SAND2013-5301C

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@bsc.es,michael.klemm@intel.com,kli@ca.ibm.com,
xavier.martorell@bsc.es,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 parallel 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. 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

[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.

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.

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.

```
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.

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 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.

```
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

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 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.

[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)

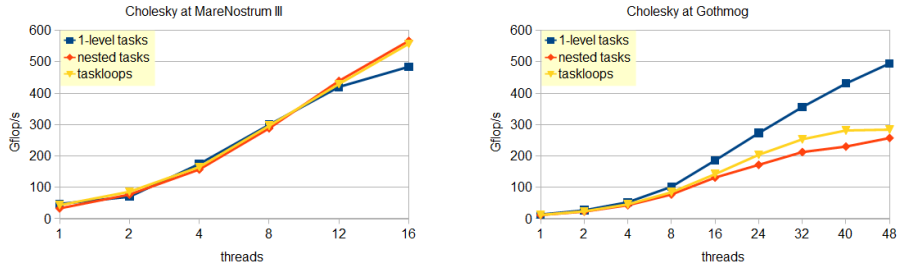
[3] Based on git repository (http://pm.bsc.es/git/mcxx.git) revision mcxx 1.99.0 (git b51a11c 2013-04-10 09:27:34 +0200 developer version)

```
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

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 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.



(a) MareNostrum III (b) Gothmog

Fig. 1. Cholesky performance results.

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.

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.

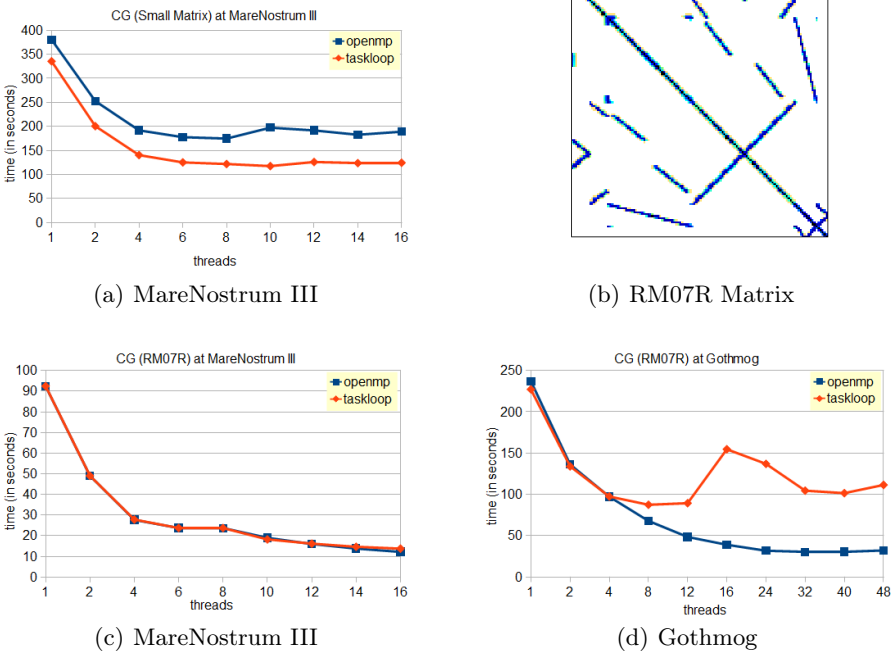
5 Related Work

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

[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.

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 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 (Mar 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'04). 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 (Sep 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: Languages and Compilers for Parallel Computing. Lecture Notes in Computer Science, vol. 5898, pp. 416–423. Springer Berlin/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 (March 2010)
8. Microsoft: Task Parallel Library (2013), `http://msdn.microsoft.com/en-us/library/dd460717.aspx`, last accessed 2013-06-21
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, CA (July 2007)
12. Terboven, C., Schmidl, D., Cramer, T., an Mey, D.: Task-parallel Programming on NUMA Architectures. In: Euro-Par 2012 Parallel Processing, Lecture Notes in Computer Science, vol. 7484, pp. 638–649. Springer Berlin/Heidelberg (2012)