A Prototype Implementation of OpenMP Task Dependency Support

Priyanka Ghosh, Yonghong Yan, Deepak Eachempati and Barbara Chapman

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

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

1 Introduction

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

Thus, the need for point-to-point synchronization among specific tasks is particularly important when dealing with applications that can be expressed

through a task graph, exhibit pipeline parallelism, or require wavefront synchronization. Computations in these types of problems exhibit a data dependency pattern within certain periods of its execution, and stand to benefit from the exploitation of fine-grained parallelism.

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

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

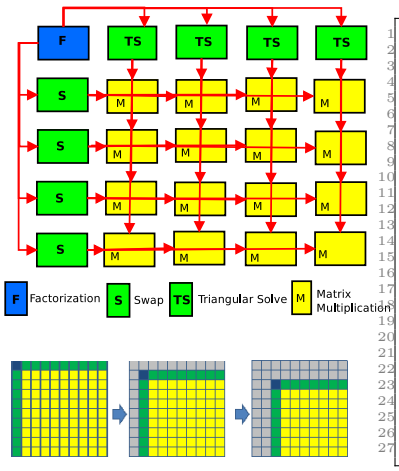
2 Motivation

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

2.1 LU decomposition

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

as shown in Listing 1.1. Such an approach constrains the execution order of those tasks, that is, each of the M blocks has to wait for the completion of all the S and TS blocks. Thus the algorithmic parallelism will not be fully exploited due to the lack of this particular language feature.




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

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

Listing 1.1: Tasking implementation for LU decomposition

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

2.2 Smith-Waterman algorithm

The Smith-Waterman algorithm is used primarily in the field of DNA and protein sequencing to determine similarities between biomolecule sequences according to a scoring system defined by a substitution matrix and gap penalty function. It employs a dynamic computational matrix technique that makes the algorithm more computationally intense, especially in the presence of data dependencies which restrict it from scaling well for parallel applications.

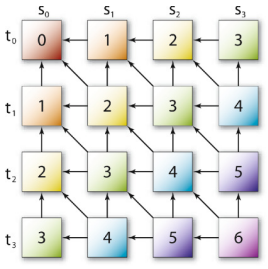


Fig. 2: 2D wavefront of smith-waterman algorithm [7]

Figure 2 represents the algorithm using 2D wavefront operations. Updating an element in the matrix requires updating the previous neighboring elements, resulting in a computation that resembles a diagonal sweep across the elements in the logical plane. Each element in the scoring matrix has three explicit dependencies on: a) its immediate north neighbor, b) its immediate west neighbor, and c) its immediate north-west neighbor.

The computations start at the extreme upper-left corner of the matrix and a gradual sweep moves along the diagonal down to

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

3 Approaches to handling *task* dependencies

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

3.1 The OmpSs programming model

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

3.2 QUARK runtime API

QUARK (QUeuing And Runtime for Kernels) [16] developed primarily at the University of Tennessee, provides a runtime environment which enables the dynamic execution of tasks with data dependencies in a multi-core, multi-socket, shared-memory environment. QUARK infers data dependencies and precedence constraints among tasks from the way that the data is being used, and then executes the tasks in an asynchronous, dynamic fashion in order to achieve a high utilization of the available resources. Even though the main focus for the development of the API was catered to support basic linear algebraic algorithms (BLAS) for the PLASMA [3] library, it is capable of supporting other data-driven

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

3.3 Extensions implemented in OpenUH compiler

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

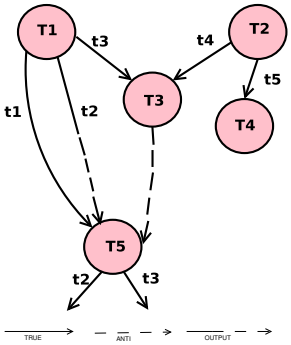
- *#pragma omp task* ***out*** *[t_1,t_2,....,t_n]* A generated task is denoted to have "*out*" dependence if it compute's variables required by succeeding tasks.
- *#pragma omp task* ***in*** *[t_1,t_2,....,t_n]* A generated task is denoted to have "*in*" dependence if it requires variables that have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

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

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

3.4 Comparison

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




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

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

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

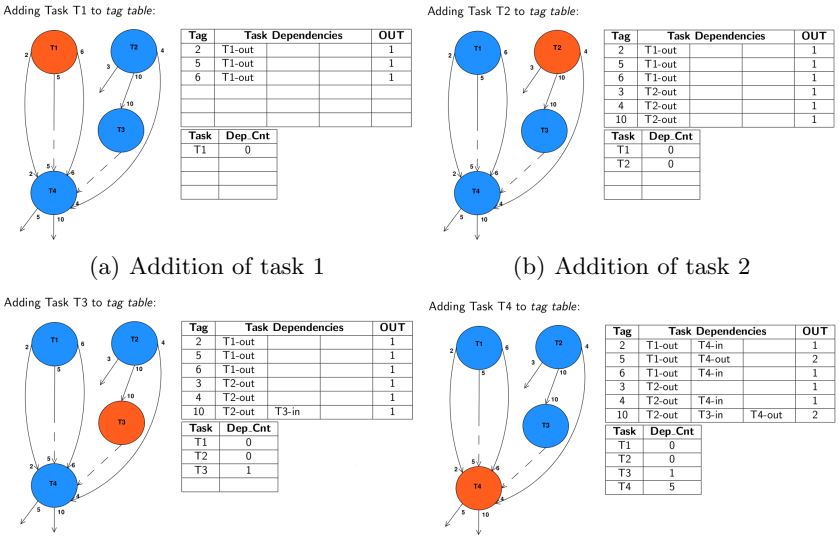
Secondly, OmpSs constructs a task dependency graph (DAG) and maintains a table data structure to store and manage the data dependencies among tasks, whereas OpenUH employs only a *tag table* (unordered hash map) to account for the same. Dependencies among tasks in OmpSs are expressed based on arbitrary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

4 Implementation of extensions in OpenUH

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

For instance, for greater control over task scheduling, the programmer can choose at runtime an appropriate task queue organization as well as control the order in which tasks are added/removed from a task queue.

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



(a) Addition of task 1

(b) Addition of task 2

(c) Addition of task 3

(d) Addition of task 4

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

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

The dependencies specified by the programmer are individually associated with a "*tag*". Each *tag* holds a unique entry in the parent table as a hash key. Its corresponding hash value is a linked list representation of all dependent tasks sharing the same *tag*. At any time when a task's *Dep_Counter* reaches zero, it has satisfied all its related dependencies and can now be placed in the task pool for it to be scheduled. However, if the *Dep_Counter* is not zero, it implies that it has to wait for the subsequent dependent tasks to finish execution before it

could be placed on the task pool. Once a task is placed in the task pool it is available for execution. Similarly tasks after execution need to be removed from the *tag table* at the time of task deletion. Before the tasks are removed from the table, it is essential that the *Dep_Counter* of its subsequent dependent tasks are updated accordingly, so that they may be scheduled for execution after their dependencies has been resolved.

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

5 Experimental results

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

5.1 Performance analysis for LU decomposition algorithm

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

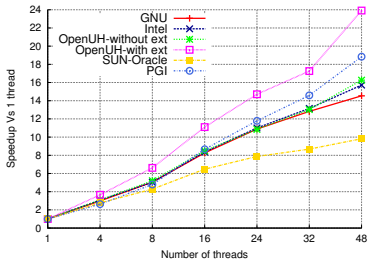
Listing 1.3 below represents the pseudo-code for implementing the LU decomposition algorithm using the OpenUH *task* extensions thereby eliminating the need to apply the first *taskwait*, depicted in Line 17 of Listing 1.1. The flexibility of task execution in the hands of the programmer warrants the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

Figure 5(a) measures the speedup obtained by varying number of cores, with 16 blocks per dimension on a matrix of size 4096 X 4096. We observe that the version implemented with the OpenUH task extensions scales well up to 24x on a single core. Figure 5(b) measures the performance obtained by varying the number of blocks per dimension (8,16,32 and 48), on 48 threads, wherein the matrices are partitioned into smaller blocks so that they could fit in memory registers and cache. This allows spatial locality to improve by increasing reuse of the data in cache memory and can be acknowledged as a significant

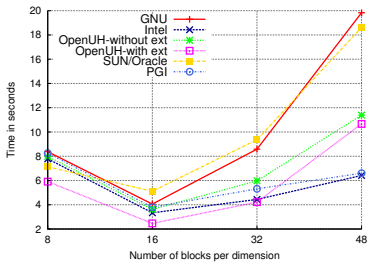
memory optimization uplifting code performance. We observe improvement in performance as we gradually increase the number of blocks per dimension. However, implementing 32 blocks per dimension degrades performance mainly due to the creation of excessive tasks, adding additional overhead attributed to task creation/deletion as well as task synchronization. Additionally by gradually decreasing the size of blocks we reduce the chances of achieving higher data reuse causing the performance to suffer.

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

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



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

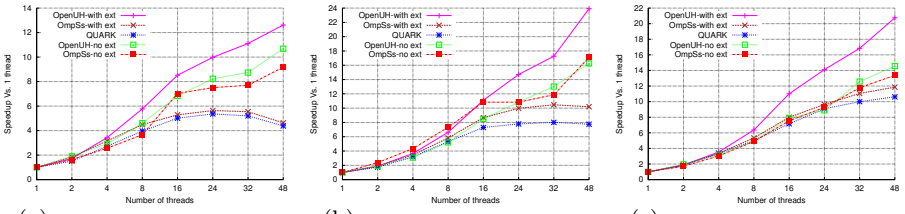


(b) Performance varying blocks per dimension-matrix size 4096 with -O3 optimization with 48 threads

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

Figure 6 allows us to make an assessment on how well OmpSs and QUARK scale in comparison to OpenUH, with varying matrix sizes. With the increase in size of input data, both OmpSs and QUARK show consistent improvement in terms of scalability. In the case of OpenUH, where, with the application of the extensions, we account for an average performance improvement of 20% (compared to the version excluding the extensions), there is a subsequent degradation in performance estimated close to an average 30%, observed for OmpSs. This may be attributed to that fact that the master thread invests significant amount of time in maintaining the task graph at runtime along with the overhead encountered by worker threads waiting for tasks to populate the ready pool [8]. We observe that QUARK scales poorly as well for smaller data sizes and the drop

in scalability for large number of cores can be attributed to overhead incurred due to use of broader thread mutex locks resulting in contention.



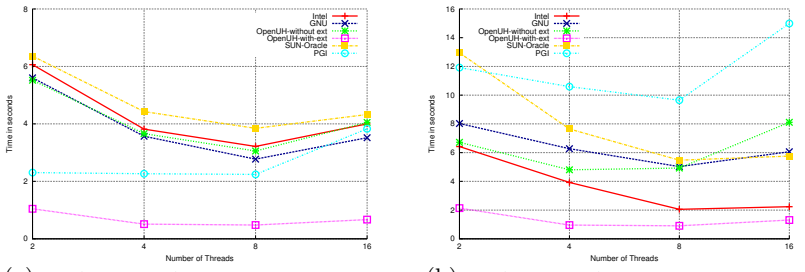
(a) matrix size 2048 X 2048 (b) matrix size 4096 X 4096 (c) matrix size 6144 X 6144

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

5.2 Performance analysis for Smith-Waterman algorithm

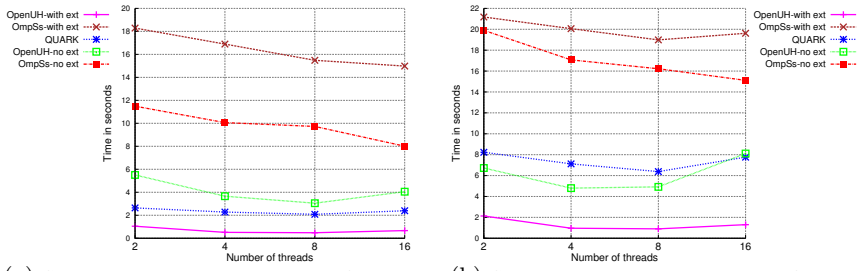
Our testbed for the following set of experiments comprises of a Dual Intel Nehalem - E5520 processor with 16 cores. 32GB of total memory capacity, L1 cache 32K, L2 cache 256K and an L3 cache of 8MB.

Figures 7(a) and 7(b) present the performance obtained for sequences of size 4096 and 8192, with -O0 and -O2 levels of optimization respectively, for a standard tasking version of the Smith-Waterman kernel where tasks have been created as chunks of 320 and 512 per diagonal. The version with the OpenUH task extensions outperforms the version without the use of the extensions by a factor of 6x. This improvement is attributed to the use of less constrained synchronization, allowing tasks to execute as soon as their respective dependencies have been satisfied. This performance is consistent even when tested with -O0 optimization validating that the improvement observed is due to the elimination of the *taskwait*, and without the interference of any other optimizations. Another observation suggests none of the compilers produce scalable results beyond 8 threads. This is attributed to the memory bound nature of the application, wherein tasks being very fine grained add additional overhead of task scheduling.



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

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



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

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

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

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

6 Related work

Other than the OmpSs and QUARK presented in Section 2, data-driven tasks (DDT) have been widely advocated by researchers to replace the use of potentially expensive global barriers [13]. Some of these efforts rely on compiler transformation to decompose data parallel computations into tasks with dependencies, and to achieve higher degree of overlap and concurrency between these tasks [14]. Other extension to task parallelism, described as data-driven futures (DDFs) in [12], allows users to create write-once tags as input and output events that could trigger other tasks. The write-once restriction, same as in the Intel Concurrent Collection [1] programming model for data-flow parallelism, simplify the programming logic and algorithms reasoning, as well as the runtime implementations, but may introduce overheads when handling a large number of tags requiring multiple read/write.

The standardization of the task dependency model in OpenMP represents a major step toward its adoption from research community to applications in

real world. We believe our work and other related efforts have demonstrated its effectiveness and usability toward this direction.

7 Conclusions and future Work

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

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

8 Acknowledgments

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

References

1. Intel Concurrent Collections . `http://software.intel.com/en-us/articles/intel-concurrent-collections-for-cc/`.
2. OpenMP 4.0 release candidate 2. `http://www.openmp.org/mp-documents/OpenMP_4.0_RC2.pdf/`.
3. E. Agullo, J. Demmel, J. Dongarra, B. Hadri, J. Kurzak, J. Langou, H. Ltaief, P. Luszczek, and S. Tomov. Numerical linear algebra on emerging architectures: The plasma and magma projects. In *Journal of Physics: Conference Series*, volume 180, page 012037. IOP Publishing, 2009.

4. Barbara Chapman, Deepak Eachempati, and Oscar Hernandez. Experiences developing the openuh compiler and runtime infrastructure. *International Journal of Parallel Programming*, pages 1–30, 2012.
5. Tamer Dallou and Ben Juurlink. Hardware-based task dependency resolution for the starss programming model. In *Parallel Processing Workshops (ICPPW), 2012 41st International Conference on*, pages 367–374. IEEE, 2012.
6. Frederic Desprez, Stéphane Domas, and Bernard Tourancheau. Optimization of the scalapack lu factorization routine using communication/computation overlap. In *Euro-Par'96 Parallel Processing*, pages 1–10. Springer, 1996.
7. A.J. Dios, R. Asenjo, A. Navarro, F. Corbera, and E.L. Zapata. Evaluation of the task programming model in the parallelization of wavefront problems. In *High Performance Computing and Communications (HPCC), 2010 12th IEEE International Conference on*, pages 257–264. IEEE, 2010.
8. A. Duran, J. Perez, E. Ayguadé, R. Badia, and J. Labarta. Extending the openmp tasking model to allow dependent tasks. *OpenMP in a New Era of Parallelism*, pages 111–122, 2008.
9. P. Ghosh, Y. Yan, and B. Chapman. Support for dependency driven executions among openmp tasks. Workshop on Data-Flow Execution Models for Extreme Scale Computing (DFM 2012) in conjunction with PACT, September 2012.
10. Azzam Haidar, Hatem Ltaief, Piotr Luszczek, and Jack Dongarra. A comprehensive study of task coalescing for selecting parallelism granularity in a two-stage bidiagonal reduction. In *Parallel & Distributed Processing Symposium (IPDPS), 2012 IEEE 26th International*, pages 25–35. IEEE, 2012.
11. Stephen L. Olivier, Bronis R. de Supinski, Martin Schulz, and Jan F. Prins. Characterizing and mitigating work time inflation in task parallel programs. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '12, pages 65:1–65:12, Los Alamitos, CA, USA, 2012. IEEE Computer Society Press.
12. Sağnak Taşırlar and Vivek Sarkar. Data-Driven Tasks and their Implementation. In *Proceedings of the International Conference on Parallel Processing*, Sep 2011.
13. S. Vajracharya, S. Karmesin, P. Beckman, J. Crotinger, A. Malony, S. Shende, R. Oldehoeft, and S. Smith. Smarts: Exploiting temporal locality and parallelism through vertical execution. In *Proceedings of the 13th international conference on Supercomputing*, pages 302–310. ACM, 1999.
14. T.H. Weng. *Translation of OpenMP to Dataflow Execution Model for Data locality and Efficient Parallel Execution.* PhD thesis, Department of Computer Science, University of Houston, 2003.
15. Yonghong Yan, Sanjay Chatterjee, Daniel A. Orozco, Elkin Garcia, Zoran Budimlić, Jun Shirako, Robert S. Pavel, Guang R. Gao, and Vivek Sarkar. Hardware and software tradeoffs for task synchronization on manycore architectures. In *Proceedings of the 17th international conference on Parallel processing - Volume Part II*, Euro-Par'11, pages 112–123, Berlin, Heidelberg, 2011. Springer-Verlag.
16. A. YarKhan, J. Kurzak, and J. Dongarra. Quark users guide: Queueing and runtime for kernels. *University of Tennessee Innovative Computing Laboratory Technical Report ICL-UT-11-02*, 2011.