Selection of Parallel Runtime Systems for Tasking Models

Chun-Kun Wang
Department of Computer Science
Univ. of North Carolina at Chapel Hill
Chapel Hill, NC 27599, USA
Email: amos@cs.unc.edu

***Abstract*—The ubiquity of multi- and many-core processors means that many general purpose programmers are beginning to face the difficult task of using runtime systems designed for large-scale parallelism. Not only do they have to deal with finding and exploiting irregular parallelism through Tasking, but they have to deal with runtime systems that require an expert tuning of task granularity and scheduling for performance.**

This paper provides hands-on experiences to help programmers to select an appropriate tasking model and design programs. It investigates the scheduling strategies of three different runtime tasking models: Cilk, OpenMP and High Performance ParalleX (HPX-5). Six different simple benchmarks are used to expose how well each runtime performs when provided untuned implementations of irregular code fragments. The benchmarks, which have irregular and dynamic structures, provide information about the pros and cons of each system's runtime model, particularly the differences to the programmer between help-first and work-first scheduling.

***Index Terms*—Tasking Model; Parallel Programming; Cilk; OpenMP; HPX-5**

I. Introduction

The increasing complexity of multi- and many-core processors has made it difficult to achieve both high performance and reasonable programming productivity. Not only do they have to deal with finding and exploiting irregular parallelism through Tasking, but they have to deal with runtime systems that require an expert tuning of task granularity and scheduling for performance. This paper provides practical observations for programmers to determine an appropriate tasking model according to their application.

The runtime system schedules tasks efficiently to complete different sized units of work. Through runtime scheduling, tasking is able to balance workloads and keep all processors working. There are many parallel programming models that address some of the barriers to large-scale performance, like C++, Fortran, Chapel, MPI, OpenMP [1], and the HPX-5 library [3]. This paper investigates the scheduling strategies of three different runtime tasking models: Cilk, OpenMP, and High Performance ParalleX (HPX-5).

The OpenMP tasking model, first introduced in OpenMP 3.0, allows users to exploit the parallelism of irregular and dynamic program structures. Although OpenMP 4.0 supports expressing task dependences, this work focuses on OpenMP 3.0 since expressing task dependences requires more expertise. Cilk provides the user some keywords to implement task parallelism. (Here, Cilk refers to Cilk Plus, implemented by Intel Corp.) HPX-5, developed by CREST (Indiana University), is a state-of-the-art distributed programming model that is designed according to the ParalleX execution model [4].

A variety of well-known benchmarks can be run to compare parallel computing performance, including PARSEC, SPLASH-2, and NAS Parallel Benchmarks. In this paper, all systems are tested using benchmark sets designed for OpenMP. There are no recent well-known benchmark sets for HPX-5. Cilk and OpenMP have similar execution models and frameworks; thus, benchmark sets that can be used with the OpenMP framework can also be used with the Cilk framework. HPX-5 is tested using the same OpenMP benchmark sets. The HPX-5 runtime system is relatively similar to the benchmark designed for the OpenMP tasking model used in the Barcelona OpenMP Tasks Suit (BOTS) evaluations [5]. BOTS evaluation benchmarks, which are based on task parallelism, are used to test all the systems mentioned except for OmpSCR [6] and PARSEC.

This paper examines the difference between the design principles and throughput of Cilk, OpenMP, and HPX-5 runtime systems. Six representative benchmarks are applied, including *Fibonacci*, *Knight*, *Pi*, *Sort*, *N-Queens*, and *Unbalanced-Tree-Search*, in three distinct versions. These benchmarks have the coding patterns that are used by most general purpose programmers. This paper also discusses the pros and cons of each runtime system, particularly the differences to the programmer between help-first and work-first scheduling.

The remainder of this work is structured as follows. Other works relevant to ours are introduced in Section II. Section III presents the design principles of the Cilk, OpenMP, and HPX-5 runtime systems. Section IV introduces the benchmarks and shows the results. Finally, observations and conclusions are provided in Section V.

II. Related Work

Task-scheduling strategies fall into two broad groups: help-first schedulers and work-first schedulers. Cilk [2], provided by Intel, applies a work-first scheduler and uses a *work-stealing* technique. Work-first scheduling follows the serial execution path in choosing new tasks. Processors that are looking for tasks "steal" continuations for additional sequential tasks that haven't begun yet. OpenMP builds on this, attempting to add Intel's *work-queueing* model for dynamic task generation. The OpenMP tasking model, which was officially introduced to OpenMP language in Ayguade [8].

Several papers have studied how to schedule tasks. Korch et al. [9] discuss task-based algorithms and describe the implementation of different task pools for shared-memory multiprocessors. Duran [10] evaluates different scheduling strategies, including centralized breadth-first and fully-distributed depth-first *work-stealing* schedulers, and compares OpenMP and Cilk's results.

OpenMP and Cilk have been widely studied. In contrast, HPX-5 is a relatively modern parallel runtime system that has not yet been carefully investigated. Firoz et al. [11] compare three different single-source shortest-path algorithms running on two recent asynchronous many-task runtime systems: AM++ and HPX-5. In Zhang [12], the Fast Multipole Method [14] is implemented into four versions including Cilk, C++11, HPX-5, and OpenMP.

HPX-5 provides not only help-first schedulers but work-first schedulers. However, none of the studies above explore how HPX-5 performs compared to applications that combine runtime systems and tasking strategies, such as Cilk and OpenMP. Although some work has studied both tasking and runtime systems, these studies each targeted one specific program for testing; these types of studies cannot explain the comprehensive differences between runtime systems. The goal in this work is to investigate the pros and cons of different runtime systems and provide hands-on experiences for programmers.

III. Runtime System

A runtime system principally serves to implement the parallel execution model, and performance varies according to how different strategies are used in the design pattern. The OpenMP tasking model is presented in Section III-A. Section III-B explores the strategies of the Cilk runtime system. Section III-C reveals the design philosophy of HPX-5 and gives an example of how to make an HPX-5 program.

A. OpenMP Tasking Model

The OpenMP tasking model allows users to exploit parallelism in irregular and dynamic program structures, such as conditional loops, recursive algorithms, and producerconsumer patterns. The syntax of the OpenMP task construct includes clauses and select task regions. The supported data clauses, which control the data-sharing attributes of the variables, are *shared*, *private*, *firstprivate*, and *default*. Users need to select task regions and insert proper task constructs to enclose the chosen task regions. In other words, users are responsible for utilizing OpenMP's task model correctly, a problem that is well-discussed in Wang [15].

Besides data clauses, OpenMP provides *tied* and *untied* directives to illustrate a task's affinity for a particular thread. A *tied* task is one that is always executed by the same thread upon its resumption from the status of being suspended. Otherwise, it is an *untied* task. One key difference is that *untied* tasks may be interrupted at arbitrary points in execution while *tied* tasks suspend only at a specific point. OpenMP also provides the *taskwait* construct to synchronize the execution of tasks and to preserve dependent relationships among tasks.

B. Cilk Tasking Model

Cilk supports nested data and task parallelism; it provides three keywords (*cilk_spawn*, *cilk_for*, and *cilk_sync*) that developers can use to specify task parallelism. Unlike OpenMP, users are not required to specify the data-sharing attributes of the variables; users are, however, responsible for specifying *cilk_spawn* to a function call, which is allowed to execute in parallel. *cilk_sync* requires that all spawned calls in a function finish before execution continues. But whether or not a function called with *cilk_spawn* runs asynchronously is determined by the Cilk runtime system.

Cilk runtime scheduler follows a work-first policy, which allows tasks to be stolen by other processors. When a thread has no tasks in its work queue, it can steal a task from the local work queue (last-in, first-out) or from another thread pool (first-in, first-out).

C. HPX-5 Tasking Model

The architecture of the HPX-5 runtime system includes several components: localities, global memory, lightweight threads and actions, LCOs (Lighted Control Objects), and parcels. Actions are accessed and executed by HPX-5 lightweight threads. The lightweight threads are usually destined to execute a process that changes a local address to a specific global address. That allows HPX5 runtime to send an action to the physical location in which data is located. Otherwise, synchronization and communication between actions are carried out by LCOs. The design of LCOs permits actions to proceed with their execution as far as possible—in other words, to be *event-driven*. Thread execution is not required to wait for particular blocking operations, like busy-waits and polling, to finish.

An HPX-5 thread manager compiles several scheduling policies, including work-first, help-first, and hybrid policies, which provide users the freedom to configure the scheduling of the HPX-5 runtime system. The thread manager implements a work queue-based execution strategy with work stealing and the default scheduling policy is a hybrid policy. Whether newly generated parcels will be executed using work-first or help-first is controlled by the *–hpx-sched-wfthreshold* parameter. When the number of tasks is under the threshold, the scheduling policy is help-first; when it exceeds the threshold, it is work-first.

HPX-5 also supports both random and hierarchical policies. During work stealing, a random policy means that a worker will select a random victim from the rest of the workers in its locality from which to steal tasks. The hierarchical policy, in contrast, dictates that the worker first attempt to steal one parcel from another worker within the local NUMA node, then

TABLE I
CONFIGURATION OF THE EXPERIMENTAL ENVIRONMENT

	Item	Value
Hardware	CPU	Intel(R) Xeon(R) CPU E5-2660 v3 @ 2.60GHz (20 cores in total)
	Cache	L1: 32 KB, L2: 256 KB, L3: 25 MB
	Bandwidth	68GB/s
	Memory	DDR4 64GB with 4 memory channels
Software	OS	CentOS release 6.6 (kernel: 2.6.32)
	Compiler	icc 14.0.2 with ”-O3” option

attempt to steal a much larger number of parcels from a worker in a different NUMA node.

IV. Evaluation

In this section, the differences between the OpenMP, Cilk, and HPX-5 runtime systems are explored. Section IV-A describes applications that are used to test the scalability of runtime systems. Section IV-B shows the experiment setup, including hardware and software specifications. Finally, the results are shown in Section IV-C.

A. Benchmarks

In order to evaluate runtime systems' tasking models, some untuned benchmarks have been selected, the parallelisms of which are frequently adopted by most general-purpose programs. Many were chosen from BOTS [5] and one from OmpSCR [6]. Based on the OpenMP versions, these benchmarks were designed to be used with the parallel frameworks of both Cilk and HPX-5. So that all applications are compared based on the same foundation, all benchmarks have the same parallel task regions and the same points for synchronization. All benchmarks are identical from the point of view of algorithms and parallelism. For all OpenMP versions, the default task directive that uses *tied* tasks is adopted.

B. Experimental Setup

This paper evaluates the runtime systems of OpenMP, Cilk, and HPX-5 by running the experiments on an Intel Haswell machine. Table I lists the configuration of the experimental environment. The OpenMP benchmarks were compiled using Intel icc 14.0.2; the Cilk benchmarks were compiled using Intel Cilk++ 1.0 (based on icc 14.0.2), for comparison with the OpenMP implementation; and the HPX-5 benchmarks adopted HPX-5 v2.2.0, compiled with icc 14.0.2. The ”O3” option is always used. Unless otherwise mentioned, reported results represent the average of 10 trials.

C. Results

Figure 1 shows the performance of the three runtime systems on the six untuned benchmarks. The speed-up on the *Fibonacci*, *Knight*, and *Pi* benchmarks indicates that the parallel version runs slower than sequential versions. None of the runtime systems is capable of obtaining a reasonable speed-up. *N-Queens*, *Sort*, and *UTS* demonstrate more information about runtime systems. The *N-Queens* benchmark shows an 11X

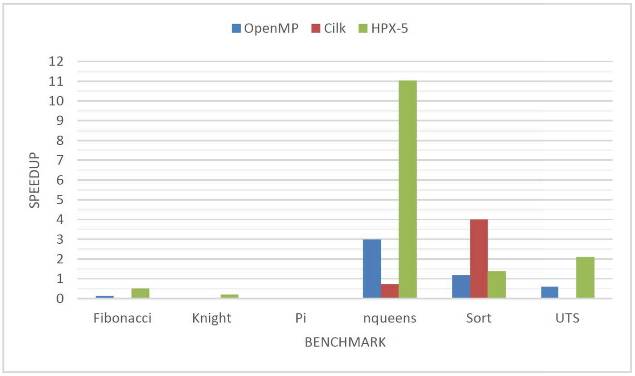

Fig. 1. Performance comparisons on the six benchmarks of OpenMP, Cilk, and HPX-5 runtime systems, as run on the system described in Table I. Each benchmark has its own sequential version as the baseline for speed-up. *Fibonacci* computes to the 50th Fibonacci number; *Knight* has a board size of 7 by 7; *Pi* calculates π to the accuracy of 10^4; *N-Queens* finds all solutions on a 16x16 chessboard; *Sort* sorts a random list of size 10^9; and *UTS* explores 1.6 billion nodes in an unbalanced tree.

speed-up using HPX-5, whereas OpenMP has approximately a 3X speed-up. On the other hand, in the *Sort* benchmark, Cilk achieves a 4X speed-up—more advantageous than either OpenMP or HPX-5. Some benchmarks have no bar because either a) the runtime system cannot handle the problem size (Cilk is not available to run *UTS* when the tree size is large), or b) the performance is so low (slowdown) that no bar is visible. The following provides insight into each benchmark individually.

1) Fibonacci: The *Fibonacci* benchmark uses a recursive parallelism on a simple test case of a deep tree. As shown in Figure 2, none of the runtime systems do well under the naive circumstance, regardless of their scheduling policies and system design; this is because the benchmark is composed of very fine-grained tasks with very little real work done. When the Fibonacci number is 50, the difference in average time among the three runtime systems ranges from 8% to 20%. The OpenMP version runs faster than others.

Figure 2 also reveals the overhead of all runtime systems. When input is below 35, all runtime systems run slower than they should in theory. Surprisingly, the OpenMP version runs faster with an input of 20 than it does with an input of 15. Overall, the overhead of HPX-5 is less than that of OpenMP. In a simple program like *Fibonacci*, the overhead is still noticeable.

2) Pi and Knight: The *Knight* and *Pi* benchmarks have similar program structures and outcomes, and will therefore be discussed together. Figure 3 presents how the different runtime systems perform in executing the *Pi* program. Since the *Pi* program calls itself recursively to the desired depth, Cilk and OpenMP finish all work with the same levels of accuracy, although Cilks work-first policy has an advantage given *Pi*'s parallelism. Initially, Cilk finishes the work in the smallest execution time; after the accuracy point of 10^4, however, Cilk is not available. Like Cilk, OpenMP does not

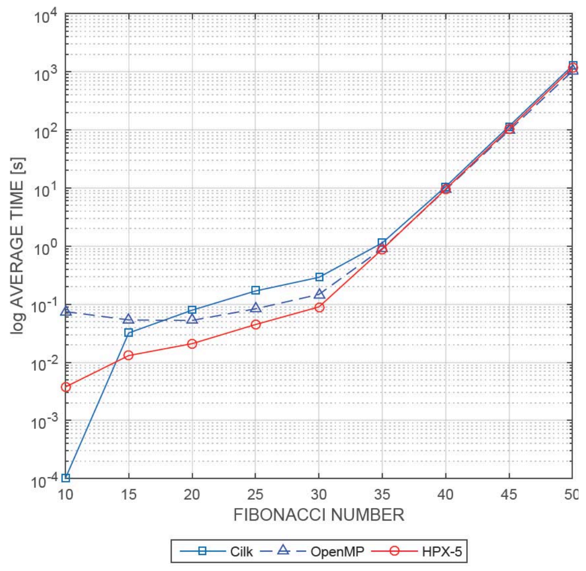

Fig. 2. The average execution time of the *nth* Fibonacci number.

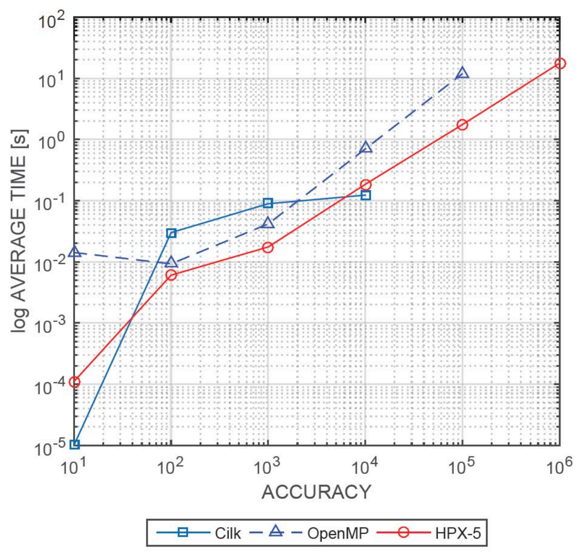

Fig. 3. The average execution time of various runtime systems for different desired precisions on the Pi program.

allow the program to generate tasks recursively with unlimited depth. Infinitely recursive function calls not only blow up the stack size, but also break the balance between reasonable depth and memory usage. The HPX-5 runtime system virtualizes all memory access, so it is limited by the program stack size. It turns out that HPX-5 creates spawns as deep as possible until it runs out of memory and gets killed by the operating system.

3) N-Queens: Figure 4 shows the runtime systems' average speed-ups for *N-Queens* on a 16 by 16 chessboard. Both Cilk and OpenMP versions have poor speedup; HPX-5 achieves an 11X speed-up. The program architecture of *N-Queens* represents an individual node as one task or spawn that it is able to

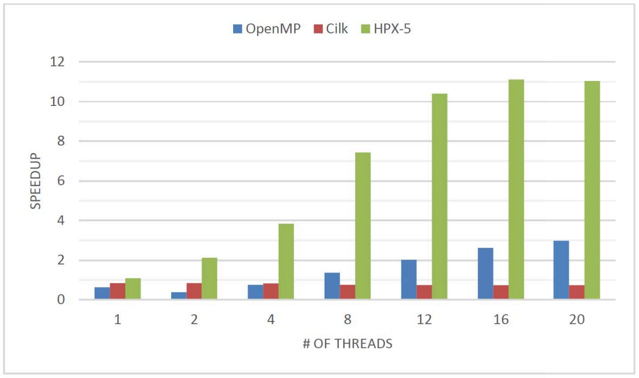

Fig. 4. The speed-up on *N-Queens* using the different task implementations of OpenMP, Cilk, and HPX-5. We use a chessboard of size 16 by 16 in the experiment. The performance of the sequential version is the comparison baseline.

execute in parallel, and tree depth is determined by whether *N-Queens* can position queens. *N-Queens* generates a flat, shallow tree architecture that does not benefit Cilk's work-first policy. It turns out that the victim node (the shallowest node) provides an insufficient workload; Cilk therefore displays no speed-up. According to Duran [10], a work-first policy with tied tasks severely degrades performance; this performance decay can be mitigated by using untied tasks.

The help-first policy applied by both OpenMP and HPX-5 is good for traversing a broad, shallow tree structure, and this accounts for their performance increases. The difference in performance improvements between HPX-5 and OpenMP comes from the different implementations of these two runtime systems. OpenMP adopts heavyweight POSIX threading that causes much overhead in generating new tasks. By contrast, the lightweight threading applied by HPX-5 significantly reduces the overhead of generating tasks. For these reasons, the HPX-5 runtime system on the *N-Queens* benchmark shows a progressive speed-up when the number of threads increases.

4) Sort: The *Sort* benchmark performance comparison results are shown in Figure 5. The baseline for comparison was the performance of the sequential version. In the experiment, two versions of HPX-5 are implemented: one is *HPX-5*, which is the basic HPX-5 application, and the other is *HPX-5 Cut*, which applies a cut-off strategy that reduces degradation in the obtained performance.

When the sorted number goes above 10^5, *HPX-5* starts to slow down, but the speed-up of *HPX-5 Cut* continues to grow, because *HPX-5 Cut* avoids the overhead of generating overwhelming spawns. Even though HPX-5 applies lightweight threading, the overhead of creating spawn must still be an issue taken into consideration. OpenMP has no significant speed-up; no matter the size of the input number. Compared to *OpenMP*, *HPX-5*, which does not control the depth of generating spawn, achieves 2.5X speed-up when the size is 10^5. Cilk's performance seems relatively sound and sustainable, since the way Cilk generates spawn does not require support from the operating system.

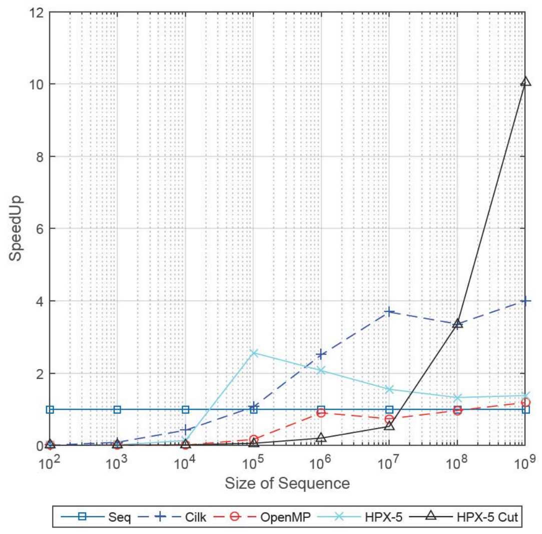

Fig. 5. The average execution time of *Sort* of various runtime systems for different sizes of sequences. The performance of the sequential version is the comparison baseline. *HPX-5 Cut* stands for the HPX-5 version with a cut-off mechanism.

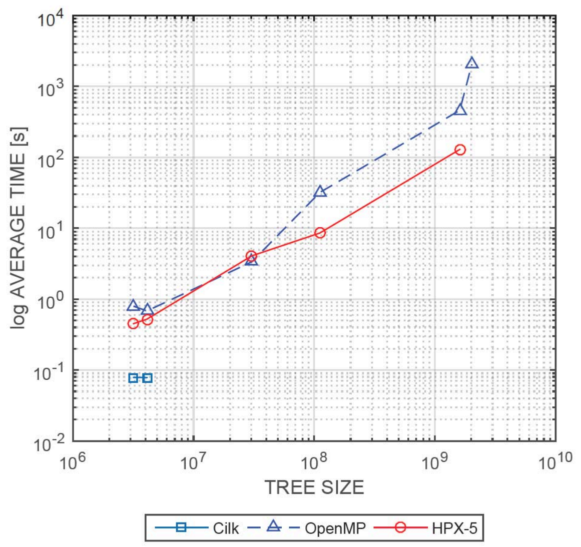

Fig. 6. The average execution time of Unbalanced-Tree-Search for various runtime systems on different tree sizes.

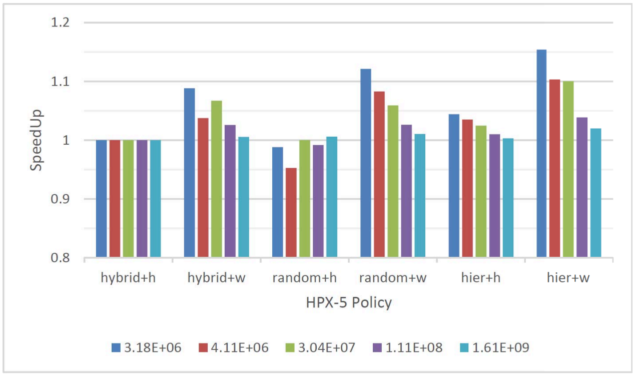

Fig. 7. Speed-up comparison of the six permutations of scheduling policies for HPX-5 on the Unbalanced-Tree-Search benchmark. *hybrid+h* means hybrid policy with help-first, and *hybrid+w* represents hybrid policy with work-first. The rest are *random* policy set and *hier* (hierarchical) policy set. As a comparison base, the default scheduling strategy, which is *hybrid+h*, is used.

Sort does not require sufficient parallelism to be a good test. Besides the recursive function, the only bottleneck in *Sort* is the work of partition. According to Brent's theorem, the speedup of parallel computing is bound by step-complexity. *Sort*'s performance is limited by the sequential partition.

5) UTS: UTS generates an unbalanced tree with enormous variance in subtree sizes. *UTS* challenges the runtime systems' load balancing, since the distribution of subtree sizes follows a power law. The purpose of the work-stealing policy is to balance workload between processes. Load-balancing operations, however, incur overhead costs, leading to poor performance. Hence, it is difficult to tune *UTS*' performance for improvement.

Figure 6 shows the performance of the three runtime systems. Regarding the missing data points for Cilk, the curve of the Cilk version is unavailable because of Cilk's limited spawn depth. The Intel worker's *deque* is hard coded to 1024 entries. Given a small input size, the Cilk version runs fastest among the tested runtime systems. When input size reaches 100 million nodes, the HPX-5 version requires less time to finish jobs than OpenMP does, because HPX-5 provides an Active Global Address Space, which benefits work stealing and memory moving. The HPX-5 version fails at an input size of 2 billion nodes, because it runs out of memory and is killed by the operating system. This occurs because HPX-5 needs more memory space to sustain its Active Global Address Space, which produces a trade-off between memory usage and need. Although the OpenMP version is able to survive when the total number of tree nodes is 2 billion, its performance suffers because of the overhead produced by context-switching and cache miss, demanding more execution time.

Even though *UTS* has an unbalanced program structure, it can still benefit from the scheduling strategies of the runtime system. HPX-5 provides hybrid, random, and hierarchical (denoted as *hier* in Figure 7) policies. The default scheduling strategy, *hybrid+h* is used as a basis for comparison and the results show other scheduling strategies (except *random+h*) have better performance.

V. Discussion and Conclusion

This section offers further insights into the experimental results given in the previous section. The experimental environment has been built on the hardware architecture of a single node. The HPX-5 runtime system does not have to distribute a workload across all nodes, so it avoids the overhead of communication between nodes. All runtime systems run their programs in shared memory architecture.

Fine-grained parallelism, like that required by *Fibonacci*, *Knight*, and *Pi*, limits the ability of runtime systems' acceleration; so, their program structures should be modified to eliminate this parallelism for further performance improvements. Fine-grained parallelism also tends to cause the Cilk runtime system to abort its application because of its limited depth of tasks. Cilks work-first policy is supposed to have an advantage in executing programs with a narrow, deep tree structure; but, its unexpected overhead from spawn generation reduces performance gains. Indeed, Cilk performs much slower than its sequential version on this benchmark.

The overall speed-up of OpenMP and HPX-5 on all benchmarks is compared in Figure 1. HPX-5 runtime systems generally perform better, but OpenMP and HPX-5 show similar speed-up trends because they use help-first scheduling policies. HPX-5, however, displays higher speed-ups than OpenMP because it has lower overhead in task creation and in task stealing. Although shared memory architecture provides the same data scope to multi-threading, OpenMP threading still must copy task-related data from the task pool to thread local storage, allowing it to generate or steal tasks. HPX-5 leaves task-related data in the memory space managed by the runtime system. Its threading can recreate or steal a task by accessing the memory space where the task-related data is, without the overhead of moving data. In a nutshell, HPX-5 sends work to data, not vice versa. If a sufficient amount of memory needs to be copied for generating or stealing tasks, the cost of moving the memory weakens the performance of OpenMP. Because HPX-5 has less overhead when creating and stealing tasks, it does make high demands on memory use. *UTS* reveals the disadvantage of HPX-5: it demands much more memory from the system than the others do. *UTS*'s large tree sizes also weaken the Cilk runtime system severely. Besides the memory demand, OpenMP and HPX-5 can scale applications in a large-scale computing environment.

Cilk differs most significantly from the other systems because it uses a work-first scheduling policy rather than a help-first scheduling policy. Tasking with a work-first policy keeps data in contiguous memory and achieves better data locality. Cilk has relatively better speed-up in *Sort*, and also generates spawn without the demand of numerous operating system resources. Work-first scheduling policies become clumsy if the task held by one thread cannot bring a sufficient workload. Cilk's limitation of spawn depth also keeps it from being scalable to large-scale-level performance.

A cut-off mechanism is one practical solution that reduces performance degradation. When applying a cut-off mechanism in the HPX-5 benchmark of *Sort*, its performance dominated all other versions of the application. Whether the size of task granularity is fine-grained depends on plenty of factors associated with hardware and software architecture, like cache size, bandwidth, disk layout, programming language, overhead of function calls, etc. A certain threshold cannot be broadly applied because it varies with different problem sizes and hardware platforms.

Help-first policies work better than work-first policies if the pattern of program structure is, as in the *N-Queens* benchmark, a broad, shallow tree. Work-first policies are best suited to work on programs structured like narrow, deep trees, such as the *Fibonacci*, *Knight*, and *Pi* benchmarks. The executions of *Knight* and *Pi* show no difference between parallel programs and sequential programs; the speed-ups on the *Fibonacci*, *Knight*, and *Pi* benchmarks unexpectedly show no performance gains from Cilk's work-first scheduling policy. This is because their executions suffer from overwhelming overhead created by spawn generation.

Acknowledgment

The authors would like to thank Dr. Jan F. Prins and Dr. Allan Porterfield for their useful comments.

References

[1] Dagum, Leonardo, and Rameshm Enon. *OpenMP: an industry standard API for shared-memory programming.* Computational Science & Engineering, IEEE 5.1 (1998): 46-55.

[2] Blumofe, Robert D., et al. *Cilk: An efficient multithreaded runtime system.* Journal of parallel and distributed computing 37.1 (1996): 55-69.

[3] Tabbal, Alexandre, et al. *Preliminary design examination of the ParalleX system from a software and hardware perspective.* ACM SIGMETRICS Performance Evaluation Review 38.4 (2011): 81-87.

[4] Gao, Guang R., et al. *Parallex: A study of a new parallel computation model.* Parallel and Distributed Processing Symposium, 2007. IPDPS 2007. IEEE International. IEEE, 2007.

[5] Duran Gonzlez, Alejandro, et al. *Barcelona openmp tasks suite: A set of benchmarks targeting the exploitation of task parallelism in openmp.* 38th International Conference on Parallel Processing. 2009.

[6] Rodriguez, Claudia, and Francisco de Sande. *The OpenMP source code repository.* Parallel, Distributed and Network-Based Processing, 2005. PDP 2005. 13th Euromicro Conference on. IEEE, 2005.

[7] Frigo, Matteo, Charles E. Leiserson, and Keith H. Randall. *The implementation of the Cilk-5 multithreaded language.* ACM Sigplan Notices. Vol. 33. No. 5. ACM, 1998.

[8] Ayguad, Eduard, et al. *A proposal for task parallelism in OpenMP.* A Practical Programming Model for the Multi-Core Era. Springer Berlin Heidelberg, 2007. 1-12.

[9] Korch, Matthias, and Thomas Rauber. *A comparison of task pools for dynamic load balancing of irregular algorithms.* Concurrency and Computation: Practice and Experience 16.1 (2004): 1-47.

[10] Duran, Alejandro, Julita Corbaln, and Eduard Ayguad. *Evaluation of OpenMP task scheduling strategies.* OpenMP in a new era of parallelism. Springer Berlin Heidelberg, 2008. 100-110.

[11] Jesun Sahariar Firoz, et al. *Comparison Of Single Source Shortest Path Algorithms On Two Recent Asynchronous Many-task Runtime Systems.* In 21st IEEE International Conference on Parallel and Distributed Systems (ICPADS 2015), IEEE, December 2015.

[12] Zhang, Bo. *Asynchronous Task Scheduling of the Fast Multipole Method using various Runtime Systems.* Data-Flow Execution Models for Extreme Scale Computing (DFM), 2014 Fourth Workshop on. IEEE, 2014.

[13] Thomas Sterling, Daniel Kogler, et al. *SLOWER: A performance model for Exascale computing.* Supercomputing Frontiers and Innovations, 1:4257, September 2014.

[14] Carrier, J., Leslie Greengard, and Vladimir Rokhlin. *A fast adaptive multipole algorithm for particle simulations.* SIAM journal on scientific and statistical computing 9.4 (1988): 669-686.

[15] Wang, Chun-Kun, and Chen, Peng-Sheng. *Automatic scoping of task clauses for the OpenMP tasking model.* The Journal of Supercomputing 71.3 (2015): 808-823.

[16] Prins, Jan, et al. *UPC implementation of an unbalanced tree search benchmark.* Univ. North Carolina at Chapel Hill, Tech. Rep. TR03-034 (2003).