SUPPORT FOR DEPENDENCY DRIVEN EXECUTIONS AMONG OPENMP TASKS

A Thesis

Presented to

the Faculty of the Department of Computer Science

University of Houston

In Partial Fulfillment

of the Requirements for the Degree

Master of Science

By

Priyanka Ghosh

December 2012

SUPPORT FOR DEPENDENCY DRIVEN EXECUTIONS AMONG OPENMP TASKS

Priyanka Ghosh

APPROVED:

Dr. Barbara Chapman, Chairman
Dept. of Computer Science

Dr. Weidong Shi
Dept. of Computer Science

Dr. Dennis Adams
Dept. of Decision and Information Sciences

Dean, College of Natural Sciences and Mathematics

Acknowledgements

I would like to express my gratitude to my advisor, Dr. Barbara Chapman for providing an excellent environment for research. It was an absolute privilege to work under her tutelage, inspiring me to explore ideas for prospective research topics, obtaining constructive feedback on my performance, and providing me the opportunity to attend conferences in order to interact, gather, and share ideas with fellow members of the HPC (high performance computing) community.

I would also like to thank my mentor, Dr. Yonghong Yan. Without his unrelenting support, guidance, positive energy, and patience, this project would not have been possible. I would like to extend a special word of thanks to Deepak Eachempati for his guidance and support in conceptualization, execution and revision of the ideas presented in this thesis.

A very special thank you to my family, especially my Mum, Dad, Sister, and Sayan, for inspiring confidence and determination in me to never give up and strive to achieve the very best in all my endeavors.

A word of thanks to all my colleagues in the HPCTools lab, for all their suggestions which helped me refine the content of this thesis, as well as lending their time to review and provide feedback on the material.

I would like to acknowledge the contribution of Anibal Maldonado Agosto, a summer intern at the HPCTools lab.

Lastly, I would like to thank the Department of Computer Science for providing me the opportunity to pursue research in this distinguished university.

SUPPORT FOR DEPENDENCY DRIVEN EXECUTIONS AMONG OPENMP TASKS

An Abstract of a Thesis

Presented to

the Faculty of the Department of Computer Science

University of Houston

In Partial Fulfillment

of the Requirements for the Degree

Master of Science

By

Priyanka Ghosh

December 2012

Abstract

In order to improve its expressivity with respect to unstructured parallelism, OpenMP 3.0 introduced the concept of *tasks*, independent units of work that may be dynamically scheduled and hence support efficient load-balancing. Task synchronization is primarily accomplished via the insertion of *taskwait* and *barrier* constructs. However, these are global synchronizations and may incur significant overhead on large platforms. The performance of certain algorithms may benefit substantially if finer grained synchronization mechanisms were available. In this thesis, we extend the OpenMP tasking model to allow point-to-point synchronization among tasks in an OpenMP program. Such an approach enables us to provide support for a dataflow model within OpenMP.

We propose language extensions to the current OpenMP task directive that enable the specification of task-level granularity for synchronization of asynchronous tasks sharing the same parent. A task waits only until the explicit dependencies as specified by the programmer are satisfied, thereby avoiding the use of expensive global synchronization points. The extensions are simple to use and promise an increase in the achievable concurrency for some parallel algorithms.

We have implemented our ideas fully within the OpenUH OpenMP runtime library. The application of the extensions on two algorithms, LU Decomposition and Smith-Waterman, demonstrated significant performance improvement over the standard tasking versions of the two algorithms using the GNU, Intel, OpenUH, PGI, Oracle/Sun, and the Mercurium compilers. We compared our results with those obtained using related dataflow models - OmpSs and QUARK, and observed that the versions using our task extensions obtained an average speedup of 2 - 6X.

Contents

List of Figures

List of Tables

Chapter 1

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 the control of programmers. Conventionally, task parallelism is enabled to programmers through library APIs, notably pthreads [32]. Recently, task parallelism has gained more attention in the multi-core and many-core systems, and was introduced in several parallel programming languages owing to the flexibility and productivity it offers, with respect to handling unstructured parallelism in programs. The programming languages developed as part of the DARPA HPCS program [21, 26] (Chapel [1] and X10 [15]) identified task parallelism as one of the prerequisites for success, on highly parallel platforms.

Explicit task parallelism was introduced in OpenMP 3.0 for programming on shared-memory machines [6, 10]. In this context, 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 concept of tasks, i.e. the implicit task, is pervasive. The standard defines that, the use of of an OpenMP parallel region or worksharing construct, causes the creation of multiple implicit tasks in an SPMD style (Single Program Multiple Data), which distributes the workload and runs independently on the different threads. This approach potentially guarantees better performance and was pioneered by Tien-hsiung Weng in 2002 [39].

1.1 Problem Statement

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 may prevent the code from fully exploiting the concurrency achievable for certain parallel algorithms. The overhead of global synchronization among 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, producing a variable number of independent tasks through the traversal of such space.

Such behavior is typically characteristic to task parallelizing dense linear algorithms such as LU factorization [20] or Cholesky kernels as well as wavefront based

problems dealing with DNA sequence alignment [19].

1.2 Thesis Contribution

- We extend the OpenMP tasking model by proposing language extensions to the current OpenMP task construct, and implementing them in the OpenUH OpenMP compiler runtime library. The extensions embody the principles of a dataflow model, facilitating a more flexible approach to the ordering of asynchronous tasks based on the data access relationship among them.

- We allow the runtime detection of dependencies among sibling tasks by introducing the the notion of unique task object synchronizers of the type *output* and *input*. We highlight the two core algorithms proposed to track and resolve the data hazards among tasks at runtime. The implementation of the algorithms in the OpenUH OpenMP runtime library, at the time of *task* creation and *task* exit, limit the use global lock operations, alleviating the possibility of contention among resources.

- We perform a comparative study, comparing the results obtained with the application of our extensions on two target algorithms, in terms of performance improvement observed, the ease of programmability and average speedup obtained, with respect to related academic projects embodying similar principles of a dataflow model.

1.3 Thesis Organization

This thesis is organized into the following sections as described below:

- Chapter 2 provides the background on the origin and advantages of using a dataflow model. It describes the current research initiatives in the field of dataflow model construction and use.

- Chapter 3 provides a brief introduction to the OpenMP programming model and moves on to describes it advantages and caveats. We then introduce the OpenMP tasking model and discuss the motivation for this thesis.

- Chapters 4 and 5 introduce and describe the features of the proposed extensions to the OpenMP task construct, the target applications which would benefit with application of the extensions. It also describes the algorithm employed to implement the extensions in the OpenUH OpenMP runtime library.

- Chapter 6 presents the performance analysis and experimental results obtained after the application of the extensions to our target applications namely, LU Decomposition and Smith-Waterman algorithms. We also compare these results with those obtained from related dataflow models, OmpSs and QUARK.

- Chapter 7 contains our conclusion and future work.

Chapter 2

Related Work

2.1 Introduction

A dataflow model of computation offers an alternative to the conventional control flow model in extracting parallelism from programs, owing to a higher degree of achievable concurrency. Firstly, the dataflow model of execution is asynchronous, i.e., the execution of an instruction is based on the availability of its operand(s), in contrast to control-flow instructions, which are executed sequentially under the control of a program counter. Secondly, instructions in the dataflow model do not impose any constraints on sequencing except in the presence of data dependencies within the program. Therefore, the dataflow graph representation of a program exposes all forms of parallelism eliminating the need to explicitly manage parallel execution of a program. For high speed computations, the advantage of the dataflow approach over the control flow method stems from the inherent parallelism embedded

at the instruction level. This allows efficient exploitation of fine-grain parallelism in application programs. A dataflow model hence provides an elegant solution to the two fundamental problems of a Von Neumann (sequential control flow computing) computer: memory latency and synchronization overhead.

2.2 History of Dataflow Models

Research on dataflow models emerged in the early 1970's with the use of dataflow graphs to exhibit parallelism in programs. Computations in a program were expressed as 'actors' (or nodes), where each actor represents a basic building block in the HPC program. The dependence relationships between pairs of actors were denoted by the arcs of the graph. Kahn process networks [31] emerged as a different dataflow model wherein the actors were replaced with sequential processes. These processes communicated by sending messages along channels that conceptually consisted of unbounded FIFO queues. While the application of dataflow graphs were mainly applied to computer architecture design, Kahn dataflow was used by concurrency theorists for modeling concurrent software. In the field of computer architecture, dataflow graphs were originally used in a machine-level program representation in the two forms of dataflow architecture - Static and Dynamic. In the static approach the arc could contain only a single token (result value) and could have only one instance of a dataflow actor in execution at any time. In the dynamic approach, arcs could support multiple tokens, where each token was assigned to a tag

so that tokens associated with different activations of an actor maybe distinguishable. A history of the evolution of the dataflow computation model(s) is beyond the scope of this work and can be referenced amongst others in [27].

2.3 Caveats of the Dataflow Computation

The aforementioned dataflow model did present a few caveats [27, 24]. Firstly, the dataflow model incurs more overhead in the execution of an instruction cycle compared to its control-flow counterpart due to its fine-grained approach to parallelism. Secondly, the overhead involved in the detection of enabled instructions and the construction of result tokens generally results in poor performance in applications with low degree of parallelism.

However, in the light of these shortcomings, interest in dataflow models has recently emerged for exploiting parallelism in multi and many-core architectures. Originally the emphasis lay solely on individual data elements in the pure dataflow model emulating hardware design. The introduction of low cost on-chip (Network-on-a-chip) memory coupled with the capacity of high core density lead to a shift in the design of dataflow paradigms from fine to medium and now at a coarser grain.

2.4 Recent Developments in Dataflow Models

The ubiquity of multi-core processors in current hardware has led to the emergence of a myriad of multithreading frameworks embracing the idea of task parallelism.

Dynamic task parallelism is also being added for mainstream application in many new programming models for multi-core processors and shared-memory parallelism (apart from OpenMP 3.0), such as Cilk [23] , Java Concurrency Utilities, Intel Thread Building Blocks [33], and Microsoft Task Parallel Library.

There also exists multithreading systems based on dataflow principles, that represent computations as a Directed Acyclic Graph (DAG) and schedules tasks at runtime through resolution of data hazards. Projects falling within this category include QUARK (QUeueing And Runtime for Kernels), [42] a runtime API part of the PLASMA library [7], the OmpSs runtime environment from Barcelona Supercomputer Center [11], and StarPU [8] from INRIA Bordeaux.

2.4.1 QUARK Runtime API

QUARK (QUeuing And Runtime for Kernels) [42] 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 [7] library, developed primarily at the University of Tennessee, it is capable of supporting other data-driven applications that can be decomposed into tasks with data dependencies.

QUARK was designed to implement an easy to use application interface, that embodies the principle of a dataflow model, where scheduling is based on data dependencies between tasks in a task graph. The data dependencies are inferred through a runtime analysis of data hazards implicit in the data usage by the kernels. Figure 2.1 below represents the architecture of the QUARK shared-memory runtime environment.

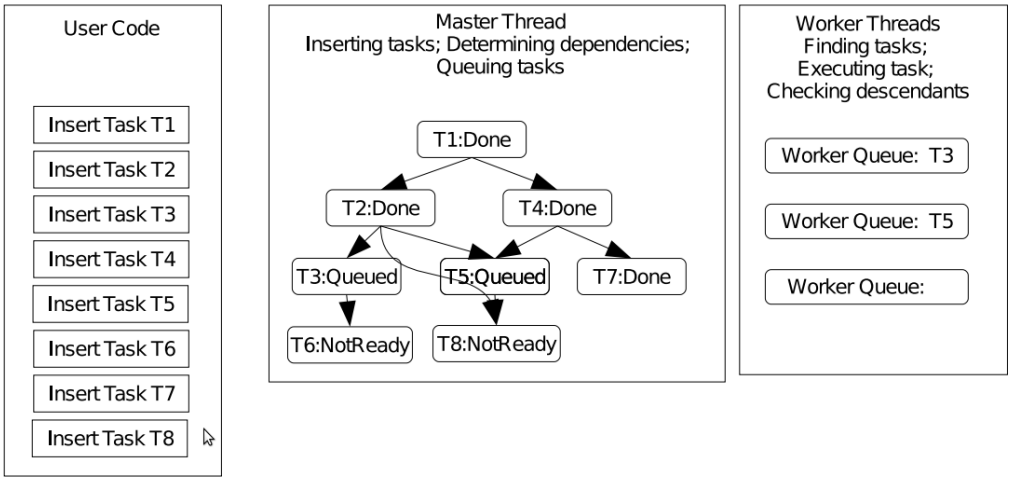


Figure 2.1: Architecture of the QUARK shared-memory runtime environment [42]

QUARK uses the parameter specifications to infer the dependencies between the various kernel routines in the application. These dependencies form an implicit DAG (Directed Acyclic Graph) connecting the kernel routines. As seen in Figure 2.1 user thread runs serial code and acting as the master inserts tasks into a (implicit) DAG based on their dependencies. Tasks can be in *NotReady*, *Queued* or *Done* states. When dependencies are satisfied, tasks are queued and executed by worker threads. Workers update dependencies when tasks are done. The use of QUARK additionally enables some optimizations such as DAG merging, loop reordering, etc. Overall

it promotes an easy to use interface which allows developers to experiment with alternative algorithmic formulations.

2.4.2 The OmpSs Programming Model

OmpSs (OpenMP SuperScalar) language is an effort to integrate features from the StarSs programming model developed by Barcelona Supercomputing Center into a single programming model. The OmpSs programming standard defines 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. Although variable names are given as arguments, the dependence information is evaluated at task creation time. This model is a prominent example of a coarse-grain dataflow programming model that offers programmers a compact set of non-intrusive language annotations in the form of the aforementioned pragmas. These pragmas are employed to remove global synchronization barriers. Although such global synchronization points limit dependence resolution, they are required at control flow points in existing programming models such as the OpenMP tasking model. The OpenMP tasking model presently dictates the use of control flow which does not fit naturally within the dataflow paradigm.

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. As described

in [11], at every nested hierarchy level of the task dependency graph, a small table is maintained in order to store the list of variables active in the system. Each variable in this table, is associated with the last graph node written to. When the code represented by a graph node completes, it removes itself as the last writer in the table. A graph node with no associated last writers, is considered to be free of the described dependence, allowing the runtime environment to pass it on to a scheduler which in turn assigns it to an available core.

The implementation proposed in this thesis differs with respect to the proposal presented by the Barcelona Supercomputing group in [22] on mainly two grounds. Firstly, in our implementation 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 the Barcelona Supercomputing group. Secondly, their implementation employs a task dependency graph and a table data structure (maintained at every hierarchical level of the graph), to store and manage the data dependencies among tasks, whereas our implementation employs a single *tag table* (unordered hash map) to account for the same. The OmpSs runtime offers expressivity in terms of application of their proposed extensions, which allows programmers to specify the dependencies among tasks at program level with ease. However, the generation and maintenance of the task dependency graph constructed at runtime, coupled with the time invested by threads waiting for tasks to be ready for execution, levies a significant overhead on their implementation. In contrast, our proposed extensions produce minimalistic overhead owing to the efficient handling of the *tag table* data structure

(explained in detail in Chapter 5) at runtime, providing performance improvement and scalability.

2.4.3 StarPU

StarPU [8] is a tasking API that allows programmers to conveniently schedule parallel tasks, develop and fine tune scheduling algorithms over a heterogeneous environment (CPU's and GPU's) while eliminating the effort to adapt their programs to the target hardware. It basically integrates an efficient data-management facility with a task execution engine. StarPU's runtime library provides support for a task-based programming model where programmers can execute tasks on multiple targets directly by means of the appropriate programming language (eg. CUDA) or libraries (eg. BLAS) based on the availability of the environment on the hardware. StarPU schedules and executes the tasks and the associated data transfers on the available hardware, freeing the programmer from issues pertaining to efficient load balancing and task offloading. It is also possible to express task dependencies owing to the presence of high level detailing on data being accessed by every task, such as the mode of data access (Read, Write, Read/Write) by the data management library. This not only facilitates data coherency but also permits programmers to express complex task graphs with ease due to its asynchronous nature allowing the reordering of the tasks when needed to further improve performance.

2.4.4 Data-Driven Tasks (DDT)

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 [37, 39]. 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 compiler's and runtime's. Another extension to task parallelism, described as Data-Driven Futures (DDFs) in [35] , 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 operations.

2.4.5 Data Versioning-based Approach

In [36], we see that instead of expressing dependencies among tasks, the authors have handled the data dependencies by attaching a version number (or a *handle*) to all the data being accessed and storing a lowest required version number for each access to that task when the task is added to the library. On the other hand in the task library

presented, a given task has no knowledge of any other tasks, instead it associates itself only with the data that it accesses. When a task has finished execution, it increases the version number of each data it has accessed. Data dependencies are monitored by the comparing the required versions stored inside the task with the version numbers attached to the accessed data. By comparing these version numbers it can be detected if a task is ready for execution, as well as it can be detected if a certain task which wants to read a certain data has finished before allowing another task to overwrite that data. The advantages of this approach is that it avoids a global perspective, i.e. each task is only aware of the *handles* it accesses and each *handle* is only aware of the tasks for which it is waiting. Also, there is no coupling in between tasks which means tasks can be deleted at any time without notifying its successors.

2.5 Summary

In this chapter we discussed the origin and highlighted the advantages a dataflow model has to offer, in comparison to a control flow model. We also discussed the application of the respective models in various fields of research including high performance computing, computer architecture, etc. We also incorporated a survey on ongoing research initiatives that employ the use of dataflow models in their compiler and runtime API construction. In the forthcoming chapters, we will compare and contrast the performance obtained from few of the aforementioned dataflow models with the extensions proposed as a part of this thesis.

Chapter 3

OpenMP Tasking Model

In this chapter we provide a brief overview of the OpenMP programming model and the discuss the features of the tasking implementation.

3.1 OpenMP Overview

OpenMP [3] is the *de facto* standard for a simple, portable, and consistent interface for shared memory programming across a host of hardware architectures and commercial compilers. It comprises of a set of worksharing and synchronization directives and runtime routines for the C/C++ and Fortran languages. With the addition of these directives to sequential programs, a user can convey instructions to an OpenMP compiler that allows it to generate multi-threaded code. To put it simply, OpenMP essentially provides a fork-join model of parallel execution (as seen in Figure 3.1) where all threads in the current thread team, have access to a shared

memory. On entering a parallel region a single master thread creates (forks) several slave threads which execute the work in parallel with the master thread. Each unit of work is bound to a specific thread for its entire lifetime and uses the data environment of the thread they are bound to. Once the slave threads have completed their work they join back with the master thread, at the end of the parallel region, and execution continues sequentially. The main merits of OpenMP (as of version 2.5) can be credited to its shared-memory abstraction, incremental parallelism, portability, scalability, support for fine-grained and coarse-grained parallelism, and ease of use.

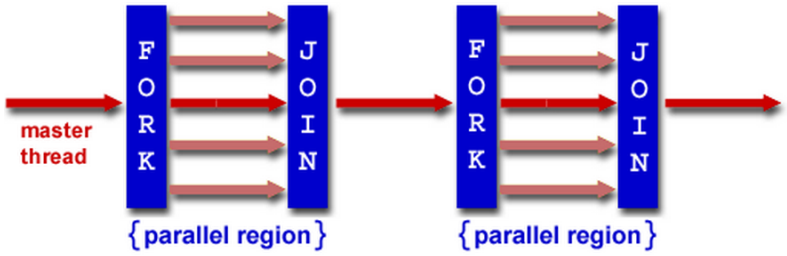


Figure 3.1: The OpenMP fork-join model [5]

3.1.1 Limitations of OpenMP 2.5

Even though OpenMP 2.5 was tailor-made for applications exhibiting loop level parallelism and employing large array-based data structures, it was yet to provide adequate support to explore unstructured parallelism. Unstructured parallelism is characterized by:

- Programs exhibiting irregular distribution of data
- Programs where the workload and data access patterns can only be discerned

at runtime.

- Programs where along with workload and data access pattern, data dependencies are also dynamic in nature and can be known only at runtime.

OpenMP 2.5 lacks the ability to specify structured dependencies among different units of work. This could be considered a significant limitation in programming hierarchical problems like those employed in: a) recursive parallelism with unbounded loops, b) list and tree traversal, c) dense linear algebra, d) multiblock grid solvers, etc.

For example, consider the code in Listing 3.1 that illustrates the traversal of a linked list using worksharing constructs. As seen the dynamic nature of the linked list suggests an unbounded loop which is essentially a characteristic of unstructured parallelism and thus difficult to parallelize with OpenMP 2.5 constructs. A possible approach, involves primarily calculating the number of nodes in the list followed by the transformation at run time of the linked list into an array, which incurs the overhead of the additional while loop and array construction thereby limiting performance owing to lack of generality and flexibility. Therefore applications exhibiting similar irregular parallelism may possess the scope for potential concurrency, which cannot be fully exploited in the absence of support for asynchronous execution of units of work, that can further benefit their performance.

```
1  while (p != NULL)
2  {
3          p = p->next;
4          count++;
5  }
6  p = head;
7  for(i=0; i<count; i++)
8  {
9          parr[i] = p;
10         p = p->next;
11 }
12 #pragma omp parallel
13 {
14         #pragma omp for schedule(static,1)
15         for(i=0; i<count; i++)
16         processwork(parr[i]);
17 }
```

Listing 3.1: Traversal of a linked list using OpenMP worksharing constructs

3.2 Introduction to OpenMP Tasks

An explicit task maybe defined as an instance of executable code and its data environment, generated when a thread encounters a task construct. Each task has some

private memory associated with it that stays persistent during a single execution. In addition tasks do not use the data environment of any thread but have a data environment of their own. The advantage of tasks over threads as highlighted in [9] mainly include:

- Tasks are more lightweight than threads i.e. the overhead for task creation and deletion are significantly lower in comparison to that of threads.
- Tasks are not bound to a specific thread and that different parts of a task may be executed by different threads.
- Tasks allow the programmer to specify units of work that maybe deferred to be executed at a later time.
- Tasks are very flexible with respect to their placement within a program. It requires only that they be nested within a parallel region. The construct can be placed within any other construct, including itself.
- Tasks enable dynamic generation of units of work, to be asynchronously executed, allowing expression for irregular parallelism, which ultimately benefits performance and program structure.

3.2.1 Task Synchronization

Synchronization among tasks maybe achieved through the use of the *taskwait* or *barrier* constructs. On encountering a *taskwait* construct the encountering master task must suspend and wait until all of its child tasks have completed up until

that point before resuming execution. When a *barrier* construct is encountered, all threads must wait until all the other threads in the current thread team reach that barrier and all tasks created prior to the barrier are completed.

Listing 3.2 below represents the code described in Listing 3.1 only with the inclusion of OpenMP tasks. In this version we see a single thread creating all the tasks (each node of the list) and all child threads executing the work within the parallel region, thereby eliminating the overhead for array construction at runtime. Tasks are dynamically created and scheduled asynchronously allowing scope for additional concurrency as well as improving the overall program structure by making it more concise.

```
1  #pragma omp parallel
2  {
3    #pragma omp single
4    {
5         p=head;
6         while (p) {
7        #pragma omp task firstprivate(p)
8         processwork(p);
9         p = p->next;
10        }
11   }
12 }
```

Listing 3.2: Traversal of a linked list using OpenMP tasks

3.3 Other Tasking Models

Dynamic task parallelism is gaining popularity for extensive mainstream use in many new programming models and languages for multi-core processors and shared memory paradigms. In this section we briefly discuss a few of the more popular paradigms that have adopted and integrated the advantages of dynamic task parallelism.

3.3.1 Cilk

Cilk [12], is a programming language and runtime system developed at MIT that generalizes the semantics of C and effectively exploits parallelism in a multithreaded environment. Its runtime system implements a *work stealing* algorithm, where each processor maintains a ready pool of threads. The algorithm allows idle processors called *thieves* to steal work from busy processors called *victims*. Cilk's scheduler guarantees that the cost of stealing contributes only to the critical-path overhead, and not to the overall work overhead thereby benefiting performance by increased load balancing. It also adopts the *work-first* principle that minimizes the scheduing overhead borne by the work of a computation.

Cilk-5 [23] closely resembles the OpenMP tasking model. The *spawn* statement corresponds to explicit OpenMP tasks and the *sync* statement behaves exactly like the *omp taskwait* directive. The most important difference between OpenMP and Cilk is that OpenMP provides constructs to express parallelism beyond the task level (i.e. loops). Cilk requires the programmer to convert loops into recursion in order to parallelize them. Cilk also requires tasks to be encapsulated in function calls,

whereas OpenMP allows any block of code to become a task. Cilk does not support the notion of a tied task; therefore, all tasks are free to migrate across threads. Cilk is closely tied to the GCC compiler, which it uses as its backend. Scheduling algorithms based on Cilk‘s work stealing scheduler are gaining popularity for its provision for dynamic lightweight task parallelism.

3.3.2 DARPA HPCS Languages

HPCS languages were developed as part of DARPAs efforts to achieve 2 petaflops of sustained performance, scalable to 4 petaflops in 2002. The three languages that evolved were: Chapel [18] by Cray, X10 [15] by IBM and Fortress by Sun Microsystems, all of which identified dynamic lightweight task parallelism as one of the prerequisites for success. Unlike the data parallel and the SPMD [17] programming models that normally assume a fixed number of concurrent threads during program execution, dynamic task parallelism allows concurrent tasks to be created and joined at any time during the execution.

The need to support irregular forms of parallelism is evident in features being included in these new programming languages, notably *activities* and *futures* in X10 and the the *cobegin* statement in Chapel. Another striking feature of X10 is clocks, that guarantees deadlock-free computation when used correctly with X10s parallel constructs like *async, final, foreach*, and *atomic*. Similar to barrier, clocks allow better synchronization among the asynchronous tasks executing at the various places.

3.3.3 Phasers

The success of clocks led to concept of phasers [34] used in Habanero-C which apart from deadlock- free synchronization, also accomplished phase-ordering providing better point-to-point synchronization in fine-grained task-parallel programming. A phaser is typically associated with four modes: *signal-wait-next, signal-wait, signal-only*, or *wait-only*. These different modes define different capabilities for tasks associated with them. Several phaser operations are available e.g. adding/dropping, next operation (similar to that in clocks) and phaser-specific signals which depend on the phasers registration mode. When executing a *next* call, a task participates in a barrier or point-to-point operation depending on the registration mode on the phaser. In addition to *next* operation, phasers have a *next* with single statement which allows advancing phasers associated with an activity to be advanced to next phase and also executing the *stmt* as a single statement. This reduces a lot of overhead and thus provides better synchronization.

3.3.4 Intel Workqueueing Model

The Intel workqueueing model was an early attempt to add dynamic task generation to OpenMP. The model consisted of the *taskq* and *task* constructs. This proprietary extension allows hierarchical generation of tasks by nesting of *taskq* constructs. Synchronization of descendant tasks was controlled by means of implicit barriers at the end of *taskq* constructs. The constructs had been intentionally designed to be similar to existing worksharing constructs, but the implementation exhibited some

performance issues.

3.4 Limitations of the OpenMP Tasking Model

The overhead of communication and synchronization between concurrent tasks typically presents an obstacle to getting high performance and scalability on parallel systems. In order to support diverse workloads in multi and many-core architectures it is desirable to have a high level synchronization mechanism that is able to support point-to-point synchronizations among asynchronous tasks. The execution of a task normally (as of OpenMP 3.1) occurs at any point between its creation and its encounter with a task synchronization point namely *taskwait* or *barrier* (to name a few) which serve as a global barrier. For example in case of *barrier*, threads in a thread team must wait for the slowest one to reach the barrier in the program, which may typically degrade performance.

We identify the following roadblocks in the current OpenMP 3.1 specification:

- There are no means to specify waiting on a specific task(s).
- Support for expressing data dependencies among tasks is absent.
- The programmer has limited control over scheduling of tasks.
- Increased data synchronization overheads limits scalability and performance of the application.

Since the current model provides no means to specify waiting on a particular

task or a set of tasks, the programmer is forced to make use of the global barriers *taskwait* or *barrier* which prevents a given parallel algorithm from fully exploiting the maximum concurrency achievable.

Code in Listing 3.3 illustrates the problem mentioned above. The for loop generates a number of tasks, each writing to an element of the array A. The statement at line 10 reads from A[j]. Since OpenMP does not provide any means to wait specifically on the task that writes to A[j], and the programmer must instead rely on the the *taskwait* which waits for all created tasks to complete. This is necessary since we must avoid a situation where a read in line 10 may happen prior to a write in line 5 for elements in array A, in order to maintain data integrity.

```
1  #pragma omp parallel
2  {
3    for( i = 0 ; i < N; i ++) {
4  #pragma omp task shared(A)
5    A[ i ] = f( ) ;
6    }
7  / * process some element of A * /
8    j = getindex( 0 ,N ) ;
9    #pragma omp taskwait
10   g(A[ j ] ) ;
11 }
```

Listing 3.3: Absence of point-to-point synchronization among OpenMP tasks

The need for controlling synchronization among such tasks is particularly important when dealing with applications that a) can be expressed through a task graph, b) exhibit pipeline parallelism, and c) require wavefront synchronization.

We explore ways to reduce the need for global synchronization, by including additional language features to the current specification, which tailor a macro-dataflow model strategy of execution. In the next chapter we shall discuss this approach which entails extending the current OpenMP tasking construct to support a specific ordering of tasks based on the dependence relationships existing among the individual tasks.

3.5 Summary

In this chapter we began by providing a brief overview to the OpenMP programming model. We discussed the limitations of this model, OpenMP 2.5, which led to the introduction of the tasking model (in the OpenMP specification 3.0). We further described how other mainstream programming models have gradually leaned towards adopting the concept of dynamic task parallelism. Lastly, we spoke of the limitations accompanying the present OpenMP tasking model. In the next chapter we focus on how to overcome these limitations which primarily serve as motivation for this thesis.

Chapter 4

Proposed Extensions to the OpenMP Task Construct

In this chapter we propose to extend the current OpenMP tasking model, by proposing language extensions to the OpenMP task construct. Our objective is to lend support for unstructured parallelism to the current OpenMP specification by allowing point-to-point synchronization among asynchronous tasks. This serves as our main motivation for this thesis. We also discuss target algorithms which may benefit in performance, owing to a specific ordering of tasks expressed by the programmer.

4.1 Extensions to the OpenMP Task Construct

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 next taskwait or barrier construct (whichever is encountered first). Thus, the order of task execution is 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 [25]. 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 data dependencies amongst tasks that are modifying and reading a shared resource, and ensuring that they are not violated. If the tasks do not access a shared resource they are considered independent.

Similar extensions to the OpenMP tasking model [22] has been proposed by the Barcelona Supercomputing Center (BSC). However, our approach may be considered more concise and expressive in terms of implementing task-level granularity. It supports fine-grained synchronization based on the data dependencies among the

tasks, without the addition of extreme overhead for maintaining task synchronization. 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" 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). We show the extensions proposed and Listing 4.1 explains very briefly the functionality of the extensions:

- *#pragma omp task* ***out*** *[t_1,t_2,....,t_n]* A task is proposed to have "*out*" dependence if a task might compute variables required by succeeding tasks.
- *#pragma omp task* ***in*** *[t_1,t_2,....,t_n]* A task is proposed to have "*in*" dependence if it requires variables that have been computed previously.
- *#pragma omp task* ***inout*** *[t_1,t_2,....,t_n]* The combination of the two extensions explained above which entails a task having an *in* and *out* dependence on the same *tag*.

t_1,t_2,....,t_n specifies the argument list that can be represented in the form of integer expressions (which may include constants, an constant expression, variables, an address etc) denoting the *tags* as specified by the programmer.

In the Listing 4.1 below, *task 3*, due to the existence of an *in* dependence owing to tags t_1 and tag t_2, is put on hold until *task 1* and *task 2* (which have tags t_1 and tag t_2 respectively as *out* dependencies) complete execution. 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*. By doing so, *task 4* would have to wait for a global synchronization point and will not be scheduled in parallel with *task 1*.

```
1  #pragma omp parallel num_threads(2)
2   {
3   #pragma omp master
4    {
5     #pragma omp task out(t1)    /* task 1 created */
6        x = f1();
7     #pragma omp task out(t2)    /* task 2 created */
8        y = f2();
9     #pragma omp task in(t1) in(t2) out(t3) /* task 3 created */
10       z = f3(x,y);
11  /* task 3 has to wait for task 1 and task 2 to complete execution
       */
12    #pragma omp task in(t2) out(t4) /* task 4  created */
13       w = f4(y,z)
14  /* task 4 has to wait on only task 2. Can scheduled in parallel
       with task 1 and task 3 */
15   }
16  }
```

Listing 4.1: sample code quoting the proposed extensions.

4.2 Categories of Data Dependence

Before we discuss the target applications that may benefit from the proposed extensions and explain the details of the extension's implementation, in the OpenUH runtime library, it is important to enumerate the different categories of dependencies the extensions are capable of handling, and explain the semantics of using those extensions.

Figure 4.1 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 Figure 4.1 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 is also referred to as the *read after write* (RAW) dependence. In Figure 4.1, 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

is also referred to as the *write after read* (WAR) dependence. In Figure 4.1, 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 4.1, an *output* dependence is noticed for tasks T_1 and T_4 for tag 5.

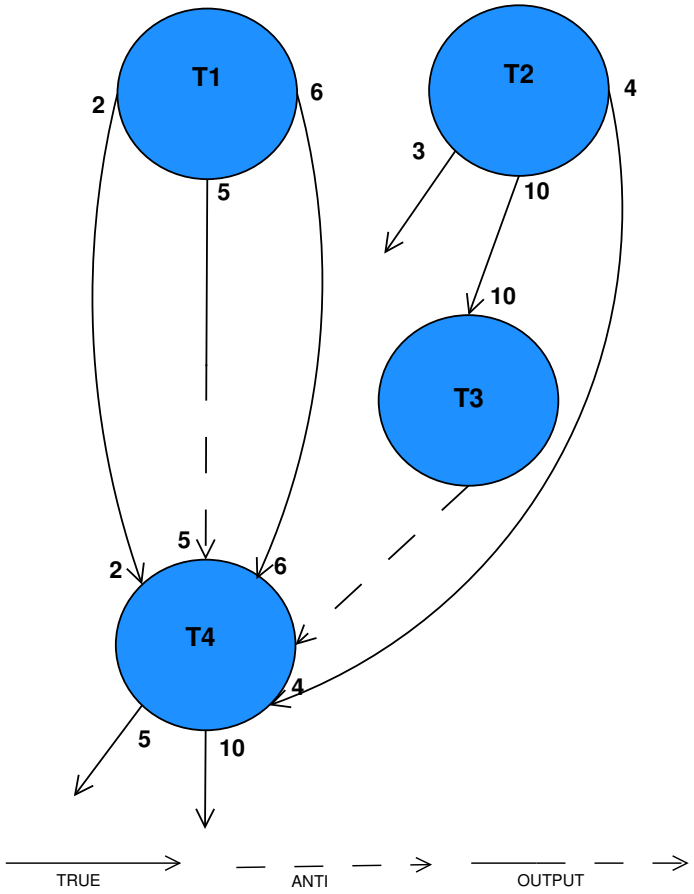


Figure 4.1: Dependency graph with *true, anti*, and *output* dependencies

In Figure 4.1 each task has been labeled with its respective *tags* establishing the existence of dependence relationships among them. 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 dependencies to exist and the runtime library will ensure that the execution of those dependent tasks follow the order in which they appear in the program (or are created).

4.3 Target Algorithms

We applied three algorithms to evaluate the advantages of obtaining high scalability and improved performance, with the application of the proposed extensions to the OpenMP task construct.

1. The LU decomposition algorithm - characteristic of dense linear algebra computations found within the LINPACK benchamrks used to rank supercomputers collected by the TOP500 website.

2. Smith-Waterman algorithm - a wavefront-based programming paradigm used in scientific applications such as those based in sequence alignment, biological sequence, etc.

3. Matrix Multiplication - found in Level 3 BLAS libraries.

In this context our challenge was the combination of two goals: achieving high performance and maintaining the accuracy of the scientific algorithms.

4.3.1 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 dynamic blocking [38], and pipeline processing [30]. 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:

$$A = L * U \tag{4.1}$$

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

We focus on a blocked version of the algorithm which 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.

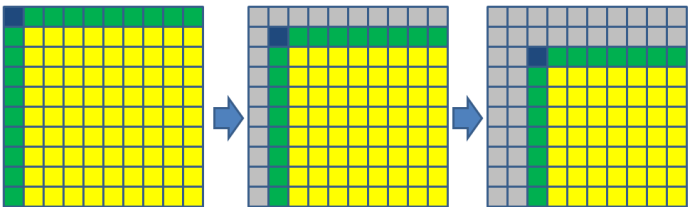


Figure 4.2: Progress of Blocked LU Decomposition per iteration [41]

With the assistance of Figure 4.2, 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 Figure 4.2, 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 Figure 4.3. Figure 4.3 represents the dependency graph for a 5 X 5 block matrix where we can clearly observe the dependence relations where each arrow illustrates an existent data dependence across neighboring blocks.

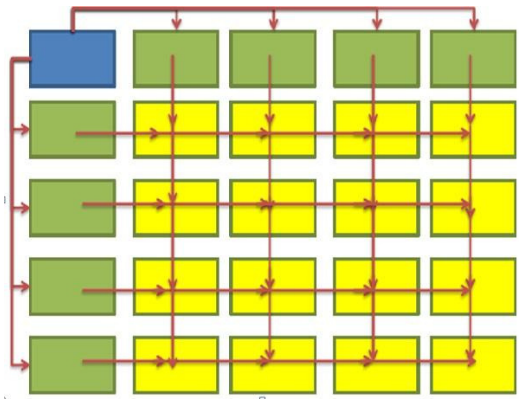


Figure 4.3: A single iteration highlighting data dependence relations across blocks [41].

In every iteration of the outermost diagonal block is been factorized. The factorization of that block enables the execution of the *swap* and *triangular solve* operations on all blocks in the same column and row accordingly. All the column and row blocks can be updated in parallel since there are no dependencies among them. Blocks of the inner part of the matrix can be updated with a matrix multiplication operation as soon as their dependencies are solved. This means that for each of the blocks their corresponding *swap* and *triangular solve* blocks must both finish executing before they are available for execution. The iteration of this algorithm over all the diagonal blocks produces the final decomposition of the matrix. Synchronization between steps of the same iteration uses global synchronization points (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.

Listing 4.2 represents the pseudo-code for execution of the LU Decomposition algorithm using OpenMP tasks. The use of a *taskwait* depicts the presence of a global synchronization point. This indicates that before *ProcessInnerBlock* can execute, all tasks executing *ProcessBlockOnRow* and *ProcessBlockOnColumn* must complete. The presence of a *taskwait* compromises the flexibility with which tasks can be optimally scheduled at runtime and also limits the margin of achievable concurrency.

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

  for (i=1;i<M;i++) {
  /**** 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 4.2: Tasking implementation for LU decomposition

4.3.2 Smith-Waterman Algorithm

The Smith-Waterman algorithm is used primarily in the field of DNA and protein sequencing, where with the help of local sequence alignment, we are able 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. The time complexity of this algorithm for comparing two sequences is O(mn), where m and n are the lengths of the two sequences being compared in order to obtain the most optimal local alignment between the two.

The basics steps involved in the algorithm are as follows:

1. Populate the dynamic programming scoring matrix
2. Calculate the maximal score in the scoring matrix
3. Trace back the path that leads to the maximal score to find the optimal local alignment.

Scoring Matrix H is determined as follows:

$$
\begin{aligned}
&H(i,0) = 0, 0 \leq i \leq m \\
&H(0,j) = 0, 0 \leq j \leq n \\
&\quad if\ a_i = b_j \text{ then } w(a_i, b_j) = w(match) \text{ or} \\
&\quad if\ a_i \neq b_j, w(a_i, b_j) = w(mismatch) \\
&H(i,j) = MAX(0, \\
&\qquad H(i-1, j-1) + w(a_i, b_j), \text{ (Match/Mismatch)} \\
&\qquad H(i-1, j) + w(a_i, -), \text{ (Deletion)} \\
&\qquad H(i, j-1) + w(-, b_j), \text{ (Insertion)} \\
&\qquad)
\end{aligned}
\tag{4.2}
$$

where :

- a,b = individual alphabets of the two sequences.
- m = length(a)
- n = length(b)
- H(i,j) is the maximum Similarity-Score between a suffix of a[1...i] and a suffix of b[1...j]
- w(x,y) is the gap penalty

For the purpose of this thesis we have employed a symmetric square scoring matrix whose costs are derived from the observed substitution frequencies in alignments

of related sequences.

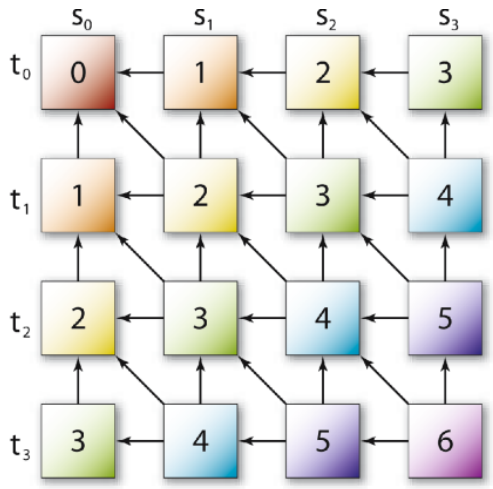


Figure 4.4: Data dependencies in the wavefront pattern exhibited by Smith-Waterman algorithm [29].

Figure 4.4 [19] represents an example of a 2D wavefront, where updating an element in the matrix requires the updating of 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 corner of the matrix and a gradual sweep moves along the diagonal up to the next corner. This diagonal represents the number of computations or elements that could be executed in parallel without dependencies among them. 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. As illustrated in Figure 4.4,

elements of a similar color on each diagonal can be executed in parallel provided their respective dependencies has been satisfied. The number of such independent elements gradually increases as seen in Figure 4.5 to the maximum length of the diagonal. Then it will decrease to end in the opposite corner of the grid. This diagonal sweep is the reason for the name *wavefront*. In Figure 4.5 we observe that in the first cycle, only one element could be calculated. In the second cycle, two elements could be calculated. In the third cycle, three elements could be calculated, and so on. This feature implies that the algorithm has a very good potential for parallelism.

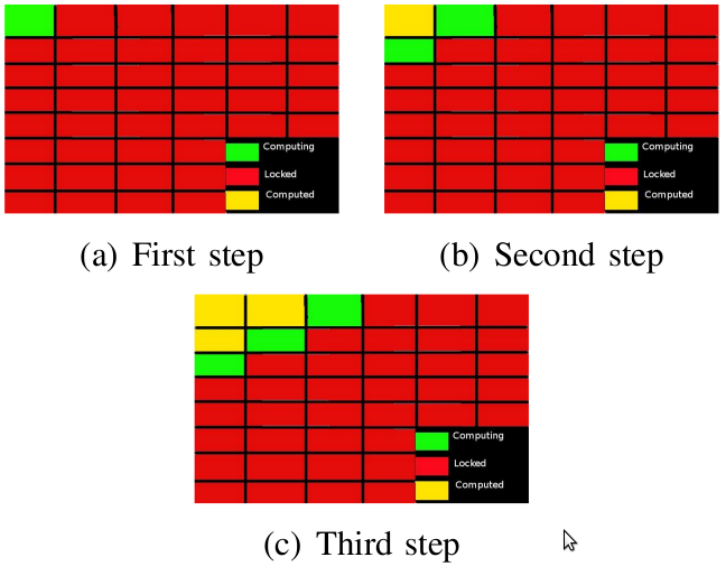


Figure 4.5: Progression in the scoring matrix generation.

By applying the equation in 4.2, all values of the scoring matrix can be obtained. The gap cost in above equation is the penalty for inserting a gap character "-" in the event of a mismatch in sequence alignment.

We advocate the use of a task-based approach over a thread-based approach primarily because a) tasks are much more lightweight compared to threads, and b) in a task-based model, the task scheduler is capable of manipulating the scheduling of tasks based on the discretion of the programmer. We can tweak the runtime task scheduler to adapt itself to suit the requirements of the application. For instance, in this particular case we implemented and enabled a *lockless* approach (described in Chapter 5) for accessing the task queue's so that we could reduce the amount of contention among threads when they try to steal work from the task queues simultaneously. We noticed a significant improvement in performance when utilizing the *lockless* queue implementation.

The initial OpenMP tasking implementation parallelized the diagonals using a brute force approach where we compute every single element of each diagonal as a task in parallel. This approach caused a degradation in performance with the increase in the number of threads primarily due to: a) enormous overhead generated for creating numerous tasks (a task for each element on the scoring matrix) especially for larger problem sizes, b) the additional overhead incurred in synchronization of all these numerous tasks.

In order to improve the parallelization further and take advantage of the available hardware, a second OpenMP tasking version was implemented. In this version instead of creating a task for computing a single element of the matrix, we divide all the work into smaller chunks, where a cluster of elements on each diagonal comprises of a task. Such an approach coarsens the granularity of the parallelizing application leading to better performance in comparison to the fine grained approach initially

implemented.

```
#pragma omp parallel
 {
#pragma omp master
   {
      for(wave = 0; wave < waves; ++wave) {
      /* obtain number of element in the present diagonal */
      obtain_num_elements();
      /* traversing elements of each diagonal */
      for(ii = 0; ii < elements; ii+=chunk) {
      min = MIN(elements,ii + chunk);
#pragma omp task firstprivate(ii, np, mp, chunk, elements)
      {
        /* chunk of elements comprising of each task */
         for (i = ii; i < min; i++)
         {
          /* acquiring north-west neighbor */
           temp[0] = H[(np-i)-1][(mp+i)-1] +
                     similarity(seq_a[a][(np-i)-1],seq_b[a][(mp+i)-1]);
          /* acquiring west neighbor */
           temp[1] = H[(np-i)-1][(mp+i)]-gap;
         /* acquiring north neighbor */
           temp[2] = H[(np-i)][(mp+i)-1]-gap;
           temp[3] = 0;
           H[(np-i)][(mp+i)] = find_array_max(temp,4);
         }
       } // task
      } // inner for loop
#pragma omp taskwait /* global barrier */
     } // outer for loop
   } // master
 } // parallel region
```

Listing 4.3: Tasking implementation for Smith-Waterman

Listing 4.3 represents the pseudo-code for the chunked task implementation of the Smith-Waterman algorithm. A chunk of elements as specified in the inner *for* loop in line 19 denotes a single task. Array H denotes the scoring matrix where each element represents the maximum of its north, west and north-west neighboring elements. However the presence of a *taskwait* in line 34 of Listing 4.3 prevents us from accessing more than a single diagonal at a given time as in Figure 4.6. 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. Since the current OpenMP 3.1 specification is not yet equipped to handle explicit dependencies among tasks of the different diagonals, we envision an approach with our proposed extensions to the OpenMP task construct that might allow us to eliminate this *taskwait*, thereby allowing tasks on multiple diagonals to be executed concurrently only after their respective dependencies in the previous two diagonals have been resolved as seen in Figure 4.7.

It is to be noted that previous efforts have been made to accelerate this algorithm using both homogeneous and heterogeneous accelerator architectures as well as hybrid programming models. However the scope of our implementation solely lies within a shared memory environment with the use of OpenMP 3.1/4.0 constructs.

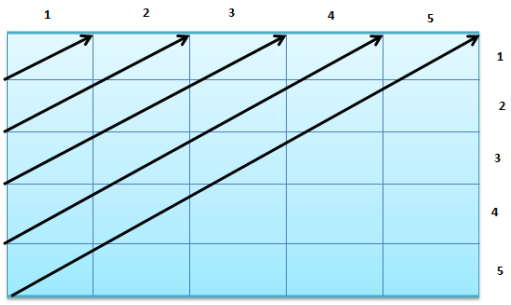


Figure 4.6: Scoring matrix traversal in the presence of global barrier.

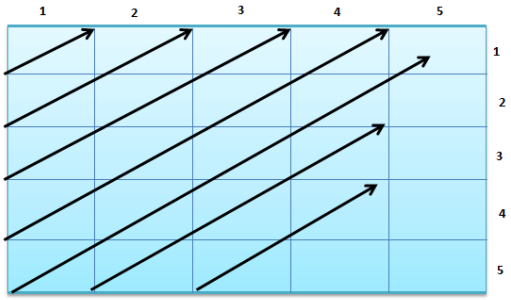


Figure 4.7: Scoring matrix traversal without the global barrier

4.3.3 Matrix - Matrix Multiplication

We use the matrix-matrix multiplication microbenchmark for assessing the overhead generated by the proposed extensions at runtime. We also apply this benchmark in gauging the overhead generated by related dataflow model implementations like the OmpSs runtime and QUARK (as described in Chapter 2) API. We perform this analysis in Chapter 6 of this thesis.

A naive version of a matrix - matrix multiplication [16], is seen in Listing 4.4, where A, B and C are the naive matrices of size N X N, the algorithm has the complexity of O(n^3) flops while operating on O(n^2) data elements.

```
for i = 1 to N
    for j = 1 to N
        for k = 1 to N
            C(i,j) = C(i,j) + A(i,k) * B(k,j)
```

Listing 4.4: Pseudo-code for Naive Matrix Multiplication

When working with large matrices, it becomes impossible to store the computations within the cache hierarchy of the processor. As a workaround, it is advisable to perform certain optimizations, for example, break the workload into smaller chunks or blocks of computation. Since each block uses a smaller piece of the data it fits well into caches thus allowing scope to improve temporal and spatial locality.

```
for i = 1 to B
  for j = 1 to B
    load block C(i,j) into cache
    for k = 1 to B
      load block A(i,k) into cache
      load block B(k,j) into cache
      C(i,j) = C(i,j) + A(i,k) * B(k,j)
       do a matrix multiply on blocks
       writes block C(i,j) back to memory
```

Listing 4.5: Pseudo-code for Blocked Matrix Multiplication

The pseudo-code of such a blocked matrix multiplication algorithm is shown in Listing 4.5, where A, B, and C are the matrices of size N X N divided into b X b blocks where b = N/B. Matrix multiplication is an *embarrassingly parallel* algorithm that allows the calculation of the product of each block in parallel without any

dependencies.

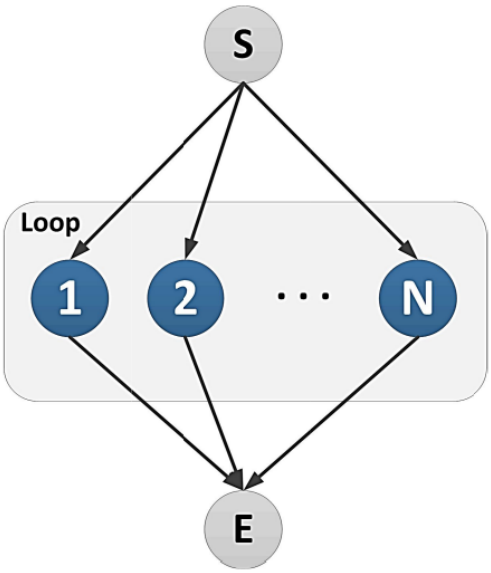


Figure 4.8: Dependency graph for Matrix Multiplication [16]

The dependency graph shown in Figure 4.8 illustrates how each sub-block of the matrix can be executed independently in parallel with other blocks on available cores. The only synchronization point is at the end of the computation, where it becomes essential to wait for all the blocks to finish processing. At the return of the function, we have the result for the whole matrix, with each core completing its own part of the result.

4.4 Summary

In this chapter, we have described at length the proposed extensions to the OpenMP task construct, which provides a mechanism for expressing point-to-point synchronizations among sibling tasks. We provide, as example, two applications namely - LU Decomposition (involving dense linear algebraic computations), and Smith-Waterman algorithm (exhibiting wavefront parallelism). Each shows potential for improvement in performance if a specific ordering of tasks is enforced. This would allow them to exploit higher degrees of concurrency by the eliminating a global synchronization point (like *taskwait*) so long as their respective data dependencies are correctly honored. We also briefly discussed the matrix-matrix multiplication microbenchmark. The purpose for utilizing this benchmark is to make an assessment with regards to the overhead generated, when using the proposed extensions and compare it with overhead generated from related dataflow model implementations.

Chapter 5

Algorithm and Implementation

In this chapter we describe the algorithm, which adds support to detect and process data dependencies among OpenMP explicit tasks, implemented at the OpenUH OpenMP runtime library at length. Table 5.1 introduces a few terminologies that we shall be referring to throughout the rest of the chapter.

5.1 Introduction to the OpenUH Compiler

The OpenUH compiler [14] as seen in Figure 5.1 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, a back-end translation implemented jointly with Tsinghua University, and an efficient runtime task scheduling infrastructure which holds support for improved nested parallelism.

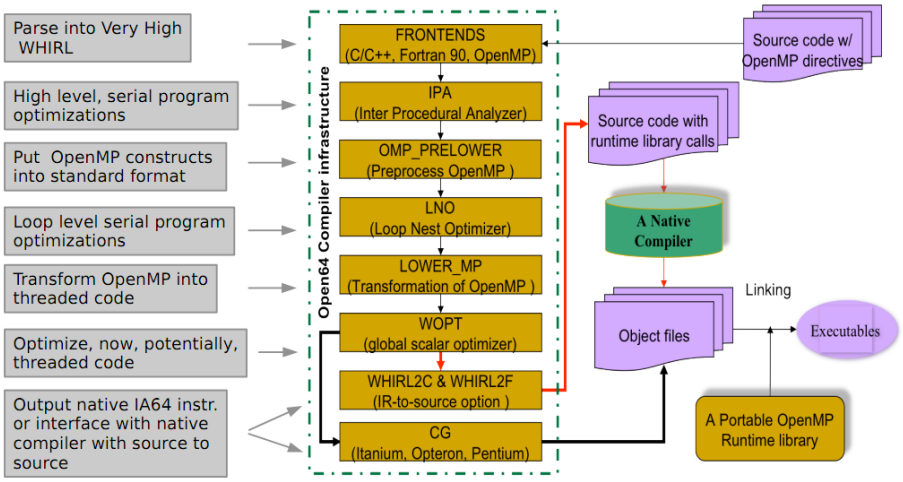


Figure 5.1: OpenMP implementation in the OpenUH Compiler

Its implementation supports a configurable task pool framework that allows the user to adapt the runtime environment based on the needs of the application. 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. As described in [28] the efficiency of the runtime implementation will therefore 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.

5.1.1 OpenUH Runtime Tasking Implementation

The OpenUH OpenMP runtime library (RTL) offers the following environment variables which allows the programmer to tweak the runtime in order to extract maximum performance from a given application:

- O64_OMP_TASK_POOL: allows the user to control the task pool environment. The OpenUH RTL provides a variety of distributed queue strategies with work stealing, to help configure the task pool environment -
  - *default*: single-level task pool with 1 task queue per thread holding tied and another task queue holding untied tasks.
  - *simple*: single-level task pool, 1 task queue per thread holding both tied and untied tasks.
  - *2level*: two-level task pool with 1 task queue per thread, and 1 community queue for the team. Each queue holds both tied and untied tasks. Work can be stolen from the community queue in chunks.

  The use of multiple public queues instead of a shared global queue, allows the contention to be distributed across the system.

- O64_OMP_TASK_QUEUE: allows user to control how the task queues operate. This environment variable controls how tasks are entered and removed from the queues as well as determines the execution order of the tasks. The available options are:
  - *DEQUE*, a doubled-ended queue where tasks can be stolen from the head or the tail of the queue.
  - *FIFO*, a First In First Out queue where tasks are stolen from queue's own head or from a victim's head. This allows a breadth-first execution of tasks.

  - *LIFO*, a Last In First Out queue where tasks are stolen from queue's own tail or from a victim's tail. This allows a depth-first execution of tasks. This method benefits from the data locality of depth-first execution but requires more memory for storing the excess tasks.

  There is also a CFIFO or concurrent FIFO ordering of tasks which allows tasks to access the pool concurrently from the head and tail.

- O64_OMP_TASK_CUTOFF: allows user to control the cutoff condition for task generation. The OpenUH runtime allows the programmer to choose from a number of values, each of which sets the task create condition that determines whether tasks will be queued or executed immediately.

  - *always*: always cuts off explicit tasks which means that explicit tasks will never get created.
  - *never*: never cuts off explicit tasks implying that explicit tasks will always get created.
  - *num_threads*: cuts off task generation if team size is less than specified value (by default is set to 2).
  - *switch*: cuts off task generation if the "switching depth" reaches this specified value (by default is set to 100).
  - *depth*: cuts off task generation if current depth in the task tree reaches this specified value (by default set to 100).

- O64_OMP_QUEUE_STORAGE: controls how slots in the queue are stored. The available options include:

- *ARRAY*: where the slots in the task queues are stored in an array. This is the default configuration.
- *LIST*: where the slots in the task queues can be stored in a linked list. This allows the size of the list to be dynamically altered based on the number of tasks in the queue.
- *DYN_ARRAY*: an optimization which applies to the ARRAY implementation where work is grabbed from a GLOBAL queue when the array if completely full. When the GLOBAL queue is full, it dynamically doubles in size.
- *LOCKLESS*: provides a lockless queue implementation inspired from Habenero C [13], where we use a compare-and-swap operation to obtain atomicity of operations. This configuration was implemented to alleviate the contention overhead generated from a mutual lock based approach, in the event that the task queues are bombarded by threads simultaneously looking for work.

Explicit details explaining the scheduling of tasks (taking explicit data dependencies into consideration), in the OpenUH runtime, will be explained in the next section.

5.2 OpenUH Runtime Implementation for Proposed Extensions

In this section we describe the algorithm implemented at the OpenUH OpenMP runtime library that provides support for the extensions proposed in Chapter 4. We introduce a data structure (an unordered hashmap) called the "tag" table as described in Table 5.1 in the runtime which dynamically detects the dependencies associated among the tasks in the form of *tags* as specified by the programmer.

5.2.1 Dependency Setup

Algorithm 1 explains the procedure with which the runtime library interprets the dependencies parsed by the compiler 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 behavior 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.

Table 5.1: Important Terminology

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 the 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, the task has no associated dependencies and is free to be scheduled. When Dep_Count != 0, 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.

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

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 based on the task pool and task queue organization applicable at runtime.

Figures 5.2 to 5.5 represent the manner in which tasks with dependencies (*tags*) once created, are added to the *tag table*.

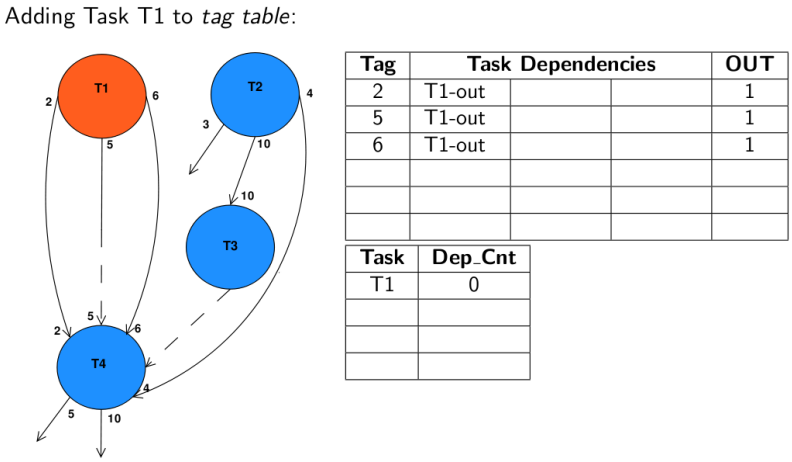
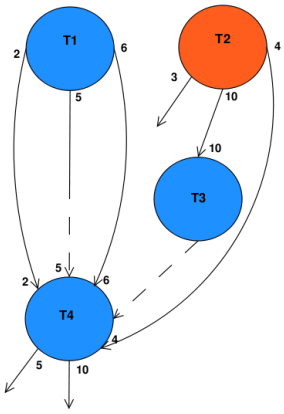


Figure 5.2: Addition of Task 1 to the *tag* table

Adding Task T2 to *tag table*:

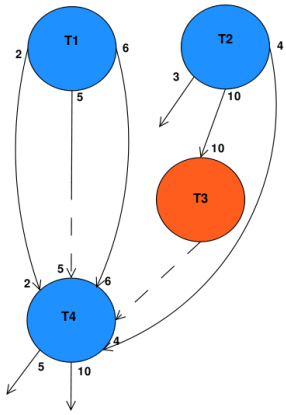


Tag	Task Dependencies			OUT
2	T1-out			1
5	T1-out			1
6	T1-out			1
3	T2-out			1
4	T2-out			1
10	T2-out			1

Task	Dep_Cnt
T1	0
T2	0

Figure 5.3: Addition of Task 2 to the *tag* table

Adding Task T3 to *tag table*:

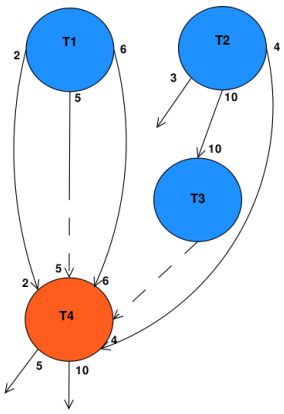


Tag	Task Dependencies			OUT
2	T1-out			1
5	T1-out			1
6	T1-out			1
3	T2-out			1
4	T2-out			1
10	T2-out	T3-in		1

Task	Dep_Cnt
T1	0
T2	0
T3	1

Figure 5.4: Addition of Task 3 to the *tag* table

Adding Task T4 to *tag table*:



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

Figure 5.5: Addition of Task 4 to the *tag* table

We notice that the value of the *Dep_Counter* varies according to the number of *IN* and *OUT* dependencies specified for each and every task. Figure 5.5 represents the state of the parent table (and *Dep_Counter*) at runtime, after all the tasks in Figure 4.1 have been created.

5.2.2 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 the scheduling of the subsequent tasks which had been waiting (in the WAIT state) for the current task to complete execution.

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 Algorithms 1 and 2, tasks T_1 to T_4 (represented in Figures 5.2 to 5.5) are scheduled in the following order:

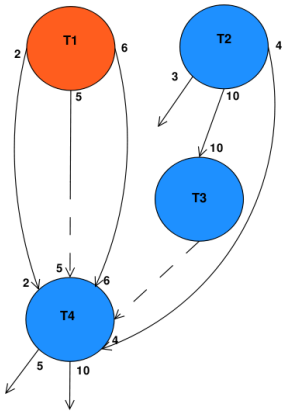
1. Task T_1 has three associated *tags* 2, 5, and 6, all of which have a *Dep_Count* with value zero. This is owing to the fact that task T_1 has three *out* dependencies (identified by *T1-out* in the parent table) that form the first node in the *Dep_List* and hence free from prior dependencies. Similarly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

Figures 5.6 to 5.9 represent the manner in which tasks after completing execution are deleted from the *tag table*. Before the tasks are removed from the table, it is essential that the *Dep_Counter* is updated for their subsequent dependent tasks accordingly, after the dependency has been resolved.

Deleting Task T1 from *tag table*:

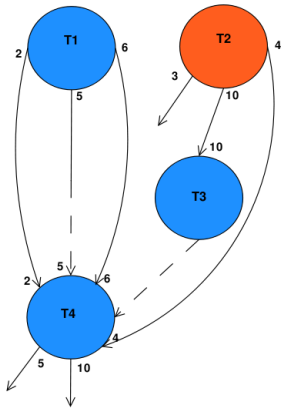


Tag	Task Dependencies			OUT
2	T1-out	T4-in		1
5	T1-out	T4-out		2
6	T1-out	T4-in		1
3	T2-out			1
4	T2-out	T4-in		1
10	T2-out	T3-in	T4-out	2

Task	Dep_Cnt
T1	0
T2	0
T3	1
T4	5

Figure 5.6: Deletion of Task 1 from the *tag* table

Deleting Task T2 from *tag table*:

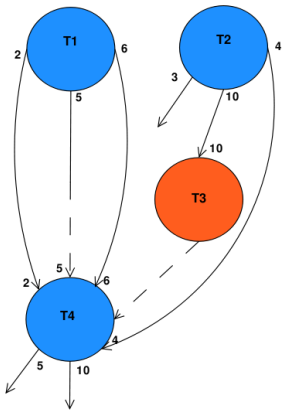


Tag	Task Dependencies			OUT
2	T4-in			0
5	T4-out			1
6	T4-in			0
3	T2-out			1
4	T2-out	T4-in		1
10	T2-out	T3-in	T4-out	2

Task	Dep_Cnt
T2	0
T3	1
T4	2

Figure 5.7: Deletion of Task 2 from the *tag* table

Deleting Task T3 from *tag table*:

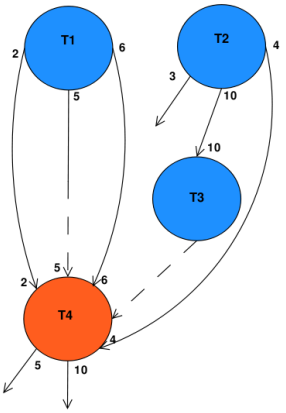


Tag	Task Dependencies			OUT
2	T4-in			0
5	T4-out			1
6	T4-in			0
4	T4-in			0
10	T3-in	T4-out		1

Task	Dep_Cnt
T3	0
T4	1

Figure 5.8: Deletion of Task 3 from the *tag* table

Deleting Task T4 from *tag table*:



Tag	Task Dependencies			OUT
2	T4-in			0
5	T4-out			1
6	T4-in			0
4	T4-in			0
10	T4-out			1

Task	Dep_Cnt
T4	0

Figure 5.9: Deletion of Task 4 from the *tag* table

Please refer to Appendix A, for a more detailed explanation on the process of the actual implementation of the two aforementioned algorithms in the OpenUH OpenMP RTL.

5.2.3 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 introduce 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 update.

5.3 Summary

In this chapter we provided an overview to the OpenUH OpenMP runtime library (RTL). We discussed the two algorithms created for the purpose of implementing the proposed extensions introduced in Chapter 4 in the OpenUH OpenMP RTL. With the help of an example we illustrated the manner in which the runtime is capable of detecting the dependencies among tasks, expressed by the programmer, and carefully process them without compromising data integrity.

Chapter 6

Performance Evaluation And Results

In this chapter we will be discussing the results obtained on applying the OpenUH task extensions on the applications proposed in Chapter 4.

At the time of implementing the extensions and procuring the results on each of the applications, we lay emphasis on the following:

- **Performance:** How well the applications perform with and without the use of the extensions. The measure of scalability with respect to the number of cores being utilized is crucial. A performance comparison between the implemented extensions has been made with respect to other similar dataflow models.
- **Programmability:** We also lay emphasis primarily on the ease of implementation. We suggest approaches with which the extensions can be implemented

with minimalistic changes in lines of code.

- **Accuracy:** Our objective was to implement the extensions in a manner that does not hamper the integrity of the results obtained. All our applications (with and without the use of the extensions) have been tested thoroughly and are error free.

6.1 Testbed

All three programs tested have been programmed in the C language. The performance results have been obtained for the following compilers:

1. GNU (gcc) version 4.7.1
2. Intel (icc) version xe12.0
3. OpenUH (uhcc) version 3.0
4. PGI (pgcc) version 11.9
5. Sun/Oracle (suncc) version 12.3
6. Barcelona Supercomputing group's OmpSs programming model - Mercurium compiler with Nanos runtime support (current updated version on website) [4]
7. We also tested our applications using the QUARK runtime API [42]

Experiments have been conducted on two testbeds:

1. 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.

2. 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.

Table 6.1 represents a summary of the experiments conducted in the chapter, explained in greater detail in the forthcoming sections.

Table 6.1: Summary of the conducted experiments

LU Decomposition (Testbed 1)	**Smith-Waterman (Testbed 2)**	**Matrix-Matrix Multiplication (Testbed 1)**
Problem size = 2048 x 2048 (-O0 optimization)	Sequence size = 4096 X 4096 (-O0 optimization)	matrix size = 2048 (-O0 optimization)
Problem size = 2048 x 2048 (-O3 optimization)	Sequence size = 8192 X 8192 (-O0 optimization)	matrix size = 2048 (-O2 optimization)
Problem size = 4096 x 4096 (-O3 optimization)	Sequence size = 8192 X 8192 (-O2 optimization)	matrix size = 8192 (-O0 optimization)

NOTE: We used the *numactl* policy for running our experiments on both the test beds. *Numactl* allows processes to run with specific NUMA scheduling or memory placement policy. We utilized the –interleave=nodes, -i nodes policy which sets a memory interleave policy for multiple nodes. This policy allows memory to be allocated using round robin on nodes. When memory cannot be allocated on the current interleave, target fall back to other nodes.

6.2 Performance Analysis for LU Decomposition Algorithm

6.2.1 Implementation

Listing 6.1 below represents the pseudo-code for implementing the LU Decomposition algorithm using the OpenMP *task* construct with the added use of the proposed extensions thereby eliminating the need to apply the first *taskwait* acting as a global synchronization point, denoted by the code depicted in Listing 4.2 in Chapter 4.

We specify an *out* dependence on *ProcessBlockOnColumn* (line 10) and *ProcessBlockOnRow* (line 13) with unique *tags* '2*i' and '2*i+1' respectively. With an *in* dependence specified in line 19, we allow the tasks in *ProcessInnerBlock* to begin execution immediately after their corresponding dependencies in blocks *ProcessBlockOnColumn* (2*i) and *ProcessBlockOnRow* (2*i+1) have been addressed at runtime. This flexibility of task execution in the hands of the programmer warrants the reduction of overheads normally encountered (in the absence of the proposed extensions) while having to wait until all the tasks executing *ProcessBlockOnColumn* and *ProcessBlockOnRow* have concluded, prior to the execution of *ProcessInnerBlock* (the first *taskwait* in Listing 4.2) in order to maintain data integrity.

```
#pragma omp parallel
{
 #pragma omp master
 {
   for ( i=0; i<matrix_size; i++ ) {
/**** Processing Diagonal block ****/
   ProcessDiagonalBlock(.......);
   for (i=1;i<M;i++){
/**** Processing block on column ****/
   #pragma omp task out(2*i)
     ProcessBlockOnColumn(........);
/**** Processing block on row ****/
   #pragma omp task out(2*i+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 6.1: LU Decomposition Algorithm using OpenMP tasks with the proposed extensions [41]

6.2.2 Results Obtained with Elimination of Global Synchronization Points

6.2.2.1 Results obtained with -O0 (no) optimization

The objective for obtaining results with no optimization is crucial to validate, that the performance improvement observed with the use of the OpenUH task extensions, could solely be attributed to the elimination of the *taskwait*, without the interference of any other compiler optimizations. Any improvement observed in such a case could only be attributed to the tasks executing in a specific order adhering to their respective data dependencies, without compromising the accuracy of the computation.

6.2.2.1.1 Experiment: Matrix size 2048 X 2048, with varying number of threads - Figure 6.1 measures the speedup obtained by varying the number of cores from 1 to 48 for 16 blocks per dimension on a matrix of size 2048 X 2048.

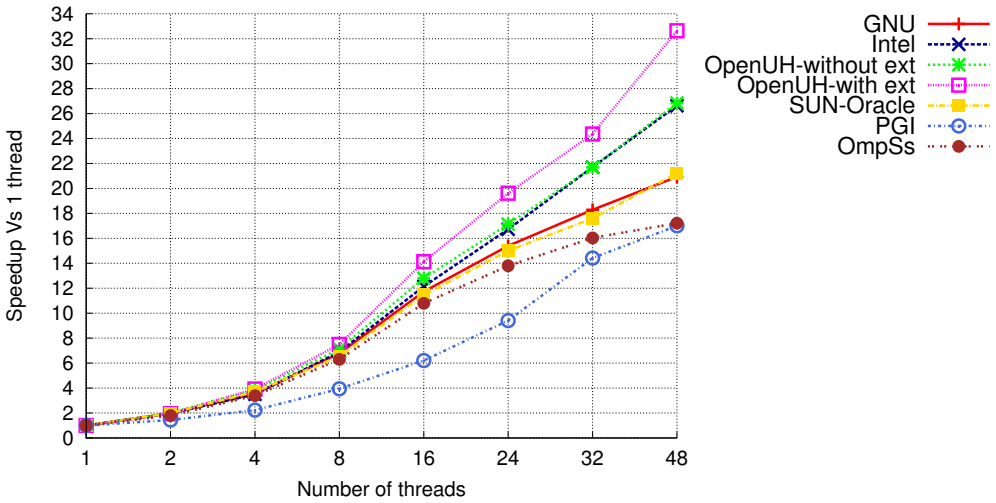


Figure 6.1: Scalability measure for matrix size 2048 X 2048 with no optimization

Table 6.2 shows the performance obtained for the LU decomposition algorithm in seconds, for a matrix of size 2048 X 2048 with 16 blocks per dimension.

Table 6.2: Performance obtained for matrix 2048 X 2048 (in seconds) with no optimization

Threads	**GNU**	**Intel**	**OpenUH_noext**	**OpenUH_ext**	**Sun**	**PGI**	**OmpSs**
1	40.874	39.298	45.044	45.006	43.8	20.107	33.337
2	20.586	19.904	22.269	23.11	22.039	13.941	18.569
4	11.555	11.225	11.766	11.466	11.856	9.012	9.815
8	6.028	5.696	6.31	6.001	6.69	5.107	5.287
16	3.485	3.234	3.518	3.186	3.821	3.239	3.089
24	2.655	2.349	2.629	2.294	2.919	2.138	2.416
32	2.235	1.81	2.077	1.847	2.489	1.394	2.079
48	1.952	1.476	1.678	1.379	2.065	1.182	1.936

We observe that the version implemented with the OpenUH task extensions scales up to 32X for 48 cores in comparison to the performance on a single core. This is 1.23 times better than the speedup obtained for Intel and OpenUH (without extensions) compilers, which scale up to 26X for 48 cores. However, it is interesting to note that for the PGI compiler, which delivers speedup of only 17X on 48 cores, in terms of performance, is the only compiler which outperforms the performance obtained on the OpenUH compiler (with task extensions), by approximately 15% on 48 cores.

6.2.2.1.2 Experiment: Matrix size 2048 X 2048, with varying number of blocks per dimension - Figure 6.2 measures the performance of the implemented extensions in comparison to the conventional tasking version, by varying the number blocks per dimension on a matrix of size 2048 X 2048 with 48 threads.

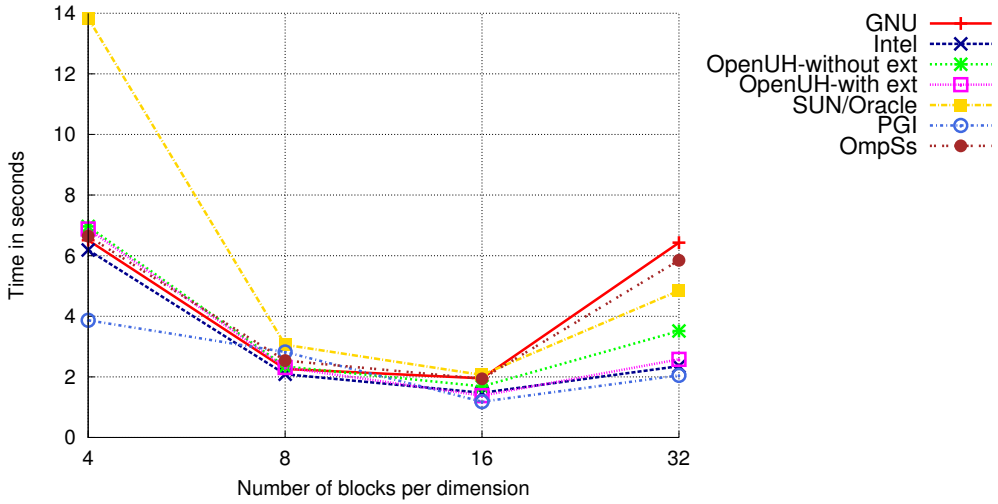


Figure 6.2: Performance measure varying blocks per dimension for matrix size 2048 with no optimization

As mentioned earlier in Chapter 4 we employ a blocking strategy in order to promote data reuse and decrease memory latency, wherein the matrices are partitioned into smaller blocks so that they could fit in memory registers. This allows spatial locality to improve and can be acknowledged as a significant memory optimization uplifting code performance.

We denote each block explicitly as an individual OpenMP task. We observe improvement in performance as we gradually increase the number of blocks per dimension from 4 to 8 up to 16. However, implementing 32 blocks per dimension degrades performance mainly due to the creation of excessive tasks (too fine grained), which adds additional overheads attributed to task creation/deletion as well as task synchronization.

6.2.2.2 Results obtained with -O3 optimization

In this section we test the LU Decomposition kernel using -O3 level optimization. Our focus is to draw a comparison with respect to the performance obtained with -O3 enabled optimization, on OpenUH (with task extensions) and the other commercial compilers.

6.2.2.2.1 Experiment: Matrix size 2048 X 2048, with varying number of threads - Figure 6.3 represents the speedup obtained across 48 cores, observed for data size 2048 X 2048, comparing tasking implementations of different compilers in addition to the extensions implemented in the OpenUH runtime library. The number of blocks per dimension is set to 8.

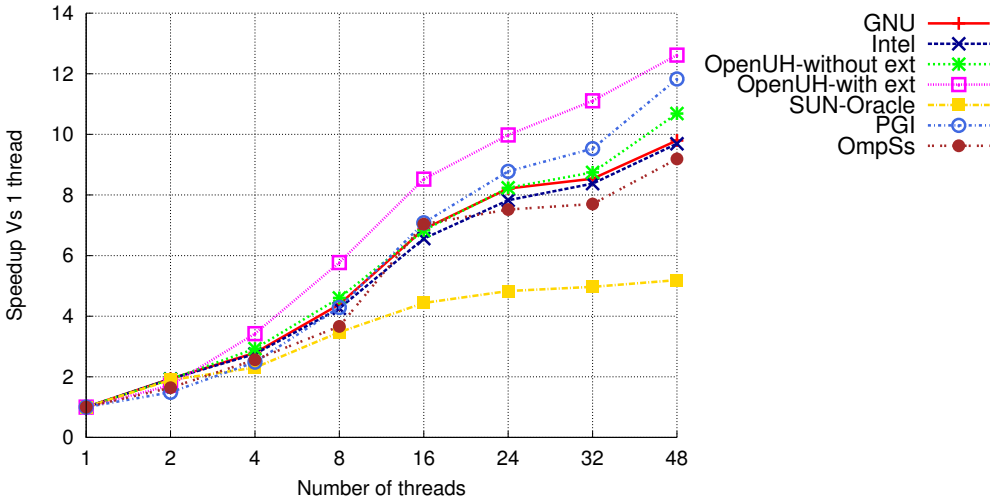


Figure 6.3: Scalability measure for matrix size 2048 with -O3 optimization

Similarly Table 6.3 refers to the corresponding performance numbers obtained for data size 2048 X 2048, with 8 blocks per dimension.

Table 6.3: Performance obtained for matrix 2048 X 2048 (in seconds) with 8 blocks per dimension

Threads	GNU	Intel	OpenUH_noext	OpenUH_ext	Sun	PGI	OmpSs
1	5.54	4.86	5.47	5.47	4.72	7.02	6.75
2	2.86	2.52	2.85	3.18	2.49	4.72	4.12
4	1.99	1.76	1.87	1.6	2.05	2.83	2.64
8	1.26	1.14	1.19	0.95	1.36	1.64	1.84
16	0.81	0.74	0.8	0.64	1.06	0.99	0.96
24	0.67	0.62	0.66	0.55	0.98	0.8	0.9
32	0.65	0.58	0.62	0.49	0.95	0.74	0.88
48	0.57	0.5	0.51	0.43	0.91	0.59	0.73

We observe that OpenUH (with task extensions) obtains a maximum speedup of more than 12X, and outperforms the speedup obtained by the GNU, Intel, PGI, Sun/Oracle, OmpSs and OpenUH (without task extensions) compilers, by 1.32X, 1.16X, 1.37X, 2.11X, 1.69X and 1.18X respectively. It is interesting to note that, unlike the previous test case (with no optimization), the OpenUH task extensions, with -O3 optimization, contributed a 27% performance improvement, in comparison to the PGI compiler.

6.2.2.2.2 Experiment: Matrix size 2048 X 2048, with varying number of blocks per dimension - Figure 6.4 represents the performance obtained for matrix size 2048 X 2048 with -O3 level optimization by varying the number of blocks per dimension.

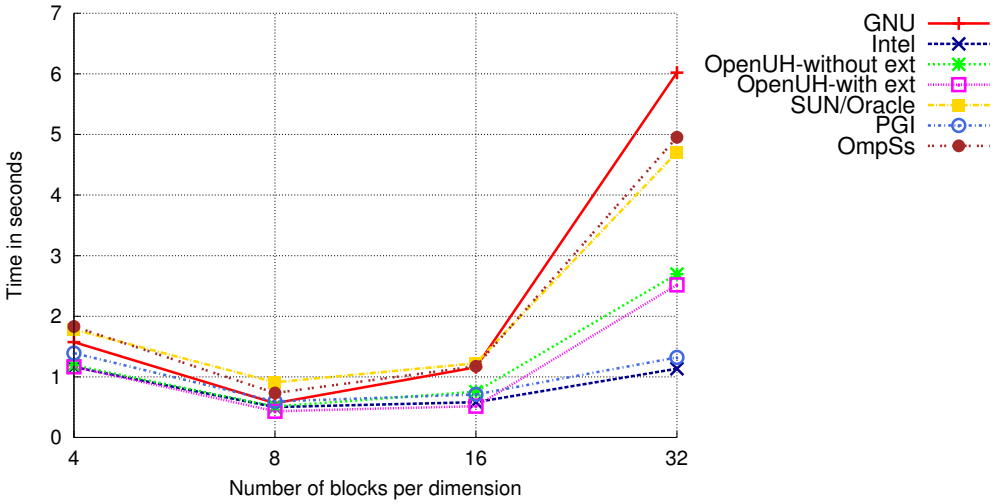


Figure 6.4: Performance measure varying blocks per dimension for matrix size 2048 with -O3 optimization on 48 cores

The extensions provide improved performance with increasing number of blocks per dimension in comparison to the remaining compilers depicted in Figure 6.4. This could be attributed to a more flexible distribution of the workload amongst the threads owing to an improved scheduling strategy invoked with the use of the extensions in the OpenUH runtime library (RTL). Extending the value of the number of blocks per dimension beyond 16 resulted in performance degradation due to excessive task creation and synchronization overheads as explained earlier.

6.2.2.2.3 Experiment: Matrix size 4096 X 4096, with varying number of threads - Figure 6.5 presents the speedup obtained across 48 cores when applying LU decomposition on a matrix of size 4096 X 4096 with -O3 level optimization, with 16 blocks per dimension.

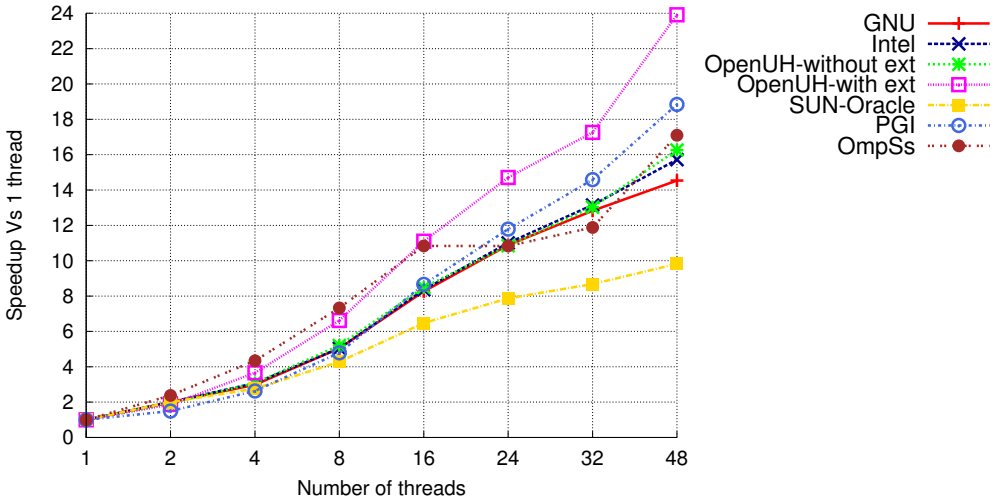


Figure 6.5: Speedup obtained for matrix size 4096 X 4096 with -O3 optimization

Table 6.4: Performance measuring scalability (in seconds) for matrix 4096 X 4096, with 16 blocks

Threads	GNU	Intel	OpenUH_noext	OpenUH_ext	Sun	PGI	OmpSs
1	58.94	52.49	58.84	58.9	50.22	71.56	93.24
2	29.57	26.24	30.28	31.51	25.06	47.93	39.2
4	19.77	17.05	19.14	16.1	18.22	27.2	21.5
8	11.69	10.41	11.3	8.9	11.73	14.94	12.72
16	7.13	6.28	6.93	5.3	7.76	8.26	8.61
24	5.41	4.77	5.42	4	6.38	6.07	8.61
32	4.6	3.99	4.52	3.41	5.79	4.9	7.85
48	4.05	3.34	3.62	2.46	5.11	3.8	5.45

Figure 6.5 indicates that OpenUH (with task extensions) obtains a speedup of up to 24X, which outperforms the performance obtained from the Intel and PGI compilers (having obtained 16X and 18X speedup respectively) by 27% and 36% respectively.

6.2.2.2.4 Experiment: Matrix size 4096 X 4096, with varying number of blocks per dimension - Figure 6.6 represents the speedup obtained on varying the number of blocks per dimension, for a matrix of size 4096 X 4096 with -O3 level optimization.

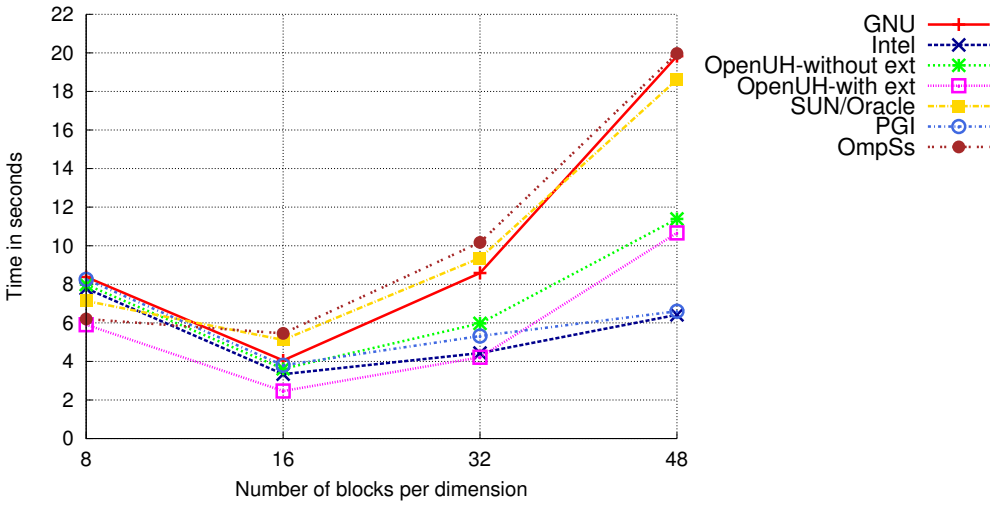


Figure 6.6: Performance measure varying blocks per dimension for matrix size 4096 with -O3 optimization on 48 cores

The choice for the number of blocks per dimension in order to obtain the optimum performance depends on the size of the last level cache. Performance results from Figure 6.4 indicate, that use of 8 blocks per dimension for matrix size 2048 X 2048 proved better than the usage of 16 or 32 blocks per dimension. From Figure 6.6 we gather that the use of 16 blocks per dimension for matrix size 4096 X 4096 proved to be more advantageous compared to the usage of 8 or 32 blocks. Therefore in order to obtain the optimum performance for a given size of input data, we need to take into consideration the size of the block of data to be prefetched within the cache, in order to exploit the advantages of cache affinity, to obtain increased spatial

locality. However, it is also important to take into consideration the availability of space within the last level of cache. Hence, the trade-off between the distribution of workload with appropriate block size and the available cache size is crucial in terms of obtaining the best performance.

6.2.3 Performance Comparison to other Dataflow Models

We implemented LU Decomposition on runtime API's which mirror dataflow models such as the OmpSs programming model and QUARK runtime API. Our objective lay mainly in comparing the performance obtained using the OpenUH task extensions with related dataflow models as mentioned above. We wish to make an assessment on how well the extensions provided by OmpSs, QUARK scale in comparison to OpenUH for the LU Decomposition benchmark, with varying data sizes.

6.2.3.1 Experiment: Matrix size 2048 X 2048, with varying number of threads, -O0 optimization

Figure 6.7 shows the scalability obtained on testing a matrix of size 2048 X 2048, with 16 blocks per dimension, using -O0 optimization across OpenUH (with the use of the extensions), the OmpSs Programming model and QUARK.

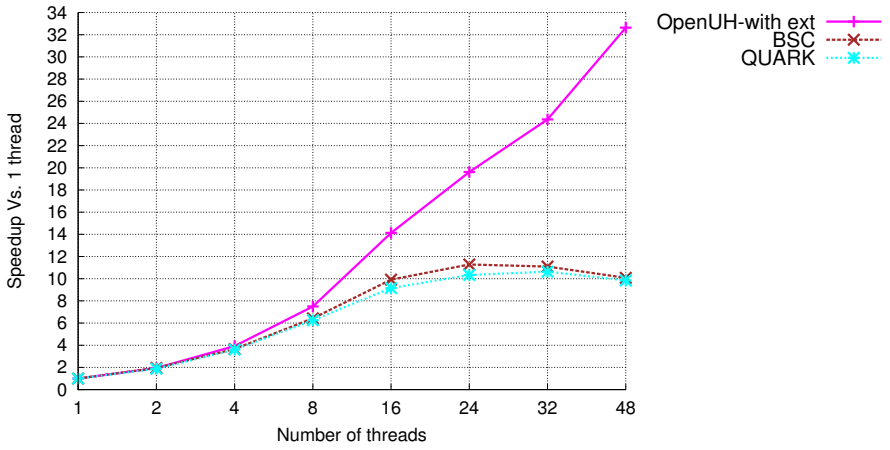


Figure 6.7: Performance measure varying the number of threads for matrix size 2048 with no optimization

6.2.3.2 Experiment: Matrix size 2048 X 2048, with varying number of threads, -O3 optimization

Figure 6.8 represents the speedup obtained for the implemented extensions for matrix size 2048 X 2048, with 8 blocks per dimension, using -O3 optimization, in comparison to dataflow models - OmpSs and QUARK.

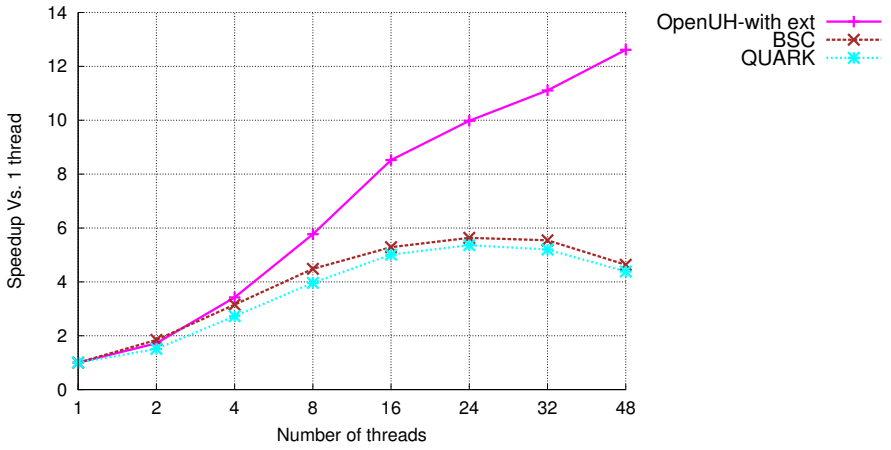


Figure 6.8: Performance measure varying the number of threads for matrix size 2048 with -O3 optimization

The performance obtained from OpenUH far supersedes the performance obtained from OmpSs and QUARK in terms of both scalability and overall speedup. Both OmpSs and QUARK scale to a maximum of 24 cores, beyond which we notice performance degradation. We notice a similar pattern for data size 2048 X 2048, for both -O0 and -O3 level of optimizations. OpenUH shows a performance benefit of 2.7X and 2.9X in comparison to the OmpSs and QUARK extensions respectively, for matrix size 2048 X 2048 with -O3 level of optimization enabled.

However it is curious to note that, in the case of OpenUH, where, with the application of the extensions on a matrix of size 2048 X 2048 (with -O0 optimization), we observe a performance improvement of 18.7%, there is a noticeable degradation in performance estimated close to 50%, observed for OmpSs. This can be attributed to the overhead generated in maintaining a task dependency graph, in order to track the explicit dependencies among tasks by the OmpSs runtime.

6.2.3.3 Experiment: Matrix size 4096 X 4096, with varying number of threads, -O3 optimization

Figure 6.8 represents the speedup obtained for the implemented extensions for matrix size 4096 X 4096 with -O3 optimization in comparison to dataflow models - OmpSs and QUARK.

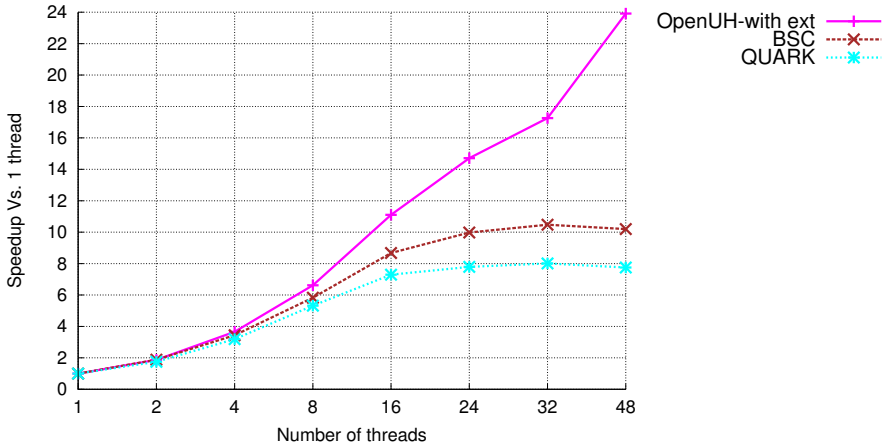


Figure 6.9: Performance measure varying the number of threads for matrix size 4096 with -O3 optimization

We observe that with the increase in data size (4096 X 4096), OpenUH continues to perform 2.3X and 3X times better in comparison to OmpSs and QUARK respectively. However, the OmpSs scales upto 32 cores in this particular test case, and manages to reduce the performance degradation from 50% (as mentioned in the prior test case) to 26%.

6.3 Performance Results for Smith-Waterman Algorithm

6.3.1 Implementation

We have implemented two approaches incorporating the OpenUH task extensions in order to support a specific ordering among tasks, keeping true to their respective data dependencies.

Both approaches provide a significant improvement over the conventional tasking versions discussed in Chapter 4. The first approach (Version 1) shows improved scalability and performance in comparison to the second approach (Version 2), although Version 2 is easier to program when compared to Version 1. As mentioned earlier in Chapter 4, both approaches have been implemented on top of the chunked tasking implementation of the Smith-Waterman algorithm as depicted in Figure 6.10.

Figure 6.10, represents the chunked tasking version with a chunk size of two. Every task (as denoted by the individual numbers included in every element of the matrix) is composed of two elements (owing to chunk size two) and is represented by an alternate color.

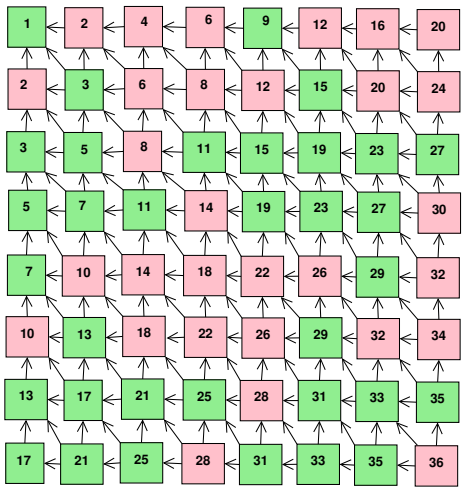


Figure 6.10: Smith Waterman - chunked tasking implementation

6.3.1.1 Version 1 approach

Listing 6.2 illustrates the implementation of the OpenUH task extensions to the chunked tasking version of the algorithm, eliminating the *taskwait* in line 43.

```
1  #pragma omp parallel
2   {
3  #pragma omp master
4     {
5         for(wave = 0; wave < waves; ++wave) {
6         /* computation of nbr of elements per wave & its location within the matrix */
7         wave2 = wave1;  wave1 = task_count;  task_count=0;
8         for(ii = 0; ii < elements; ii+=chunk) {
9          tid++;
10         if (line == -1 && elements <= chunk) /* Region 1 */
11         {
12 #pragma omp task in(tid-1) in(tid -2) out(tid)
13             {
14               for (i = ii; i < MIN(elements,ii + chunk); i++)
15                    SW_kernel();
16             } // task
```

```
17          }
18          else if( line <= 0 && elements > chunk) /* Region 2 */
19          {
20  #pragma omp task in(tid-wave1) in(tid-wave1-1) in(tid-(wave1 + wave2))
21                                  in(tid-(wave1 + wave2 -1)) out(tid)
22              {
23              for (i = ii; i < MIN(elements,ii + chunk); i++)
24                  SW_kernel();
25              } // task
26          }
27          else if( line > 0 && elements >= chunk) /* Region 3 */
28          {
29  #pragma omp task in(tid-wave1) in(tid-wave1+1) in(tid-(wave1 + wave2))
30                                  in(tid-(wave1 + wave2 +1)) out(tid)
31              {
32              for (i = ii; i < MIN(elements,ii + chunk); i++)
33                  SW_kernel();
34              } // task
35          }
36          else                        /* Region 4 */
37          {
38  #pragma omp task in(tid-1) in(tid-2) in(tid-3) out(tid)
39              {
40              for (i = ii; i < MIN(elements,ii + chunk); i++)
41                  SW_kernel();
42              } // task
43          }
44        task_count++;
45  /* elimination of the taskwait */
46     } // inner for loop
47    } // outer for loop
48   } // master
49  } // parallel
```

Listing 6.2: Implementation for Smith-Waterman with task extensions - Version 1

A specific ordering of the tasks as represented in Listing 6.2 for Version 1, relies on four separate *IF* conditions, each depicting a particular region in the scoring matrix. It is particularly challenging to specify the dependencies for every task in a wavefront-based pattern since, unlike the LU Decomposition algorithm where the data dependencies were fixed, the dependencies for every task (or chunk of elements) for this algorithm, vary based on the location of that task in the scoring matrix. We make an assessment, based on Figure 6.11, where a task belonging to either of the four regions demarcated from 1 to 4, will possess dependencies, on a different set of tasks, as expressed in Listing 6.2. We track the dependencies for each task, (pertaining to the region to which the task belongs) by storing the number of elements processed in the prior two diagonals, within variables *wave1* and *wave2* as seen in line 7. We estimate a pattern where the dependencies of every task relies on specific tasks in the previous two diagonals. We notice that this pattern is common to all tasks in their respective regions, which we incorporate in the four separate *IF* conditions.

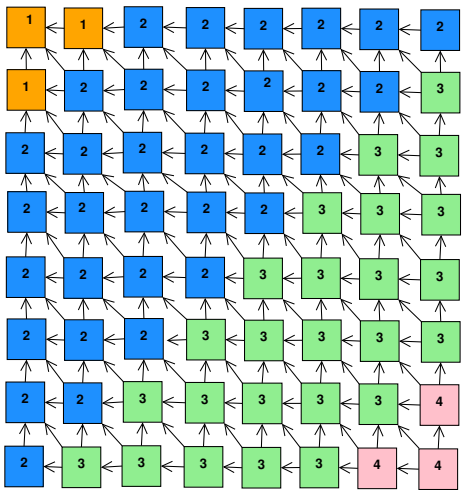


Figure 6.11: Smith Waterman OpenUH task implementation depicted in Listing 6.2, displaying the 4 separate regions

The explicit expression for dependent North, West, and North-West neighbors of every task individually, negates the need for the *taskwait* in line 43, thereby allowing multiple diagonals (or waves) to be processed concurrently, providing scope for more parallelism.

6.3.1.2 Version 2 approach

Listing 6.3 illustrates an alternative implementation of the OpenUH task extensions to the chunked tasking version of the algorithm, eliminating the *taskwait* in Line 26. Listing 6.3 represents the pseudo-code for implementing the second version, that allows specifying the requisite dependencies for each task, thereby eliminating the need to specify a *taskwait* between iterations of the *for* loop in line 5 processing elements on each wave in the process of constructing the final scoring matrix. As seen the code for this version is more concise and allows specifying dependencies on the source and sink elements of each dependent task (or chunk) in the north, west and north-west directions. However, this approach incorporates a local 1-D array (as seen line 20) to store the values of each task (as *out* dependence), so that we could later track its respective *in* dependencies as seen in line 13.

Even though this approach (Version 2) is easier to implement and guarantees fewer changes in lines of code, its performance cannot surpass the performance obtained from Version 1. This is owing to an increase in the number of load and store operations in comparison to the number of floating-point operations, thereby compromising the achievable scalability.

```
#pragma omp parallel
 {
#pragma omp master
   {
      for(wave = 0; wave < waves; ++wave) {
      /* computation of nbr of elements per wave & its location within the matrix */
      for(ii = 0; ii < elements; ii+=chunk)
      {
         tid++;
         NW1 = TASK_ID(np-ii-1,mp+ii-1);  NW2 = TASK_ID(np-(min-1)-1,mp+(min-1)-1);
         N1 = TASK_ID(np-ii-1,mp+ii);  N2 = TASK_ID(np-(min-1)-1,mp+(min-1));
         W1 = TASK_ID(np-ii,mp+ii-1);  W2 = TASK_ID(np-(min-1),mp+(min-1)-1);
#pragma omp task in(NW1) in(NW2) in(N1) in (N2) in(W1) in(W2) out(tid)
         {
           for (i = ii; i < MIN(elements,ii + chunk); i++)
                SW_kernel();
         } // task
         for(i=ii; i< min; i++)
            {
              H1[(np-i)*N_a+(mp+i)] = tid;
            }
      } // inner for loop
/* elimination of taswait */
    } // outer for loop
  } // end of master
} // end of parallel
```

Listing 6.3: Implementation for Smith-Waterman with task extensions - Version 2

We will discuss the performance results obtained for either version in the next section. For the purpose of this thesis, we will solely consider a symmetric square scoring matrix, implying that the two different sequences being tested for performing

local alignment, will always be of the same size.

6.3.2 Results Obtained with Elimination of Global Synchronization Points

6.3.2.1 Sequence size 4096 with -O0 level optimization

Figure 6.12 represents the performance obtained for sequences of size 4096, with -O0 level optimization where tasks have been created as chunks of 320 (grouping every 320 consecutive elements) per diagonal. It can deduced that compilers - Intel, GNU, Oracle/Sun and OpenUH perform relatively in a similar fashion scaling up to 8 threads, whereas PGI compiler demonstrates the least amount of scalability. Both versions 1 and 2 implemented with the OpenUH task extensions, perform well in terms of performance securing an improvement of 84% and 79% respectively, in comparison to the OpenUH compiler, without the use of the extensions.

Another observation suggests that none of the compilers are able to produce scalable results beyond 8 threads. This can be attributed to the fact the amount of work performed by a task is significantly less, compared to the number of load and store operations executed. Hence, the implementation is deemed more memory bound. The master thread creates all the tasks, which are acquired by the slave threads to perform the work. The tasks being smaller in size, allow the slave threads to execute it fast enough and bombard the master thread's queue looking for more work to steal. With the increase in the number of threads, the amount of work performed by each thread decreases, thereby increasing the amount of contention

on the master thread's queue. This results in poor load balancing with the master thread performing majority of the work, which adversely affects the scalability as seen in Table 6.5.

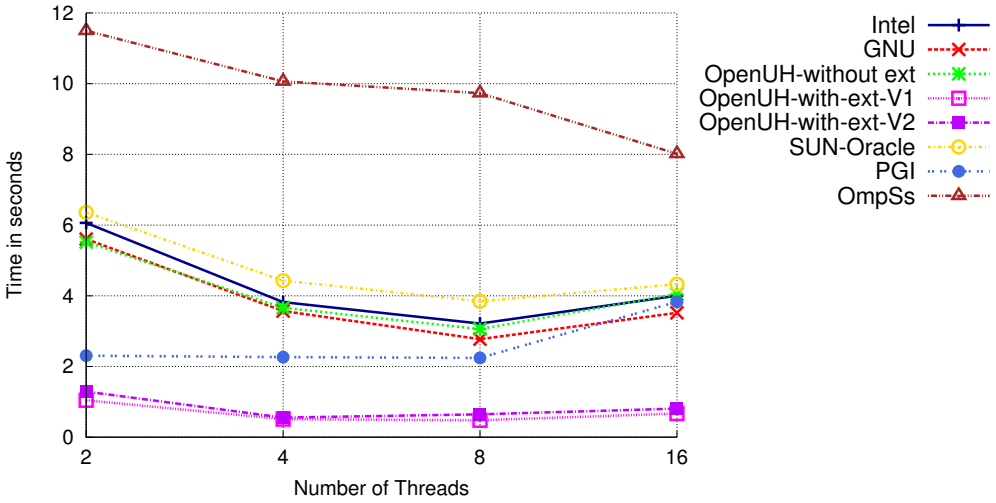


Figure 6.12: Performance obtained by varying the number of threads for sequence size 4096 with -O0 optimization

Table 6.5: Performance of Smith-Waterman algorithm (in seconds) for sequence size 4096 for task chunk of 320 with -O0 optimization

Threads	**Intel**	**GNU**	**OpenUH no ext**	**OpenUH ext V1**	**OpenUH ext V2**	**Sun**	**PGI**	**OmpSs**
2	6.062	5.604	5.519	1.045	1.291	6.360	2.305	11.497
4	3.819	3.569	3.661	0.511	0.553	4.429	2.266	10.065
8	3.215	2.774	3.057	0.480	0.646	3.845	2.242	9.733
16	4.001	3.518	4.051	0.669	0.807	4.327	3.830	8.014

6.3.2.2 Sequence size 4096 with -O0 level optimization, varying chunk size

Table 6.6 and Figure 6.13 depict the performance (in seconds) obtained for sequence size 4096, on 8 threads by varying the task chunk size, with -O0 level optimization.

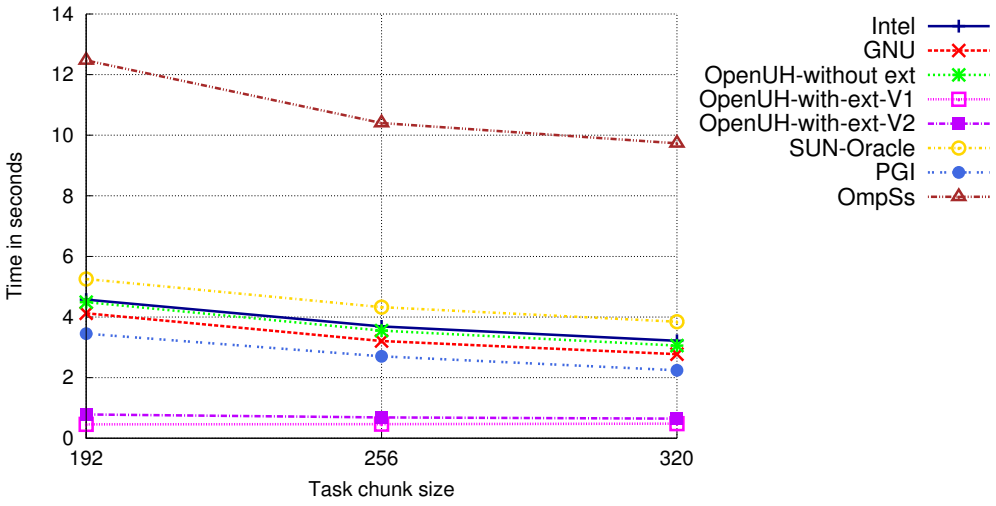


Figure 6.13: Performance obtained by varying the task chunk size for sequence size 4096 with -O0 optimization

Table 6.6: Performance of Smith-Waterman algorithm (in seconds) for sequence size 4096 on 8 threads with varying chunk size

Chunk size	**Intel**	**GNU**	**OpenUH no ext**	**OpenUH ext V1**	**OpenUH ext V2**	**Sun**	**PGI**	**OmpSs**
192	4.576	4.122	4.490	0.457	0.783	5.254	3.452	12.467
256	3.689	3.209	3.550	0.466	0.686	4.324	2.703	10.401
320	3.215	2.774	3.057	0.480	0.646	3.845	2.242	9.733

We observe based on the readings specified in Table 6.6 that the OpenUH extensions version 2, shows improvement in performance in comparison to version 1

with gradual increase in chunk size. However, judging by the overall performance and scalability, OpenUH task extensions version 1 provides the optimum result securing a speedup of 6.6X, 5.6X, 7.9X, 4.5X and 16X compared to its Intel, GNU, Sun/Oracle, PGI, and OmpSs counterparts.

6.3.2.3 Sequence size 8192 with -O0 level optimization, varying number of threads

Figure 6.14 and Table 6.7 represents the performance obtained for sequences of size 8192, with -O0 (no) level optimization where tasks have been generated as chunks of 512 elements per diagonal. We observe that OpenUH compiler without the task extensions performs significantly better than its Intel, GNU, Sun/Oracle, PGI, and OmpSs counterparts by a margin of 35%, 27%, 44%, 11%, and 62% respectively. However, the version 1 and 2 implementing the OpenUH task extensions beat OpenUH version by a margin of 5.5X and 4X respectively.

Table 6.7: Performance obtained (in seconds) for sequence size 8192 with no optimization, on varying number of threads

Threads	**Intel**	**GNU**	**OpenUH no ext**	**OpenUH ext V1**	**OpenUH ext V2**	**Sun**	**PGI**	**OmpSs**
2	29.469	27.228	14.509	4.478	5.580	30.421	12.803	41.327
4	18.465	17.253	10.037	1.818	2.053	20.661	11.652	35.660
8	14.708	13.040	9.529	1.780	2.305	17.079	10.793	29.007
16	17.510	15.376	14.995	2.780	2.905	18.870	14.762	25.354

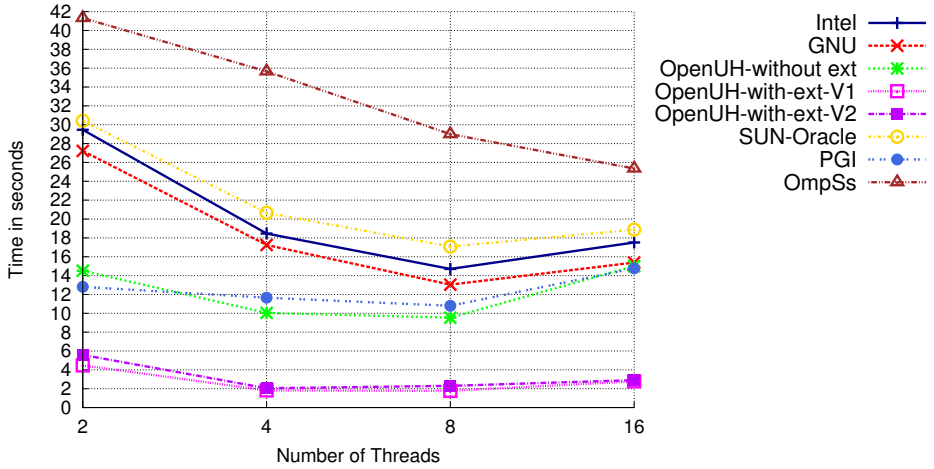


Figure 6.14: Performance obtained by varying the number of threads for sequence size 8192 with -O0 optimization

6.3.2.4 Sequence size 8192 with -O0 level optimization, varying chunk size

Figure 6.15 and Table 6.8 represents performance obtained by varying the task chunk size for sequence size 8192 on 8 threads with no optimization.

We observe that with the gradual increase in the chunk size, the noticeable improvement in performance, is owing to creation of fewer tasks. We also notice that OmpSs scales not only with the increase in chunk size, but also with the increase in the number of threads. However, this comes at a cost of overall inefficient performance (brought by the overhead generated for maintaining the task dependencies), negating the merits achieved with scalability.

Table 6.8: Performance (in seconds) for sequence size 8192, with varying chunk size on 8 threads

Chunk_size	Intel	GNU	OpenUH no ext	OpenUH ext V1	OpenUH ext V2	Sun	PGI	OmpSs
256	26.253	23.871	15.632	1.732	2.813	29.164	18.055	40.152
320	21.368	19.247	13.027	1.718	2.566	24.237	14.601	34.588
512	14.708	13.040	9.529	1.780	2.305	17.079	10.793	29.007

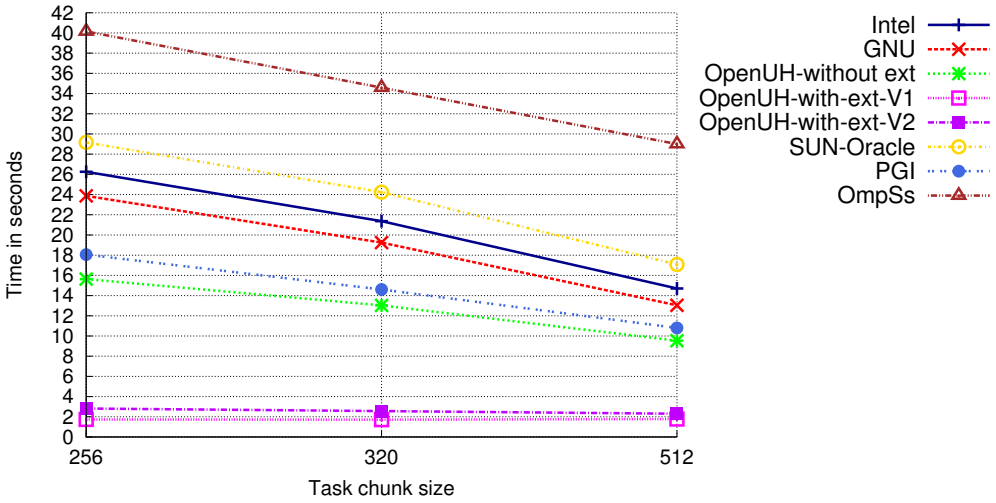


Figure 6.15: Performance obtained by varying the task chunk size for sequence size 8192 with -O0 optimization

6.3.2.5 Sequence size 8192 with -O2 level optimization

Figure 6.16 and Table 6.9 represent the performance obtained for sequence size 8192, across varying number of threads, with task chunk size 512 and -O2 level optimization.

We notice that the Intel compiler performs almost at par with the version 2

implementation of the OpenUH task extensions, whereas the OpenUH compiler does not scale well beyond 4 threads, unlike the Sun/Oracle and GNU compilers which continue to scale well upto 8 threads.

However, version 1 OpenUH task extensions continues to outperform the performance of the Intel, GNU, Sun/Oracle, PGI, and OmpSs by factors of 2.2X, 5.5X, 6X, 10X, and 18X respectively.

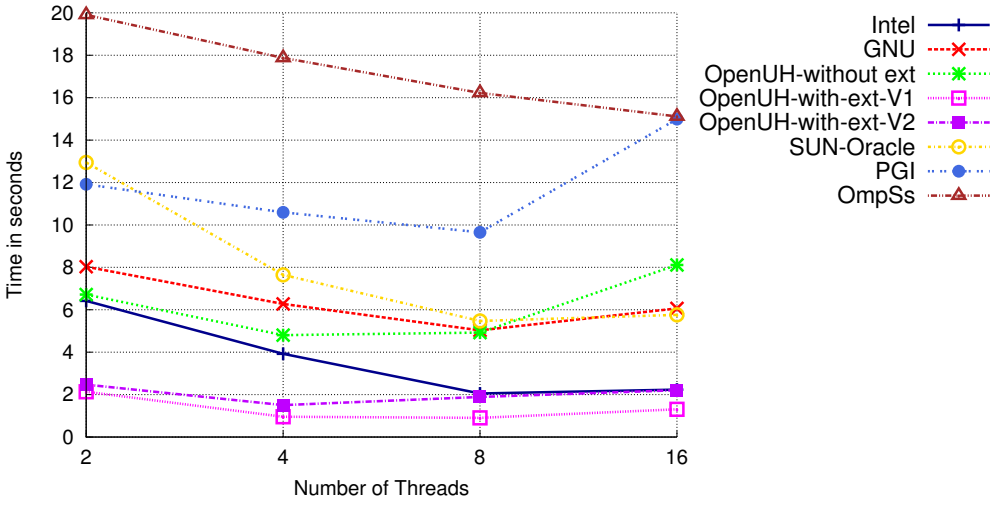


Figure 6.16: Performance obtained by varying the number of threads for sequence size 8192 with -O2 optimization

Table 6.9: Performance (in seconds) for sequence size 8192, task chunk 512 and -O2 optimization

Threads	**Intel**	**GNU**	**OpenUH no ext**	**OpenUH ext V1**	**OpenUH ext V2**	**Sun**	**PGI**	**OmpSs**
2	6.422	8.029	6.720	2.137	2.473	12.945	11.918	19.902
4	3.925	6.275	4.804	0.959	1.505	7.646	10.593	17.873
8	2.055	5.029	4.926	0.906	1.889	5.474	9.651	16.225
16	2.236	6.063	8.113	1.309	2.207	5.759	14.991	15.113

6.3.3 Performance Comparison to other Dataflow Models

We implemented the Smith-Waterman algorithm on the aforementioned dataflow models - OmpSs and QUARK.

The implementation for the OmpSs programming model entails, tracking the respective source and sink memory addresses of the north, west, and north-west neighboring dependent tasks as the *input* parameters, and specifying the current task's source and sink memory addresses as the *output* parameters.

Tables 6.10 to 6.12 denote the performance (in seconds) obtained from the two dataflow models, across varying threads for sequence size 4096 and 8192 tested with task chunk sizes 320 and 512 with -O0 and -O2 optimizations respectively.

Table 6.10: Performance comparison with dataflow models (in seconds) for size 4096, chunk 320, with varying number of threads,-O0 optimization

Threads	**OpenUH_ext**	**OmpSs_ext**	**Quark**
2	1.045	52.251	2.639
4	0.511	50.640	2.278
8	0.480	48.645	2.081
16	0.669	46.256	2.395

Table 6.11: Performance comparison with dataflow models (in seconds) for size 8192, chunk 512, with varying number of threads, -O0 optimization

Threads	**OpenUH_ext**	**OmpSs_ext**	**Quark**
2	4.47	392.92	10.63
4	1.81	344.02	9.11
8	1.77	314.22	8.13
16	2.78	308.15	8.32

Table 6.12: Performance comparison with dataflow models (in seconds) for size 8192, chunk 512, with varying number of threads - O2 optimization

Threads	**OpenUH_ext**	**OmpSs_ext**	**Quark**
2	2.13	388.39	8.23
4	0.96	324.09	7.12
8	0.90	308.44	6.38
16	1.30	301.76	7.76

For all the above test cases, we observe that the OpenUH task extensions not only sustains the overall best performance, but also in the absence of global synchronization points, incurs the least overhead by effectively scheduling the tasks.

We noticed that OmpSs, incurred large overhead which significantly impacted its performance. This can primarily be attributed to the fact that the master thread invests significant amount of its time in the maintenance of the task graph (which allows tracking of the dependencies among the tasks), and only around 65% of its time is actually invested for execution of the tasks. The worker threads also suffer some overheads, not only due to the maintenance of the task graph, but also for the time the threads are kept waiting for ready tasks to be executed. Additional overhead is incurred due to the time invested in updating the tables nested in each hierarchical level of the task graph storing variables (depicting active dependencies) associated with the last node being written to in the graph.

Similar to OmpSs, the QUARK API manages the data dependencies among tasks with a task graph implementation. However it differs from the OmpSs model on the grounds that it integrates the PLASMA linear algebra library. With the PLASMA

library incorporating a static scheduler incurring no overhead, coupled with optimizations enabled in QUARK (e.g DAG merging, loop reordering), justifies its better performance in comparison to OmpSs. The drop in performance for QUARK is partially due to the fact that, the master thread remains completely devoted to detecting dependencies and inserting the tasks in the DAG, without participating in executing them. Despite these features, QUARK has been designed to cater performance, especially for linear algebraic problems, possessing dense computation intensive kernels. For a memory bound algorithm such as Smith-Waterman, it is unable to surpass the performance obtained from the OpenUH task extensions.

Both OmpSs and QUARK allow created tasks to immediately populate the task pool, mandating the need for the runtime scheduler to scan all the tasks within the pool in order to identify a task free from data dependencies. Once identified, this task is assigned to an available core to be executed. This poses an additional overhead on the runtime scheduler which is required to scan the entire task pool before assigning work to available cores. In the OpenUH OpenMP runtime library, we eliminate this overhead by placing tasks in the task pool only after their respective data dependencies have been resolved.

6.4 Dataflow Model Overhead Analysis

We have used the Matrix Multiplication kernel, an *embarassingly parallel* microbenchmark, to draw an assessment in terms of the amount of overhead generated by each of the dataflow models discussed in the previous sections. Our objective was to choose

an application, which required little to no need for coordination among tasks, so that, the overhead estimated for each model, could be directly accountable to their respective implementations in the runtime.

The experiments have been performed on a blocked matrix multiplication kernel in order to promote more efficient cache usage by fitting smaller chunks of data into cache, thereby improving spatial locality.

Figure 6.17 and Table 6.13 represents the speedup and performance obtained with matrix size 2048 with no optimization with block size 256.

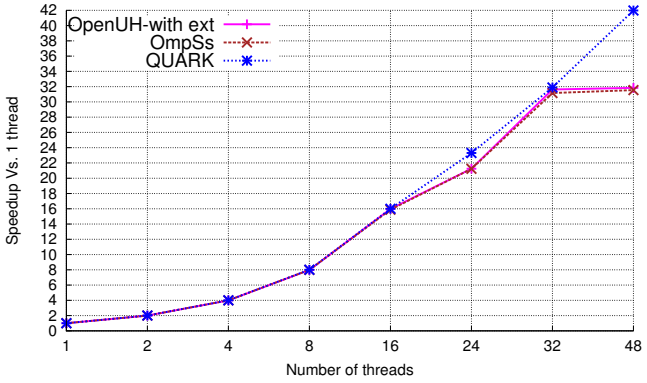


Figure 6.17: Speedup obtained for matrix size 2048 with block 256

Table 6.13: Performance obtained (in seconds) for matrix size 2048 with block 256

Threads	**OpenUH**	**OmpSs**	**Quark**
1	131.735	97.431	132.229
2	65.927	48.811	65.903
4	33.027	24.365	33.031
8	16.503	12.201	16.503
16	8.247	6.138	8.263
24	6.199	4.588	5.672
32	4.164	3.126	4.146
48	4.137	3.088	3.150

Figure 6.18 and Table 6.14 represent the speedup and performance obtained with matrix size 2048 with -O2 optimization with block size 256.

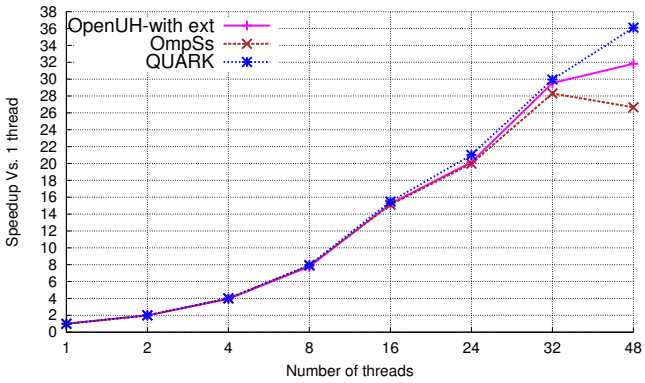


Figure 6.18: Speedup obtained for matrix size 2048 with block 256 with -O2 optimization

Table 6.14: Performance obtained (in seconds) for matrix size 2048 with block 256 with -O2 optimization

Threads	**OpenUH**	**OmpSs**	**Quark**
1	11.876	15.689	12.065
2	5.949	7.873	6.029
4	3.014	3.911	3.011
8	1.527	1.988	1.516
16	0.780	1.038	0.770
24	0.589	0.785	0.574
32	0.402	0.551	0.402
48	0.373	0.588	0.334

With the introduction of optimizations we observe that, unlike OmpSs which scales up to 32 threads, both QUARK and OpenUH with task extensions, continues to scale up to 48 threads. In Figure 6.18 it is clearly visible that due to overhead

generated in the scheduling and synchronization of the tasks, OmpSs fails to scale beyond 32 threads causing rapid performance degradation for 48 threads.

Figure 6.19 and Table 6.15 represents the speedup and performance obtained with matrix size 8192 with -O0 optimization with block size 1024.

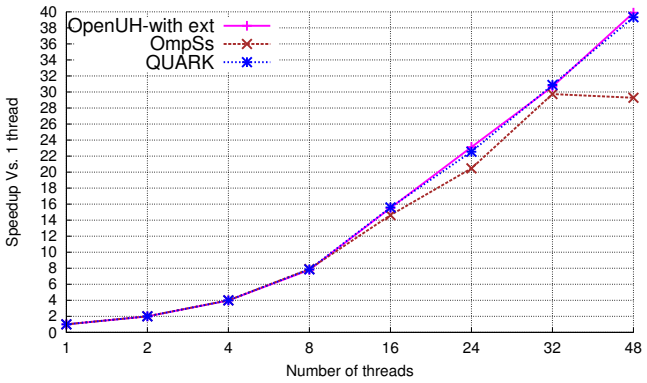


Figure 6.19: Speedup obtained for matrix size 8192 with block size 1024 with -O0 optimization

Table 6.15: Performance obtained (in seconds) for matrix size 8192 with block 1024 with -O0 optimization

Threads	**OpenUH**	**OmpSs**	**QUARK**
1	8597.01	6626.1	8621.62
2	4317.99	3303.75	4303.57
4	2158.44	1658.28	2158.59
8	1102.36	838.21	1096.36
16	554.07	452.86	552.92
24	371.92	323.64	382.51
32	280.34	222.83	279.28
48	215.52	226.32	219.21

We observe that for matrix size 8192, OpenUH and QUARK perform head to head

in terms of performance, scalability, and speedup. However, we do notice that OmpSs continues to scale poorly beyond 32 threads even though in terms of performance, it is comparable to both OpenUH and QUARK. OpenUH task extensions performs well by incurring the least amount of overhead and providing a performance benefit of close to 5% and 3% when compared OmpSs and QUARK respectively, for the above test case.

6.5 Summary

In this chapter we revisited the applications initially described in Chapter 4, and applied the proposed OpenUH task extensions to each of them. We obtained performance improvement for each of the applications, due to the elimination of the *taskwait* directive (acting as a global synchronization point). We observed an improvement in the overall scalability achieved for each of the applications. This is attributed to the

- Reduction in task synchronization and scheduling overheads when dealing with larger input data sizes.
- Improved load balancing among the threads owing to flexible scheduling of computations.
- Lesser overhead generated by the implementation at runtime (compared to the performance obtained from related dataflow model implementations - OmpSs and QUARK).

Chapter 7

Conclusion and Future Work

7.1 Conclusion

In this thesis we presented an approach that aimed at handling fine-grained inter-task synchronizations in a more dynamic and flexible manner. The proposed extensions implemented in the OpenUH OpenMP runtime library provide support for the specific ordering among asynchronous tasks. Thereby producing a point-to-point synchronization among the tasks needed to enable the efficient expression of algorithms that rely on patterns such as wavefront propagation and pipeline parallelism. Exhaustive experiments conducted on the LU Decomposition and Smith-Waterman algorithms, exhibit marked improvement in performance, when compared to the results obtained for the standard tasking versions (as per OpenMP 3.1 specification). This is attributed primarily to the reduction of synchronization overheads when dealing with larger input data sizes owing to the creation of a larger number of

tasks. It presented significant speedup and scalability for both the applications When compared to the performance obtained from the GNU, Intel, PGI, Oracle/Sun, and Mercurium compilers, as well as dataflow models (which demonstrate similar capabilities for addressing dependencies among tasks at runtime), QUARK and OmpSs. We utilized the matrix-matrix multiplication microbenchmark to perform overhead estimation analysis, comparing the overall performance of the OpenUH task extensions with QUARK and OmpSs runtime. We observed that the OpenUH task extensions incurred the least overhead and provided an average speedup of 1.5X, when tested with varying data sizes.

7.2 Future Work

As future work our focus lies in further improving the implementation in the OpenUH OpenMP runtime library by firstly, including 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 creating numerous tasks. Secondly, we would like to incorporate a sliding window approach, targeted for applications generating a number of tasks, wherein we populate the *tag table* with only a subset of active tasks at a time, in order to reduce the overhead of task scheduling and modulate the memory usage. Lastly we would like to include support, for handling data dependencies among tasks not sharing the same parent as well.

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 a dependency graph [40]. 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.

We also wish to extend our implementation of such data driven algorithms in the direction of a distributed memory environment. Our intention is to relax the synchronization among tasks by employing an efficient scheduling strategy obtained from analyzing the task dependency graph, before mapping them on to processors.

Bibliography

[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] OpenMP Application Program Interface, 3.1 edition, July 2011. http://www.openmp.org/mp-documents/OpenMP3.1.pdf.

[4] Programming models at BSC. http://pm.bsc.es/ompss.

[5] The OpenMP Fork-Join Model. `https://computing.llnl.gov/tutorials/totalview/part3.html`.

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

[7] 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.

[8] C. Augonnet, S. Thibault, R. Namyst, and P. Wacrenier. Starpu: A unified platform for task scheduling on heterogeneous multicore architectures. *Concurrency and Computation: Practice and Experience*, 23(2):187–198, 2011.

[9] E. Ayguadé, N. Copty, A. Duran, J. Hoeflinger, Y. Lin, F. Massaioli, E. Su, P. Unnikrishnan, and G. Zhang. A proposal for task parallelism in openmp. *A Practical Programming Model for the Multi-Core Era*, pages 1–12, 2008.

[10] 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.

[11] N. Azuelos, Y. Etsion, I. Keidar, A. Zaks, and E. Ayguadé. Introducing speculative optimizations in task dataflow with language extensions and runtime support. Accepted to the DFM 2012 Workshop in conjunction with PACT 2012, September 2012.

[12] R. Blumofe, C. Joerg, B. Kuszmaul, C. Leiserson, K. Randall, and Y. Zhou. *Cilk: An efficient multithreaded runtime system*, volume 30. ACM, 1995.

[13] V. Cavé, J. Zhao, J. Shirako, and V. Sarkar. Habanero-java: the new adventures of old x10. In *Proceedings of the 9th International Conference on Principles and Practice of Programming in Java*, pages 51–61. ACM, 2011.

[14] B. Chapman, D. Eachempati, and O. Hernandez. Experiences developing the openuh compiler and runtime infrastructure. Cetus Users and Compiler Infastructure Workshop in conjunction with PACT 2011, October 2011.

[15] P. Charles, C. Grothoff, V. Saraswat, C. Donawa, A. Kielstra, K. Ebcioglu, C. Von Praun, and V. Sarkar. X10: an object-oriented approach to non-uniform cluster computing. In *ACM SIGPLAN Notices*, volume 40, pages 519–538. ACM, 2005.

[16] C. Christofi, G. Michael, P. Trancoso, and P. Evripidou. Exploring hpc parallelism with data-driven multithreating. Accepted to the DFM 2012 Workshop in conjunction with PACT 2012, September 2012.

[17] F. Darema. The spmd model: Past, present and future. *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, 19:1–1, 2001.

[18] R. Diaconescu and H. Zima. An approach to data distributions in chapel. *International Journal of High Performance Computing Applications*, 21(3):313–335, 2007.

[19] A. Dios, R. Asenjo, A. Navarro, F. Corbera, and E. 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.

[20] J. Dongarra, M. Faverge, H. Ltaief, and P. Luszczek. Achieving numerical accuracy and high performance using recursive tile lu factorization. *University of Tennessee Computer Science Technical Report UT-CS-11-688 (also LAPACK Working Note 259), Submitted to Journal of Concurrency and Computation: Practice and Experience*, 2011.

[21] J. Dongarra, R. Graybill, W. Harrod, R. Lucas, E. Lusk, P. Luszczek, J. Mcmahon, A. Snavely, J. Vetter, K. Yelick, et al. Darpa's hpcs program: History, models, tools, languages. *Advances in Computers*, 72:1–100, 2008.

[22] 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.

[23] M. Frigo, C. Leiserson, and K. Randall. The implementation of the cilk-5 multithreaded language. *ACM Sigplan Notices*, 33(5):212–223, 1998.

[24] J. Gaudiot and L. Bic. *Advanced topics in data-flow computing*. Prentice Hall, 1991.

[25] P. Ghosh, Y. Yan, and B. Chapman. Support for dependency driven executions among openmp tasks. Accepted to the DFM 2012 Workshop in conjunction with PACT 2012, September 2012.

[26] Y. Guo, R. Barik, R. Raman, and V. Sarkar. Work-first and help-first scheduling policies for async-finish task parallelism. In *Parallel & Distributed Processing, 2009. IPDPS 2009. IEEE International Symposium on*, pages 1–12. IEEE, 2009.

[27] A. Hurson and K. Kavi. Dataflow computers: Their history and future. *Wiley Encyclopedia of Computer Science and Engineering*.

[28] J. LaGrone, A. Aribuki, C. Addison, and B. Chapman. A runtime implementation of openmp tasks. *OpenMP in the Petascale Era*, pages 165–178, 2011.

[29] Y. Liu, W. Huang, J. Johnson, and S. Vaidya. Gpu accelerated smith-waterman. *Computational Science–ICCS 2006*, pages 188–195, 2006.

[30] P. D. Michailidis and K. 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.

[31] W. Najjar, E. Lee, and G. Gao. Advances in the dataflow computational model. *Parallel Computing*, 25(13):1907–1929, 1999.

[32] B. Nichols, D. Buttlar, and J. Farrell. *Pthreads programming*. O\'Reilly, 1998.

[33] A. Robison, M. Voss, and A. Kukanov. Optimization via reflection on work stealing in tbb. In *Parallel and Distributed Processing, 2008. IPDPS 2008. IEEE International Symposium on*, pages 1–8. IEEE, 2008.

[34] J. Shirako, D. Peixotto, V. Sarkar, and W. Scherer. Phasers: a unified deadlock-free construct for collective and point-to-point synchronization. In *Proceedings of the 22nd annual international conference on Supercomputing*, pages 277–288. ACM, 2008.

[35] S. Tasırlar and V. Sarkar. Data-driven tasks and their implementation. In *ICPP11: Proceedings of the International Conference on Parallel Processing*, pages 652–661. IEEE, 2011.

[36] M. Tillenius and E. Larsson. An efficient task-based approach for solving the n-body problem on multicore architectures. *PARA 2010: State of the Art in Scientific and Parallel Computing*, pages 74:1–4, 2010.

[37] 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 *International Conference on Supercomputing: Proceedings of the 13th international conference on Supercomputing*, volume 20, pages 302–310. Citeseer, 1999.

[38] I. E. Venetis and G. 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.

[39] T. Weng and B. Chapman. Implementing openmp using dataflow execution model for data locality and efficient parallel execution. In *Parallel and Distributed Processing Symposium., Proceedings International, IPDPS 2002*, volume 2, pages 107–114. IEEE, 2002.

[40] T.-H. Weng and B. Chapman. Toward optimization of openmp codes for synchronization and data reuse. In *Journal of High Performance Computing and Networking (IJHPCN)*, volume 1, pages 43–54, 2004.

[41] Y. Yan, S. Chatterjee, D. A. Orozco, E. Garcia, Z. Budimlić, J. Shirako, R. S. Pavel, G. R. Gao, and V. 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.

[42] 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.

Appendix A

Runtime Implementation of OpenUH Task Extensions

This appendix describes the implementation of the OpenUH task dependency extensions in the OpenMP Runtime Library (RTL) in greater detail. We will refer to terminologies discussed in Table 5.1 introduced in chapter 5.

A.1 Addition of task *Tags* at task creation

In this section we describe the implementation of the Algorithm 1 introduced in chapter 5 in greater detail.

The *omp_task_create()* function in the OpenUH OpenMP RTL is responsible for

creating the tasks at runtime. Amongst the many arguments received, there contains an array which holds dependency information for all *tags* provided by the programmer. The dependency information comprises of a) the type of dependence encountered - *IN/OUT*. b) the task with which the *tag* is associated with, etc.

For every *tag* encountered amongst the arguments passed to this function, it initiates a search within the *tag table* (implemented as an unordered hash map) to ensure if that particular *tag* already exists. If the *tag* exists, it calls the *omp_append_dep* function which firstly, appends the *tag* to the *Dep_list* for that task. Secondly, it appends the dependency information (implemented as an additional node added to the linked list) to the end of the *Dependency_List* for that *tag*. Lastly, it updates the *Dep_Counter* of the associated task in accordance to the *IN/OUT* dependence encountered.

If the *tag* does not exist in the *tag_table*, it firstly locks the *tag_table*, creates a new entry for that *tag* in the table. It then calls the *omp_append_dep* function to append the dependency information as the first node to the *Dependency_List* and lastly unlocks the *tag_table*.

In the *omp_append_dep()* function, if the *tag* is identified to possess an *IN* dependence, and the corresponding *OUT counter* for that *tag* is greater than one, we atomically (with the use of compare and swap operations) increment the value of the *Dep_Counter*. This indicates that this particular task has a prior *OUT* dependence to honor. If the *tag* is identified to contain an *OUT* dependence instead, we conduct a backtrace on that *tag's Dependency_List* until we encounter an *OUT* dependence or the end of the list. In the event that the previous node itself is an *OUT* dependence,

we increment the *Dep_Counter* for that corresponding task by one, else we continue backtracking, counting the number of *IN* dependencies encountered in between, until an *OUT* dependence is encountered. We then increment the *Dep_Counter* of the task by the total number of *IN's* encountered. This implies that the task has several designated *IN* dependencies to satisfy, prior to its own execution.

We check the *Dep_Counter* of every task after all its dependencies have been added to the *tag table*. If the *Dep_Counter* is equal to zero, it implies that the task has no prior dependencies to satisfy and therefore it ready for execution. In such a case we immediately place that task in the task pool, where it can grabbed by an available resource and executed. In the event that the *Dep_Counter* is greater than zero, it suggests that the task is not ready for execution due to the existence of pending dependencies. We put the task on hold (in a WAIT state) until all its previous dependencies get resolved.

A.2 Deletion of task *Tags* at task exit

In this section we describe the implementation of the algorithm 2 introduced in chapter 5 in greater detail.

The *omp_task_exit()* function in the OpenUH OpenMP RTL is responsible for the deletion of tasks, after their execution at runtime. We examine the *Dep_List* of every task executing this function. The *Dep_List* contains information of all the associated dependencies for that task and by extension their respective *tags* which are retrievable from the *tag table*.

We begin by examining the first dependence (*IN/OUT*) listed in the *Dep_List*. We retrieve its succeeding dependence node (say T_{next}) on the *Dependency list* acquired from the *tag table*. The objective is to correctly update the *Dep_Counter* for that task and its subsequent dependent tasks, associated with T_{next}. If T_{next} has an *IN* dependence, we perform a forward trace on its corresponding *Dependency_List*, until we encounter an *OUT* dependence say T_{out}. We then decrement the *Dep_Counter* of the task associated with T_{out} and check to see if it is zero. If so, we place that task in the task pool, else we place the task on hold. If T_{next} has an *OUT* dependence, we initiate a forward tracking from that position in its corresponding *Dependency_List* until we encounter another *OUT* dependence. In the event that the next node itself is an *OUT* dependence, we decrement the *Dep_Counter* of the associated task by one and place it in the task pool if its *Dep_Counter* hits zero. If the next node has an *IN* dependence, we continue forward tracking. We decrement the respective *Dep_Counter*'s of all the tasks associated with *IN* dependencies encountered in between (and place them in the task pool if their *Dep_Counter*'s hit zero), until an *OUT* dependence is encountered. This ensures that we have removed the dependency constraint's the exiting task exercised, on subsequent tasks.

This process is followed for all the dependencies present in the exiting tasks's *Dep_List*. After making sure all the dependencies on subsequent tasks have been encountered for, we delete the task by deallocating the memory allotted for that task.

Appendix B

Explanation of Abbreviations

Table B.1: Abbreviations used in this document

Abbreviation	**Acronym**	**Meaning**
RTL	Runtime Library	A special program library used by a compiler to implement functions built into a programming language during the execution (runtime) of a program.
NUMA	Non Uniform Memory Access	Memory design used in multiprocessing, where the memory access time depends on the memory location relative to a processor.
BLAS	Basic Linear Algebra Subprograms	De facto programming interface standard for publishing software libraries to perform linear Algebra operations.
API	Application Programming Interface	Source code based specification used by software components to communicate to each other.
SPMD	single program, multiple data	A technique employed to achieve parallelism where tasks are split up and run simultaneously on multiple processors with different input in order to obtain results faster.

Table B.2: Abbreviations used in this document

Abbreviation	**Acronym**	**Meaning**
OpenMP	Open Multiprocessing	A programming model that supports multi-platform shared memory multiprocessing programming in languages C, C++, and Fortran
FLOPS	Floating Point Operations Per second	The number of FLOPS signify the performance of an application
QUARK	Queuing And Runtime for Kernels	a library that enables the dynamic execution of tasks with data dependencies in a multi-core, multi-socket, shared-memory environment.
CPU	Central Processing Unit	Portion of computer that performs the basic arithmetical, Logical, and input/output operations of the system
HPCS	High Productivity Computing Systems	An assessment on how we define and measure performance, programmability, portability, robustness and ultimately, productivity of applications in the HPC domain.
DARPA	Defense Advance Research Project Agency	A project focused on providing a new generation of economically viable high productivity computing systems for national security and for the industrial user community.