Executing Optimized Irregular Applications Using Task Graphs Within Existing Parallel Models

Christopher D. Krieger, Michelle Mills Strout, Jonathan Roelofs
Colorado State University
Fort Collins, CO 80526
krieger|mstrout|roelofs@cs.colostate.edu

Amanreet Bajwa
Colorado School of Mines
Golden, CO 80401
amanreetbajwa@gmail.com

***Abstract*—Many sparse or irregular scientific computations are memory bound and benefit from locality improving optimizations such as blocking or tiling. These optimizations result in asynchronous parallelism that can be represented by arbitrary task graphs. Unfortunately, most popular parallel programming models with the exception of Threading Building Blocks (TBB) do not directly execute arbitrary task graphs. In this paper, we compare the programming and execution of arbitrary task graphs qualitatively and quantitatively in TBB, the OpenMP doall model, the OpenMP 3.0 task model, and Cilk Plus. We present performance and scalability results for 8 and 40 core shared memory systems on a sparse matrix iterative solver and a molecular dynamics benchmark.**

I. Introduction

Computations involving irregular data access patterns figure prominently in many important scientific applications. These include solving partial differential equations over irregular grids, molecular dynamics simulations, and sparse matrix computations.

In some cases, these computations are intrinsically parallel and can be performed using direct doall parallelism across all elements or across particular elements in a computational sweep. Unfortunately, in many cases these straightforward parallelization strategies encounter performance problems due to the irregularity of the data accesses. Irregular accesses decrease the efficiency of caching and data prefetching, and therefore increase the demand on memory bandwidth. To avoid memory bottlenecks, these algorithms are modified by performance programmers to improve the algorithm's temporal and spatial locality. These optimization techniques include irregular cache blocking [1], full sparse tiling [2], and communication avoiding algorithms [3].

The benefits realized by these approaches can be illustrated by applications as simple as a Jacobi solver. The Jacobi solver algorithm is an iterative algorithm for determining solutions to a system of linear equations. The recurrence equation that describes Jacobi is as follows: $u_i^{(k)} = \frac{1}{A_{ii}}\left(f_i - \sum_{j \neq i} A_{ij} * u_j^{(k-1)}\right)$. Figure 1 shows the iteration space for Jacobi. Each point is a computation of a specific $u_i^{(k)}$. Arrows between the points are data-flow dependences. The example in Figure 1 shows the iteration space that would occur for a tri-diagonal sparse matrix, where each unknown has two neighboring unknowns with which it is coupled. The right side of Figure 1 shows a possible sparse tiling of Jacobi. That tiling results in the task graph shown in Figure 2.

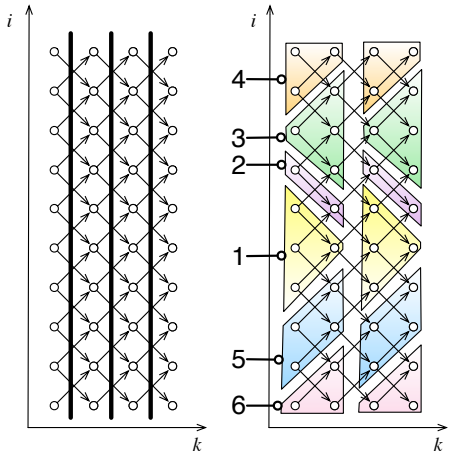


Fig. 1. At left, the Jacobi iteration space for a tri-diagonal sparse matrix. Typically sparse matrices are not this structured. The heavy black lines show barriers that occur when performing a doall parallelization. On the right, a full sparse tiling of the computation.

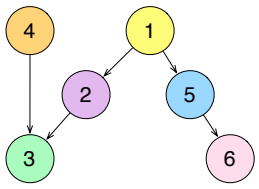


Fig. 2. Task graph for the full sparse tiled Jacobi computation. Each node represents a group of computations and edges represent the partial ordering due to data dependencies.

While the Jacobi method admits a straightforward doall parallel solution on the i loop, this approach quickly hits scalability issues due to bandwidth demands. Figure 3 shows the performance benefit of a full sparse tiled Jacobi solver compared to a doall parallelized version as threads are increased on a 40 core machine. As threads are added, the

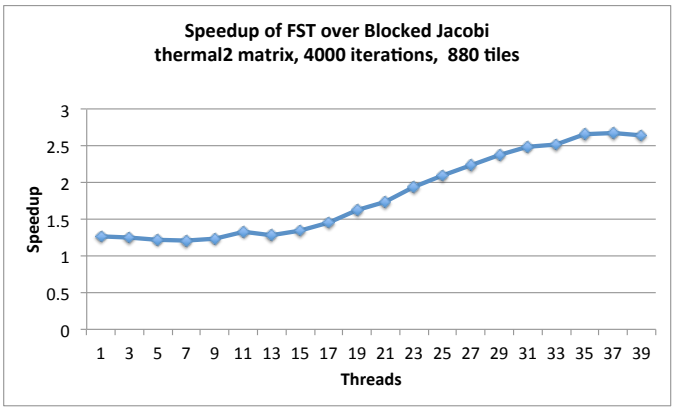


Fig. 3. Relative speedup of full sparse tiled Jacobi solver compared with blocked version.

performance benefit widens from approximately 25% to 165%.

The computation resulting from tiling optimizations is usually still highly parallel, but now requires a partial ordering on the execution of the sparse tiles in order to preserve correctness. Such a series of partially ordered steps can be represented naturally using an *arbitrary task graph* like the one shown in Figure 2.

Custom task graph execution code could be written from scratch, but there are several advantages to using an established parallel programming model. Doing so allows arbitrary task graph execution to take advantage of parallel schedulers and runtimes that are finely tuned by vendors or the parallel programming community. Leveraging existing parallel models enables the use of model specific debugging and performance analysis tools [4] such as Cilk Plus' *cilkscreen* [5]. The TAU tool suite [6] and Intel's Inspector XE also include support specifically for OpenMP.

For these reasons, we would like to express arbitrary task graph parallelism within existing parallel programming models. However, doing so is not straightforward. In Section II, we discuss mismatches between task graph parallelism and other programming models. We present high-level strategies for bridging these differences and provide implementation details for the Threading Building Blocks' graph model, the OpenMP 3.0 task model, the OpenMP doall model, and Cilk Plus. Section III presents a quantitative analysis of the performance of each task graph execution engine for two scientific applications: a Jacobi solver and a molecular dynamics simulation. Section IV presents related work and Section V concludes.

II. Implementing Task Graphs Using Common Parallel Patterns

Arbitrary task graphs are directed acyclic graphs (DAGs) where the vertices represent work or *tasks* to be performed. A task can represent work to be done in parallel or series. Each task can itself be a subordinate task graph or contain a different style of parallelism, such as doall parallelism. Directed edges between task vertices in the graph represent partial ordering between the connected tasks, usually, though

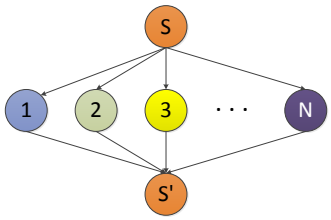


Fig. 4. Graphical representation of doall parallelism

not exclusively, due to data flow dependencies. A task graph is termed *arbitrary* because no further restrictions are imposed on the structure of the task graph beyond it being a DAG.

The expressive power of an arbitrary task graph lies in a task's ability to have multiple predecessors and successors. A task only executes after all its predecessors have completed. This implies that during execution, the execution system must maintain dynamic state information regarding what tasks have completed in order to determine if a task is eligible to start. Unmodified doall (e.g. OpenMP parallel for) and fork-join (e.g. Cilk Plus, OpenMP tasks) do not fulfill this requirement.

In the remainder of this section, we examine parallel patterns that are directly supported in common parallel programming models and contrast their characteristics with those of arbitrary task graph parallelism. The patterns we discuss here are related to those identified in [7]. Two general strategies emerge for expressing arbitrary task graphs within other parallel models. The first approach is to augment the model with additional capabilities until the enhanced model is sufficiently expressive to encompass arbitrary task graphs. The second approach is to restrict the parallelism of the task graph such that it can be expressed within a given parallel model. We use combinations of these strategies to express task graph parallelism in each of the patterns under discussion.

A. Doall Parallel Pattern

The most common parallel pattern, supported by many parallel programming libraries, is doall parallelism. In this pattern, all specified work can be executed in parallel. The most commonly specified work group is the independent iterations of a loop, e.g. OpenMP's *parallel for* pragma, although the application of a function to each element in a container is also often supported, e.g. TBB's *parallel_for*. Figure 4 is a diagram showing the work flow for doall parallelism. The main program or execution thread spawns multiple parallel processes, then waits at an implicit barrier for all work to complete. The top node, labeled S, represents the spawning of the parallel work, while the bottom node, labeled S', represents the barrier that waits for work completion.

Doall is a simple pattern that is limited to loops where all iterations can be executed in parallel. For arbitrary task graphs we can remove the asynchronous nature and create a *frontier schedule* with stages of tasks, where each stage is simply a level set of tasks at a given depth in the graph. For example, in Figure 2, nodes 1 and 4 are placed in the first frontier stage,

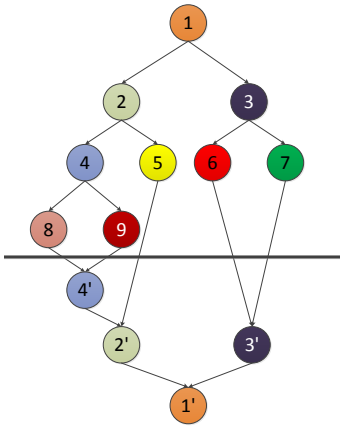


Fig. 5. Graphical representation of fork-join parallelism. Numbered circles above the black line represent the fork portion. Circles labeled with a number plus an apostrophe found below the black line represent the join portion.

nodes 2 and 5 are placed in the second stage. Note that because of this approach, nodes 2 and 5 now behave as though there were an ordering edge from node 4 ending at each of them.

The use of the frontier schedule reduces the fully asynchronous parallel execution of a task graph to a series of parallel frontier stages, each of which follows the doall pattern. It removes the burden of tracking what tasks have completed from the dynamic execution engine, A consequence of this approach is that many valid execution interleavings are arbitrarily eliminated, leading to potentially faster schedules. This scheme also reduces the scheduler's ability to load balance by introducing frequent barriers. Balancing can occur within a stage, but the barrier between stages prevents load balancing across stages.

We implemented a doall based task graph execution engine using OpenMP[8][9] `parallel for` pragmas to parallelize across all the elements within a frontier.

B. Fork-Join Parallel Pattern

Nearly as ubiquitous as doall parallelism is *fork-join* parallelism. In the fork-join pattern, executing code determines that multiple forward paths are desirable, so the single thread spawns multiple execution paths. At some point in the future, when these forked threads complete, execution will return to and continue from the fork point. A common use of this fork-join parallel pattern is in the implementation of *divide and conquer* algorithms, in which work is recursively subdivided into smaller chunks until some criterion is met. Our exemplars for parallel models using fork-join parallelism are Intel's Cilk Plus [5][10] and the OpenMP 3.0 task model [11].

An example of the fork-join pattern is shown in Figure 5. The portion above the solid horizontal line represents the fork portion of the pattern, while nodes below the line, denoted with apostrophes, represent the continuation of execution after the forked execution paths have returned. For example, node $4'$ combines the results from leaf child nodes 8 and 9. In turn, it passes that result to node $2'$. Eventually the entire call tree unwinds and execution continues at the point from which task 1 was called.

The fork-join pattern is far more expressive than the doall pattern and in fact can fully express doall parallelism as a degenerate case. Fork-join parallelism comes close to being able to express arbitrary task graph parallelism. It supports the notion of a predecessor and a partial ordering on node execution via the fork and join edges seen in Figure 5. However, the limitation is that in the fork-join pattern each non-entry node has exactly one predecessor node and that execution flow returns to that predecessor during the fork-join stack unwinding process. To take advantage of the more dynamic nature of fork-join parallelism over doall parallelism, we track the dynamic state of each task. To this end, the task graph itself is needed, along with a per-task count of the number of that task's pending predecessors. Before executing the task graph, this data structure is initialized with the predecessor count for each task in the task graph. When a task completes, the task graph is referenced to determine each of the just-completed task's successors. The pending predecessor count for each of these successors is then atomically decremented. If the count is found to be zero, the engine spawns execution of that successor task. To the underlying fork-join pattern, each node meets the single predecessor requirement by having its last-completing task graph predecessor function as its parent in the fork-join pattern. Note that Nabbit [12] uses Cilk++ to execute arbitrary task graphs in much the same way.

Looking at Figure 5, we observe that for arbitrary task graphs, the portion of the diagram below the horizontal line is unneeded. A successor task never returns control flow or data to a predecessor. Therefore, during task graph execution within a fork-join pattern, a fork-join stack is dynamically and unnecessarily created.

There is a subtle but significant difference in the semantics of the Cilk Plus and OpenMP task models. In the OpenMP tasking model, a thread will wait for all tasks generated since the beginning of the current task to complete *only* if a `taskwait` directive is *explicitly* given. In contrast, Cilk Plus *implicitly* inserts a `cilk_sync` at the end of every function that contains a `cilk_spawn`. This means that Cilk Plus must always create a full fork-join call stack, while OpenMP can avoid the winding and unwinding of the fork-join stack.

Although OpenMP allows for task spawning without waiting for their completion, doing so introduces a new issue. The join portion of the fork-join pattern, while not executing code, serves the critical role of determining when the overall task graph has completed its execution. As long as the main code can launch task graph execution, then wait for the fork-join stack to unwind, execution can continue from that point with the knowledge that the graph has fully executed. If OpenMP asynchronous tasks are used, control returns immediately to the main code. Without the use of additional mechanisms, the main code cannot determine if the task graph has completed execution. Resolving this issue requires an additional mechanism outside the fork-join pattern. One technique is for the execution engine to count how many leaf nodes are present in

Processor Type	Procs x Cores	L1D Cache	L2 Cache	Last Level Cache	Memory
Intel Xeon E5450	2x4	32 kB	—	6 MB shared/core pair	16 GB
Intel Xeon E7-4860	4x10	32 kB	256 kB	24 MB shared/10 cores	256 GB

TABLE I
HARDWARE USED IN THE PERFORMANCE EVALUATION

the task graph and store this value. Upon completion of a leaf node, the engine atomically decrements this leaf counter. If the decrement causes the counter to reach zero, all leaf nodes and therefore all nodes have been executed and the engine signals the suspended main code to continue execution.

For completion detection on OpenMP, we implemented both leaf node counting and the fork-join stack unwinding techniques. Neither showed a significant performance advantage over the other, even on deep task graphs with many nodes. Ultimately, we chose the native stack unwinding method as the default behavior.

These engines are fairly simple because they use the work queues provided by the underlying model. Both OpenMP and Cilk Plus manage thread pools and dynamically balance work across the threads in the pool using scheduling and work stealing.

III. PERFORMANCE EVALUATION

We conducted our performance evaluation using two kernels drawn from representative irregular scientific codes. Each has unique tiling and memory access patterns and will be detailed in turn below.

Our tests were run on the machines described in Table I. The high core count machine was made available for our research by Intel's Manycore Testing Lab. The Intel icpc compiler, version 12.1.0 (20110811) was used. For comparison purposes, in addition to the models previously discussed, results are also presented for a task graph execution engine coded from scratch using only pThreads. The reported performance statistics refer only to the time during which the task graph is being executed. Any time required by the inspector to generate the task graph or to read in and process data files is excluded.

A. Jacobi Solver

The first algorithm in our performance evaluation is a sparse Jacobi solver. The Jacobi algorithm was described in detail in Section I. The Jacobi kernel is tiled across two convergence iterations of the main loop and uses a temporary buffer to store the intermediate vector results (ping-pong storage allocation). Each tile was sized to access approximately 200kB of data, so as to fit within the caches of the processors used. The amount of time to execute a single typical task for Jacobi was between 200 and 500 μs. All of the sparse matrices were drawn from the University of Florida Matrix Market and are listed in Table II. Due to space constraints, we present full results for only a single sparse matrix. Scalability was slightly reduced for larger input sets, but performance of the engines relative to one another was consistent across input sets.

As can be seen in Figure 6, performance was remarkably homogeneous for the different engines when executing the Jacobi algorithm. The 8 core machine showed scalability of 3.9x on 8 cores, while the 40 core machine scaled similarly to 17.2x on 40 cores.

Name	Rows	Nonzero/Row	Mem (MB)
xenon2	157464	24	48
thermal2	1228045	6	130
pwtk	217918	53	138
audikw_1	943695	82	913

TABLE II
CHARACTERISTICS OF SPARSE MATRICES USED IN THE PERFORMANCE EVALUATION FOR THE JACOBI SOLVER.

Name	Atoms	Interactions	Memory (MB)
3NZ8	6677	19639	0.61
3RFZ	19068	56098	1.74
2IA5	28415	80130	2.56

TABLE III
CHARACTERISTICS OF PROTEINS SIMULATED WITH MOLDYN

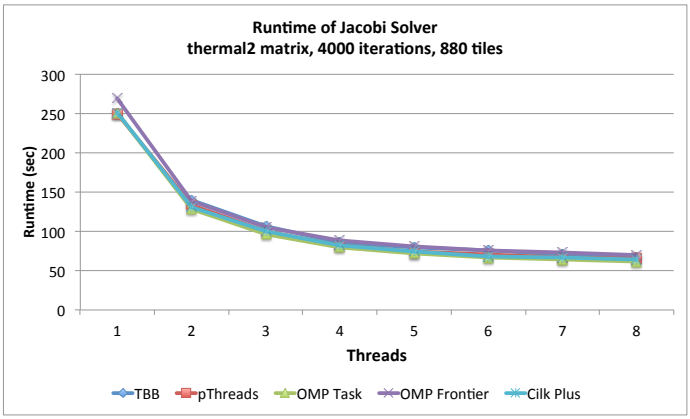
B. Molecular Dynamics Benchmark

We also evaluated performance executing task graphs generated from full sparse tiling a Moldyn molecular dynamics benchmark. After the initial locations of the atoms are read from a file, an interaction list is populated with pairs of atoms that are within a given cutoff distance from one another. This interaction list is not updated during the simulation. The atom data is stored in a single two dimensional dense array of double precision numbers. The input data sets represent proteins from the RCSB Protein Data Bank and are listed in Table III. Note that the data sets for Moldyn easily fit within the last level caches of all our evaluation machines.

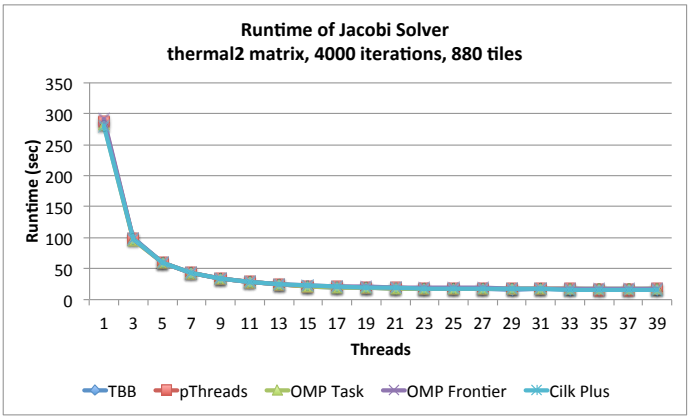
The Moldyn kernel consists of an outer timestep loop surrounding three inner loops. The inner loops update the position of atoms based on their velocities, recompute the forces between atoms based on their new locations, then update the velocities of the atoms due to the forces. The first and third loops access only the tiling information and the data arrays and do simple computation. The middle loop has higher computational intensity and accesses the interaction array in addition to the other information.

We choose to tile across 1024 tiles for the 2IA5 input set, whose results are shown in Figure 7. This gave an average degree of parallelism of approximately 114. It also reduced the amount of work done per tile to less than 100 atom updates and 100 atom interactions per tile on average. This led to a fine work grain of approximately 5-7μs.

On the 8 core machine, as more threads are added the runtime generally decreases as expected. Scalability at 8 threads

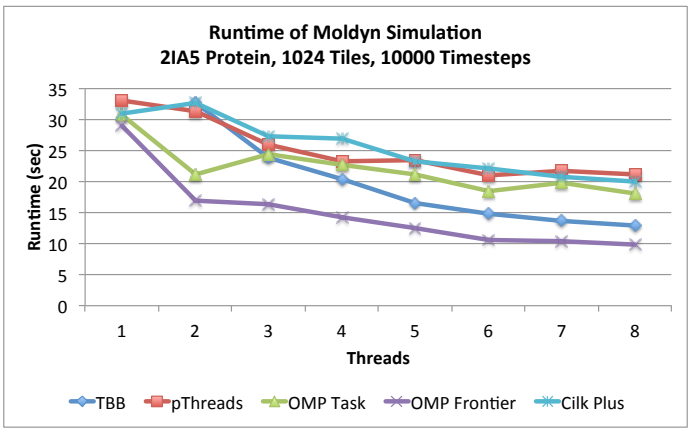


(a) 8 Core Results

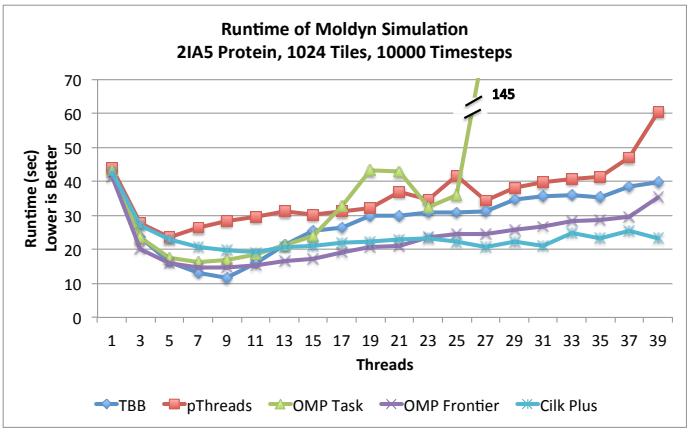


(b) 40 Core Results

Fig. 6. Comparison of runtimes when executing the Jacobi solver with different engines.



(a) 8 Core Results



(b) 40 Core Results

Fig. 7. Comparison of runtimes when executing the Moldyn benchmark with different engines.

is fairly poor, reaching between 1.2x and 2.96x at 8 threads. A reduction in runtime is observed on the 40 core system only until between 5 and 10 threads are used, depending on the engine. At this point, the runtime begins to increase and continues to increase through the maximum of 40 threads.

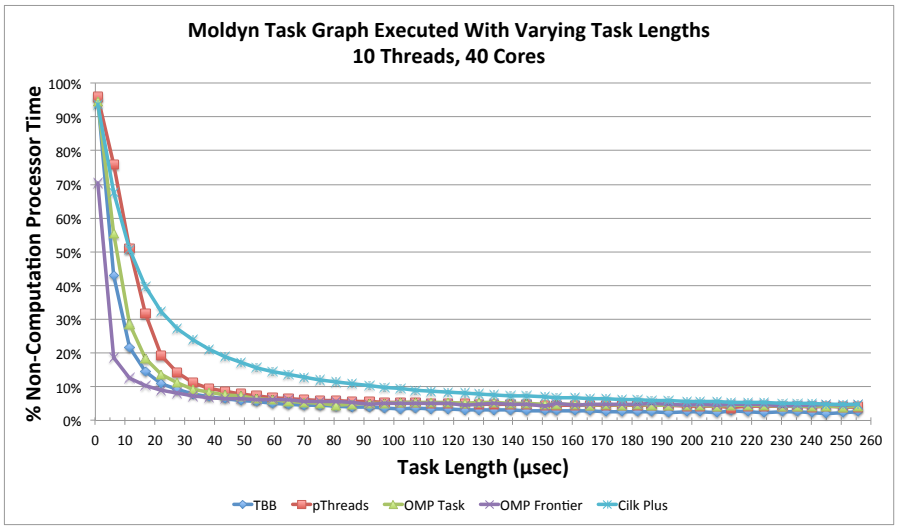
There is a much larger variance in performance between the different engines when running a moldyn task graph. For example, at 30 threads, the worst runtime is 1.72x that of the best runtime. The simple pThreads engine unsurprisingly delivers the worst performance overall. In general, for thread counts up to 22, the frontier scheduler performs best before being surpassed by the Cilk Plus engine.

Note that the behavior of the OpenMP task engine is erratic. The runtime under the OpenMP task engine drastically increases at 25 threads and does not recover as additional threads are added. This same general behavior was observed with the Intel OpenMP implementation and with the version of OpenMP included with gcc 4.4. By way of contrast, the frontier based runtime, also using OpenMP but not explicitly using tasks, performed well and did not exhibit this pathological behavior.

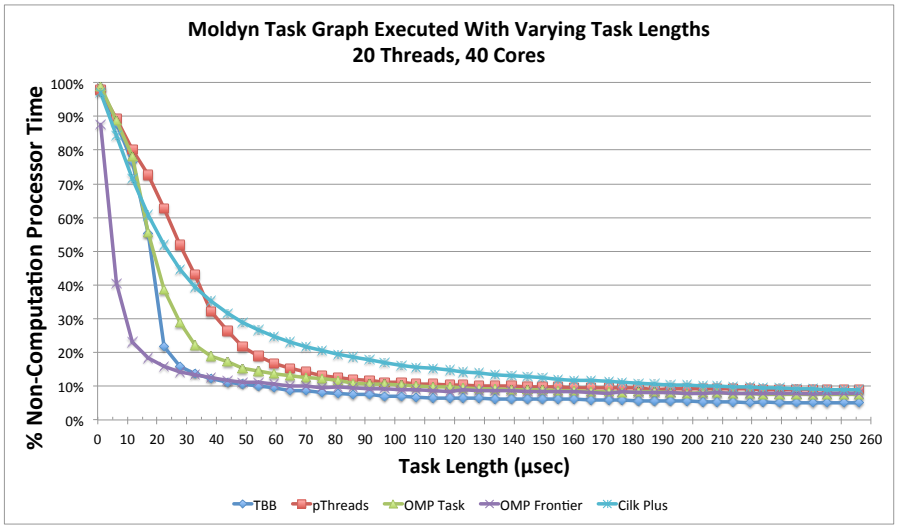
C. Controlled Task Length Experiments

To better understand the Moldyn results, we conducted an additional experiment. We reran the Moldyn solver, but replaced the actual molecular dynamics computation with a simple test kernel. The kernel does basic floating point computation in a loop. It makes no memory accesses. By varying the number of iterations of this computation loop, we are able to control the amount of time spent in a task. Each task should take the same amount of time.

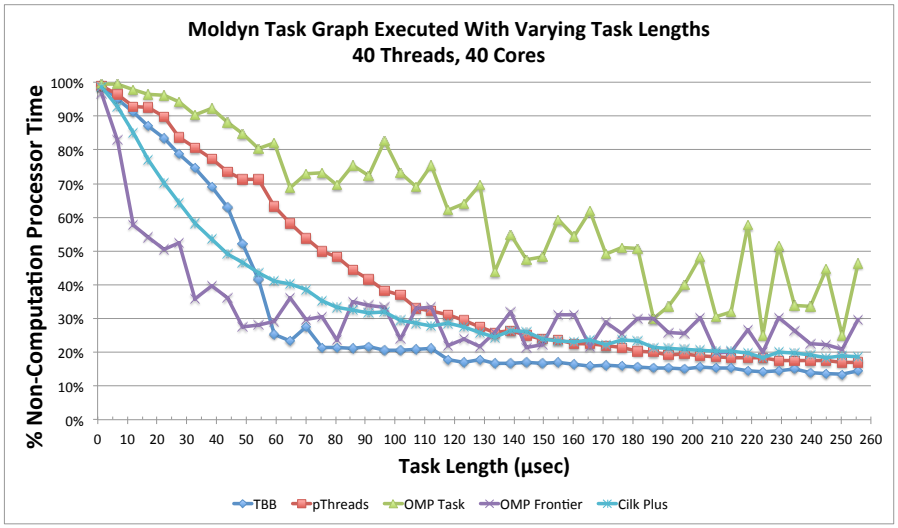
Figure 8 shows the results of executing the moldyn task graph for the 2IA5 protein, but executing the test kernel in place of the molecular dynamics simulation code. The length of each task is increased from 1 to 256 μs and the percentage of unproductive, non-computation time is plotted on the vertical axis. We calculate the unproductive time by first directly measuring the total execution time using calls to a TBB tbb::tick_count object. We then multiply this time by the number of threads used. This yields the total amount of processor seconds that were available during the computation. Next, we measure the time spent in the computation portion



(a) 10 Thread Results



(b) 20 Thread Results



(c) 40 Thread Results

Fig. 8. Percentage of processor time spent on unproductive work while executing the Moldyn 2IA5 task graph, but replacing the actual work with a simple work function of controllable length.

of each task and sum that across all tasks to compute the total time actually spent doing test kernel work. The unproductive time is the difference between these two values.

Unproductive time may be due to a number of sources. For example, overhead in the queueing system, time spent determining if successor tasks are ready to execute, or time during which a processor is idle due to load imbalance may all contribute. At the left side of each graph there is essentially no work being done, so the non-useful work approaches 100%. As the test work length is increased, the graph execution overhead is amortized over an increasingly lengthy total execution time. Therefore, the non-useful work becomes an increasingly smaller percentage until it reaches a floor. The floor is due to load imbalance and increases proportionally with the test task size, thus contributing a roughly constant percentage.

Figure 8(a) shows the graph when the experiment is run using only 10 threads on a 40 core system. The behavior largely matches expectations. The overhead for Cilk Plus takes longer to amortize, while the other engines behave similarly to one another. When the test task length reaches between 10 and 30 μs, the overhead has reached its minimum value for all engines except Cilk Plus, which requires a test task length of almost 150 μs before converging on its minimum. Note that the frontier schedule has lower initial overhead than the other engines.

The results from running the same experiment with 20 threads are shown in Figure 8(b) . Note that neither the task graph nor the hardware was changed. The curves in general have shifted to the right and a larger difference in performance is now evident between the different execution engines. In particular, the pThreads scheduler, with a single shared work queue, now requires tasks of length between 50 - 80 μs to reach the floor value. TBB overhead increases from 15μs to 25 μs. The floor value has also risen by approximately 5%. We believe this degradation in performance is due to increased contention for shared resources such as work queues, and increased work stealing.

Figure 8(c) presents the results when the thread count is increased further to 40 threads. At this point, the number of threads exceeds the amount of parallelism available during much of the task graph execution. The behavior of the OpenMP frontier and OpenMP task engines has become erratic. The other engines continue to be well behaved but now do not reach the floor value until a much longer task length that varies widely between engines. Note also that the floor has increased to approximately 20%.

This experiment sheds some light on the difference in behavior seen between the Moldyn benchmark and the Jacobi solver in Figures 6(b) and 7(b). The average task length for a Moldyn task is 5-7μs and for a Jacobi task is 200-500 μs. This places Moldyn tasks at the left edge of the graphs in Figure 8 and tasks from the Jacobi solver near or beyond the right edge of the graphs. Therefore, for Moldyn simulations, overhead is the majority of the runtime. As threads are added, the overhead rises and the runtime engines become increasingly stressed. These results suggest that the number of threads used should not significantly exceed the amount of available parallelism. While it is well understood that using more, smaller tasks incurs more overhead, these results further suggest that task sizes should be kept above a quantifiable minimum. Furthermore, in this experiment, the OpenMP engines are found to be less tolerant of fine grained tasks than the engines based on other runtime systems.

IV. Related Work

There is a considerable body of work related to optimal scheduling of task graphs for multiprocessor systems. For example, in [13], Casavant and Kuhl survey over 50 different algorithms for statically scheduling task graphs. Work by Kwok et al. [14][15], surveys task graph scheduling algorithms as well. Our work differs in that we focus on dynamically scheduling tasks to processors as they become available, rather than generating an optimal static schedule beforehand. We also build our implementations on readily available parallel programming models. This work emphasizes challenges using existing programming models to execute task graphs, making it complementary to much of the existing literature.

The memory wall problem has been extensively studied in the context of dense matrix computations. In this class of computations, some form of aggregation (e.g., tiling [16], [17]) is often used to improve cache line utilization and avoid false sharing. To provide even better parallel scaling, asynchronous versions of dense matrix algorithms have been developed [18], [19], [20], [21], [22]. The general approach is to tile parts of the computation and create a task graph where each tile is a task and the dependences between tasks are exposed to a runtime scheduler. This improves the performance on multicores over bulk synchronous approaches because the tiling leads to better utilization of the cache hierarchy and the asynchronous parallelism improves tolerance of memory latency and enables dynamic load balancing.

The idea of exposing aggregated, asynchronous parallelism at inspector time is not new. Mirchandaney et al. [23] presented a C library called PARTY for expressing the DAG at runtime. Our work differs from their work in that it uses sparse tiling and focuses on how to map the resulting DAGs (arbitrary task graphs) to various existing parallel programming models.

Other approaches to parallelizing sparse iterative matrix computations have expose asynchronous parallelization within each outer iteration, perform aggregation across the outer iterations to improve bulk synchronous parallelization approaches, or expose blocked synchronous parallelism.

The communication avoiding algorithms [24] still use a bulk synchronous parallelization approach, but they improve upon the synchronization for each time step approach in two ways. First, they only synchronize after some constant number of time steps k. And second, they avoid communication (or false sharing) between processors by duplicating some of the computation being done between processors.

Mark Adams presented an asynchronous parallel algorithm for use within one sweep of the Gauss-Seidel sparse iterative computation [25]. The computation within a single sweep

is partitioned into pieces based on the matrix graph. Each piece is then divided into two types of interior nodes and three kinds of boundary nodes based on their numbering in relation to their neighbors. Each processor then executes a task graph where each node grouping is essentially a task and has certain dependences with other node groupings. The experimental results we present in this paper explore task graphs that span the outer loop of iterative methods, and we are looking at Jacobi, which does not have any dependences within one sweep over the sparse matrix. The BeBOP group at Berkeley has done an extensive amount of research on how to improve the serial [26] and parallel performance [27] of sparse matrix vector computations. One sweep, or outer iteration, of Jacobi is identical to the dependence pattern in a sparse matrix-vector multiply. The optimizations that they perform including sparse register tiling, sparse cache blocking, TLB blocking, reordering, and software prefetching are all complementary to an asynchronous parallelization approach.

V. Conclusions

Arbitrary task graphs are a powerful tool for expressing parallelism. Mapping arbitrary task graphs to existing models requires varying levels of effort with TBB providing the most straight-forward implementation. These graphs can be efficiently executed using existing parallel programming models. In general, most models deliver equivalent performance when the work grain is sufficiently large, leaving the choice of model to programmer preference. For fine grained parallelism, the overhead of task graph scheduling becomes considerable. In these cases, careful tuning of the number of threads and task length may be required to achieve high performance.

Acknowledgements

The authors gratefully acknowledge the use of Intel's Manycore Testing Lab. We also thank Samantha Wood of the University of California, San Diego for her contributions toward the algorithms used in this paper's evaluations. This project is supported by the CSCAPES Institute, which is supported by the U.S. Department of Energy's Office of Science through grant DE-FC-0206-ER-25774, as part of its SciDAC program, a Department of Energy Early Career Grant DE-SC0003956, by a National Science Foundation CAREER grant CCF 0746693, and by the Department of Energy CACHE Institute grant DE-SC04030.

References

[1] C. C. Douglas, J. Hu, M. Kowarschik, U. Rüde, and C. Weiss, "Cache optimization for structured and unstructured grid multigrid," *Electronic Tranactions on Numerical Analysis*, vol. 10, pp. 21–40, 2000.

[2] M. M. Strout, L. Carter, J. Ferrante, J. Freeman, and B. Kreaseck, "Combining performance aspects of irregular Gauss-Seidel via sparse tiling," in *Proceedings of the 15th Workshop on Languages and Compilers for Parallel Computing (LCPC)*, College Park, Maryland, July 2002.

[3] M. Mohiyuddin, M. Hoemmen, J. Demmel, and K. Yelick, "Minimizing communication in sparse matrix solvers," in *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, ser. SC '09. New York, NY, USA: ACM, 2009, pp. 36:1–36:12.

[4] M. S. Mohsen, R. Abdullah, and Y. M. Teo, "A survey on performance tools for OpenMP," in *Proceedings of the World Academy of Science, Engineering and Technology*, 2009.

[5] *Cilk++ SDK Programmer's Guide*, Intel Corporation, 2009.

[6] S. S. Shende and A. D. Malony, "The TAU parallel performance system," *The International Journal of High Performance Computing Applications*, vol. 20, pp. 287–331, 2006.

[7] K. Keutzer and T. Mattson, "Our Pattern Language (OPL): A design pattern language for engineering (parallel) software," in *ParaPLoP Workshop on Parallel Programming Patterns*, June 2009.

[8] "OpenMP (http://openmp.org)."

[9] L. Dagum and R. Menon, "OpenMP: An industry-standard API for shared-memory programming," *IEEE Computational Science & Engineering*, vol. 5, no. 1, pp. 46–55, 1998.

[10] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou, "Cilk: An efficient multithreaded runtime system," Cambridge, MA, USA, Tech. Rep., 1996.

[11] R. van der Pas, "An overview of OpenMP 3.0," in *International Workshop on OpenMP*, 2009.

[12] K. Agrawal, C. E. Leiserson, and J. Sukha, "Executing task graphs using work-stealing," *2010 IEEE International Symposium on Parallel Distributed Processing IPDPS*, pp. 1–12, 2010.

[13] T. L. Casavant, Jon, and G. Kuhl, "A taxonomy of scheduling in general-purpose distributed computing systems," *IEEE Transactions on Software Engineering*, vol. 14, pp. 141–154, 1988.

[14] Y.-K. Kwok and I. Ahmad, "Benchmarking and comparison of the task graph scheduling algorithms," 1999.

[15] ——, "Static scheduling algorithms for allocating directed task graphs to multiprocessors," *ACM Comput. Surv.*, vol. 31, no. 4, pp. 406–471, 1999.

[16] M. J. Wolfe, "Iteration space tiling for memory hierarchies," in *Third SIAM Conference on Parallel Processing for Scientific Computing*, 1987, pp. 357–361.

[17] F. Irigoin and R. Triolet, "Supernode partitioning," in *Proceedings of the 15th Annual ACM SIGPLAN Symposium on Priniciples of Programming Languages*, 1988, pp. 319–329.

[18] G. Quintana-Ortí, F. D. Igual, E. S. Quintana-Ortí, and R. A. van de Geijn, "Solving dense linear systems on platforms with multiple hardware accelerators," in *Proceedings of the 14th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPOPP)*. New York, NY, USA: ACM, 2009, pp. 121–130.

[19] M. M. Baskaran, N. Vydyanathan, U. K. R. Bondhugula, J. Ramanujam, A. Rountev, and P. Sadayappan, "Compiler-assisted dynamic scheduling for effective parallelization of loop nests on multicore processors," *PPOPP*, vol. 44, no. 4, pp. 219–228, 2009.

[20] A. Buttari, J. Langou, J. Kurzak, and J. Dongarra, "A class of parallel tiled linear algebra algorithms for multicore architectures," *Parallel Comput.*, vol. 35, no. 1, pp. 38–53, 2009.

[21] J. D. Hogg, J. K. Reid, and J. A. Scott, "A dag-based sparse cholesky solver for multicore architectures," Science and Technology Facilities Council, Tech. Rep., April 27, 2009.

[22] A. Chandramowlishwaran, K. Knobe, and R. W. Vuduc, "Performance evaluation of Concurrent Collections on high-performance multicore computing systems," in *Proceedings of the IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2010.

[23] R. Mirchandaney, J. H. Saltz, R. M. Smith, D. M. Nico, and K. Crowley, "Principles of runtime support for parallel processors," in *Proceedings of the 2nd International Conference on Supercomputing*, 1988, pp. 140–152.

[24] M. Mohiyuddin, M. Hoemmen, J. Demmel, and K. Yelick, "Minimizing communication in sparse matrix solvers," in *Supercomputing*. New York, NY, USA: ACM, 2009.

[25] M. F. Adams, "A distributed memory unstructured Gauss-Seidel algorithm for multigrid smoothers," in *SC2001: High Performance Networking and Computing. Denver, CO*, ACM, Ed., 2001.

[26] R. Vuduc, J. W. Demmel, K. A. Yelick, S. Kamil, R. Nishtala, and B. Lee, "Performance optimizations and bounds for sparse matrix-vector multiply," in *Proceedings of the ACM/IEEE Conference on Supercomputing*, 2002, pp. 1–35.

[27] S. Williams, L. Oliker, R. Vuduc, J. Shalf, K. Yelick, and J. Demmel, "Optimization of sparse matrix-vector multiplication on emerging multicore platforms," in *Proceedings of the 2007 ACM/IEEE Conference on Supercomputing*. New York, NY, USA: ACM, 2007, pp. 1–12.