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

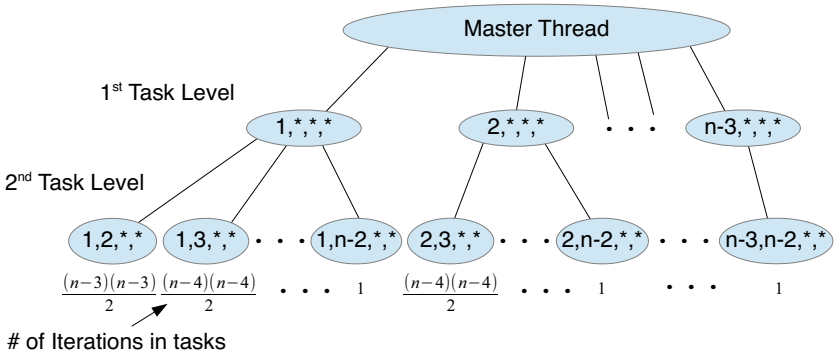


Fig. 1. The task graph of a network with n nodes

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

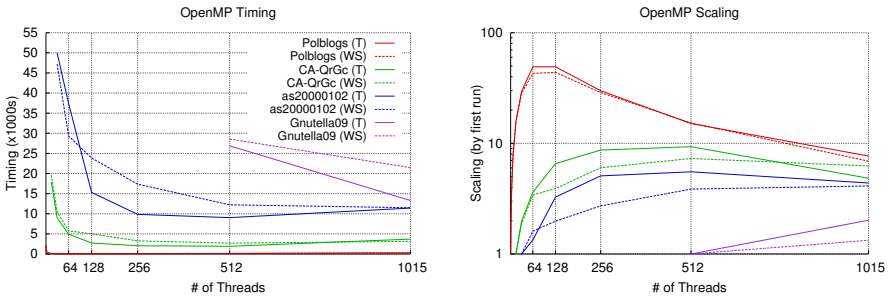
of jobs using even the smallest of our test networks, which we believe can be attributed to calls to the Boost Graph Library.

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

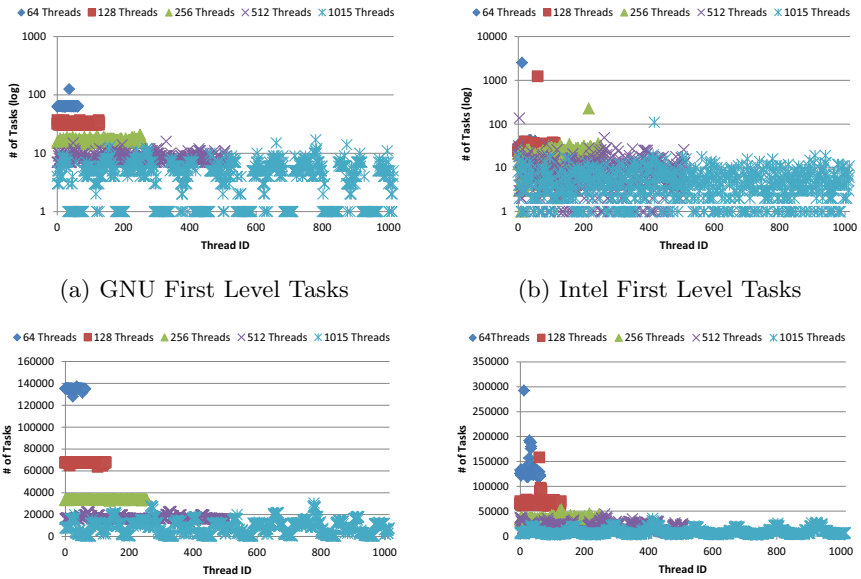
Table 2. Computation time (in seconds) of tasking versus workshare

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

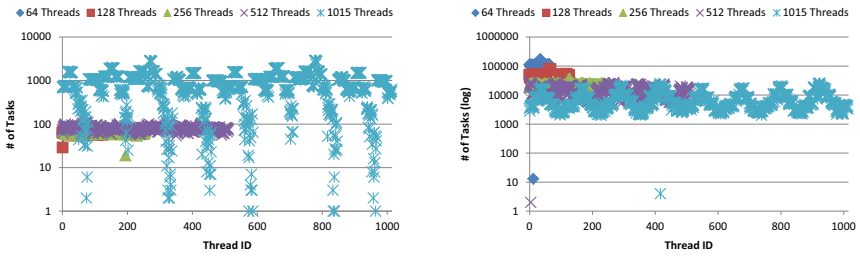
Fig. 3. Number of first and second level tasks executed per thread in CA-GrQc

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

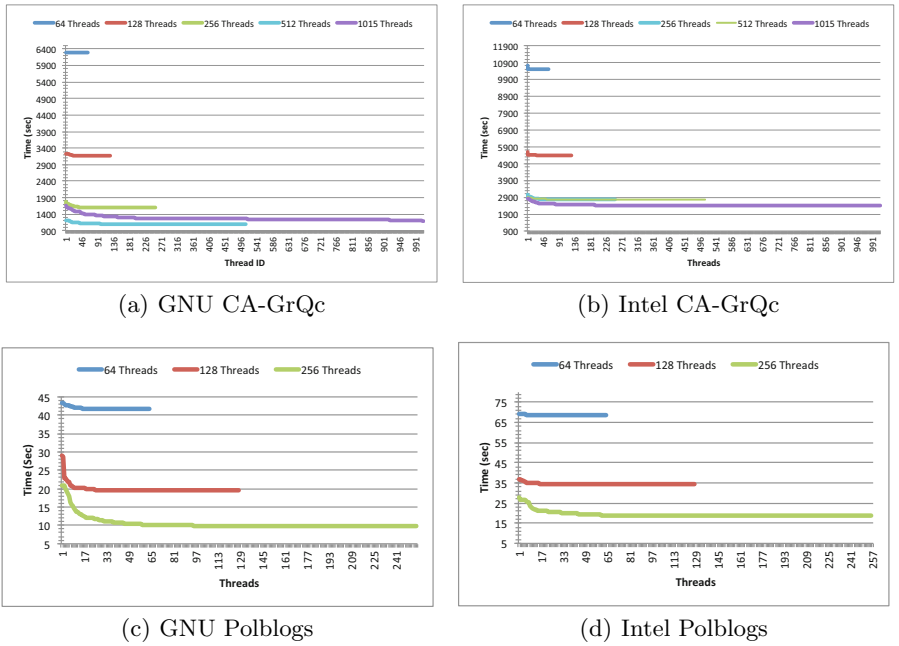
Table 3. Performance ratio (original runtime / optimized runtime) on GNU-compiled code

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

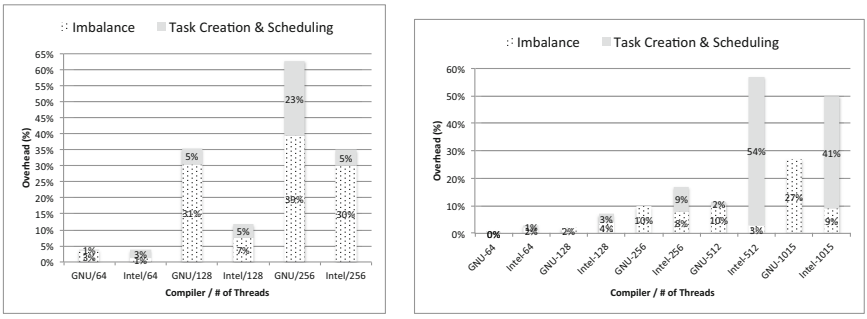


(a) GNU CA-GrQc (b) Intel CA-GrQc

(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)