.* [8] introduced two main metrics to model the performance of work-stealing-based algorithms: the *work* W, *i.e.* the time to sequentially execute the loop, and the *depth* D, also called the critical path, *i.e.* the execution time on an infinite number of processors.

Considering these metrics, the average completion time of the parallel loop is $O(W/P + D)$. If the work is $W = \sum_{i=0}^{n-1} w_i$, where w_i is the work associated to the i-th iteration, then $D = O(\log n + \max\{w_i\})$.

3.2 Extending the Adaptive Scheduler to Deal with Locality

Ensuring memory locality is crucial to achieve good performance on NUMA architectures. We extended the *adaptive* scheduler in order to benefit from shared memory

[1] If the memory consistency is not sequential, memory barriers have to be inserted between lines 15-16 and 28-29.

```
/* get the number of locality domains of a parallel region */
#pragma omp parallel
#pragma omp master
      printf("Number of locality domains = %i\n",
          omp_get_num_locality_domains());

/* sample of the modified STREAM benchmark with the adaptive
 scheduler and bloc memory distribution on locality domains */
#pragma omp parallel
#pragma omp master
    a = (double*)omp_locality_domain_allocate_bloc1d(
                      sizeof(double)*STREAM_ARRAY_SIZE+OFFSET);
```

Fig. 3. Code sample using our extended OpenMP runtime APIs

banks of NUMA multicore machines. This is done at two levels. First, we make the cores attached to the same memory bank work on contiguous iterations. This step is useful to abstract the OS identification of cores that may not be contiguous on a NUMA node. *adaptive* makes idle threads steal work from cores that belong to the same NUMA node. Thanks to this strategy, a successful steal will hopefully reduces the number of remote memory transfers. Moreover, this local steal strategy may not be enough to balance the workload among all the cores. That is why, if the number of unsuccessful steal operations reaches a threshold, the idle thread emits a steal request to a victim randomly selected over the whole machine.

The second feature is to provide ways of distributing the application data over the NUMA nodes. Our extension of the libGOMP library comes with APIs to distribute data as proposed in MAMI [4], implementing bloc and bloc-cyclic data distributions as runtime extensions. For now, we only support data distribution within parallel regions where the participating threads are bound to cores, for example using the GOMP_CPU_AFFINITY environment variable. As in [10,16,15], we refer to NUMA nodes as *locality domains*.

omp_get_num_locality_domains() : returns the number of locality domains holding threads from the current OpenMP parallel region.

omp_get_locality_domain_num() : returns the locality domain of the running thread.

omp_locality_domain_allocate_bloc1dcyclic (size, blocsize) : allocates an array of `size` bytes in a `blocsize`-cyclic distribution over the locality domain of the parallel region.

omp_locality_domain_allocate_bloc1d (size) : allocates an array of `size` bytes using a bloc distribution over the locality domain of the parallel region.

These routines performs allocation following the OS constraints: sizes are rounded up to a multiple of page size. If the OS does not support NUMA allocation routines, the implementation triggers calls to the libc's malloc function.

Figure 3 illustrates the use of the runtime APIs we propose. In order to allow reuse of memory mapping across parallel regions, we ensure that the $i-th$ thread of a parallel region will be bound to the same core across parallel regions if and only if the following parallel regions have the same size and are nested in the same parallel region or are at the top level.

3.3 Discussion

Defining the best granularity of work is certainly one of the most difficult challenge a parallel application programmer has to face to exploit HPC platforms at their full potential. For example, the best *chunk_size* for a specific OpenMP loop may depend on the target architecture and the input data of the parallel application. In other words, application programmers have to consider the underlying system state to specify the granularity that will achieve the best performance. This is an old problem for the OpenMP community: defining the *right* number of threads, the *right* level of nested parallel regions and the *right* chunk size for parallel loops are a few examples of the many crucial steps to achieve good performance and scalability.

The addition of tasks to the OpenMP standard provides new ways of expressing parallelism with a finer granularity. One can consider tasks as another way of dealing with irregular workload, as tasks can move from one OpenMP thread to another to perform load balancing. However, tasks will not solve the problem of granularity, as defining the *right* number of tasks can be challenging, as studied in our previous work [3].

Our proposal introduces a runtime-level approach to deal with granularity and has been implemented inside a loop scheduler. The same approach could be applied to task parallelism as well, considering ways of *splitting* OpenMP tasks when necessary. Our group has carried out research in this context [19] that could be extended to OpenMP. The application programmer could provide functions to split a running task into smaller ones, similarly to the way our adaptive loop scheduler splits ranges of iterations. This idea was applied to more general iterative algorithms where dependencies may exist between iterations [20].

4 Implementation Details on Extending libGOMP with Adaptive Loop Scheduling

We implemented our *adaptive* loop scheduler inside the original LIBGOMP library that comes with GCC 4.6.2. Our loop scheduler can be experimented with parallel loops stated as `schedule(runtime)` by setting the `OMP_SCHEDULE` environment variable to `"adaptive,chunk_size"` before running the application. This allows us to experiment with our proof-of-concept implementation without modifying the compiler.

The implementation (figure 2) of the stealing mechanism used in the adaptive loop scheduler is greatly inspired from Cilk's THE algorithm [8] designed to limit the perturbation of the serial execution from stealing-related overheads. Unlike other OpenMP loop schedulers, *adaptive* uses a per-thread data structure describing the range of iterations assigned to the considered thread. This structure basically contains the boundaries of this range (*[begin, end)*) and an atomic-based lock used to synchronize the stealing thread and its victim. Each thread pops *chunk_size* iterations to execute out of its own range (*begin += chunk_size*), until there are no more iterations left (*begin == end*). Stealing a range of iterations from a busy thread is simply a matter of shrinking the *end* bound of the victim's data structure down to *end - N*, *N* being the number of iterations we want to steal. The THE algorithm uses an optimistic approach to minimize the need for a thread to lock its own data structure on a pop operation. This can be done

Table 1. Overhead measured by EPCC benchmark (in μs) of the *adaptive* loop scheduler versus *static*, *dynamic* and *guided* on the AMD48 platform.

chunk size	1	2	4	8	16	32	64	128
static	30.44	28.20	25.97	25.03	24.40	24.50	24.18	24.50
dynamic	1328.43	594.30	232.61	75.43	36.29	35.20	34.02	33.21
guided	77.86	69.49	55.55	45.47	42.90	43.16	58.66	30.54
adaptive	55.92	50.74	48.26	47.72	47.69	47.97	48.90	49.44
adaptive (no steal)	30.29	27.48	25.67	24.48	25.21	24.48	24.16	23.23

by detecting conflicting accesses to the same data structure, by comparing the value of *end* before and after the pop operation. If the value has changed, someone accessed the data structure during the pop: the thread will then undo this last pop and acquire the lock before trying again. Such implementation greatly minimizes the overhead added to threads performing local work (Cilk's work first principle [8]).

The memory binding routines rely on *libNUMA* and the `mbind` system call. The current prototype was developed on Linux. libGOMP maintains a pool of threads (`gomp_thread` and `gomp_thread_pool`) attached to each parallel region. We extended the data structures in order to maintain simple per-thread NUMA-related information, like the core id, the NUMA node id and a list of threads per NUMA node that can be used by the *adaptive* scheduler to select a victim. Based on this information, the scheduler initializes of per-loop data structure to balance the iterations over the NUMA nodes taking the number of cores per NUMA node into account.

5 Performance Evaluation

We conducted our experiments on two different ccNUMA configurations.

The first one holds 8 AMD Magny Cours processors for a total of 48 cores. Each core has access to 64 KB of L1 cache, 512 KB of L2 cache. Both L1 and L2 caches are private, while L3 cache is shared between the 6 cores of a processor. This configuration provides a total of 256 GB (32 GB per NUMA node) of main memory. We will refer to this configuration as **AMD48** in the following of the paper.

The second configuration holds 12 groups containing two Intel Sandy Bridge processors each for a total of 192 cores. 32 GB of main memory is attached to each socket, for a total of 768 GB. Inter-groups communications use the SGI NUMAlink technology, while standard Intel QuickPath interconnect provides inner-group communications. We will refer to this configuration as **Intel192** in the following of the paper.

All experiments were performed with the libGOMP library distributed with GCC 4.6.2.

5.1 EPCC: Overhead of the Adaptive Loop Scheduler

The EPCC benchmark [5] reports runtime-related overheads when performing OpenMP loop scheduling with respect to the corresponding serial execution. The measured overheads of the four loop schedulers are reported in table 1 for different chunk sizes.

A larger chunk size implies less calls to the runtime and thus a smaller overhead. Reported measures reported represent the average performance over 10 runs. The libGOMP implementation of both the *dynamic* and the *guided* schedulers suffer from a high overhead for the three smallest chunk size values tested in this experiment.

The *adaptive* scheduler adds an extra overhead to $25\mu s$ with respect to the *static* scheduler. By disabling the steal operations from the *adaptive* scheduling algorithm, we are able to provide finer estimation of the overheads. The performance of this modified scheduler, named *adaptive (no steal)* in the table, reports no overhead induced by the initial work distribution over the *static* scheduler. We can thus infer that an extra $25\mu s$ includes the overheads of the work-stealing operations and the detection of the termination.

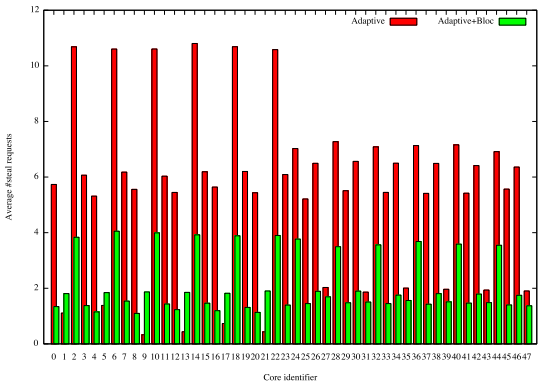
Obviously, our code is not as optimized as the other scheduler implementations from libGOMP. We will add some optimizations (memory alignment of data structures, lazy initialization) to reduce overheads, and we believe that those may improve the performance of all libGOMP schedulers as well.

5.2 STREAM: Impact of the Memory Hierarchy

The STREAM benchmark [14] measures the maximal achievable bandwidth over four memory bound kernels (`copy`, `scale`, `add` and `triad`). We evaluated the behavior of the *static* and the *adaptive* schedulers with two memory allocation strategies: a *first-touch* strategy and an explicit bloc distribution of the arrays over the 8 NUMA nodes of the AMD48 platform using the API presented in section 3.2 . The memory per array is 150.0 MB and the number of iterations is set to 500. Measures are reported in table 4a.

	static		*adaptive*	
	first-touch	bloc	first-touch	bloc
Copy	6.9	6.9	6.4	6.8
Scale	6.7	6.8	6.2	6.7
Add	7.2	7.4	6.8	7.3
Triad	6.9	7.5	6.6	7.4

(a) Achieved bandwidth (GB/s) reported by the STREAM benchmark for the *static* and *adaptive* loop schedulers.



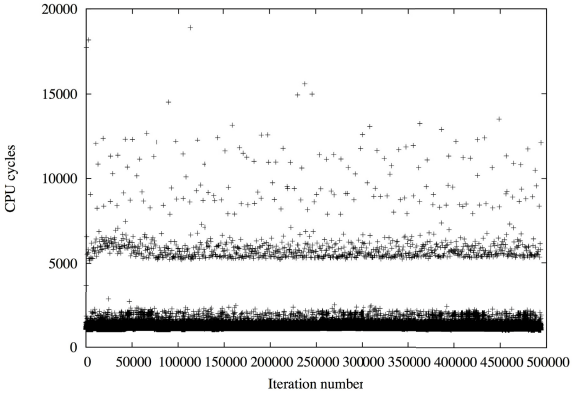
(b) Number of steal operations with the *adaptive* scheduler with and without bloc memory allocation.

Fig. 4. Performance evaluation of the STREAM benchmark on the AMD48 platform

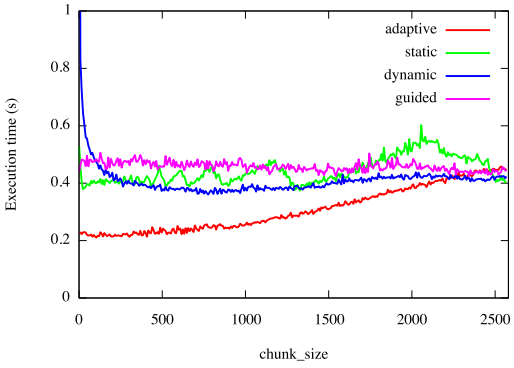
On the Triad kernel, the *bloc* allocation strategy increases the performance of at most 7% with the *static* scheduler and of at most 12% with the *adaptive* scheduler. The two schedulers reach comparable performances on this highly regular benchmark, with a slight advantage for *static* over *adaptive*.

This difference comes from the steal operations performed by the *adaptive* scheduler. Figure 4b shows the average number over 500 iterations of successful steal requests per core. We can see that the *adaptive* scheduler with the NUMA-aware extension taking the memory distribution into account helps reducing the number of steal operations.

On this memory-intensive benchmark, *adaptive* is able to reach performances that are similar to *static* as the load balancing strategy first conside the cores from the same NUMA node to perform work-stealing, thus favoring memory locality on such architectures.



(a) Execution time, in CPU cycles, of each iteration of the kmeans kernel main loop on a single core of the Intel192 platform.



(b) Average performance of different OpenMP loop schedulers for varying values of chunk_size on the Intel192 platform.

Fig. 5. Performance evaluation of the K-Means benchmark on the Intel192 platform

Table 2. Comparison of the four loop schedulers on PMA on the AMD48 platform

Time in ms	static	dynamic chunk=1	dynamic chunk=3000	guided	adaptive
numactl	17.8	57.4	12.1	15.2	11.6
bloc distribution	16.3	57.2	14.2	14.9	6.95

5.3 K-Means: Benefits of Adaptive Granularity for OpenMP Loops

We evaluated the *adaptive* loop scheduler with the OpenMP version of the K-Means kernel coming from the Rodinia benchmark suite [6]. This kernel implements a clustering algorithm commonly used by data-mining applications. Its parallel implementation involves an OpenMP parallel loop with an irregular workload.

Figure 5a reports the execution times of each one of the 494020 iterations of this loop executed on a single core of the Intel192 platform. We can distinguish at least two main classes of iterations on this figure with different execution times, but we can only consider this as a rough source of information, as the execution times of the same iterations may vary when executing them in parallel, depending for example on the capacity of threads to efficiently communicate through shared cache memory.

The results obtained running this kernel with the *adaptive* scheduler confirms K-Means can benefit from dynamic load balancing. Figure 5b shows a performance comparison between the *adaptive*, *static*, *dynamic* and *guided* schedulers on the K-Means kernel. We experimented with different values for the *chunk_size* clause of the for loop. These tests were executed on the 192 cores of the Intel192 platform. We tested every value of *chunk_size* from 1 to 2574, corresponding to **imax / nthr** here. The best performance is achieved by our scheduler. *adaptive* performs especially well for executions with small chunk sizes, as they offer more options to perform load balancing. We can also note that, even if the workload of this kernel is irregular, the best performance of the *dynamic* scheduler can almost be achieved by *static* for a tuned value of *chunk_size*.

5.4 PMA: Dealing with Both Load Balancing and Locality

We applied our *adaptive* loop scheduler to a practical situation in physical simulations considering elements of a 3D space that evolve with respect to physical laws. Maintaining these elements ordered is a key factor to improve memory efficiency as elements are likely to interact with their neighbors [11].

The Packed Memory Array (PMA) [7] data structure has been proposed to help maintaining its elements ordered in an efficient way. This sparse data structure was designed to reduce the amount of memory movement induced by reordering operations.

We focus on the loop that handles both the detection of the moving elements and their copy in a dense array. In real applications, the workload gets irregular since some parts of the physical space go through a lot of changes while others report only a few changes. We extracted actual change distribution from a memory-intensive fluid simulation [9] ran with 2 900 000 elements.

In this code, the data structure is initialized from reading sequentially a file. Without major rewrite of the initialization phase, it is not possible control the

affinity with the simple first touch strategy as OpenMP standard preaches it. Table 2 reports average times for each of the four loop schedulers with two memory distribution strategies. The first strategies, called *numactl* in the table, distributes memory pages in a round robin fashion among the NUMA nodes using `numactl --interleave`. The second strategy leads to a bloc distribution using the API `omp_locality_domain_allocate_bloc1d` presented in section 3.2.

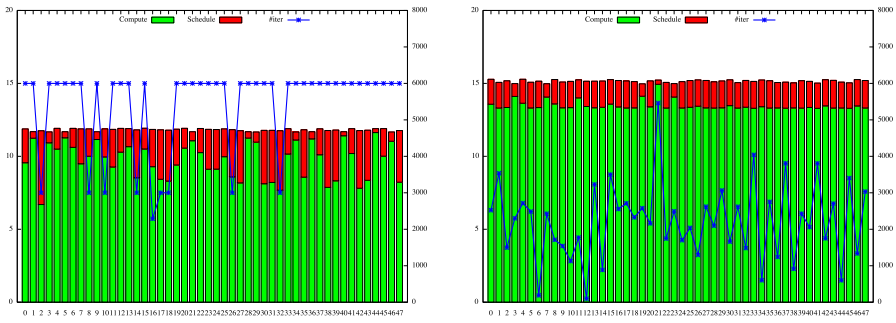
When the array is distributed with the bloc distribution, contigous elements in the array are mostly on the same NUMA node. Even with this distribution, the static scheduler does not reach the best time, which illustrates the irregularity of the application. The dynamic and guided schedulers are able to improve performance but without control of the locality while workload is balanced. The *adaptive* scheduler obtains the best times for the two memory distribution strategies. This is due to a good compromise between a good balance of iterations to control affinity and a dynamic balance of the workload.

The plots in figures 6, 7 and 8 correspond to execution where the memory is bound using bloc distribution strategy among the NUMA nodes.

Figures 6a and 6b report a per-core execution using the libGOMP *dynamic* and *guided* schedulers with a bloc data distribution over the NUMA nodes. The "compute" (green) part of the graph represents the time spent, in CPU cycles, during the execution of the loop body. The "schedule" part represents the time spent to perform the required computations, apply the scheduling decisions and wait until the loop completion.

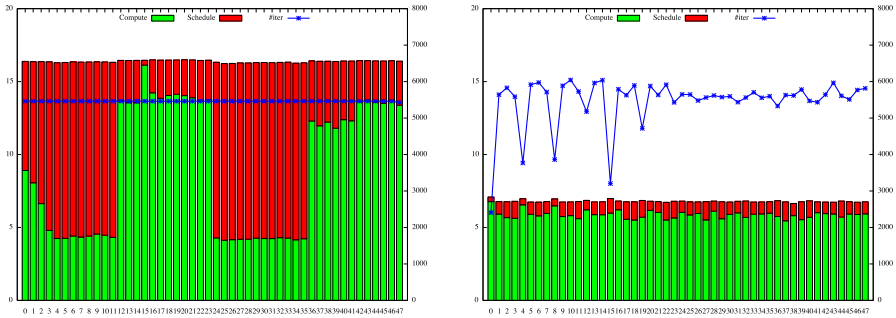
The histogram represents the number of iterations performed by each core. We can conclude from the top two plots that:

1. the libGOMP *dynamic* scheduler with a chunk size of 1 spends a lot of time in the runtime system. This is mainly due to contention generated by concurrent accesses to internal data structures,
2. the number of iterations and the time spent executing iterations vary from one core to another. The computation of a correct chunk size helps decreasing the schedule time. The performance is thus increased as the average time per iteration decreases from $57.2ms$ to $14.2ms$.



(a) Dynamic scheduler, chunk size=3000 (b) Guided scheduler

Fig. 6. Times (ms) per core for the same PMA iteration with dynamic and guided schedulers. The histograms have the same scale.



(a) Static scheduler (b) Adaptive scheduler

Fig. 7. Times (ms) per core for the same PAM iteration with static and adaptive schedulers. The histograms have the same scale.

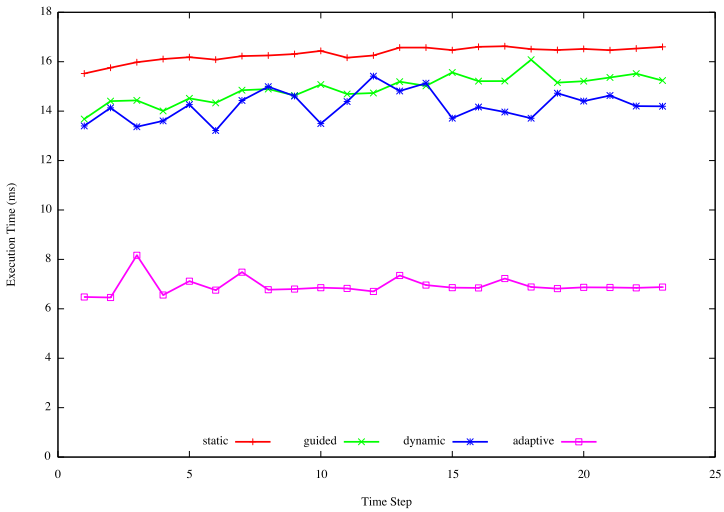


Fig. 8. Comparison of loop schedulers with respect to the time step of PMA simulation

Figures 7a and 7b report results with the *static* and our *adaptive* loop schedulers with a bloc data distribution. The blue line in the bottom left plot validates the behavior of the static scheduler, as every thread executes the same number of iterations. Nevertheless, the CPU times are highly variable: the distribution of iterations fails at balancing the workload. On the other hand, our *adaptive* scheduler is able to keep the workload well balanced at the expense of an irregular distribution of iterations. The average execution time is $6.95ms$ for the *adaptive* scheduler and $16.3ms$ for the *static* scheduler.

Figure 8 reports the behavior of the 4 loop schedulers on PMA simulation with respect to the time step. Even if the iterations are well balanced among the cores, the *static* scheduler is unable to balance the workload. Both the *dynamic* and the *guided* schedulers reach the same level of performances and are able to better balance the workload,

even if memory affinity is not ensured. Our adaptive scheduler is a good compromise between the static and the dynamic schedulers, and reaches a speed-up of 2.35 over the *static* scheduler.

6 Conclusion and Future Work

This paper introduced *adaptive*, a new OpenMP loop scheduler implementing a runtime-level approach to deal with irregular memory-bound applications on NUMA architectures. Instead of distributing statically-fixed portions of work over the threads of a parallel region, this scheduler adapts the granularity of work on demand by making idle threads steal a subset of the victim's remaining iterations, thus introducing the notion of dynamic per-thread granularity. Our scheduler is also capable of adapting its work-stealing algorithm to fit different memory bindings on NUMA architectures and outperforms OpenMP-based approaches to deal with memory locality, like the joint use of the *static* loop scheduling and the *first-touch* allocation policy, on several benchmarks and applications.

This work could be extended to task parallelism, providing the OpenMP application programmer with ways of annotating *splitter* functions called to generate parallelism on demand by splitting a running task into smaller ones. We also consider extending our approach using the concept of *places* recently added to the OpenMP standard that could help the programmer transmitting valuable and portable information on the way memory should be allocated on hierarchical architectures.

Acknowledgement. This work has been partially supported by the ANR 09-COSI-011-05 Repdyn project and the FP7-PEOPLE-2011-IRSES HPC-GA project.

References

1. Ayguadé, E., Blainey, B., Duran, A., Labarta, J., Martínez, F., Martorell, X., Silvera, R.: Is the Schedule clause really necessary in openMP? In: Voss, M.J. (ed.) WOMPAT 2003. LNCS, vol. 2716, pp. 147–159. Springer, Heidelberg (2003)
2. Broquedis, F., Aumage, O., Goglin, B., Thibault, S., Wacrenier, P.-A., Namyst, R.: Structuring the execution of OpenMP applications for multicore architectures. In: Proceedings of 24th IEEE International Parallel and Distributed Processing Symposium (IPDPS 2010), Atlanta, GA. IEEE Computer Society Press (April 2010)
3. Broquedis, F., Gautier, T., Danjean, V.: LIBKOMP, an efficient openMP runtime system for both fork-join and data flow paradigms. In: Chapman, B.M., Massaioli, F., Müller, M.S., Rorro, M. (eds.) IWOMP 2012. LNCS, vol. 7312, pp. 102–115. Springer, Heidelberg (2012)
4. Broquedis, F., Furmento, N., Goglin, B., Namyst, R., Wacrenier, P.-A.: Dynamic Task and Data Placement over NUMA Architectures: An OpenMP Runtime Perspective. In: Müller, M.S., de Supinski, B.R., Chapman, B.M. (eds.) IWOMP 2009. LNCS, vol. 5568, pp. 79–92. Springer, Heidelberg (2009)
5. Bull, J.M.: Measuring synchronisation and scheduling overheads in openmp. In: Proceedings of First European Workshop on OpenMP, pp. 99–105 (1999)
6. Che, S., Boyer, M., Meng, J., Tarjan, D., Sheaffer, J.W., Lee, S.-H., Skadron, K.: Rodinia: A benchmark suite for heterogeneous computing. In: Proceedings of the 2009 IEEE International Symposium on Workload Characterization, IISWC 2009, pp. 44–54. IEEE Computer Society, Washington, DC (2009)

7. Durand, M., Raffin, B., Faure, F.: A Packed Memory Array to Keep Moving Particles Sorted. In: 9th Workshop on Virtual Reality Interaction and Physical Simulation (2012)
8. Frigo, M., Leiserson, C.E., Randall, K.H.: The implementation of the cilk-5 multithreaded language. SIGPLAN Not. 33(5), 212–223 (1998)
9. Hoetzlein, R.C.: Fluids v2.0, open source, fluid simulator (2008)
10. Huang, L., Jin, H., Yi, L., Chapman, B.: Enabling locality-aware computations in openmp. Sci. Program. 18(3-4), 169–181 (2010)
11. Ihmsen, M., Akinci, N., Becker, M., Teschner, M.: A parallel sph implementation on multi-core cpus. Computer Graphics Forum 30(1), 99–112 (2011)
12. Mahéo, A., Koliaï, S., Carribault, P., Pérache, M., Jalby, W.: Adaptive openmp for large numa nodes. In: Chapman, B.M., Massaioli, F., Müller, M.S., Rorro, M. (eds.) IWOMP 2012. LNCS, vol. 7312, pp. 254–257. Springer, Heidelberg (2012)
13. Marowka, A., Liu, Z., Chapman, B.: Openmp-oriented applications for distributed shared memory architectures: Research articles. Concurr. Comput.: Pract. Exper. (2004)
14. McCalpin, J.D.: Memory bandwidth and machine balance in current high performance computers. In: IEEE Computer Society Technical Committee on Computer Architecture (TCCA) Newsletter, pp. 19–25 (December 1995)
15. Olivier, S.L., de Supinski, B.R., Schulz, M., Prins, J.F.: Characterizing and mitigating work time inflation in task parallel programs. In: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis, SC 2012, pp. 65:1–65:12. IEEE Computer Society Press, Los Alamitos
16. Olivier, S.L., Porterfield, A.K., Wheeler, K.B., Spiegel, M., Prins, J.F.: Openmp task scheduling strategies for multicore numa systems. Int. J. High Perform. Comput. Appl. 26(2), 110–124 (2012)
17. OpenMP Architecture Review Board (1997-2008), `http://www.openmp.org`
18. Subramaniam, S., Eager, D.L.: Affinity scheduling of unbalanced workloads. In: Proceedings of the 1994 ACM/IEEE Conference on Supercomputing, Supercomputing 1994, pp. 214–226. IEEE Computer Society Press, Los Alamitos (1994)
19. Tchiboukdjian, M., Danjean, V., Gautier, T., Le Mentec, F., Raffin, B.: A work stealing scheduler for parallel loops on shared cache multicores. In: Proceedings of the 2010 Conference on Parallel Processing, Euro-Par 2010, pp. 99–107. Springer (2011)
20. Traoré, D., Roch, J.-L., Maillard, N., Gautier, T., Bernard, J.: Deque-free work-optimal parallel stl algorithms. In: Proceedings of the 14th International Euro-Par Conference on Parallel Processing, Euro-Par 2008, Berlin, Heidelberg, pp. 887–897 (2008)
21. Yan, Y., Jin, C., Zhang, X.: Adaptively scheduling parallel loops in distributed shared-memory systems. IEEE Trans. on Parallel and Distributed Systems 1 (January 1997)

Locality-Aware Task Scheduling and Data Distribution on NUMA Systems

Ananya Muddukrishna[1], Peter A. Jonsson[2], Vladimir Vlassov[1], and Mats Brorsson[1,2]

[1] KTH Royal Institute of Technology
[2] SICS Swedish ICT AB

Abstract. Modern parallel computer systems exhibit Non-Uniform Memory Access (NUMA) behavior. For best performance, any parallel program therefore has to match data allocation and scheduling of computations to the memory architecture of the machine. When done manually, this becomes a tedious process and since each individual system has its own peculiarities this also leads to programs that are not performance-portable.

We propose the use of a data distribution scheme in which NUMA hardware peculiarities are abstracted away from the programmer and data distribution is delegated to a runtime system which is generated once for each machine. In addition we propose using task data dependence information now possible with the OpenMP 4.0RC2 proposal to guide the scheduling of OpenMP tasks to further reduce data stall times.

We demonstrate the viability and performance of our proposals on a four socket AMD Opteron machine with eight NUMA nodes. We identify that both data distribution and locality-aware task scheduling improves performance compared to default policies while still providing an architecture-oblivious approach for the programmer.

1 Introduction

According to conventional wisdom, memory intensive OpenMP programs must distribute data wisely across NUMA nodes to minimize remote memory accesses. However, NUMA architectures have reached such complexity that even simple memory-oblivious algorithms such as the recursive fibonacci have begun to suffer from NUMA effects [1]. Careful data distribution, regardless of memory footprint, has become an absolute necessity for achieving good performance on NUMA systems.

Despite the importance of data distribution there are no mechanisms available to application programmers for dealing with the issue, even in the latest OpenMP 4.0 Public Review RC2 version. Programmers cope by using third-party tools and APIs [2–4] or by re-purposing the OpenMP `for` work-sharing construct to allocate and distribute data to different NUMA nodes. The third-party tools are fragile and might not be available on all machines, and the clever use of the `for` work sharing construct [5] depends on a particular OS page management policy and require that the programmer knows about the NUMA node topology on the target machine.

A.P. Rendell et al. (Eds.): IWOMP 2013, LNCS 8122, pp. 156–170, 2013.

Expert programmers can still work around the current situation, but even for them the process can be described as fragile and error prone. The average programmers that do not manage to cope with all the complexity at once pay a performance penalty when running their applications, a penalty that might be partially mitigated from clever caching by the hardware. The current situation will get increasingly worse for everybody since NUMA effects are exacerbated by growing network diameters and increased cache coherence complexity [6] that will inevitably follow from increasing sizes of NUMA machines.

We present a runtime system assisted data distribution scheme that allows programmers to control data distribution in a portable fashion without forcing them to understand low-level system details. The scheme relies on the programmer to provide high-level hints on the granularity of the data distribution in calls to `malloc`. Programs without hints will work and have the same performance as before, which allows gradual addition of hints to programs to get partial performance benefits. Our runtime system assisted distribution scheme requires nearly the same programmer effort as regular calls to `malloc` and yet doubles the performance for some scientific workloads.

We also present a locality-aware task scheduling algorithm which reduces memory access times by leveraging page locality information gained from data distribution and task data footprint information from the programmer. Our scheduling algorithm demonstrates a performance improvement of up to 11% compared to a work-stealing scheduler when NUMA effects degrade application performance and remains competitive for other applications.

2 Potential for Performance Improvements

We indicate the performance improvement that can be gained by distributing application data across NUMA nodes by means of an experiment. We execute task-based OpenMP applications taken or derived from the Barcelona OpenMP Task Suite (BOTS) [7] with two different memory allocation strategies: the first strategy uses malloc with the *first-touch* policy and the second distributes memory pages evenly across NUMA nodes using the numactl [4] tool. We use first-touch as a short hand for malloc with first-touch policy in the rest of the paper.

We perform the experiment on a four socket, eight NUMA node machine built using four AMD Opteron 6172 processors. We use the OpenMP implementation from the Intel C compiler. We measure the execution time of the parallel section of each application and count the number of dispatch stall cycles to quantify the amount of time spent waiting for memory. The dispatch stall cycles include Load/Store Unit stall cycles [8].

Five out of nine applications show a significant reduction in execution time when data is distributed across NUMA nodes as shown in Figure 1. The reduction in dispatch stall cycles contributes to the reduction in execution time. The performance of the remaining applications, except for strassen, is approximately the same with data distribution and first-touch cases. Speedup compared to sequential execution is close to ideal which implies low communication as seen in

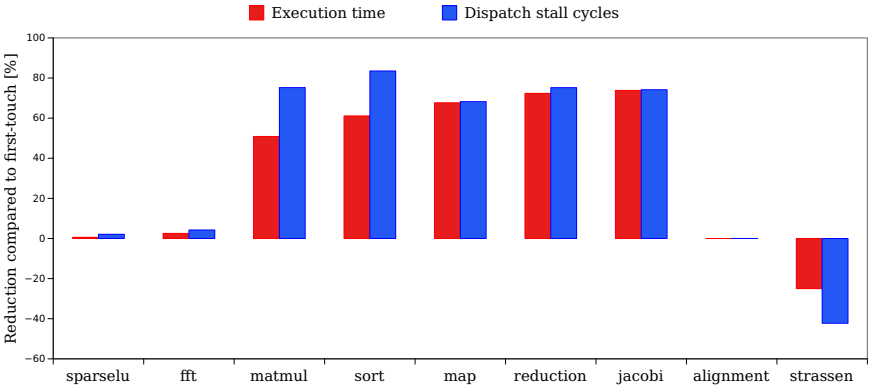


Fig. 1. Execution time and dispatch stall cycle reduction when application data is distributed across NUMA nodes in comparison to first-touch. Execution time corresponds to the critical path of parallel section. Dispatch stall cycles are aggregated over all application tasks.

the alignment benchmark or that page-wise interleaved data distribution does not relieve the memory sub-system as seen in the fft, sparselu and strassen benchmarks.

In summary, overheads from memory access latencies are significant in OpenMP applications. Our goal is to provide simple and portable abstractions that mitigate these overheads by performing data distribution.

3 Runtime System Assisted Data Distribution

Runtime system assisted data distribution is one mechanism for increasing performance portability – handling specific OS and hardware details can be delegated to an architecture-specific runtime system which has a global view of the execution of the application.

We propose a memory allocation and distribution mechanism controlled by a simple data distribution policy which is chosen by the programmer. The distribution policy choice is deliberately kept simple with only a few choices in order to provide predictable behavior and be easy to understand for the programmer, just like process binding hints in OpenMP are defined. There are two different policies available to the programmer:

Fine: distributes memory with a page-wise round-robin interleaving across all NUMA nodes. This policy is a good choice if data structures are allocated using a single call to `malloc` in the original application and many tasks operate on the same data structures.

Coarse: distributes memory with a block-wise round-robin interleaving across all NUMA nodes. This policy is a good choice if the data structures are allocated using multiple calls to `malloc` in the original application and tasks operate exclusively on the data structures.

We demonstrate how the data distribution policies work at the machine level and propose preliminary interfaces for policy selection using an example C program in Figure 2. The example program requests memory using a proposed interface called `omp_malloc` whose signature is similar to `malloc`. The two memory allocation requests, A and B, are eight memory pages each. The user selects the data distribution policy by setting a proposed environment variable OMP_DATA_DISTRIBUTION to one of `standard`, `fine` or `coarse` prior to the program invocation. The `standard` data distribution policy choice refers to the machine default – first-touch. The memory requested using `omp_malloc` is distributed to different NUMA nodes based on the global data distribution policy selected.

```
int main(...)
{
    ...
    /* Allocate data */
    size_t sz = 8 * PAGE_SIZE;
    void * A = omp_malloc(sz);
    void * B = omp_malloc(sz);
    ...
    /* Initialize data */
    init(A, sz, ...);
    init(B, sz, ...);
    ...
    /* Work in parallel */
    ...
}

$ export OMP_DATA_DISTRIBUTION=<standard | fine | coarse>
$ <invoke application>
```

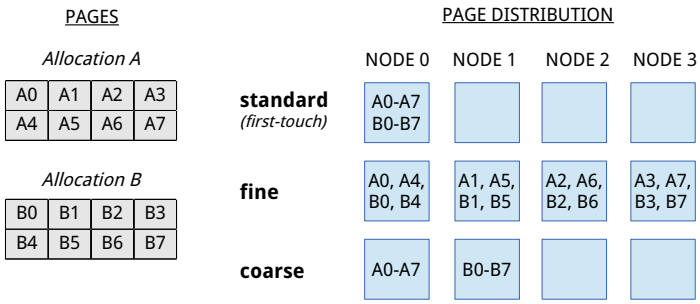


Fig. 2. Proposed preliminary interfaces for selecting the policy-based data distribution and machine level results

We provide heuristics in Table 1 to assist in the choice of data distribution policy based on the number of `malloc` based data allocations in the original application and the number of tasks operating on those allocations.

We have built the runtime system assisted distribution scheme using readily available OS support - libnuma on Linux. The overheads of the distribution scheme is low since the implementation is essentially a wrapper for libnuma API with a few additional book-keeping instructions. The book-keeping instructions track the round-robin node selection counter for the coarse distribution policy and cache the NUMA node affinity of memory pages when requested by the locality-aware scheduling policy as described in the Section 4.

Programmers do not need to be concerned about NUMA node identifiers and topology in order to use our data distribution scheme. The distribution policy choice is kept simple with only those choices that are easy to predict and understand for the programmer. In addition, programmers can incrementally distribute data by targeting specific memory allocation sites. We also provide precise control for expert programmers in our implementation allowing them to over-ride the global data distribution policy and request fine or coarse data distribution for a specific allocation as shown in Figure 3.

We have implemented two simple distribution policies to demonstrate the potential of our data distribution scheme. Runtime system developers can use the extensibility of our scheme to provide more advanced distribution policies as plug-ins. Programmers can be educated about distribution policies in a manner similar to existing education about for loop scheduling policies within the OpenMP specification.

4 Locality-Aware Task Scheduling

Our implementation of locality-aware scheduling aims to further leverage the performance benefits of data distribution. The main idea behind our locality-aware scheduler is to schedule tasks with minimal memory access latencies. Our machine model consists of a number of NUMA nodes where each node has local access to DRAM and is connected directly to a set of cores. The locality-aware scheduler use one task queue per NUMA node and takes locality-aware decisions both during work-dealing and work-stealing. Work-dealing are to actions taken at the point of task creation and work-stealing are actions taken when cores are idle.

Knowing the data footprint of tasks is crucial for the scheduler. We expect data footprint information to come from the programmer through expressive task definition clauses which do not yet exist in the OpenMP specification. We currently estimate the data footprint of each task through the information provided by the depend clause which is new in the OpenMP specification version 4.0RC2.

Table 1. Heuristics to select data distribution scheme

		Number of tasks operating on data	
		One	Many
Number of `malloc` calls	One	Regular `malloc`	Fine
	Many	Coarse	Coarse

```
int main(...)
{
    ...
    /* Allocate data */
    size_t sz = 8 * PAGE_SIZE;
    void * A = omp_malloc_specific(sz, OMP_MALLOC_COARSE);
    void * B = omp_malloc_specific(sz, OMP_MALLOC_COARSE);
    void * C = omp_malloc_specific(sz, OMP_MALLOC_FINE);
    ...
    /* Initialize data */
    init(A, sz, ...);
    init(B, sz, ...);
    init(C, sz, ...);
    ...
    /* Work in parallel */
    ...
}
```

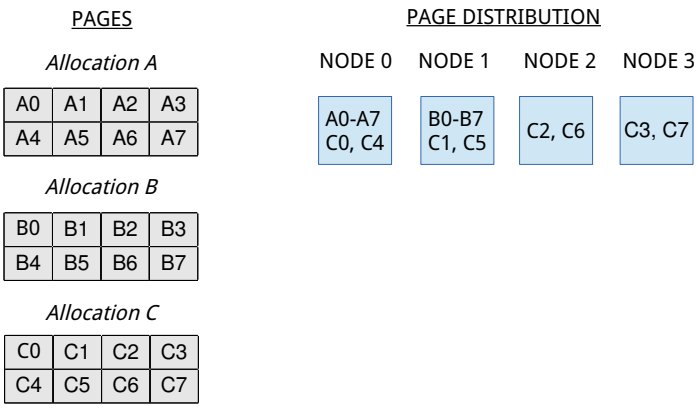


Fig. 3. Proposed interfaces for specific data distribution

This estimate is fragile when programmers specify an incomplete depend clause that is sufficient for scheduling decisions but underestimate the data footprint. The limitation can be overcome if programmers use low-effort expressive constructs such as array-sections to express a large fraction of the data footprint in the depend clause in return for improved execution performance.

We describe the work-dealing algorithm of the locality-aware scheduler in Algorithm 1. The scheduler deals a task at the point of task creation to the node queue having the least total memory access latency for pages not in the last-level cache (LLC). The individual access latencies are computed by weighting NUMA node distances with the node-wise distribution D of the data footprint of the task.

NUMA node distances are obtained from OS tables which are cached by the scheduler for performance reasons. The distribution D is calculated using page

locality information cached by the data distribution mechanism. The complexity of the access cost computation is $O(N^2)$ where N is the number of NUMA nodes in the system, typically a small number.

```
1  Procedure deal-work(task T, queues Q_1, ..., Q_N, current node n, cores per
   node C)
2      Populate D[1:N] with bytes in T.depend_list;
3      if sum(D) > sizeof(LLC)/C and Standard_Deviation(D) > 0 then
4          find Q_l with least NUMA distance-weighted cost to D ;
5          i = l;
6      else
7          i = n;
8      end
9      enqueue(Q_i, T);
10 end
```

Algorithm 1: Work-dealing algorithm

Tasks are immediately added into the local queue if two thresholds make it clear that scheduling costs outweigh the performance benefits. The first threshold - *Sum(D) > sizeof(LLC)/C* - ensures that tasks have a working set size exceeding the LLC size per core. The second threshold - *Standard_Deviation(D) > 0* - ensures that scheduling effort will not be wasted on tasks with a perfect data distribution.

Distributed task queues may lead to load-imbalance and in our experience the performance benefits from load-balancing often trumps those from locality. We have therefore implemented a work-stealing algorithm to balance the load. The cycles spent while dealing tasks are wasted when steals occur but stealing is still preferred over idle cores.

```
1  Procedure find-work(queues Q_1, ..., Q_N, current node n, cores per node C)
2      if empty Q_n then
3          Attempt to steal work;
4          for Q_i in (Sort Q_1 ... Q_N by NUMA distance from n) do
5              if sizeof(Q_i) > distance(i, n)*C then
6                  Run dequeue(Q_i);
7                  break;
8              end
9          end
10     else
11         Run dequeue(Q_n);
12     end
13 end
```

Algorithm 2: Work-finding algorithm

We show the stealing algorithm of the scheduler in Algorithm 2. Cores attempt to steal when there is no work in the local queue. Candidate queues for steals

are ranked based on NUMA node distances. The algorithm includes a threshold which prevents tasks from being stolen from nearly empty task queues which would incur further steals for the cores in the victim node. In addition, there is an exponential back-off for steal attempts when work cannot be found.

5 Experimental Setup

We perform our evaluation on an AMD Opteron 6172 processor based system with eight NUMA nodes, four sockets and 48 cores. Cache sizes of the processor are: 64 KB DL1, 512 KB DL2 and 5MB LLC. The maximum NUMA distance of the system obtained from OS tables is 22. NUMA interconnects of the system are configured for maximum performance with an average NUMA factor of 1.19 [9]. Memory latencies of the system measured using the BenchIT tool are similar to the measurements reported by Molka et al. [10]. The OS is Linux v2.6.32. We use the Intel C compiler v13.1.1 with -O3 to compile both the runtime system and benchmarks.

The benchmarks we use in the evaluation are described in Table 2. The benchmarks are executed using MIR, a task-based runtime system library which we have developed. MIR supports the OpenMP tied tasks model. We have programmed the evaluation benchmarks using the runtime system interface directly since MIR does not currently have a source-to-source translation front-end. MIR provides plugin based scheduling and data allocation policies which allows us to compare different policies within the same system.

Table 2. Benchmarks used in the evaluation. T_{48} in baseline speedup computation corresponds to page-wise interleaved data distribution using numactl.

Benchmark	Behavior	Input	Baseline speedup (T_{48}/T_1)
matmul	Blocked matrix multiplication with BLAS task kernel	Dimension = 4096; Block size = 128	32
map	1D vector scaling	48 vectors of 1MB each	18
jacobi	Blocked 2D heat equation solver	Dimension = 16834; Block size = 512	36
sparselu	LU factorization of sparse matrix from BOTS [7]	Dimension = 8192; Block size = 256	35
reduction	Iterative implementation of the merge phase of Sort from BOTS	Depth = 10; Array size = 256 MB	6

We use a work-stealing scheduler as the baseline for comparing the locality-aware scheduler. The work-stealing scheduler has one task queue per core. Each core adds newly-created tasks to its own task queue. Cores look for work in their own task queue first. Cores with empty task queues select victims for stealing in a round-robin fashion. Both queueing and stealing decisions of the work-stealing scheduler are fast but can result in drastic memory latencies during task execution since the work-stealing scheduler is oblivious to data locality.

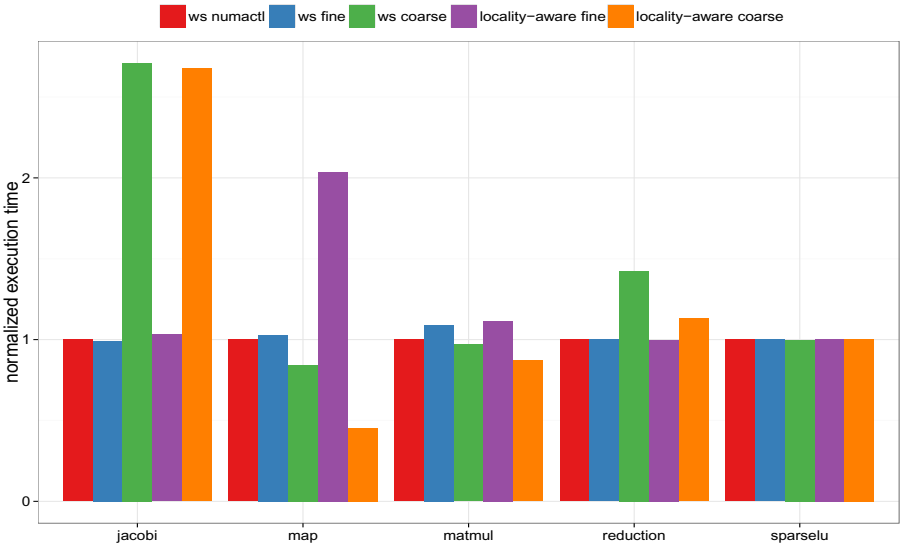


Fig. 4. Performance of data distribution combined with work-stealing and locality-aware scheduling. Execution time is normalized against the first column for each benchmark.

We use cycle counters and dispatch stall cycle counters to measure the execution time and memory access latency of tasks. We also collect execution traces of thread state which can be viewed using Paraver [11].

We run each benchmark 20 times for all combinations of scheduling and data distribution policies except for the combination of numactl page-wise interleaving and locality-aware scheduling. The locality-aware scheduler does not currently support querying numactl for page locality information. We record the execution time of the critical path of the parallel section as the main performance metric to measure speedup. We collect execution traces and performance counter events on an additional set of runs to conduct a detailed analysis using Paraver.

6 Results

We show the performance of our set of evaluation benchmarks for combinations of data distribution and scheduling policy in Figure 4. The fine distribution is a feasible replacement for numactl since execution times with the work-stealing scheduler are comparable to page-wise interleaving using numactl. As expected, when coarse distribution is used according to the guidelines in Table 1 the execution time is reduced in each benchmark for both schedulers and when using coarse distribution against the guidelines the execution time increases for both schedulers. The locality-aware scheduler coupled with the best data distribution choice for each benchmark gives execution times that are comparable with or reduced compared to the work-stealing scheduler.

We use execution timelines for the map and matmul benchmark in Figure 5 to explain that reduced memory page access time is the main reason behind the difference in task execution times of the work-stealing and locality-aware scheduler. The execution timeline captures the entire program execution. The timeline is essentially a 3D figure drawn with cores on the Y-axis, time on the X-axis and dispatch stall cycles on the Z-axis. The Z-axis is represented in using a linear green to blue gradient which encodes dispatch stall cycles. We filter out all core states except the state of task execution. On the Z-axis, green presents lower dispatch stall cycles and blue higher. The execution timelines are time aligned (same X-axis span) and gradient aligned (same Z-axis span) for all benchmarks.

We can explain why the locality-aware scheduler reduces task execution time by understanding the structure of the benchmarks. Each task in the map benchmark scale the values of a single array chosen from a list and with coarse distribution all pages of the array reside on a single node whereas fine distribution spreads the pages evenly across all nodes. Tasks of the matmul benchmark update a block in the output matrix using a chain of blocks from the input matrices.

For the map benchmark, the locality-aware scheduler combined with coarse distribution ensures that each task accesses its unique vector from the local node. The behavior is consistent with the low number of dispatch stall cycles seen in Figure 5. The work-stealing scheduler steals tasks without consideration for locality and risks increasing remote memory accesses which is witnessed by the relatively large number of stall cycles appearing as dark green and blue. The locality-aware scheduler with fine distribution detects that pages are evenly balanced across nodes and places all tasks on the same local queue. The imbalance can not be completely recovered from since steals are restricted (Table 3). The work-stealing scheduler is able to balance the load more effectively (Table 3) which results in reduced execution times. We note that fine distribution is a case of going against the guidelines and also results in the slowest execution for both schedulers since pages are interleaved across all nodes.

For the matmul benchmark, it may seem surprising to see the 11% reduction in task execution time produced by the locality-aware scheduler. The memory pages touched by a task are located on different nodes for both coarse and fine

Table 3. Work-dealing time and task steal ratio for matmul and map benchmarks aggregated over all cores. Steal ratio is the number of tasks stolen divided by the number of tasks created.

	matmul		map	
	Work-deal time (cycles)	Steal ratio	Work-deal time (cycles)	Steal ratio
ws numactl	775293	1003/1024	44762	47/48
ws fine	480245	1002/1024	51157	47/48
ws coarse	498719	1003/1024	47799	47/48
locality-aware fine	116170458	0/1024	106536909	6/48
locality-aware coarse	26370885	0/1024	512435	0/48

distribution. With fine distribution, the locality-aware scheduler detects that data is evenly distributed and falls back to work-stealing by queuing tasks in its own node. Both schedulers show increased execution time for tasks as revealed by similar amounts of dispatch stall cycles (similar intensity levels of green and blue) in the timeline. However, with coarse distribution the locality-aware scheduler is able to exploit the locality arising from distributing blocks of pages in round-robin as evident by the relatively smaller amount of dispatch stall cycles (lighter intensity levels of green and blue) in comparison to the work-stealing scheduler.

The locality-aware scheduler can safely be used as the default scheduler for all work loads without performance degradation. For workloads which provide strong locality with data distribution, there is a performance benefit in using the locality-aware scheduler. For workloads which do not improve locality with data distribution, the locality-aware scheduler behaves similar to the work-stealing scheduler. The locality-aware scheduler can also work for OpenMP untied tasks since it preserves locality by restricting steals at node boundaries. Untied tasks are more likely to be stolen by cores of the same node for balanced applications.

7 Related Work

Numerous ways of how to distribute data programmatically on NUMA machines have been proposed in the literature. We discuss the proposals that are closest to our approach.

Huang et al. [12] propose extensions to OpenMP to distribute data over an abstract notion of locations. The primary distribution scheme is a block-wise distribution which is similar to our coarse distribution scheme. The scheme allows precise control of data distribution but relies on compiler support and additionally requires changes to the OpenMP specification. Locations provide fine-grained control over data distribution at the expense of programming effort.

The Minas framework [3] incorporates a sophisticated data distribution API which gives precise control on where memory pages end up. The API is intended to be used by an automatic code transformation in Minas that uses profiling information for finding the best possible distribution for a given application. The precise control is powerful but requires expert programmers who are capable of writing code that will decide on the distribution required.

Majo and Gross [13] use fine grained data distribution API to distribute memory pages. Execution profiling is used to get data access patterns of loops and used for both guiding code transformation as well as data distribution. Data distribution is performed in between loop iterations which guarantee that each loop iteration accesses memory pages locally.

Runtime tracing techniques that provide automatic page migration based on hardware monitoring through performance counters have the same end goal as we do: to provide good performance with low programming effort. Nikolopoulos

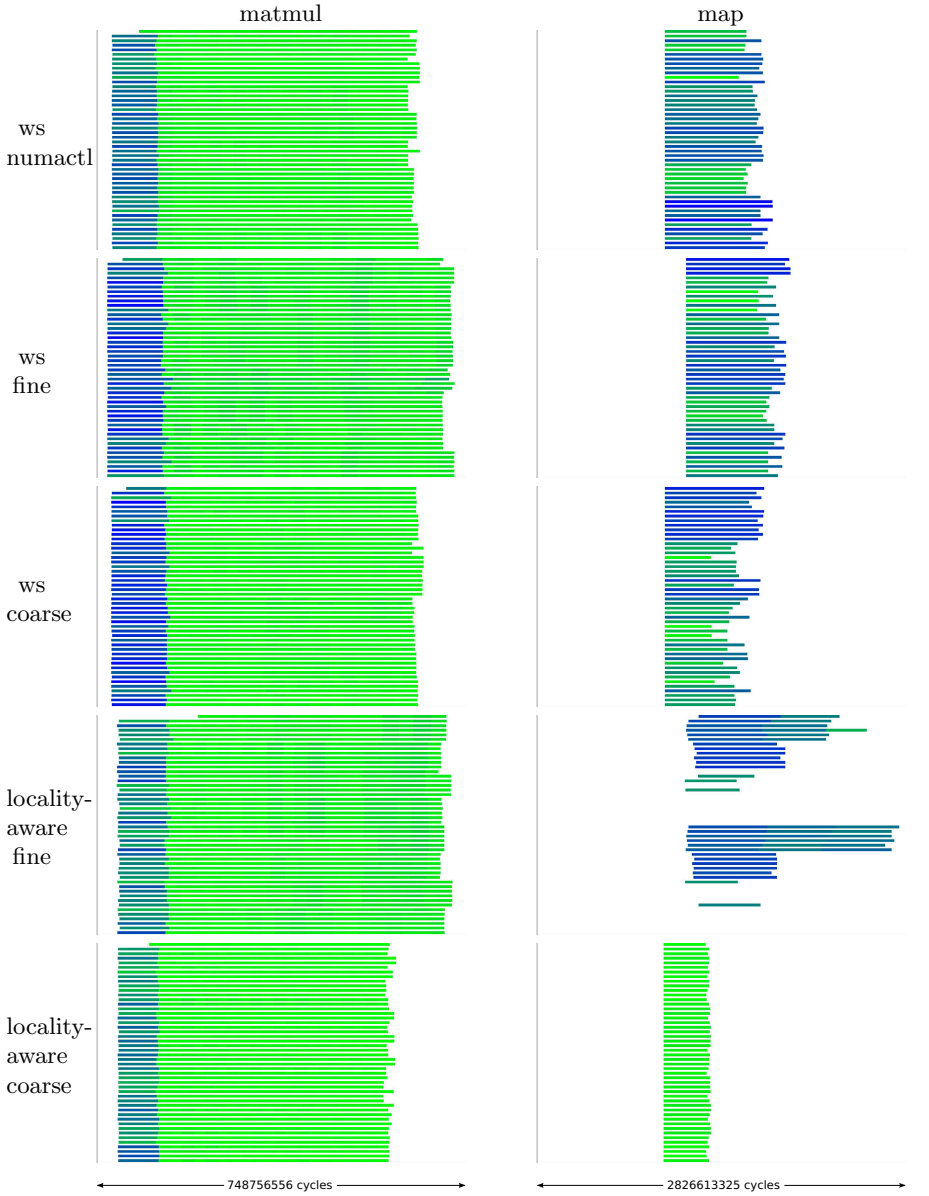


Fig. 5. Execution timelines to demonstrate reduced task execution time and dispatch stall cycles. The timeline for the matmul benchmark is zoomed-in.

et al. [14] pioneered the idea of page migration with user-level framework. Page accesses are traced in the background and *hot* pages are migrated closer to the accessing node. Terboven et al. [15] presented a next-touch dynamic page migration implementation on Linux. An alternative approach to migrating pages

which is costly is to move threads instead, an idea exploited by Broquedis et al. [16] in a framework where decisions to migrate threads and data are based on information about thread idleness, available node memory, and hardware performance counters.

Dynamic page migration requires zero effort from the programmer, which is a double edged sword. The benefit of getting good performance without any effort is obvious, but when the programmer experiences bad performance it is difficult to analyze the root cause of the problem. Performance can also be affected by input changes. Attempts at reducing the cost of page migration by providing native kernel support give promising results for matrix multiplication on large matrices [17].

Locality-aware scheduling for OpenMP has been studied extensively. Since our approach is based on tasks we indicate task based approaches relevant for our work.

Locality domains where programmers manually place tasks in abstract bins have been proposed [1, 18]. The tasks are scheduled within their domain which reduces remote memory accesses. MTS [19] is a scheduling policy structured on the socket hierarchy of the machine. MTS uses one task queue per socket which is similar to our task queue per NUMA node. Only one idle core per socket is allowed to steal bulk work from other sockets. Charm++ uses NUMA topology information and task communication information to reduce communication costs between tasks [20].

Memphis uses hardware monitoring techniques and provide methods to fix NUMA problems on general class of OpenMP computations [6]. Monitoring cross-bar (QPI) related and LLC cache miss related performance counters are used to measure network activity. Memphis provides diagnostics to the programmer for when to pin threads, distribute memory and keep computation in a consistent shape throughout the execution. Their recommendations have inspired the design of our locality-aware scheduler and our evaluation methodology.

Schmidl et al. propose the keywords scatter and compact for guiding thread placement using SLIT-like distance matrices [21]. Our names for data distribution, fine and coarse, are directly inspired by their work.

8 Conclusions

We have presented a data distribution and memory page locality-aware scheduling technique that gives good performance in our tests. The major benefit is simplicity to use which allows ordinary programmers to reduce their suffering from NUMA effects by reduced performance. Our technique can be built on standard components provided by the operating system without special frameworks and third party tools. The locality-aware scheduler can be used as the default scheduler since it will fall back to behaving like a work-stealing scheduler when locality is missing, something also indicated from our measurements.

Acknowledgments. This work was partially funded by European FP7 project ENCORE Project grant agreement nr. 248647 and Artemis PaPP Project nr. 295440.

References

1. Olivier, S.L., de Supinski, B.R., Schulz, M., Prins, J.F.: Characterizing and mitigating work time inflation in task parallel programs. In: 2012 International Conference on High Performance Computing, Networking, Storage and Analysis (SC), pp. 1–12 (2012)
2. Broquedis, F., Clet-Ortega, J., Moreaud, S., Furmento, N., Goglin, B., Mercier, G., Thibault, S., Namyst, R.: hwloc: A generic framework for managing hardware affinities in HPC applications. In: 2010 18th Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP), pp. 180–186 (2010)
3. Ribeiro, C.P., Méhaut, J.F.: Minas: Memory affinity management framework (2009)
4. Kleen, A.: A NUMA API for Linux. Novel Inc. (2005)
5. Terboven, C., Schmidl, D., Cramer, T., an Mey, D.: Assessing openMP tasking implementations on NUMA architectures. In: Chapman, B.M., Massaioli, F., Müller, M.S., Rorro, M. (eds.) IWOMP 2012. LNCS, vol. 7312, pp. 182–195. Springer, Heidelberg (2012)
6. McCurdy, C., Vetter, J.: Memphis: Finding and fixing NUMA-related performance problems on multi-core platforms
7. Duran, A., Teruel, X., Ferrer, R., Martorell, X., Ayguade, E.: Barcelona OpenMP Tasks Suite: A set of benchmarks targeting the exploitation of task parallelism in OpenMP. In: International Conference on Parallel Processing, ICPP 2009, pp. 124–131 (2009)
8. AMD: BIOS and Kernel Developers Guide for AMD Family 10h Processors
9. Conway, P., Kalyanasundharam, N., Donley, G., Lepak, K., Hughes, B.: Cache hierarchy and memory subsystem of the AMD Opteron processor. IEEE Micro. 30(2), 16–29 (2010)
10. Molka, D., Schöne, R., Hackenberg, D., Müller, M.S.: Memory performance and SPEC openMP scalability on quad-socket x86_64 systems. In: Xiang, Y., Cuzzocrea, A., Hobbs, M., Zhou, W. (eds.) ICA3PP 2011, Part I. LNCS, vol. 7016, pp. 170–181. Springer, Heidelberg (2011)
11. Pillet, V., Labarta, J., Cortes, T., Girona, S.: Paraver: A tool to visualize and analyze parallel code. WoTUG-18, 17–31 (1995)
12. Huang, L., Jin, H., Yi, L., Chapman, B.: Enabling locality-aware computations in OpenMP. Scientific Programming 18(3), 169–181 (2010)
13. Majo, Z., Gross, T.R.: Matching memory access patterns and data placement for NUMA systems. In: Proceedings of the Tenth International Symposium on Code Generation and Optimization, pp. 230–241 (2012)
14. Nikolopoulos, D.S., Papatheodorou, T.S., Polychronopoulos, C.D., Labarta, J.: Is data distribution necessary in OpenMP? In: Proceedings of the 2000 ACM/IEEE conference on Supercomputing (CDROM), p. 47 (2000)
15. Terboven, C., Schmidl, D., Jin, H., Reichstein, T.: Data and thread affinity in OpenMP programs. In: Proceedings of the 2008 Workshop on Memory Access on Future Processors: A Solved Problem?, pp. 377–384 (2008)
16. Broquedis, F., Furmento, N., Goglin, B., Namyst, R., Wacrenier, P.-A.: Dynamic task and data placement over NUMA architectures: An openMP runtime perspective. In: Müller, M.S., de Supinski, B.R., Chapman, B.M. (eds.) IWOMP 2009. LNCS, vol. 5568, pp. 79–92. Springer, Heidelberg (2009)

17. Goglin, B., Furmento, N.: Enabling high-performance memory migration for multithreaded applications on Linux. In: IEEE International Symposium on Parallel & Distributed Processing, IPDPS 2009, pp. 1–9 (2009)
18. Wittmann, M., Hager, G.: Optimizing ccNUMA locality for task-parallel execution under OpenMP and TBB on multicore-based systems. arXiv preprint arXiv:1101.0093 (2010)
19. Olivier, S.L., Porterfield, A.K., Wheeler, K.B., Spiegel, M., Prins, J.F.: OpenMP task scheduling strategies for multicore NUMA systems. International Journal of High Performance Computing Applications 26(2), 110–124 (2012)
20. Pilla, L.L., Ribeiro, C.P., Cordeiro, D., Mhaut, J.F.: Charm++ on NUMA platforms: the impact of SMP optimizations and a NUMA-aware load balancer. In: 4th workshop of the INRIA-Illinois Joint Laboratory on Petascale Computing, Urbana, IL, USA (2010)
21. Schmidl, D., Terboven, C., an Mey, D.: Towards NUMA support with distance information. In: Chapman, B.M., Gropp, W.D., Kumaran, K., Müller, M.S. (eds.) IWOMP 2011. LNCS, vol. 6665, pp. 69–79. Springer, Heidelberg (2011)

OMPT: An OpenMP Tools Application Programming Interface for Performance Analysis

Alexandre E. Eichenberger[1], John Mellor-Crummey[2], Martin Schulz[3], Michael Wong[4], Nawal Copty[5], Robert Dietrich[6], Xu Liu[2], Eugene Loh[5], and Daniel Lorenz[7]
and other members of the OpenMP Tools Working Group

[1] IBM T.J. Watson Research Center
[2] Rice University
[3] LLNL
[4] IBM Canada
[5] Oracle
[6] TU Dresden, ZIH
[7] Jülich Supercomputer Center

Abstract. A shortcoming of OpenMP standards to date is that they lack an application programming interface (API) to support construction of portable, efficient, and vendor-neutral performance tools. To address this issue, the tools working group of the OpenMP Language Committee has designed OMPT—a performance tools API for OpenMP. OMPT enables performance tools to gather useful performance information from applications with low overhead and to map this information back to a user-level view of applications. OMPT provides three principal capabilities: (1) runtime state tracking, which enables a sampling-based performance tool to understand what an application thread is doing, (2) callbacks and inquiry functions that enable sampling-based performance tools to attribute application performance to complete calling contexts, and (3) additional callback notifications that enable construction of more full-featured monitoring capabilities. The earnest hope of the tools working group is that OMPT be adopted as part of the OpenMP standard and supported by all standard-compliant OpenMP implementations.

1 Introduction

Over the last decade, multicore processors have become ubiquitous. As a result, programming models and tools that help an application developer exploit thread-level parallelism within and across multicore chips have attracted increasing interest. In this context, OpenMP [1] has emerged as a popular directive-based programming model for multithreaded parallel programming. However, constructing portable, efficient, and vendor-neutral performance tools for OpenMP has remained a challenge. To date, there has not been a standard application programming interface (API) to support tools for OpenMP. In this paper, we describe OMPT—an emerging application programming interface for OpenMP

A.P. Rendell et al. (Eds.): IWOMP 2013, LNCS 8122, pp. 171–185, 2013.

performance tools. OMPT is designed to enable performance tools to gather useful performance information from applications and to hide low-level OpenMP implementation idiosyncrasies from users. At the same time, OMPT is designed to minimize the runtime overhead for supporting performance tools. The development of OMPT was a community effort by the tools working group of the OpenMP Language Committee. The earnest hope of the tools working group is that OMPT be adopted as part of the OpenMP standard and supported by all standard-compliant OpenMP implementations.

1.1 Design Objectives

OMPT tries to satisfy several design objectives for a performance tool interface for OpenMP. These objectives are listed in decreasing order of importance.

- The API should enable tools to gather sufficient information about an application executing under control of an OpenMP runtime system to associate costs both with application source code and the runtime system.
  - The API should provide an interface sufficient to construct low-overhead performance tools based on asynchronous sampling.
  - The API should enable a profiler that uses call stack unwinding to identify which frames in its call stack correspond to routines in the OpenMP runtime system.
  - An OpenMP runtime system should associate the activity of a thread at any point in time with a *state*, which will enable a performance tool to interpret program execution behavior.
  - Certain API routines must be defined as async signal safe so that they can be invoked in a signal handler by a profiler while processing asynchronous events.
- Incorporating support for the API in an OpenMP runtime system should add negligible overhead to an OpenMP runtime system if the interface is not in use by a tool.
- The API should define support for trace-based performance tools.
- Adding the API to an OpenMP implementation must not impose an unreasonable development burden on runtime implementers.
- The API should not impose an unreasonable development burden on tool implementers.

1.2 Prior Work

The design of OMPT is based on experience with two prior efforts to define a standard tools API for OpenMP: the POMP API [2] and the Sun/Oracle Collector API [3,4]. POMP is geared toward trace-based measurement. A disadvantage of POMP's trace-based measurement is that it can lead to large measurement overhead because operations to be traced, e.g., an iteration of an OpenMP work-sharing loop, can take less time than recording an event in a trace. As an alternative to POMP's trace-based approach, the Sun/Oracle Collector API

was designed primarily to support measurement and attribution of performance information using asynchronous sampling of call stacks. This sampling-based design enables construction of tools that attribute costs to full calling contexts without the drawbacks of tracing; namely, tools can record compact profiles with low runtime overhead.

OMPT builds on ideas of both POMP and the Sun/Oracle collector API to support asynchronous sampling as well as define an optional interface to support trace-based tools. In addition, OMPT defines a series of optional callbacks to support *blame shifting* [5,6], which helps tools shift attribution of costs from symptoms to causes. The OMPT interface can be implemented entirely by the OpenMP runtime system, entirely by a compiler, or using a combination of compiler and runtime support. Using a prototype version of OMPT, Rice University's HPCToolkit performance tools have been able to provide deep insight into the performance of OpenMP programs with low runtime and space overhead [7].

1.3 OMPT Interface

To support the OMPT interface for tools, an OpenMP runtime system has two responsibilities: maintain information about the state of each OpenMP thread and provide a set of API calls that tools can use to interrogate the OpenMP runtime. Maintaining information about the state of each thread in the runtime system is not free and thus an OpenMP runtime system need not maintain state information unless a tool has registered itself, an environment variable directed the tool to track runtime state, or a debugger has demanded that runtime state information be maintained. Without any explicit request for tool support to be enabled, an OpenMP runtime need not maintain any information to support tools and may provide trivial (and thus, perhaps useless) answers to any API queries.

The API is designed so that callbacks enable a tool to keep track of various phases of an OpenMP application. Callback signatures are customized so that most information that we anticipate a tool will need is provided as callback arguments. For example, callbacks associated with a new parallel region receive information about the parallel region and the parent task creating the parallel region. Similarly, callbacks associated with lock waiting receive information about the lock awaited. If necessary, callbacks may gather additional information using API queries. Tools that gather information using asynchronous sampling rather than tracing callbacks have access to runtime information via API queries.

In addition to the callback and queries, the API enables a tool to leave "cookies" associated with threads and tasks within the internal structures of the OpenMP runtime. These cookies enable a tool to quickly retrieve its own information (e.g., pointers to tool data-structures associated with OpenMP constructs) without costly hash maps.

Most routines in the OMPT API are intended only for use by tools rather than for direct use by applications. As a result, almost all OMPT API functions have a C binding only. A Fortran binding is provided only for a few application-facing inquiry and control functions, described in Section 6.

1.4 Document Roadmap

Section 2 describes state information maintained by the OpenMP runtime system for use by tools. Section 3 describes callback events for tools supported by the OpenMP runtime system. Section 4 describes tool data structures. Section 5 describes runtime system inquiry operations for tools. Section 6 describes runtime system inquiry and control operations available to applications.

2 Runtime State

Table 1. OpenMP runtime states

/* work states (0..15) */	
`ompt_state_work_serial`	= 0x00, /* serial work */
`ompt_state_work_parallel`	= 0x01, /* parallel work */
`ompt_state_work_reduction`	= 0x02, /* performing a reduction */
/* idle (16..31) */	
`ompt_state_idle`	= 0x10, /* waiting for work */
/* overhead states (32..63) */	
`ompt_state_overhead`	= 0x20, /* non-wait overhead */
/* barrier wait states (64..79) */	
`ompt_state_wait_barrier`	= 0x40, /* waiting at any barrier */
`ompt_state_wait_barrier_explicit`	= 0x41, /* waiting at an explicit barrier */
`ompt_state_wait_barrier_implicit`	= 0x42, /* waiting at an implicit barrier */
/* task wait states (80..95) */	
`ompt_state_wait_taskwait`	= 0x50, /* waiting at a taskwait */
`ompt_state_wait_taskgroup`	= 0x51, /* waiting at a taskgroup */
/* wait states mutex (96..111) */	
`ompt_state_wait_lock`	= 0x60, /* waiting for lock */
`ompt_state_wait_nest_lock`	= 0x61, /* waiting for nest lock */
`ompt_state_wait_critical`	= 0x62, /* waiting for critical */
`ompt_state_wait_atomic`	= 0x63, /* waiting for atomic */
`ompt_state_wait_ordered`	= 0x64, /* waiting for ordered */
/* miscellaneous (112..127)*/	
`ompt_state_undefined`	= 0x70, /* undefined thread state */
`ompt_state_first`	= 0x71, /* initial enumeration state */

To enable a tool to understand what an OpenMP thread is doing, when tools support has been enabled, an OpenMP runtime will maintain state information for each OpenMP thread that may execute user code; we call such threads *OpenMP threads*. Some OpenMP runtime systems launch a helper thread to shepherd worker threads; OMPT doesn't consider such helper threads to be OpenMP threads. The state maintained for each OpenMP thread by the OpenMP runtime is an approximation of the thread's instantaneous state.

Table 1 shows the states defined by OMPT. States are divided into seven groups. A *work* state indicates that a thread is performing either serial or parallel

work. The *idle* state is typically reported by a worker thread waiting for work outside a parallel region. The *overhead* state may be reported by a thread when it is executing runtime system code. A *barrier* wait state indicates that a thread is waiting at a barrier. A *task* wait state indicates that a thread is waiting for tasks to complete. The OpenMP runtime will report the *undefined* state for any thread that is not an OpenMP thread.

Each *mutex* wait state has the obvious interpretation. When a thread is in a mutex wait state, the thread also maintains an identifier that represents the identity the mutex awaited. This capability is described in Section 4.3.

To enable low overhead implementations, OMPT does not precisely specify if or when an OpenMP runtime must report thread state transitions. For example, consider a thread acquiring a lock. One compliant runtime may transition the thread state to *lock wait* before attempting to acquire a lock. Another may transition the thread state to *lock wait* only if the thread begins to spin or block for an unavailable lock. A third compliant runtime may transition the state to *lock wait* only after a thread waits for a significant amount of time. A comprehensive discussion of OMPT states can be found in a draft OpenMP Tools API Technical Report [8].

3 Events

OMPT prescribes events for which an OpenMP runtime may provide callback notification for tools. There are two classes of events: *mandatory events* and *optional events*. Mandatory events must be implemented in any compliant runtime implementation. Optional events are grouped in sets of related events. While each event can be individually included or omitted, we encourage runtimes and tools to consider implementing all or none of the events in a given set. A callback need not be registered for an event. An OpenMP runtime system will not make any callback unless a tool has registered to receive it. Each callback has a specified type signature.

Mandatory Events. Figure 1 lists the mandatory events for which an OpenMP runtime system must provide callback notification. Mandatory events include creation and destruction of worker threads, entry and exit from a parallel region, creation and destruction of explicit tasks, and notification of when an OpenMP runtime is shutting down. Finally, the *control* event, inspired by `MPI_pcontrol` enables an application to directly control a

```
ompt_event_thread_create
ompt_event_thread_exit
ompt_event_parallel_create
ompt_event_parallel_exit
ompt_event_task_create
ompt_event_task_exit
ompt_event_runtime_shutdown
ompt_event_control
```

Fig. 1. Mandatory OMPT events

tool. If the user program calls `ompt_control`, the OpenMP runtime invokes the control callback. The callback executes in the environment of the user control call. The arguments passed to the callback are the ones passed by the user to `ompt_control`. Details about the type signatures for other callbacks are available in the draft OpenMP Tools Technical report [8].

3.1 Optional Events

This section describes two sets of events. One set of events is used by sampling-based performance tools that employ a strategy known as *blame shifting* to attribute waiting by one or more threads to activity by contexts that cause other threads to wait, e.g., a thread holding a lock, rather than the contexts in which the waiting is observed, e.g., threads waiting to acquire a lock. Support for any events described in this section is optional for a compliant runtime system.

Events for Blame Shifting. Figure 2 lists synchronous 'blame shifting' events. These events enable sampling-based performance tools to transfer blame for waiting from contexts where waiting is observed to code responsible for the waiting.[1] A tool can use these callbacks to blame time worker threads idle between parallel regions on the serial code executing. Similarly, using these callbacks, a tool can attribute time spent waiting at barriers, waiting for tasks to complete, or waiting for a thread to release a construct that provides mutual exclusion (e.g., a lock, critical section, atomic, region, or ordered section) on the thread or threads responsible for the waiting.

```
ompt_event_idle_begin
ompt_event_idle_end
ompt_event_wait_barrier_begin
ompt_event_wait_barrier_end
ompt_event_wait_taskwait_begin
ompt_event_wait_taskwait_end
ompt_event_wait_taskgroup_begin
ompt_event_wait_taskgroup_end
ompt_event_release_lock
ompt_event_release_nest_lock_last
ompt_event_release_critical
ompt_event_release_atomic
ompt_event_release_ordered
```

Fig. 2. Optional events for blame shifting

Events for Trace-Based Measurement Tools. Figure 3 lists optional events to support trace-based measurement tools. Most of these callback events consist of start/end pairs that bracket an activity. A few events, such as lock initialization, lock destruction, and flush merit only a single event. Additional details about these events and the type signatures of their callbacks can be found in the draft OpenMP Tools Technical report [8].

[1] Blame shifting is effective for attributing idle time while threads await work [5] and idle time while threads spin wait for a lock [6]. The utility of such blame shifting has also been demonstrated for performance analysis of OpenMP programs [7].

ompt_event_implicit_task_create	ompt_event_taskgroup_end
ompt_event_implicit_task_exit	ompt_event_release_nest_lock_prev
ompt_event_task_switch	ompt_event_wait_lock
ompt_event_loop_begin	ompt_event_wait_nest_lock
ompt_event_loop_end	ompt_event_wait_critical
ompt_event_section_begin	ompt_event_wait_atomic
ompt_event_section_end	ompt_event_wait_ordered
ompt_event_single_in_block_begin	ompt_event_acquired_lock
ompt_event_single_in_block_end	ompt_event_acquired_nest_lock_first
ompt_event_single_others_begin	ompt_event_acquired_nest_lock_next
ompt_event_single_others_end	ompt_event_acquired_critical
ompt_event_master_begin	ompt_event_acquired_atomic
ompt_event_master_end	ompt_event_acquired_ordered
ompt_event_barrier_begin	ompt_event_init_lock
ompt_event_barrier_end	ompt_event_init_nest_lock
ompt_event_taskwait_begin	ompt_event_destroy_lock
ompt_event_taskwait_end	ompt_event_destroy_nest_lock
ompt_event_taskgroup_begin	ompt_event_flush

Fig. 3. Optional events for tracing

4 Tool Data Structures

4.1 Thread and Task Data

Each OpenMP thread and task instance provides an `ompt_data_t` object, which is a union of an integer and a pointer:

```
typedef union ompt_data_u {
    uint64_t value;    /* data under tool control    */
    void *ptr;         /* pointer under tool control */
} ompt_data_t;
```

The lifetime of an `ompt_data_t` object for a thread or task instance begins when the instance is created and ends when the instance is destroyed. While the value of an `ompt_data_t` is preserved throughout the life of its associated thread or task instance, tools should not assume that the address of an `ompt_data_t` remains constant over its lifetime.

When a thread or task instance is created, the callback associated with the instance creation event must initialize the `ompt_data_t` object. If there is no callback associated with the event, the OpenMP runtime initializes the structure value field to 0. The address of the `ompt_data_t` structure is passed to callbacks associated with the creation/destruction of threads/tasks. The address of the structure can also be retrieved on demand, e.g., by invoking an inquiry function in a signal handler.

If the `ompt_data_t` value field is 0 for a thread or task instance at the point that an exit callback would be made, the exit callback is not invoked. The tool is responsible for coordinating any concurrent accesses to `ompt_data_t` objects.

4.2 Parallel Region Identifier

Each OpenMP parallel region instance has `ompt_parallel_id_t` that uniquely identifies the region instance. An `ompt_parallel_id_t` is represented as a 64-bit unsigned integer. The `ompt_parallel_id_t` for a parallel region instance is unique across all instances of all parallel regions.

An `ompt_parallel_id_t` is defined when a parallel region instance is created and passed to callbacks associated with creation/destruction of the parallel region instance. A parallel region's ID can be retrieved on demand, e.g., by invoking an inquiry function in a signal handler. Tools should not assume that `ompt_parallel_id_t` values for adjacent region instances are consecutive.

4.3 Wait Identifier

When a thread enters a mutex wait state, it assigns a unsigned 64-bit integer `ompt_wait_id_t` that indicates a program variable, lock, or internal runtime object associated with a critical section, ordered section, or atomic region that caused the thread to wait. The value of the `ompt_wait_id_t` structure is passed to callbacks associated with wait events, and can also be retrieved on demand by invoking the `ompt_get_state` operation, described in Section 5.1. The value of a thread's `ompt_wait_id_t` is undefined if the thread is not awaiting a mutex.

4.4 Pointers to Support Classification of Stack Frames

Each implicit or explicit task instance provides an `ompt_frame_t` data structure which contains pointers to OpenMP runtime procedure frames that appear above and below procedure frames associated with user task code.

```
typedef struct ompt_frame_s {
    void *exit_runtime_frame;    /* next frame is user code */
    void *reenter_runtime_frame; /* prev frame is user code */
} ompt_frame_t;
```

The structure's lifetime begins when a task instance is created and ends when the task instance is destroyed. While the value of the structure is preserved over the lifetime of the task, tools should not assume that the address of a structure remains constant over its lifetime. Frame data is passed to some callbacks; it can also be retrieved for a task (e.g. by a signal handler). Frame data contains two components:

`exit_runtime_frame` This value is set when a task exits the runtime to begin executing user code. This field points to the stack frame of the runtime procedure that called the user code. This value is NULL until just before the task exits the runtime.

`reenter_runtime_frame` This value is set when the current task re-enters the runtime to create new (implicit or explicit) tasks. This field points to the stack frame of the runtime procedure called by a task to re-enter the runtime. This value is NULL until just after the task re-enters the runtime.

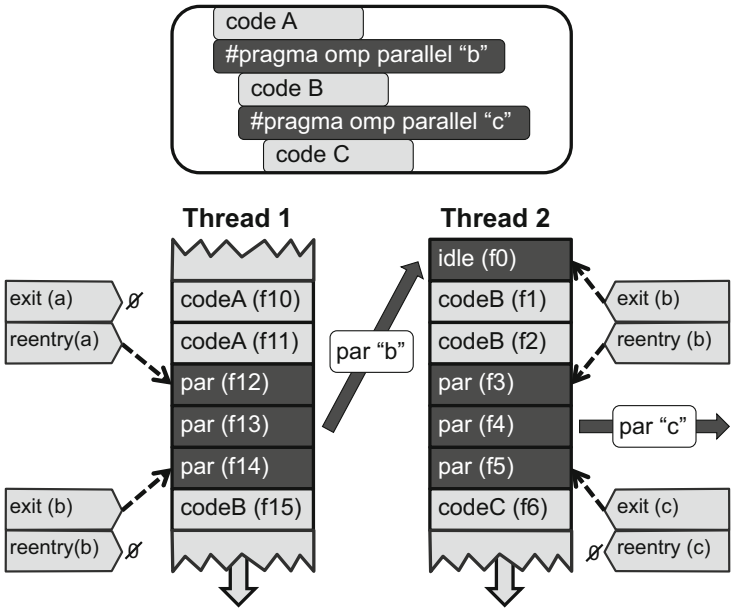


Fig. 4. Frame information

Figure 4 illustrates a program executing a nested parallel region, where code A, B, and C represent, respectively, code associated with the sequential, outer-parallel, and inner-parallel regions. Figure 4 also depicts the stacks of two threads, where each new function call installs a new stack frame below the previous frames. When Thread 1 encounters the outer-parallel region (parallel "b"), it jumps into the runtime functions responsible for creating a new parallel region. This functionality is performed here by 3 consecutive function calls labeled "par" and frames f12 to f14. The OMPT API guarantees that the `reenter_runtime_frame` field of the parent task will points to the first frame of the runtime upon reentering the runtime (f12 here).

Before starting the parallel work, the runtime instantiates the `exit_runtime_frame` to the last runtime frame for each of the implicit tasks created by the parallel region: f14 for Thread 1 and f0 for Thread 2 here. Let us focus now on Thread 2, which is created by the runtime to execute the outer-parallel region "b". This thread first exits a runtime function, labeled "idle" here, which is responsible for initializing a worker thread prior to executing user work. Before exiting the runtime, the runtime instantiates the `exit_runtime_frame` field to Frame f0. When Thread 2 later encounters the inner-parallel region "c", the execution returns to the runtime where the runtime will instantiate the `reentry_runtime_frame` field of the parent task to Frame f3.

Table 2. Meaning of values for `exit_runtime_frame` and `reenter_runtime_frame`

exit / reentry	reentry = null	reentry = defined
exit = null	case 1) initial task in user code case 2) explicit task that is created but not yet scheduled	initial task in runtime because of a parallel region or a task creation
exit = defined	non-initial task in user code	non-initial task in runtime because of a parallel region or a task creation

The function stack frames introduced by the OpenMP runtime can thus be elided by walking back the stack and eliminating every frames between the `exit_runtime_frame` of a child's task and the `reenter_runtime_frame` of its parent's task. When reaching the first frame of a stack, for example, when walking back the stack of Thread 2, a tool can resume the walking in the parent's stack starting above the `reentry_runtime_frame` field associated with its parent's task.

As a reference for tool designers, Table 2 enumerates the meaning of possible NULL or non-NULL entries in exit / re-enter pairs. In the presence of nested parallelism, a tool may observe a sequence of pairs for a thread. The pairing between reenter and exit events is worth noting. A exit event in an `ompt_frame_t` at level k always pairs with the reenter event in the frame at level $k + 1$. Tools must be prepared to observe frame exit and reenter values that have not yet been set or reset as the program exits the runtime or returns into the runtime. In such cases, an exit or reenter pointer may point above the frame at the top of the thread's stack, or the `ompt_frame_t` at level 0 may contain only NULL pointers.

5 Inquiry Functions for Tools

Inquiry functions retrieve data from the execution environment for the tools. All inquiry functions are async signal safe.

5.1 Enumerate States Supported by an OpenMP Runtime

An OpenMP runtime system is allowed to support other states in addition to those described herein. For instance, a particular runtime system may want to provide finer-grain information about the nature of runtime overhead, e.g., to differentiate between the overhead associated with setting up a parallel region and the overhead associated with setting up a task. Further, a tool may not report all states defined herein, e.g., if state tracking for a particular state would be too expensive. To enable a tool to identify all states that an OpenMP runtime

system implements, OMPT provides the following interface for enumerating all possibly reported runtime states.

```
int ompt_enumerate_state(int current_state,
 int *next_state, const char **next_state_name);
```

When this interface is invoked for the first time, the value `ompt_state_first` should be supplied for `current_state`. The argument `next_state` is a pointer to an integer that will be set to the code for the next state in the enumeration. The argument `next_state_name` is a pointer to a location that will be filled in with a pointer to the name associated with `next_state`. Subsequent invocations of `ompt_enumerate_state` should pass the code returned in `next_state` by the prior call. The enumeration is complete when `ompt_enumerate_state` returns 0. The canonical way to enumerate the states supported by an OpenMP runtime system is shown below:

```
int state; const char *state_name;
for (int ok = ompt_enumerate_state(ompt_state_first, &state,
  &state_name); ok; ok = ompt_enumerate_state(state, &state,
  &state_name))
   { /* "state" is supported with name "state_name" */ }
```

Thread Data Inquiry. Function `ompt_get_thread_data` is an inquiry function to access data stored by the OpenMP runtime system for the current thread for use by a tool.

```
ompt_data_t *ompt_get_thread_data(void);
```

This inquiry function returns NULL prior to OpenMP initialization or when no tool is attached to the runtime. This function is async signal safe.

Thread State Inquiry. Function `ompt_get_state` is the inquiry function to determine the state of the current thread.

```
ompt_state_t ompt_get_state(ompt_wait_id_t *wait_id);
```

The location specified by `wait_id` is updated point to the wait identifier associated with the current state, if any, or NULL otherwise. This function returns `ompt_state_undefined` prior to OpenMP initialization or when no tool is attached to the runtime. This function is async signal safe.

Parallel Region Inquiry. OMPT defines two inquiry functions to access data stored by the OpenMP runtime for parallel regions. The first region inquiry function returns the unique parallel id associated with an enclosing parallel region instance:

```
ompt_parallel_id_t ompt_get_parallel_id(int ancestor_level);
```

Outside a parallel region or in the idle state, `ompt_get_parallel_id` returns 0. In other cases, except as we discuss below, the thread should return the identity of the enclosing parallel region at the requested level.

The second region inquiry function returns a pointer to a compiler-generated function invoked by the OpenMP runtime to encapsulate the code of the parallel region, if any, and NULL otherwise:

```
void *ompt_get_parallel_function(int ancestor_level);
```

Both of parallel region inquiry functions take an ancestor level as an argument. By specifying different values for ancestor level (0 is current, 1 is parent, 2 is grandparent...), one can access information about each parallel region, even if parallel regions are nested. These functions return the value 0 when requesting higher levels of ancestry than available, prior to OpenMP initialization, or when no tool is attached to the OpenMP runtime. These functions are async signal safe.

Task Region Inquiry. OMPT defines three inquiry functions to access data stored by the OpenMP runtime for task regions. Function `ompt_get_task_data` returns the tool data object associated with a given task. Similarly, function `ompt_get_task_frame` returns the tool frame associated with a given task. Frame objects for tasks are used to support classification of call stack frames as being associated with user code or the OpenMP runtime system, as described in Section 4.4.

```
ompt_data_t *ompt_get_task_data(int ancestor_level);

ompt_frame_t *ompt_get_task_frame(int ancestor_level);
```

The value returned by the `ompt_get_task_function` indicates the compiler-generated function used by the OpenMP runtime to encapsulate the code of the task construct, if any, and NULL otherwise.

```
void *ompt_get_task_function(int ancestor_level);
```

These functions return NULL when requesting higher levels of ancestry than available, prior to OpenMP initialization, or when no tool is attached to the OpenMP runtime. These functions are async signal safe.

Tool Support Version Inquiry. OMPT provides a function to determine the version of the OMPT interface supported by a runtime:

```
int ompt_get_ompt_version(void);
```

The version of OMPT described by this document is known as version 1.

6 Inquiry and Control Functions for Applications

The functions described in this section are the only ones with a Fortran interface in addition to a C/C++ interface.

Runtime Version Inquiry. OMPT provides an interface to extract information about the version of an OpenMP runtime system in the form of a string:

```
int ompt_get_runtime_version(char *buffer, int length);
```

`ompt_get_runtime_version` fills `buffer` with a version-specific string of at most `length` characters. The suggested format is

```
<vendor>-<major version>.<minor version>[-<optional feature]*
```

Namely, a vendor name, major and minor version numbers, and, optionally, a list of zero or more features, separated by dashes. As an example, IBM's OpenMP runtime might return the following version string " IBM-1.1-blame=1-trace=0", indicating that IBM's OpenMP runtime supports the OMPT tools API core augmented with support for blame shifting, but not support for detailed tracing.

Tool Control. The function `ompt_control` can be called by an application to pass control information to a tool. The signature for this function is shown below:

```
void ompt_control(uint64_t command, uint64_t modifier);
```

A classic use case for the `ompt_control` routine might be for an application to start and stop data collection by a tool.

7 Initializing OMPT Support for Tools

An OpenMP runtime need not maintain information to support tools and may provide trivial (and thus, perhaps useless) answers in response to invocations of any API inquiry functions. Section 7.1 describes normal initialization for a tool. A further section describing the environment variable control over tool initialization and the tool initialization API for debugger can be found in the draft OpenMP Tools Technical Report [8].

7.1 Initialization of a Tool

A tool must register itself with an OpenMP runtime system and then specify callbacks for events of interest. Section 7.1 describes the initializer for a tool. Section 7.1 describes registration of callbacks for OMPT events.

Tool initializer. A tool must register itself with an OpenMP runtime by defining the following function:

```
int ompt_initialize(void);
```

The role of `ompt_initialize` is to register callbacks for specific events, e.g., creating a parallel region. A tool must register a callback for every event of interest using `ompt_set_callback`, as described in Section 7.1. The OpenMP runtime system defines a weak symbol version of `ompt_initialize` that returns 0; a tool-provided version must return 1.

Table 3. Meaning of return codes for `ompt_set_callback`

return code	meaning
0	event may occur; runtimes does not support this callback.
1	event will never occur in runtime.
2	event may occur; callback invoked when convenient.
3	event may occur; callback always invoked when event occurs.

Since only one tool-provided definition of `ompt_initialize` will be invoked by an OpenMP runtime, only one tool can be registered. Ordinarily, the tool initializer `ompt_initialize` will be invoked by an OpenMP runtime immediately after the runtime initializes itself.

An OpenMP runtime system *may* allow registration of a tool after initialization of the OpenMP runtime at a *clean point*. An OpenMP runtime is said to be at a clean point when no pthread is inside a parallel region. An OpenMP runtime system will not necessarily attempt to register a tool at a clean point unless a debugger has previously called `ompd_enable(true)`as described in the draft OpenMP Tools Technical Report [8].

After a process fork, if OpenMP is re-initialized in the child process, the OpenMP runtime system in the child process will call `ompt_initialize` under the same conditions as it would in any process.

Callback Registration. Tools register callbacks to receive notification of various events that occur as an OpenMP program executes. To register callbacks, a tool uses the following function:

```
int ompt_set_callback(ompt_event_t e, ompt_callback_t cb);
```

The function `ompt_set_callback` may only be called within the implementation of `ompt_initialize` provided by a tool, as described above. The possible return codes for `ompt_set_callback` and their meaning is shown in Table 3. Registration of supported callbacks may fail if this function is called outside `ompt_initialize`. The `ompt_callback_t` type for a callback does not reflect the actual signature of the callback; OMPT uses this generic type to avoid the need to declare a separate registration function for each actual callback type.

The function `ompt_get_callback`, as shown below, may be called at any time to inspect whether a callback has been registered or not. If a callback has been registered, `ompt_set_callback` will return 1 and set `callback` to the address of the callback function; otherwise, `ompt_set_callback` will return 0.

```
int ompt_get_callback(ompt_event_t e, ompt_callback_t *cb);
```

8 Status

This paper defines an application programming interface (API) for tools that we propose for adoption as part of the OpenMP standard and supported by all OpenMP compliant implementation. An initial implementation of OMPT in

IBM's lightweight OpenMP runtime system has been completed and tested with tools. An implementation of OMPT in Intel's open-source OpenMP Runtime Library by Rice University and the University of Oregon is nearing completion. The OMPT implementation in IBM's runtime has runtime overhead of less than 1% for OMPT.

In the spring of 2013, there has been an effort to extend the OMPT OpenMP tools API to provide an interface for debuggers as well. The design for a debugger interface, known as OMPD, embraces the OMPT design and supports a superset of OMPT interface operations in a shared library intended to be loaded into a debugger interacting with the state of an OpenMP application in a core file or a live process. The design for OMPD builds upon an earlier effort to design a debugger library [9].

A more detailed description of OMPT and the proposed OMPD extensions are part of a draft OpenMP Tools API Technical Report [8].

References

1. OpenMP Architecture Review Board: OpenMP Application Programming Interface, version 3.1. (July 2011), http://www.openmp.org/mp-documents/OpenMP3.1.pdf
2. Mohr, B., Malony, A.D., Shende, S., Wolf, F.: Design and prototype of a performance tool interface for OpenMP. The Journal of Supercomputing 23, 105–128 (2002)
3. Itzkowitz, M., Mazurov, O., Copty, N., Lin, Y.: An OpenMP runtime API for profiling Sun Microsystems, Inc. OpenMP ARB White Paper, http://www.compunity.org/futures/omp-api.html
4. Jost, G., Mazurov, O., An Mey, D.: Adding new dimensions to performance analysis through user-defined objects. In: Mueller, M.S., Chapman, B.M., de Supinski, B.R., Malony, A.D., Voss, M. (eds.) IWOMP 2005 and IWOMP 2006. LNCS, vol. 4315, pp. 255–266. Springer, Heidelberg (2008)
5. Tallent, N.R., Mellor-Crummey, J.M.: Effective performance measurement and analysis of multithreaded applications. In: Proceedings of the 14th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP 2009, pp. 229–240. ACM, New York (2009)
6. Tallent, N.R., Mellor-Crummey, J.M., Porterfield, A.: Analyzing lock contention in multithreaded applications. In: Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP 2010, pp. 269–280. ACM, New York (2010)
7. Liu, X., Mellor-Crummey, J., Fagan, M.: A new approach for performance analysis of OpenMP programs. In: Proceedings of the 27th International ACM Conference on International Conference on Supercomputing. ICS 2013, pp. 69–80. ACM, New York (2013)
8. Eichenberger, A., Mellor-Crummey, J., Schulz, M., Copty, N., DelSignore, J., Dietrich, R., Liu, X., Loh, E., Lorenz, D.: OMPT: An openMP tools application programming interface for performance analysis. In: Rendell, A.P., Chapman, B.M., Müller, M.S. (eds.) IWOMP 2013. LNCS, vol. 8122, pp. 171–185. Springer, Heidelberg (2013), http://openmp.org/mp-documents/ompt-tr.pdf
9. Cownie, J., Del Signore, J., de Supinski, B.R., Warren, K.: DMPL: An openMP DLL debugging interface. In: Voss, M.J. (ed.) WOMPAT 2003. LNCS, vol. 2716, pp. 137–146. Springer, Heidelberg (2003)

Open Source Task Profiling by Extending the OpenMP Runtime API

Ahmad Qawasmeh[1], Abid Malik[1], Barbara Chapman[1], Kevin Huck[2], and Allen Malony[2]

[1] University of Houston, Dept. of Computer Science, Houston, Texas
{arqawasm,malik,chapman}@cs.uh.edu
www2.cs.uh.edu/~hpctools
[2] University of Oregon, Dept. of Computer and Information Science, Eugene, Oregon
{khuck,malony}@cs.uoregon.edu
www.cs.uoregon.edu/research/tau/home.php

Abstract. The introduction of tasks in the OpenMP programming model brings a new level of parallelism. This also creates new challenges with respect to its meanings and applicability through an event-based performance profiling. The OpenMP Architecture Review Board (ARB) has approved an interface specification known as the "OpenMP Runtime API for Profiling" to enable performance tools to collect performance data for OpenMP programs. In this paper, we propose new extensions to the OpenMP Runtime API for profiling task level parallelism. We present an efficient method to distinguish individual task instances in order to track their associated events at the micro level. We implement the proposed extensions in the OpenUH compiler which is an open-source OpenMP compiler. With negligible overheads, we are able to capture important events like task creation, execution, suspension, and exiting. These events help in identifying overheads associated with the OpenMP tasking model, e.g., task waiting until a task starts execution or task cleanup etc. These events also help in constructing important parent-child relationships that define tasks' call paths. The proposed extensions are in line with the newest specifications recently proposed by the OpenMP tools committee for task profiling.

Keywords: OpenMP, OpenMP Runtime API for Profiling, Open-Source Implementation, OpenMP Tasks.

1 Introduction

OpenMP is a standard API for shared memory programming. It provides a directive-based programming approach for generating parallel versions of programs from the sequential ones. The compiler generated code invokes the OpenMP runtime library routines to create and manage threads and tasks. The lack of standards in the runtime layer has hampered the development of third-party tools to

A.P. Rendell et al. (Eds.): IWOMP 2013, LNCS 8122, pp. 186–199, 2013.

support OpenMP application development. The OpenMP Runtime API (ORA) for profiling OpenMP applications was presented in [9]. The ORA has been accepted by the tools committee of the OpenMP Architecture Review Board (ARB). The API is designed to permit a tool, known as a collector, to gather information about an OpenMP program from the runtime system in such a manner that neither the collector nor the runtime system needs to know any details about each other. The ORA is designed to ensure that tool developers are not required to have an insight into the details of the different OpenMP implementations. However, these tools should maintain information about the OpenMP execution model in order to trouble-shoot the OpenMP specific performance problems.

The ORA is an event-based interface that relies on bi-directional communications between a performance tool, i.e. collector, and an OpenMP runtime library. The communication is established through a collection of requests that take a send-receive protocol with a distinct functionality for each request. The main advantage of the ORA is that no modifications to an application's source code are required. Consequently, compiler analysis and optimizations will not be affected. Hence, the performance measurements of an application, collected by a performance tool, are more accurate and specific.

Traditionally, parallel programming models for shared memory multiprocessors had focused on scientific applications using large arrays and exhibiting loop level parallelism. In order to exploit the new massive parallelism provided by the modern architectures, the parallel programming models had to propose a new dimension of concurrency to cap available parallelism within high performance computing applications. Applications exhibiting irregular parallelism in the form of recursive algorithms and pointer based data structures were not taken care of before the introduction of tasking in the OpenMP programming model. Tasking has added a new dimension of concurrency, represented by the task construct, to OpenMP applications. The task construct allows a developer to dynamically create asynchronous units of work to be scheduled at runtime. Two types of tasks have been introduced in the OpenMP specification; **1) Tied tasks 2) Untied tasks**. Tied tasks can be suspended at specific scheduling points that include the creation of tasks, taskwait constructs, barriers, and completion of tasks etc. Untied tasks can be suspended at any point in an OpenMP program according to OpenMP 3.1 specifications. Moreover, a tied task can be resumed only by the thread that started its execution while an untied task can be resumed by any thread in the team.

In order to handle the challenges and performance issues associated with the introduction of tasks, we propose new extensions to the ORA in the OpenMP runtime library of the OpenUH compiler [17], [2]. The main motivation behind this work lies in observing the viability of monitoring the individual task instances and tracking the events associated with each one of them at the micro level. The new API we propose allows developers to:

– Distinguish the individual task instances in the same task construct by assigning a distinct ID to each task instance.

- Distinguish the task instances that get suspended at the different scheduling points.
- Construct parent-child relationships between tasks, which can be used to construct a task tree.
- Track task creation, switching, suspension, resumption, exiting, and completion.
- Allow collector tools to maintain performance measurements associated with the aforementioned events.

Our implementation supports C/C++ and Fortran programs with tied tasks and untied tasks. Furthermore, our implementation is in line with the task profiling specification proposed by the OpenMP ARB tools committee [4]. Reference implementations of the ORA are sparse since it requires compiler and OpenMP runtime library support. To the best of our knowledge, the work reported here is the first open-source implementation of the ORA with extensions to support tasks.

The remainder of the paper is organized as follows. Section 2 briefly describes the OpenMP task implementation in the OpenUH compiler runtime library. Section 3 gives details about our new extensions in the ORA for task profiling. Section 4 presents the experimental framework used to evaluate the implementation. The related work is discussed in Section 5. Finally, Section 6 concludes our contributions and discusses directions for the future work.

2 OpenMP Tasking Implementation in OpenUH

The OpenUH compiler supports OpenMP 3.0 tasking on the IA-64, IA-32, x86_64, and Opteron Linux ABI platforms. This includes the front-end support ported from the GNU C/C++ compiler, back-end translation implemented by the HPCTool group at the University of Houston jointly with the Tsinghua University, and an efficient task scheduling infrastructure developed by the HPCTools group. The HPCTools group also implemented a configurable task pool framework that allows a user to choose an appropriate task queue organization at runtime.

We use the popular Fibonacci code in Figure 1a to explain the role of the OpenMP runtime library regarding our implementation. In the code, two task constructs have been inserted to handle recursion in a dynamic parallel fashion. In the same manner, a taskwait construct has been used to get correct results by preventing parent tasks from proceeding while child tasks are still running. The number of tasks created depends on the value of integer n. The task construct will create a task instance. The execution of the task will be deferred based on the availability of threads and the status of its children. Figure 2 shows how the OpenMP tasking directives in Figure 1a are translated into the OpenMP runtime routines. A description of the tasking runtime routines is given in Table 1.

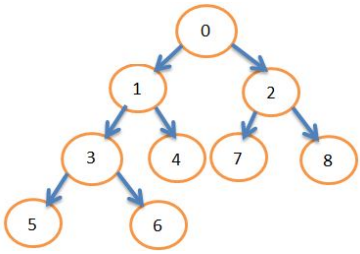
We use the OpenMP runtime routines to capture the OpenMP events and states related to the OpenMP task, such as task creation, task waiting in the task

Table 1. Description of the OpenMP tasking runtime routines in OpenUH

Routine	Description
__ompc_task_create()	creates a task and inserts it into a queue
__ompc_task_wait()	suspends a task until all of its children complete
__ompc_task_exit()	called at the end of a task to perform cleanup and schedule a new task
__ompc_task_switch()	switches the execution from one task to another
__ompc_task_firstprivates_alloc()	allocate memory for firstprivate copies
__ompc_task_will_defer()	checks if a task should be deferred or executed immediately
__ompc_task_firstprivates_free()	deallocate memory for firstprivate copies

```
int fib(int n)
{
  int x,y;
  if (n<2)
    return n;
#pragma omp task shared(x)
  x=fib(n-1);
#pragma omp task shared(y)
  y=fib(n-2);
#pragma omp taskwait
  return x+y;
}
```

(a) Fibonacci code



(b) A task tree (n=4)

Fig. 1. Fibonacci OpenMP tasking example

pool, task switching from a create state to a suspend state etc. These states and events are captured by simply modifying the OpenMP runtime routines, without modifying the OpenMP translation of the source code. The ORA extensions to support task profiling provide an API to query the OpenMP runtime library for task states and event notifications using callback functions.

3 Implementation of the OpenMP Tasking Profiling APIs

The ORA interface [9] consists of a single routine that takes the form: *int __omp_collector_api (void *arg)*. The **arg** parameter is a pointer to a byte array that can be used by a collector tool to pass one or more requests for information from the runtime. The collector requests notification of a specific event by passing the name of the event to be tracked as well as a callback routine to be invoked by the OpenMP runtime each time the event occurs. Figure 4 demonstrates the interaction between the collector and the OpenMP runtime library through the Collector API. As shown, this interaction is acheived by a set of implemented requests.

The aforementioned single routine is implemented once in the runtime and its symbol is exported in the OpenMP runtime library. This strategy allows the tool

```
extern _INT32 fib(_INT32 n)
{
  register _INT32 _w2c___ompv_task_is_deferred;
  register _UINT64 _w2c_reg3;
  register _INT32 _w2c___ompv_task_is_deferred0;
  _INT32 x;
  _INT32 y;
  struct task_args__omprg_fib_1 * __ompv_taskarg1;
  struct task_args__omprg_fib_2 * __ompv_taskarg2;
  if(n <= 1)
  {
    return n;
  }
  _w2c___ompv_task_is_deferred = __ompc_task_will_defer(1);
  if(_w2c___ompv_task_is_deferred)
  {
    __ompc_task_firstprivates_alloc(&__ompv_taskarg1, 4);
    (__ompv_taskarg1) -> n = n;
  }
  else  { __ompv_taskarg1 = (struct task_args__omprg_fib_1 *)(0ULL);}
  __ompc_task_create(&__omprg_fib_1, _w2c_reg3, __ompv_taskarg1, 1, 1, 0);
  _w2c___ompv_task_is_deferred0 = __ompc_task_will_defer(1);
  if(_w2c___ompv_task_is_deferred0)
  {
    __ompc_task_firstprivates_alloc(&__ompv_taskarg2, 4);
    (__ompv_taskarg2) -> n = n;
  }
  else  {__ompv_taskarg2 = (struct task_args__omprg_fib_2 *)(0ULL);}
  __ompc_task_create(&__omprg_fib_2, _w2c_reg3, __ompv_taskarg2, 1, 1, 0);
  __ompc_task_wait();
  return x + y;
}
```

Fig. 2. OpenUH translation of the Fibonacci code in Fig. 1a into explicitly multi-threaded code

to check whether the symbol exists via a dynamic linker in order to establish a communication with the runtime and start sending requests and monitoring events and states. The OpenMP runtime should distinguish thread's states, which are related to tasks. These states include when a task is created, suspended, existing, or being executed. When the collector tool makes a request for notification of a specified task event(s), the OpenMP runtime will start keeping track of this event inside its environment. The collector may also make requests to pause, resume, or stop event generation.

When a collector tool sends a request to register any event through the ORA, the event type *OMP_COLLECTORAPI_REQUEST* and a callback function pointer is passed as an argument to the API call in the runtime. Race conditions might occur when multiple threads try to register the same event with multiple callbacks. The callback function pointer is stored in a table in which each entry has a lock associated with it to prevent race conditions. This table contains the event callbacks shared by all the threads. The frequency in which the events are registered relies on the nature of the collector tool. Two functions, *__ompc_event_callback(event)* and *__ompc_set_state(state)*, are inserted at different positions in the OpenMP runtime task routines specifed in Table 1. These functions implement the different events and states associated with

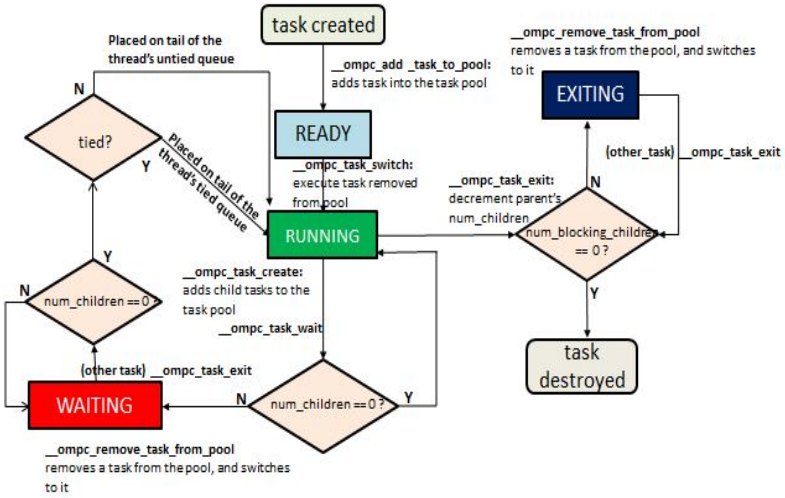


Fig. 3. OpenUH tasking execution model

the task instances. The state values are stored in the OpenMP thread descriptor in the runtime. Once a thread reaches an event point, the function *__ompc_event_callback((OMP_COLLECTORAPI_EVENT) e)* is executed and the callback function, associated with this event, is invoked. The functionality of the callback is determined by a performance tool in order to collect the required performance measurements.

The OpenMP ARB tools committee proposed a framework for task profiling in the face to face meeting recently held at the University of Houston [4]. The proposal categorizes the profiling events into two groups 1) mandatory events 2) optional events. The proposal defines the OpenMP task creation and task exiting as mandatory events, while task waiting and task switching have been defined as optional events. Our extensions include support for the mandatory events proposed during this meeting. We also provide support for some optional events in addition to some other events specific to our tasking model. Figure 3 shows the task execution model implemented in the OpenUH runtime to support OpenMP tasks. The model depicts all the different states encountered by each task instance starting from task's creation to its completion ,i.e., when the task is destroyed. In the following sections, we describe our extensions in detail.

3.1 Task Creation Events and States

These events and states are designed to capture the start and completion of a task instance creation. The following states and events have been defined:

– *THR_TASK_CREATE_STATE*: The enumerated value of this state is assigned to the thread's state field in the descriptor whenever the thread is working on a task creation.

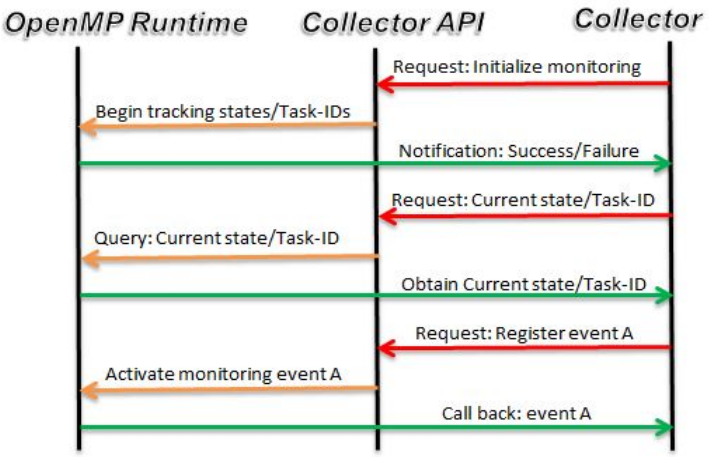


Fig. 4. Example of an interaction between collector and OpenMP runtime

- *OMP_EVENT_THR_BEGIN_CREATE_TASK*: This event indicates that the parent task creates a new explicit task before the new task starts execution.
- *OMP_EVENT_THR_END_CREATE_TASK_IMM*: This event indicates that the process of creating the task is done and its execution will start immediately.
- *OMP_EVENT_THR_END_CREATE_TASK_DEL*: This event indicates that the process of creating the task is done and its execution will start with a delay.

3.2 Task Suspension Events and States

These events and states are designed to capture the start and completion of a task instance suspension. This suspension occurs when the taskwait construct is encountered.

- *THR_TASK_SUSPEND_STATE*: The enumerated value of this state is instantly assigned to the thread's descriptor once the parent task encounters a taskwait construct. The thread, working on the parent task, will later be asssigned to another work. Child tasks, associated with the suspended task, should finish their execution in order for the suspended task to resume its execution.
- *OMP_EVENT_THR_BEGIN_SUSPEND_TASK*: This event indicates that the parent task has been suspended.
- *OMP_EVENT_THR_END_SUSPEND_TASK*: This event indicates the completion of the parent task's suspension.

3.3 Task Execution/Exiting Events and States

These events and states are designed to capture the start and completion of a task instance execution and exiting. Once a new task is created, it may start executing immediately or with some delay depending on the availability of threads.

- *THR_WORK_STATE*: The enumerated value of this state is assigned to the thread's descriptor once the thread starts the execution of a task.
- *OMP_EVENT_THR_BEGIN_EXEC_TASK*: This event is hit once the task's execution begins.
- *OMP_EVENT_THR_BEGIN_FINISH_TASK*: This event indicates that the task's execution is done. This event is hit immediately after the previous event if the task being executed does not encounter a taskwait construct.
- *OMP_EVENT_THR_END_FINISH_TASK*: This event indicates that the removal of the task from the task-pool and the required cleanup have successfully been completed.

By defining these new states, we guarantee that a thread will always have a distinct state associated with it while working on tasks. The collector tool can request the state of a thread at any given point during the execution of the program.

3.4 Task IDs and Parent Task IDs

In order to distinguish the various task instances, keep track of their associated events, and construct parent-child relationships between tasks, we have added a new OpenMP task ID field to the task data structure descriptor. It is initialized with a value corresponding to the initial implicit task. Each time a new task is created, the task ID is incremented atomically to ensure that only one thread can modify this field at any instance of time. The parent task ID is obtained by having a pointer to the parent task. Two requests are defined to enable the collector tool to obtain these IDs at any given point of the program execution.

4 Evaluation

We evaluated our implementation in the OpenUH compiler. We performed the following two analyses;

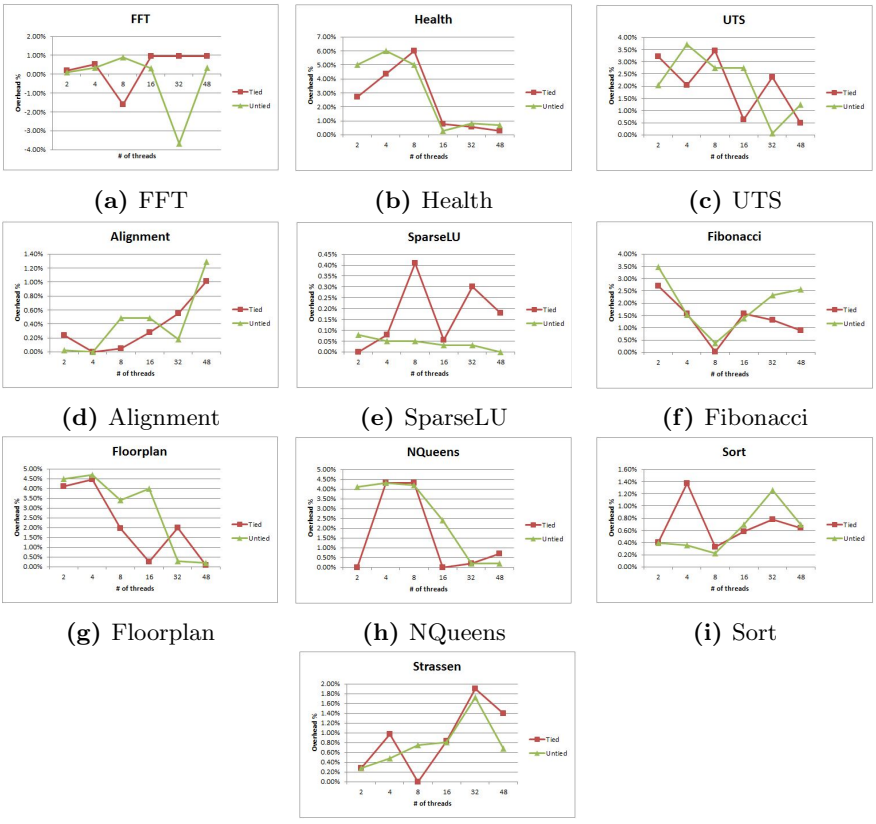
- We measured the overheads introduced by the inclusion of our implementation in the runtime.
- We tracked the newly developed task IDs, states, and events through our employed requests. This part was achieved by developing a prototype OpenMP task profiler tool.

We used the Barcelona OpenMP Task Suite (BOTS) kernels [3] as benchmark applications. The experiments were done using the x86_64 Linux system with four 2.2 GHz 12-core AMD Opteron processor (48 cores total).

4.1 Overhead Measurements

Table 2 gives details about our measurements. Each sub-table represents a kernel in the BOTS. Interested readers can consult the work [3] for full details about these kernels. We used six different numbers of threads. We collected data for both *tied* and *untied tasks*. Each kernel has two versions. One with tied tasks and the other with untied tasks. To calculate the overheads, we compiled the benchmark kernels using the OpenUH compiler. We ran the binaries with our OpenMP runtime library, while the tool is not attached. We employed a without vs. a with scenario, in which the without case excludes: assigning an ID to each task instance, assigning states to threads while working on tasks, tracking tasking events, and implementing task requests.

The results show that the overhead associted with our implementation is insignificant. The absolute overhead percentage ranges from 0% to 6% of the execution time. The average overhead percentage obtained is less than 1%. Overhead detail from Floorplan and NQueens kernels are given by Table 2g and Table 2h



(a) FFT **(b)** Health **(c)** UTS

(d) Alignment **(e)** SparseLU **(f)** Fibonacci

(g) Floorplan **(h)** NQueens **(i)** Sort

(j) Strassen

Fig. 5. BOTS overhead comparison (Tied vs. Untied)

respectively. These two kernels produced the maximum overhead. SparseLU and FFT, overhead described in Table 2e and Table 2a respectively, generated the minimum overhead. The variation in overhead is due to the behavior of these benchmarks.

Figure 5 demonstrates the overhead percentage obtained using tied vs. untied tasks for all the aforementioned kernels. The x-axis in each sub-figure represents the number of threads, while the y-axis represents the overhead percentage. As we can see from Figure 5, the behavior of tied and untied tasks, in terms of the range of deviations and the overhead's average, is very similar, except for the SparseLU benchmark shown in Figure 5e, where tied tasks have higher overheads.

Furthermore, when more threads are used, the overhead scales well with the increment of threads. As an example, when 48 threads are used, the worst overhead percentage obtained, among the different kernels, is 2.56%.

Table 2 can be consulted to obtain detailed overhead measurements about each benchmark.

4.2 Prototype OpenMP Task Profiler Tool

The motivation behind having such a profiler is to evaluate our proposed extensions in the runtime. The tool's functionality is to collect profiling measurements regarding task instances. These measurements can lead to a better utilization of task load-balancing, scheduling, and reducing overheads. OMP_REQ_START request should be used first to initialize the collector API to establish a connection with the runtime. OMP_REQ_RE

$GISTER$ request should be used next to selectively register the task events required for our calculations. $OMP\_REQ\_TASK\_ID$ and $OMP\_REQ\_TASK\_PID$ are used to get the task ID and the parent task ID respectively to construct the task-tree. By tracking the task events, we are able to request the thread-ID associated with each task as well as the thread's state while working on these tasks at any instance of time. Our tool also enables developers to obtain measurements about:

- Task instance creation time
- Task instance suspension time
- Task instance execution time
- Task instance cleanup and destroying time
- Task instance overhead waiting after creation to start execution

We have tested our tool with all the BOTS kernels. Our tool is capable of tracking millions of task instances including their IDs, states, and events. For the sake of simplicity, we show how our tool is useful by using the Fibonacci code shown in Figure 1a with different input sizes N and two threads. Figure 1b displays the task tree showing the parent and child tasks for each instance associated with their task IDs when N=4. Task 2 has to wait for tasks 7 and 8 until they finish their execution. We found that task 7 and task 8 were running in parallel since the two

Table 2. The Barcelona OpenMP Task Suite (BOTS) overhead measurements

(a) FFT

	runtime (tied/untied)		overhead(tied/untied)	
#thr	without	with	sec	%
2	10.35/10.26	10.37/10.27	0.02/0.01	0.19%/0.097%
4	5.94/5.65	5.971/5.67	0.031/0.02	0.52%/0.35%
8	3.73/3.36	3.67/3.39	(-0.06)/0.03	(-1.61%)/0.89%
16	2.839/3.3	2.866/3.31	0.027/0.01	0.95%/0.30%
32	3.17/4.33	3.2/4.17	0.03/-0.16	0.95%/-3.69%
48	4.22/5.73	4.26/5.75	0.04/0.02	0.95%/0.35%

(b) Health

	runtime (tied/untied)		overhead(tied/untied)	
#thr	without	with	sec	%
2	1.85/1.8	1.9/1.9	0.05/0.1	2.7%/5%
4	1.15/1.13	1.2/1.2	0.05/0.07	4.34%/6%
8	1/0.95	1.06/1	0.06/0.05	6%/5%
16	1.28/1.25	1.29/1.26	0.01/0.004	0.78%/0.3%
32	1.25/1.19	1.26/1.2	0.007/0.01	0.55%/0.8%
48	1.36/1.36	1.37/1.37	0.004/0.01	0.29%/0.7%

(c) UTS

	runtime (tied/untied)		overhead(tied/untied)	
#thr	without	with	sec	%
2	1.55/1.47	1.6/1.5	0.05/0.03	3.22%/2.04%
4	1.47/1.35	1.5/1.4	0.03/0.05	2.04%/3.70%
8	1.45/1.45	1.5/1.49	0.05/0.04	3.44%/2.75%
16	1.59/1.46	1.6/1.5	0.01/0.04	0.63%/2.74%
32	1.68/1.499	1.72/1.5	0.04/0.001	2.38%/0.07%
48	1.99/1.59	2/1.61	0.01/0.02	0.5%/1.25%

(d) Alignment

	runtime (tied/untied)		overhead(tied/untied)	
#thr	without	with	sec	%
2	8.28/8.27	8.3/8.272	0.05/0.002	0.24%/0.024%
4	4.13/4.14	4.13/4.14	0/0	0%/0%
8	2.079/2.07	2.08/2.08	0.001/0.01	0.05%/0.48%
16	1.046/1.044	1.049/1.049	0.003/0.005	0.28%/0.48%
32	0.546/0.541	0.549/0.542	0.003/0.001	0.55%/0.18%
48	0.396/0.389	0.4/0.394	0.004/0.005	1.01%/1.29%

(e) SparseLU

	runtime (tied/untied)		overhead(tied/untied)	
#thr	without	with	sec	%
2	1.27/1.278	1.27/1.28	0/0.001	0%/0.08%
4	3.661/3.663	3.664/3.66	0.003/0.002	0.08%/0.05%
8	1.945/1.955	1.953/1.956	0.008/0.001	0.41%/0.05%
16	1.0884/1.08	1.089/1.082	0.001/0.0003	0.055%/0.03%
32	0.658/0.659	0.66/0.659	0.002/0.0002	0.30%/0.030%
48	0.54/0.54	0.541/0.54	0.001/0	0.18%/0%

(f) Fibonacci

	runtime (tied/untied)		overhead(tied/untied)	
#thr	without	with	sec	%
2	4.38/2.899	4.5/3	0.12/0.101	2.7%/3.48%
4	3.79/1.97	3.85/2	0.06/0.03	1.58%/1.52%
8	3.529/2.59	3.53/2.6	0.001/0.01	0.03%/0.38%
16	3.19/2.86	3.24/2.9	0.05/0.04	1.56%/1.39%
32	5.28/2.15	5.28/2.2	0.07/0.05	1.32%/2.32%
48	6.68/1.95	6.74/2	0.06/0.05	0.89%/2.56%

(g) Floorplan

	runtime (tied/untied)		overhead(tied/untied)	
#thr	without	with	sec	%
2	24/11	25/11.5	1/0.5	4.1%/4.5%
4	11/7.5	11.49/8	0.49/0.5	4.45%/6.6%
8	9.7/8.7	9.89/9	0.19/0.3	1.95%/3.4%
16	11.87/12.5	11.9/13	0.03/0.5	0.25%/4%
32	11.02/9.7	11.24/9.73	0.22/0.03	1.99%/0.3%
48	10.98/9.66	10.99/9.68	0.01/0.02	0.09%/0.2%

(h) NQueens

	runtime (tied/untied)		overhead(tied/untied)	
#thr	without	with	sec	%
2	5/4.8	4/5	0/0.2	0%/4.1%
4	3.5/3.5	3.65/3.65	0.15/0.15	4.3%/4.3%
8	3.5/3.6	3.65/3.75	0.15/0.15	4.3%/4.2%
16	4.35/4.12	4.35/4.22	0/0.1	0%/2.4%
32	5.17/4.68	5.18/4.69	0.01/0.01	0.19%/0.21%
48	5.64/5.01	5.68/5.02	0.04/0.01	0.7%/0.19%

(i) Sort

	runtime (tied/untied)		overhead(tied/untied)	
#thr	without	with	sec	%
2	2.5/2.51	2.51/2.52	0.01/0.01	0.4%/0.39%
4	1.45/1.36	1.47/1.36	0.02/0.005	1.37%/0.36%
8	0.905/0.89	0.908/0.892	0.003/0.002	0.33%/0.22%
16	0.686/0.723	0.69/0.728	0.004/0.005	0.58%/0.69%
32	0.635/0.71	0.64/0.719	0.005/0.009	0.78%/1.26%
48	0.78/1.43	0.785/1.44	0.005/0.01	0.64%/0.69%

(j) Strassen

	runtime (tied/untied)		overhead(tied/untied)	
#thr	without	with	sec	%
2	0.353/0.353	0.354/0.354	0.001/0.001	0.28%/0.28%
4	0.205/0.207	0.207/0.208	0.002/0.001	0.97%/0.48%
8	1.4/0.132	1.4/0.133	0/0.001	0%/0.75%
16	0.121/0.124	0.122/0.125	0.001/0.001	0.828%/0.81%
32	0.201/0.23	0.205/0.234	0.004/0.004	1.9%/1.73%
48	0.345/0.44	0.35/0.445	0.005/0.003	1.4%/0.68%

threads were available at that instance of time. Table 3 records the timing measurements in seconds when Task-ID=2 with different values of N. The different task instance timings that include creation, execution (not including suspension), waiting before execution, and exiting were not affected by the input size, which is normal due to the fact that these events cannot be interrupted by another thread or task instance once they start. The main variation was found in the suspension time, which is due to the fact that the number of child tasks is positively proportional to the input size in a 2^N relationship. The parent task-id, which is 2 in Table 3, has to wait for all its child tasks before it can resume execution. The suspension time measurements, indicated by our tool, grow with the number of child tasks in the same relationship 2^N. These measurements present the efficiency and necessity of using our tool to get precise profiling information about the different OpenMP task applications.

Tasking collector API, proposed in this paper, are crucial to validate the various OpenMP task scheduling algorithms. OpenMP runtime library developers can consult our proposal to find the best approach in which a task should start execution and the thread that should be assigned to it. Two main optimizations (load balancing and data locality) related to task scheduling can be tested using our proposal. Load balancing should be taken care of when scheduling OpenMP tasks. Assigning tasks to the working threads in an equivalent manner is mandatory for any task scheduling algorithm. Our tool shows the thread associated with each task instance during all the different phases of the task execution. On the other hand, data locality is another concern for any scheduling algorithm. Tasks operating on the same data should be scheduled for execution on the same thread to improve data reuse, especially on non-uniform memory access (NUMA) architectures. The task tree constructed by our tool can indicate the data that was assigned to each task by mapping this tree back to the source code application.

Table 3. Fibonacci code timing measurements for Task-ID=2

Input	#Childs	Creation	Pool-waiting	Execution	Suspension	Exiting
2	0	0.0001	0.0001	0.0001	0.0	0.0001
4	2	0.0001	0.0001	0.0001	0.0002	0.0001
8	33	0.0001	0.0001	0.0001	0.0011	0.0001
16	1596	0.0001	0.0001	0.0001	0.3100	0.0001
32	3524577	0.0001	0.0001	0.0001	970	0.0001

5 Related Work

Profiler for OpenMP (POMP) [15] was the first profiling mechanism for OpenMP runtime. It enables performance tools to detect OpenMP events by specifying the names and properties of some instrumentation calls, including the invocation position and time associated with each event. The POMP adheres to the abstract OpenMP execution model and is independent of a compiler and an OpenMP runtime library. OpenMP Pragma and Region Instrumentor (OPARI) [15] is a portable source-to-source translation tool that inserts the POMP instrumentation calls in Fortran, C, and C++ programs. However, these instrumentation calls can notably affect the compiler optimizations and hence might not capture the true picture of an OpenMP program. The OPARI has been broadly used for OpenMP instrumentation in different performance tools such as TAU [18], KOJAK [16], and Scalasca [7]. Vampir [10] is another tool, which provides thread-specific measurements that can include the OpenMP static and runtime context. Another version of the POMP [14] was proposed as an attempt to standardize the OpenMP monitoring interface. However, this version was rejected by the OpenMP ARB because of its complexity and its implementation cost.

The work proposed by the Sun Microsystems [9] describes the OpenMP Runtime API (ORA) for profiling OpenMP applications. The ORA was accepted by the OpenMP Architecture Review Board (ARB). The ORA provides a framework to the performance collector tools to collect necessary information. This information is needed to enhance the performance of OpenMP programs. The OpenUH

research compiler group has developed an open source implementation [1], [8] for the ORA in the OpenUH runtime library. Another paper by Lin [11] presents a data model that captures the runtime behavior of OpenMP applications with tasks constructs. However, the work only captures the abstraction-level (construct-level) information of the OpenMP tasking constructs.

In order to measure the performance of task instances, Lorenz et al. [12] describe a portable method to distinguish individual task instances and track their suspension and resumption events using instrumentation calls implemented as an extension of OPARI. Lorenz et al. [13] also present an implementation within the Score-P performance measurement system to overcome the performance issues related to task profiling. Furlinger and Skinner [6] describe the support for task profiling using instrumentation in the ompP tool [5].

6 Conclusions and Future Work

In this work, we have presented our experiences in implementing a new API for OpenMP task profiling. The OpenMP Runtime API for profiling (ORA) was approved by the OpenMP tool committee to create a standardized tool interface for OpenMP programs. We have extended the ORA to support profiling for OpenMP tasks at the micro level. We have implemented our extensions using the OpenUH open-source compiler. Our extensions to the ORA allow the execution and scheduling of tied and untied OpenMP tasks to be tracked by a tool to collect performance measurements. These measurements assist OpenMP application developers to gain more insight into the dynamic behavior of OpenMP based applications. Our extensions adhere to the proposal recently suggested by the OpenMP tool committee for task profiling. Moreover, Our experimental results show that the overheads associated with our implementation are negligible. Finally, C/C++ and Fortran programs are supported by our implementation.

Our next step is to integrate our implementation with TAU, a powerful performance tool, to visualize the ORA measurements. We also plan to extend the ORA to support taskgroup and work-sharing constructs in order to make the ORA more powerful and comprehensive. We also plan to use the task related dynamic information, extracted through the ORA, for task-related optimizations using a feedback framework.

Acknowledgments. The authors would like to thank their colleagues in the HPCTools group, especially Deepak Eachempati, for their extensive collaboration to make this work a reality. This work is supported by the National Science Foundation under grant CCF-1148052. Development at the University of Houston was supported in part by the NSFs Computer Systems Research program under Award No. CRI-0958464.

References

1. Bui, V., Hernandez, O., Chapman, B., Kufrin, R., Tafti, D., Gopalkrishnan, P.: Towards an implementation of the OpenMP collector API. Urbana 51, 61801 (2007)

2. Chapman, B., Eachempati, D., Hernandez, O.: Experiences developing the OpenUH compiler and runtime infrastructure. International Journal of Parallel Programming, 1–30 (2012)
3. Duran, A., Teruel, X., Ferrer, R., Martorell, X., Ayguade, E.: Barcelona OpenMP tasks suite: A set of benchmarks targeting the exploitation of task parallelism in OpenMP. In: International Conference on Parallel Processing, ICPP 2009, pp. 124–131. IEEE (2009)
4. Eichenberger, A., Mellor-Crummey, J., Schulz, M., Copty, N., DelSignore, J., Dietrich, R., Liu, X., Loh, E., Lorenz, D.: OMPT: An openMP tools application programming interface for performance analysis. In: Rendell, A.P., Chapman, B.M., Müller, M.S. (eds.) IWOMP 2013. LNCS, vol. 8122, pp. 171–185. Springer, Heidelberg (2013)
5. Fürlinger, K., Gerndt, M.: ompP: A profiling tool for OpenMP. OpenMP Shared Memory Parallel Programming, 15–23 (2008)
6. Fürlinger, K., Skinner, D.: Performance profiling for openMP tasks. In: Müller, M.S., de Supinski, B.R., Chapman, B.M. (eds.) IWOMP 2009. LNCS, vol. 5568, pp. 132–139. Springer, Heidelberg (2009)
7. Geimer, M., Wolf, F., Wylie, B.J., Ábrahám, E., Becker, D., Mohr, B.: The Scalasca performance toolset architecture. Concurrency and Computation: Practice and Experience 22(6), 702–719 (2010)
8. Hernandez, O., Nanjegowda, R.C., Chapman, B., Bui, V., Kufrin, R.: Open source software support for the OpenMP runtime API for profiling. In: International Conference on Parallel Processing Workshops, ICPPW 2009, pp. 130–137. IEEE (2009)
9. Itzkowitz, M., Mazurov, O., Copty, N., Lin, Y.: An OpenMP runtime API for profiling. OpenMP ARB as an official ARB White Paper 314, 181–190 (2007), `http://www.compunity.org/futures/omp-api.html`
10. Knüpfer, A., Brunst, H., Doleschal, J., Jurenz, M., Lieber, M., Mickler, H., Müller, M.S., Nagel, W.E.: The Vampir performance analysis tool-set. In: Tools for High Performance Computing, pp. 139–155. Springer (2008)
11. Lin, Y., Mazurov, O.: Providing observability for openMP 3.0 applications. In: Müller, M.S., de Supinski, B.R., Chapman, B.M. (eds.) IWOMP 2009. LNCS, vol. 5568, pp. 104–117. Springer, Heidelberg (2009)
12. Lorenz, D., Mohr, B., Rössel, C., Schmidl, D., Wolf, F.: How to reconcile event-based performance analysis with tasking in openMP. In: Sato, M., Hanawa, T., Müller, M.S., Chapman, B.M., de Supinski, B.R. (eds.) IWOMP 2010. LNCS, vol. 6132, pp. 109–121. Springer, Heidelberg (2010)
13. Lorenz, D., Philippen, P., Schmidl, D., Wolf, F.: Profiling of OpenMP tasks with score-p. In: 2012 41st International Conference on Parallel Processing Workshops (ICPPW), pp. 444–453. IEEE (2012)
14. Mohr, B., Malony, A.D., Hoppe, H.-C., Schlimbach, F., Haab, G., Hoeflinger, J., Shah, S.: A performance monitoring interface for OpenMP. In: Proceedings of the Fourth Workshop on OpenMP, EWOMP 2002 (2002)
15. Mohr, B., Malony, A.D., Shende, S., Wolf, F.: Design and prototype of a performance tool interface for OpenMP. The Journal of Supercomputing 23(1), 105–128 (2001)
16. Mohr, B., Wolf, F.: KOJAK–a tool set for automatic performance analysis of parallel programs. In: Euro-Par 2003 Parallel Processing, pp. 1301–1304 (2003)
17. Qawasmeh, A., Chapman, B., Banerjee, A.: A compiler-based tool for array analysis in HPC applications. In: 2012 41st International Conference on Parallel Processing Workshops (ICPPW), pp. 454–463. IEEE (2012)
18. Shende, S.S., Malony, A.D.: The TAU parallel performance system. International Journal of High Performance Computing Applications 20(2), 287–311 (2006)

Author Index