Support For Dependency Driven Executions Among OpenMP Tasks

Priyanka Ghosh, Yonghong Yan, and Barbara Chapman
Department of Computer Science,
University of Houston
{pghosh06,yanyh,chapman}@cs.uh.edu

***Abstract*—OpenMP 3.x introduced the concept of creation of tasks with the aim of handling unstructured parallelism by providing support for efficient load-balancing and dynamic scheduling of asynchronous units of work, namely *tasks*. This work is focused towards extending task parallelism further to exploit task dependence relationships. The goal is to extend the OpenMP tasking model to provide more flexible synchronizations based on the data access relationship among such tasks. Our approach implements extensions to the current OpenMP task directive, which aim at providing task-level granularity for synchronization of tasks sharing the same parent. The new extensions focus primarily on adding additional functionality to the OpenUH compiler runtime referenced at the time of scheduling of tasks to honor only specified explicit dependencies. The extensions provide simplicity of use and help in achieving more parallelism owing to a more flexible approach in synchronizing tasks wherein a task waits only as long as its specified explicit dependencies are honored. The extensions when tested on a basic linear algebraic algorithm - LU Decomposition, presented a 9% and 20% speedup compared to the tasking implementations of Intel and GNU compilers respectively.**

I. Introduction

Task parallelism, as compared to data parallelism, refers to the explicit creation of multiple threads of control, or tasks, which synchronize and communicate under control of programmers. Conventionally, task parallelism is enabled to programmers through library APIs, notably *pthreads*. Recently, task parallelism has gained more attention in the multicore and manycore systems, and was introduced in several parallel programming languages because of its flexibility and productivity. The programming languages developed as part of the DARPA HPCS program (Chapel [1] and X10 [7]) identified task parallelism as one of the prerequisites for success.

Explicit task parallelism was introduced in OpenMP 3.0 for programming on shared-memory machines [4], [3]. A task is defined as an instance of executable code and its data environment, generated when a thread encounters a *task* construct. In OpenMP, the tasking concept, i.e. the implicit task, was a part of the OpenMP programming model from the beginning, for example, the OpenMP *section*. The use of OpenMP parallel region or worksharing constructs cause the creation of multiple implicit tasks in an SPMD style. We can imagine even the entire program as being enclosed within an implicit (and inactive) parallel region, such that the entire program itself can be thought of as a task.

The execution of a task can occur at any point between its creation in the program and the next task synchronization point, which is often identified by the *taskwait* or *barrier* construct. The synchronization points serve as a global barrier that causes the current execution to block until all the spawned tasks (implicit or explicit) have completed execution. In particular, there is no means to wait on a specific task, or tasks that access some specified data, to complete. The programmer is forced to make use of *taskwait* or *barrier*, which prevents the code from fully exploiting the concurrency that is possible for a given parallel algorithm. The overhead of synchronization between concurrent tasks typically presents itself as an obstacle to obtain high performance on problems that are characterized by a data dependency pattern across a data space, which can produce a variable number of independent tasks through the traversal of such space. Such behaviour is typically noticeable in task parallelizing LU factorization kernel as well wavefront based problems which mainly deal with DNA sequence alignment [8]. As previously stated, in this paper we present an approach that provides 9% speedup compared to the tasking implementation in the Intel compiler and a 20% speedup in comparison to the GNU compiler for the former problem.

The main contribution of this paper is to extend the OpenMP tasking model to provide more flexible synchronizations based on the data access relationship between asynchronous tasks. In order to support specifying such data dependencies among tasks, we propose certain extensions to the current OpenMP tasking directive that allow runtime detection of dependencies among sibling tasks by introducing the the notion of unique task object synchronizers of the type – *output* and *input*. Our discussion on the implementation highlights the core algorithms used to resolve those dependencies, without the involvement of global barrier or lock operations that limit the scope for improved scalability due to increased data synchronization overheads.

The rest of the paper is organized as follows: Section II introduces and describes the features of the proposed extensions. Section III describes the algorithm employed to implement the extensions in the OpenUH compiler and OpenMP runtime library. Section IV presents our experimental results obtained for LU Decomposition algorithm. Section V discusses the related work. Finally, Section VI contains our conclusions and future work.

II. Definition of proposed extensions.

The solution to the scheduling of dynamic tasks comprises of determining firstly *when* a given task may execute, and secondly determining *where* a task may execute. The present OpenMP task model provides limited means for a programmer to control scheduling of tasks created within a parallel region. Addressing the latter problem, there exists a one-to-one mapping of the implicit tasks to the threads constituting the regions thread team, but the explicit tasks created in the region may execute on any thread in the team. Addressing the former problem as to *when* a task may execute, it could occur at any point between its creation in the program and the the next taskwait or barrier construct (whichever is encountered first). Thus the order of task executions are solely governed by the global synchronization points created with those constructs. Finer grained control and manipulation of ordering of specific tasks have to involve those global operations, which introduce unnecessary overhead in most cases.

We address such limitations by providing three clauses to the OpenMP *task* construct. Taken together, these extensions allow the programmer to express point-to-point synchronizations between sibling tasks of the same parent in an OpenMP program. The extensions aim at enhancing synchronization among tasks. By synchronization we mean to recognize and control data dependency which is necessary to co-ordinate tasks that are modifying and reading a shared resource. If the tasks do not access a shared resource they are considered independent.

Similar extensions to the OpenMP tasking model [9] has been proposed by the Barcelona Supercomputing Center in 2009. However our approach may be considered simplistic and expressive in terms of implementation of task-level granularity that supports a finer grained synchronization, based on the data dependences. This flexible scheduling of computations to available resources also accounts for improved load balancing.

We introduce the notion of "tags" or synchronization object identifiers among sibling tasks. These "tags" could 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). Following are the extensions proposed and the Listing 1 explains very briefly the functionality of the extensions:

1) *#pragma omp task* ***out*** *[(data reference list)]* A task is proposed to have "*out*" dependence if a task might compute variables required by succeeding tasks.
2) *#pragma omp task* ***in*** *[(data reference list)]* A task is proposed to have "*in*" dependence if it requires variables that have been computed previously.
3) *#pragma omp task* ***inout*** *[(data reference list)]* The combination of the above two in an easy format.

In the Listing 1, *task 3*, due to the existence of an *in* dependence for tag 1, will have to wait until *task 1* and *task 2* finish executing who have tag 1 as *out* dependencies. Similarly, *task 4* has to wait only until *task 2* completes execution and can be executed in parallel with both *task 3* and *task 1*. In the absence of our extensions, to maintain the correct order of task execution, we may insert a *taskwait* clause between *task 2* and *task 3*. If so, *task 4* would have to wait for a global synchronization point and will not be scheduled in parallel with *task 1*.

Listing 1: sample code quoting the proposed extensions.

```
#pragma omp parallel num_threads(2)
 {
#pragma omp master
  {
   #pragma omp task out(1) /* task 1 created */
        x = f1();

   #pragma omp task out(2) /* task 2 created */
        y = f2();

   #pragma omp task in(1) in(2) out(3) /* task 3 created */
        z = x * y;

 /* task 3 has to wait for task 1 and task 2 to
    complete execution before it can be scheduled */

   #pragma omp task in(2) out(4) /* task 4 created */
        w = x / y;

 /* task 4 has to wait on only task 2.
  Hence it can scheduled in parallel with task 3 */
  }
 }
```

Before we explain the details of implementation of the proposed extensions, it is important to enumerate the different categories of dependencies the extensions are capable of handling, and explain the semantics of using those extensions.

Fig.1 represents the three dependence relationships that may exist among the several different tasks. T_1 to T_4 represents four such tasks. The numbers adjacent to each task, are the synchronization objects, namely *tags*, that were introduced in the previous section. Taking Fig.1 into consideration, we highlight the typical data dependencies encountered at the time of parallelizing scientific algorithms.

1) *True/Flow Dependence:* A *True* dependence is encountered between statement S_1 and S_2 when the former sets a value that the latter uses. This dependence described as the '*read after write* (RAW)' dependence. In figure 1, we encounter four true dependencies namely:
   a) Between T_1 and T_4 for tag 2
   b) Between T_1 and T_4 for tag 6
   c) Between T_2 and T_4 for tag 4
   d) Between T_2 and T_3 for tag 10
2) *Anti Dependence:* An *Anti* dependence is encountered between statement S_1 and S_2 when the former uses a value that the latter defines. This dependence described as the '*write after read* (WAR)' dependence. In figure 1, we notice an *anti* dependence between T_3 and T_4 for

tag 10.

3) *Output Dependence:* A *Output* dependence is encountered when both statements S_1 and S_2 define the value of some variable. In figure 1, an *output* dependence is noticed for tasks T_1 and T_4 for tag 5.

The true dependence is the relationship that forms the data-flow execution pattern for tasks. In our extensions, we also allow both anti and output dependency to exist and the runtime systems will ensure the execution of those dependent tasks to follow the order they appear in the program (or being created). This could be considered one approach to reuse the tag for new relationships.

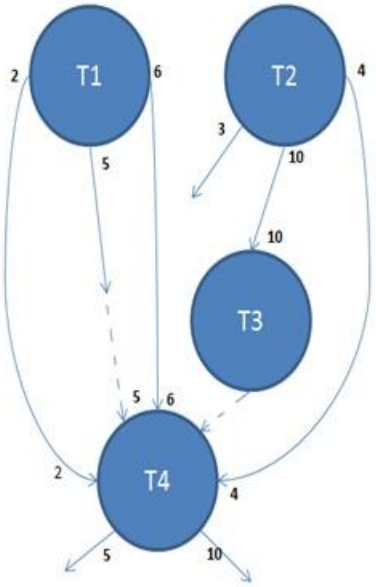

Fig. 1: Dependency graph with *true, anti* and *output* dependencies. Each task has been labeled with its respective *tags* establishing the existence of dependence relationships among them

III. Implementation

In this section we describe the algorithm implemented at the OpenUH runtime at length. Table 1 introduces a few terminologies that we shall be referring to throughout the rest of the paper.

The OpenUH compiler [5] is a branch of the open source Open64 compiler suite for C, C++, Fortran 95/2003. OpenUH includes support for OpenMP 3.x tasks. This consists of front-end support ported from the GNU C/C++ compiler, back-end translation implemented jointly with Tsinghua University, and an efficient task scheduling infrastructure developed in the runtime library that holds support for improved nested parallelism. It implementation supports a configurable task pool framework that allows the user to choose at runtime an appropriate task queue organizations to use as well as control the order in which tasks are removed from a task queue for greater control over task scheduling. Explicit details regarding the scheduling of tasks at the OpenUH runtime will be explained in the next section.

Terminology	Meaning
tag	the unique synchronization identifier associated with an output and corresponding input dependence relation.
tag/parent table	A hash table representation, with the *tags* acting as hash keys and a list of input and output dependencies associated with that particular *tag* acting as the corresponding hash value. Hence one entry in the tag table may hold dependence information of one or many tasks depending upon the data dependence relationship.
Dependency list	The corresponding hash value in parent table that represents the list of dependencies (*in* or/and *out*) associated with the same *tag* (hash key).
OUT counter	A counter that keeps track of the number of OUT dependencies associated with each *tag*. Hence every entry in the hash table has an associated OUT counter that is incremented when an OUT dependence is introduced in the table for a particular tag. It is decremented every time an OUT dependence is honored and is deleted from the table.
Dep_Count	A dependency counter flag associated with each task. When Dep_Count = 0, means the task has no associated dependencies and is free to be scheduled. When Dep_Count != 0, means the task has to wait until its associated dependencies have been honored. This counter is incremented every time an OUT/IN dependence is introduced in the table for a given *tag*, and decremented when the dependence is honored and removed from the table.
Dep_List	A list of IN/OUT dependencies associated with each task, This is useful especially at the time of task deletion in order to verify if all dependencies of current tasks have been honored If so, it decrements the Dep_Count of the subsequent dependent tasks.

TABLE I: Important Terminology

A. Dependency Setup

Algorithm 1 explains the procedure with which the runtime library interprets the dependencies parsed at the compiler end at the time of task creation. In the absence of the extensions, the OpenUH runtime environment employs a scheduling strategy that places a newly created task immediately into a task pool, to be executed by the next available resource. With the proposed extensions implemented, this behaviour is altered wherein only when a task with its corresponding *Dep_Count* equivalent to zero is allowed to be placed in the task pool. 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. 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

(uniquely identifiable). Its corresponding hash value is a linked list representation of all dependant 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.

ALGORITHM 1: Addition of *tags* - Task Creation

Input: # *pragma omp task OUT* (l_1,l_2,..,l_n) *IN* (w_1,w_2,..,w_n)
Dep_Count = 0;
repeat
  for *(All IN and OUT dependencies specified for a given task T: Dep)* **do**
    Access the global hash parent table: PTable
    if tag:*Dep found in PTable ;*
    then
      - Append dependency information to the dependency list (hash value) of *tag*: *dep* ;
      - Add dependency information to *Dep_List* for the task;
    else
      create *tag* TAG (hash key) in PTable ;
      if *Dep(TAG) == OUT* **then**
        1) OUT_counter += 1 for TAG;
        2) backtrace dependency list until another OUT dep is encountered;
        3) count the IN dep's in between (t_count);
        4) Dep_Count += t_count (of T);
      end
      if *Dep(TAG) == IN and OUT_counter != 0* **then**
        Dep_Count += t_count (of T);
      end
      - Append dependency information to the dependency list (hash value) of *tag*: *dep* ;
      - Add dependency information to *Dep_List* for the task;
    end
  end
until (l_n + w_n) == *0*;
if *Dep_Count of task T == 0* **then**
  release T for execution;
end

Fig.2 represents the state of the parent table (and *Dep_Counter*) at runtime, after all the tasks in Fig.1 have been created.

B. Dependency Resolution

Algorithm 2 illustrates the procedure with which an exiting task removes the requisite dependency information from the parent table after it has finished execution. This allows for subsequent tasks which had been waiting (in the WAIT state) for the current task to complete execution to be scheduled.

Parent Table

Task Tag	Task Dependencies			OUT
2	T1-out	T4-in		1
5	T1-out	T4-out		2
6	T1-out	T4-in		1
3	T2-out			1
4	T2-out	T4-in		1
10	T2-out	T3-in	T4-out	2

Task	Dep_cnt
T1	0
T2	0
T3	1
T4	5

Fig. 2: State of Parent table and *Dep_Counter* for tasks created in Fig.1

ALGORITHM 2: Deletion of *tags* - Task Exit

Input: exiting task T's Dep_List
repeat
  Node = Dep_List(T)->*tag* ;
  Access corresponding *Node* in PTable ;
  if *(Node(Dep) == IN)* **then**
    - forward traverse dependency list (hash value of *tag*) until an OUT dep is encountered:T_out;
    - Dep_Count(T_out->task) -= 1;
    - **if** *(Dep_Count(T_out->task)) == 0* **then**
      schedule task for execution;
      end
  end
  if *(Node(Dep) == OUT)* **then**
    - OUT_counter(*Node*) -= 1;
    - **while** *(An OUT dep not encountered on forward traversal of dependency list)* **do**
      1) Dep_Count(intermediate task) -= 1;
      2) **if** *Dep_Count(intermediate task) == 0* **then**
        schedule task for execution;
        end
      end
  end
  Next_Node = *Node*->next ;
  remove *Node* ;
  Node = *Next_Node* ;
until *Dep_List(T) == NULL*;

Abiding by the scheduling strategy explained in Algorithm 1 and 2, tasks T_1 to T_4 (represented in Fig.2) will be scheduled in the following order:

1) Task T_1 has three associated *tags* 2,5 and 6, all of which have a *Dep_Count* with value zero. This is owing to the fact that task T_1 has three *out* dependencies (identified by *T1-out* in the parent table) that form the first node in the *Dep_List* and hence free from prior dependencies.

Similarly task T_2 is associated with *tags* 3,4 and 10. Both tasks T_1 and T_2 can hence be released to the task pool and are in a position to be scheduled concurrently.

2) Task T_3, after being created has to wait until task T_2 completes execution owing to a single dependency (i.e. T_2) attributing to its *Dep_Count*. T_2 decrements the counter as soon as its finishes execution allowing T_3 to be placed on the task pool.
3) Task T_4 is accountable for three true dependencies (*tags* 2,4,6), one output dependence (*tag* 5) and one anti-dependence (*tag* 10) thereby setting the value of the *Dep_Count* to 5. T_4 is thus placed in the pool only after the execution of task T_3.

C. Efficient Handling of Tag Table and Tag Entry

In order to support the development of systems that avoid unsafe data race conditions, it is imperative to implement an efficient locking strategy to ensure atomic access to a shared resource. Locking costs introduces overheads and may limit scalability of a given implementation owing to contention of resources. The granularity of the locking mechanism is also vital. A global lock incurs far more costs than a lock on a local data structure. 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. The parent table is locked only if a new *tag* is being inserted in the hash table and not at the time of searching for existing *tags* in the table or even while appending related dependencies to the *Dependency List* (linked list associated with each *tag*, representing the corresponding hash value). This considerably limits the amount of overhead that would normally have been introduced into the system if the parent table had been entirely locked at the time of its updation.

IV. Experimental Results

In this section we shall discuss the experimental results obtained on testing the extensions on a benchmark application - LU Decomposition.

LU decomposition is a well studied algorithm for uniprocessor and multiprocessor systems. It is frequently used to characterize the performance of high-end parallel systems used in LAPACK benchmarks. Other than the conventional blocking algorithms, there are other algorithms studied to improve the performance and scalability of LU decomposition, such as the dynamic blocking [13], and the pipeline processing [10]. In linear algebra, LU Decomposition involves factorizing a matrix as a product of a lower triangular and upper triangular matrix. It is widely used in solving a system of linear equations, matrix inversion or computing the determinant of a matrix. The problem definition is as follows:

$$A = L * U \tag{1}$$

where L is a lower triangular matrix and U is an upper triangular matrix.

We focus on a blocked version of the algorithm that takes advantage of the fact that higher performance can be obtained by fitting smaller chunks of data sets in cache memory by employing a divide and conquer strategy.

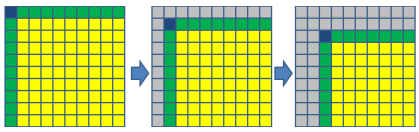

Fig. 3: Progress of Blocked LU decomposition per iteration [15]

With the assistance of Fig.3, [15] we provide a more detailed explanation of the algorithm:

An N X N matrix is divided into M X M equal blocks where M << N. Each of the M iterations have to essentially execute the following steps as demonstrated in Fig.3, considering each block to be an explicit OpenMP task.

1) Computation of the top-left corner block (in blue).
2) Computation of the first row and column blocks (green) only after step 1.
3) Computation of the rest of the blocks (yellow) based on the results obtained from step 2.

In the next iteration, blocks processed in step 3 of the previous iteration become the target of calculation as shown in Fig.4. In Fig.4 we can clearly observe the dependence relations where each arrow illustrates an existent data dependence across neighbouring blocks.

Synchronization between steps of the same iteration use barriers (such as *taskwait*) to avoid data races. The existence of such data dependencies hurts the performance and scalability of the algorithm especially for large data sets.

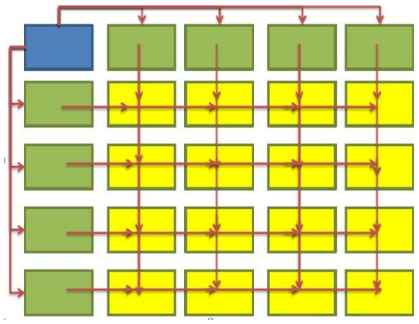

Fig. 4: A single iteration highlighting data dependence relations across blocks [15].

Our testbed comprises of a Dual Intel Nehalem - E5520 processor having a total of 16 cores. The performance results have been obtained for the following compilers:

1) GNU (gcc) compiler version 4.6.3
2) Intel (icc) compiler version 12.0
3) OpenUH compiler v3.0 without support for proposed extensions
4) OpenUH compiler v3.0 with support for proposed extensions

Fig.5 represents the performance results obtained on performing LU Decomposition on a matrix of size 2048

with 16 threads. As seen, the extensions provide improved performance with increasing number of blocks per dimension in comparison to GNU and Intel compilers. This could be attributed to a more flexible distribution of the workload amongst the threads owing to an improved scheduling strategy. The most optimum result is obtained for the version of OpenUH with the implemented extensions with 16 blocks per dimension.

In comparison to Fig.5, the optimum result for Fig.6 representing the performance results for LU decomposition on a matrix of size 4096 with 16 threads, is gathered with 32 blocks per dimension. Hence we can infer that with larger data sizes, it is appropriate to divide the workload evenly across the threads with a larger number of blocks to obtain better performance.

Similarly Fig.7 represents the performance results obtained on performing LU Decomposition on a matrix of size 8192 with number of block sizes varying from 8 to 64 on 16 threads. The choice for the number of blocks in order to obtain the optimum performance depends on the size of the last level cache. Performance results for Fig.7 indicate that use of 32 blocks is slightly better than usage of 64 blocks per dimension. Hence, it is necessary to take into account the trade-off between even distribution of workload with appropriate block size and the available cache size.

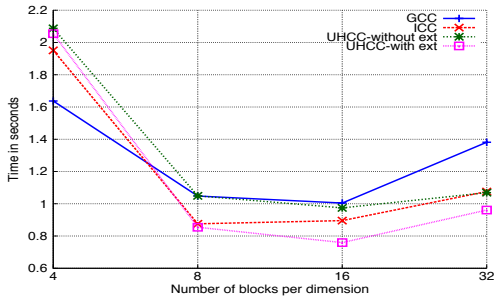

Fig. 5: Performance results with matrix size 2048 X 2048

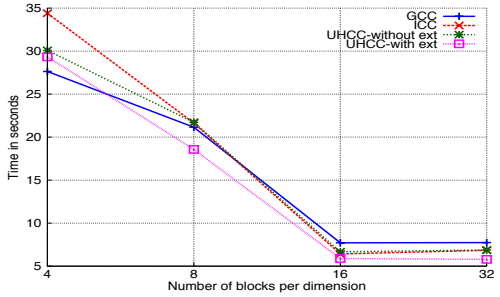

Fig. 6: Performance results with matrix size 4096 X 4096

Fig.8 measures the performance scalability of implemented extensions for 16 blocks per dimension on a matrix of size 4096. Once again we notice that the implemented extensions provide improved performance with a speedup of 1.26X and 1.10X in comparison to the GNU (gcc) and Intel (icc) compilers respectively. This is attributed to the effective load balancing of work with the increase in the number of threads. It is to be understood that an appropriate balance between data size and the number of blocks per dimension is to be maintained in order to obtain optimum performance.

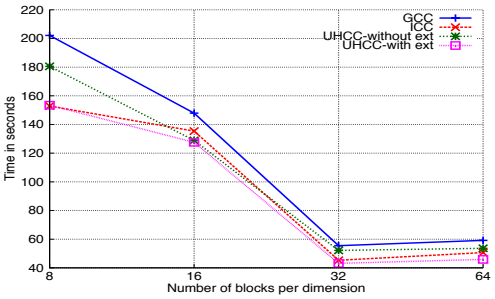

Fig. 7: Performance results with matrix size 8192 X 8192

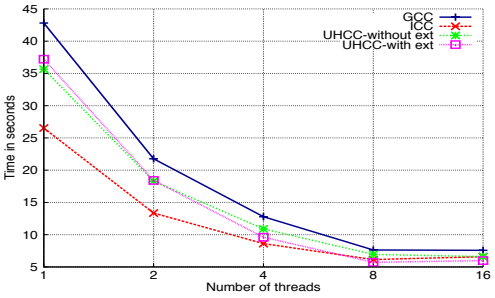

Fig. 8: Scalability across varying threads with matrix size 4096

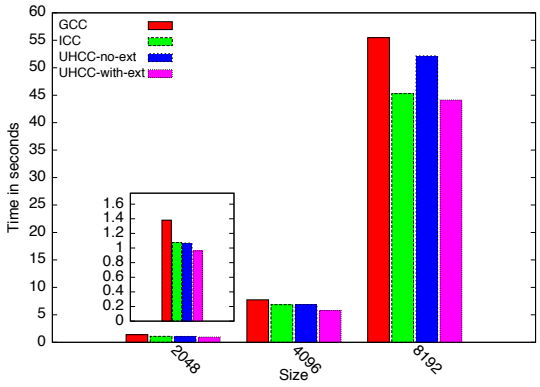

Fig. 9: Performance results varying matrix sizes with 32 blocks per dimension

Fig.9 represents a summary of the performance results obtained across varying matrix data sizes from 2048 to 8192.

As projected, we notice a consistent improvement in performance with the application of the proposed extensions in comparison to GNU and Intel compilers. Thus the extensions are a promising technique that performs reasonable well across varying workloads.

V. Related Work

For task and data parallel programs, many researchers have advocated Data-Driven Tasks (DDT) that can help avoid potentially expensive global barriers, and have shown that their use can lead to improved performance [12], [14]. These efforts rely on compiler transformation and runtime scheduling to decompose task and data parallel computations into tasks with dependencies, and to achieve higher degree of overlap and concurrency between these tasks. It does not require users to explicitly specify the data dependency, which helps on migrating legacy applications to data flow model. Yet the effectiveness of this automatic approach depends solely on the quality of the compilers and runtimes. Another extension to task parallelism, described as Data-Driven Futures (DDFs) in [11], 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 [2] 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 Barcelona SuperComputing Center has also proposed similar extensions to the OpenMP tasking model that involved the runtime detection of dependencies between generated tasks [9]. The authors proposed to extend the *taskwait* directive to *taskwait on* (data-reference-list), where a task waits on the completion of computation associated with the output or inout matching variable in the data-reference-list.

VI. Conclusions and Future Work

In this paper we presented an approach that aimed at handling fine grained inter-task synchronizations in a more dynamic and flexible manner. The proposed extensions implemented at the OpenUH compiler runtime library provide support for the specific orderings between tasks. Thereby producing a point-to-point synchronization between tasks that is needed to enable the efficient expression of algorithms that rely on patterns such as wavefront propagations and pipeline parallelism. Preliminary experiments conducted on pipeline algorithms such as LU decomposition, exhibited the marked improvement in performance brought due to primarily reduction of synchronization overheads when dealing with larger input data sizes owing to the creation of a larger number of tasks.

As future work we would also focus on implementing compiler optimizations to coalesce tasks performing similar operations in quick succession to coarsen the granularity of the tasks. This will contribute towards reducing the overhead of task creation for numerous tasks. In another direction, we will also be exploring the benefits of task decomposition within the OpenUH compiler infrastructure, by entirely decomposing an OpenMP program into a collection of tasks and represent it in the form of dependency graph [6]. This information could then be used to assist the user in placing the *in* and *out* dependence information at specific points in the program. After the initial tasks have been identified, task cost modeling information maybe be used to re-factor the tasks in the graph in a such a way that is most suitable for scheduling them onto the available hardware.

VII. Acknowledgements

We would like to sincerely thank Deepak Eachempati for his guidance and support in conceptualization and execution of the ideas presented in this paper. We would also like to extend a word of thanks to Sayan Ghosh for thoroughly reviewing our paper and suggesting corrections.

References

[1] Chapel Programming Language. http://chapel.cray.com/.

[2] Intel Concurrent Collections . http://software.intel.com/en-us/articles/intel-concurrent-collections-for-cc/.

[3] C. Addison, J. LaGrone, L. Huang, and B. Chapman. Openmp 3.0 tasking implementation in openuh. In *Open64 Workshop at CGO*, volume 2009, 2009.

[4] E. Ayguadé, N. Copty, A. Duran, J. Hoeflinger, Y. Lin, F. Massaioli, X. Teruel, P. Unnikrishnan, and G. Zhang. The design of openmp tasks. *Parallel and Distributed Systems, IEEE Transactions on*, 20(3):404–418, 2009.

[5] B. Chapman, D. Eachempati, and O. Hernandez. Experiences developing the openuh compiler and runtime infrastructure.

[6] Barbara Chapman. Toward optimization of openmp codes for synchronization and data reuse.

[7] Philippe Charles, Christian Grothoff, Vijay Saraswat, Christopher Donawa, Allan Kielstra, Kemal Ebcioglu, Christoph von Praun, and Vivek Sarkar. X10: an Object-Oriented Approach to Non-Uniform Cluster Computing. In *OOPSLA '05: Proceedings of the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 519–538, New York, NY, USA, 2005. ACM.

[8] 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.

[9] 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.

[10] Panagiotis D. Michailidis and Konstantinos G. Margaritis. Implementing parallel lu factorization with pipelining on a multicore using openmp. In *Proceedings of the 2010 13th IEEE International Conference on Computational Science and Engineering*, CSE '10, pages 253–260, Washington, DC, USA, 2010. IEEE Computer Society.

[11] Sağnak Taşırlar and Vivek Sarkar. Data-Driven Tasks and their Implementation. In *ICPP'11: Proceedings of the International Conference on Parallel Processing*, Sep 2011.

[12] 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.

[13] Ioannis E. Venetis and Guang R. Gao. Mapping the lu decomposition on a many-core architecture: challenges and solutions. In *Proceedings of the 6th ACM conference on Computing frontiers*, CF '09, pages 71–80, New York, NY, USA, 2009. ACM.

[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] Y. Yan, S. Chatterjee, D. Orozco, E. Garcia, J. Shirako, Z. Budimlic, V. Sarkar, and G. Gao. Synchronization for dynamic task parallelism on manycore architectures. 2010.