Comparison of Threading Programming Models

Solmaz Salehian, Jiawen Liu and Yonghong Yan
Department of Computer Science and Engineering, Oakland University, Rochester, MI USA
{ssalehian,jliu,yan}@oakland.edu

***Abstract*—In this paper, we provide comparison of language features and runtime systems of commonly used threading parallel programming models for high performance computing, including OpenMP, Intel Cilk Plus, Intel TBB, OpenACC, Nvidia CUDA, OpenCL, C++11 and PThreads. We then report our performance comparison of OpenMP, Cilk Plus and C++11 for data and task parallelism on CPU using benchmarks. The results show that the performance varies with respect to factors such as runtime scheduling strategies, overhead of enabling parallelism and synchronization, load balancing and uniformity of task workload among threads in applications. Our study summarizes and categorizes the latest development of threading programming APIs for supporting existing and emerging computer architectures, and provides tables that compare all features of different APIs. It could be used as a guide for users to choose the APIs for their applications according to their features, interface and performance reported.**

***Keywords*-threading; parallel programming; data parallelism; task parallelism; memory abstraction; synchronization; mutual exclusion**

I. Introduction

The High Performance Computing (HPC) community has developed a rich variety of parallel programming models to facilitate the expression of the required levels of concurrency to exploit hardware capabilities. Programming APIs for node-level parallelism, such as OpenMP, Cilk Plus, C++11, POSIX threads (PThreads), Intel Threading Building Blocks (TBB), OpenCL, Microsoft Parallel Patterns Library (PPL), to name a few, each has its unique set of capabilities and advantages. They also share certain functionalities realized in different interfaces, e.g., most of them support both data parallelism and task parallelism patterns for CPU. They are all evolving to become more complex and comprehensive to support new computer architectures and emerging applications. It becomes harder for users to choose from those APIs for their applications with regards to the features and interfaces of these models.

The same parallelism pattern could be realized using different interfaces and implemented using different runtime systems. The runtime systems that support those features vary in terms of scheduling algorithms and implementation strategies, which may cause dramatically different performance for the same applications created using different programming APIs. Thus, performance-wise selection of the right API requires efforts for studying and benchmarking for users' application.

In this paper, we provide an extensive comparison of language features and runtime systems of commonly used threading parallel programming models for HPC, including OpenMP, Intel Cilk Plus, Intel TBB, OpenACC, Nvidia CUDA, OpenCL, C++11 and PThreads. We then report our performance comparisons of OpenMP, Cilkplus and C++11 for data and task parallelism on CPU, showing the impacts of runtime systems on the performance. The paper makes the following contributions: 1) a list of features for threading programming APIs to support existing and emerging computer architectures; 2) comparison of threading models in terms of feature support and runtime scheduling strategies; 3) performance comparisons of OpenMP, Cilk Plus and C++11 for data and task parallelism on CPU using benchmark kernels and Rodinia [7].

The rest of the paper is organized as follows. In section II, a list of API features of parallel APIs are summarized. In section III, comparisons of interfaces and runtime systems are presented. Section IV presents performance comparisons. Section V provides related work study and Section VI contains our conclusion.

II. Features for Threading Programming APIs

The evolvement of programming models has been mostly driven by advances of computer system architectures and new application requirements. Computing nodes of existing and emerging computer systems might be comprised of many identical computing cores in multiple coherency domains, or they may be heterogeneous, and contain specialized cores that perform a restricted set of operations with high efficiency. Deeper memory hierarchies and more challenging NUMA effects for performance optimization have been seen in the emerging computer systems. Further, explicit data movement is necessary for nodes that present distinct memory address spaces to different computing elements, as demonstrated in today's accelerator architectures.

To facilitate programming a diversified range of computer systems, an ideal API must be expressive for the required levels of concurrency and many other unique features of hardware, while permitting an efficient implementation by system software. In this section, we categorized different features of threading model APIs. The detailed comparison of threading programming APIs based on these categories are discussed in next section.

Parallelism: A programming model provides API for specifying different kinds of parallelism that either map to parallel architectures or facilitate expression of parallel algorithms. We consider four commonly used parallelism patterns in HPC: 1) data parallelism, which maps well to manycore accelerator and vector architecture depending on the granularity of data parallel unit; 2) asynchronous task parallelism, which can be used to effectively expresses certain parallel algorithms, e.g., irregular and recursive parallelism; 3) data/event-driven

computation, which captures computations characterized as data flow; and 4) offloading parallelism between host and device, which is used for accelerator-based systems.

Abstraction of memory hierarchy and programming for data locality: Portable optimization of parallel applications on shared memory NUMA machines has been known to be challenging. Recent architectures that exhibit deeper memory hierarchies and possible distinct memory/address spaces make portable memory optimization even harder. A programming model helps in this aspect by providing: 1) API abstraction of memory hierarchy and core architecture, e.g., an explicit notion of NUMA memory regions or high bandwidth memory; 2) language construct to support binding of computation and data to influence runtime execution under the principle of locality; 3) means to specify explicit data mapping and movement for sharing data between different memory and address spaces; and 4) interfaces for specifying memory consistency model.

Synchronizations: A programming model often provides constructs for supporting coordination between parallel work units. Commonly used constructs include barrier, reduction and join operations for synchronizing a group of threads or tasks, point-to-point signal/wait operations to create pipeline or workflow executions of parallel tasks, and phase-based synchronization for streaming computations.

Mutual exclusion: Interfaces such as locks are still widely used for protecting data access. A model provides language constructs for creating exclusive data access mechanism needed for parallel programming, and may also define appropriate semantics for mutual exclusion to reduce the opportunities of introducing deadlocks.

Error handling, tools support, and language binding: Error handling provides support for dealing with faults from user programs or the system to improve system and application resilience. Support for tools, e.g., performance profiling and debugging tools, is essential to improve the productivity of parallel application development and performance tuning. For HPC, C, C++ and Fortran are still the dominant base languages. While functional languages can provide a cleaner abstraction for concurrency, it is not easy to rewrite all legacy code and library to a new base language. Ideally, a model would support at least these three languages.

III. Comparison of Language Features and Runtime Systems

In this section, we report our comparison on language features and interfaces, as well as runtime scheduling systems. The list of commonly used threading programming models for comparison includes OpenMP, Intel Cilk Plus, Intel TBB, OpenACC, Nvidia CUDA, OpenCL, C++11 and PThreads. PThreads and C++11 are baseline APIs that provide core functionalities to enable other high-level language features. CUDA (only for NVIDIA GPU) and OpenCL are considered as low-level programming interfaces for recent manycore and accelerator architectures that can be used as user-level programming interfaces or intermediate-level interfaces for the compiler-transformation targets from high-level interfaces. OpenACC provides high-level interfaces for offloading parallelism for manycore accelerators. Intel TBB and Cilk Plus are task based parallel programming models used on multi-core and shared memory systems. OpenMP is a more comprehensive standard that supports a wide variety of features we listed.

A. Language Features and Interfaces

The full comparisons of language features and interfaces are summarized in Table I, II and III. For parallelism support listed in Table I, asynchronous tasking or threading can be viewed as the foundational parallel mechanism that is supported by all the models. Overall, OpenMP provides the most comprehensive set of features to support all the four parallelism patterns. For accelerators (NVIDIA GPUs and Intel Xeon Phis), both OpenACC and OpenMP provide high-level offloading constructs and implementation though OpenACC supports mainly offloading. Only OpenMP and Cilk Plus provide constructs for vectorization support (OpenMP's simd directives and Cilk Plus' array notations and elemental functions). For data/event driven parallelism, C++'s std::future, OpenMP's depend clause, and OpenACC's wait are all for user to specify asynchronous task dependency to achieve such kind of parallelism. Other approaches, including CUDA's stream, OpenCL pipe, and TBB's pipeline, provide pipelining mechanisms for asynchronous executions with dependencies between CPU tasks.

For supporting abstraction of memory systems and data locality programming, the comparison is listed in Table II. Only OpenMP provides constructs for programmers to specify memory hierarchy (as places) and the binding of computation with data (proc_bind clause). Programming models that support manycore architectures provide interfaces for organizing a large number threads (x1000) into a two-level thread hierarchy, e.g., OpenMP's teams of threads, OpenACC's gang/worker/vector clause, CUDA's blocks/threads and OpenCL's work groups. Models that support offloading computation provide constructs to specify explicit data movement between discrete memory spaces. Models that do not support other compute devices do not require them. It is also important to note, though not listed in the table, that C++ thread memory model includes interfaces for a rich memory consistency model and guarantees sequential consistency for programs without data races [6], that are not available in most others, except the OpenMP's flush directive.

For supporting the three synchronization operations, i.e. barrier, reduction and join operations, whose comparison is summarized in Table II, OpenMP supports all the operations. Cilk Plus and TBB provide join and reducer. Note that since Cilk Plus and Intel TBB emphasize tasks rather than threads, the concept of a thread barrier makes little sense in their model, so its omission is not a problem.

The comparisons of the rest of the features are summarized in Table III. Locks and mutexes are still the most widely used mechanisms for providing mutual exclusion. OpenMP supports locks which are used with the aim of protecting shared variables and C++11, PThread and TBB provide mutex which is similar to locks.

Most of the models have C and C++ bindings, but only OpenMP and OpenACC have Fortran bindings. Most models

TABLE I: Comparison of Parallelism

	Parallelism			
	Data parallelism	Async task parallelism	Data/event-driven	Offloading
Cilk Plus	cilk_for, array operations, elemental functions	cilk_spawn/cilk_sync	x	host only
CUDA	<<<--->>>	async kernel launching and memcpy	stream	device only
C++11	x	std::thread, std::async/future	std::future	host only
OpenACC	kernel/parallel	async/wait	wait	device only (acc)
OpenCL	kernel	clEnqueueTask()	pipe, general DAG	host and device
OpenMP	parallel for, simd, distribute	task/taskwait	depend (in/out/inout)	host and device (target)
PThread	x	pthread_create/join	x	host only
TBB	parallel_for/while/do, etc	task::spawn/wait	pipeline, parallel_pipeline, general DAG (flow::graph)	host only

TABLE II: Comparison of Abstractions of Memory Hierarchy and Synchronizations

	Abstraction of memory hierarchy and programming for data locality			Synchronization		
	Abstraction of memory hierarchy	Data/computation binding	Explicit data map/movement	Barrier	Reduction	Join
Cilk Plus	x	x	N/A(host only)	implicit for cilk_for only	reducers	cilk_sync
CUDA	blocks/threads, shared memory	x	cudaMemcpy function	_synchthreads	x	x
C++11	x (but memory consistency)	x	N/A(host only)	x	x	std::join, std::future
OpenACC	cache, gang/worker/vector	x	data copy/copy in/copyout	x	reduction	wait
OpenCL	work group/item	x	buffer Write function	work_group barrier	work_group reduction	x
OpenMP	OMP_PLACES, teams and distribute	proc_bind clause	map(to/from /tofrom/alloc)	barrier, implicit for parallel/for	work_group reduction	taskwait
PThread	x	x	N/A(host only)	pthread_barrier	x	pthread_join
TBB	x	affinity_partitioner	N/A(host only)	N/A(tasking)	parallel_reduce	wait

do not provide dedicated mechanisms for error handling and many leverage C++ exceptions for that purpose. As an exception, OpenMP has its cancel construct for this purpose, which supports an error model. For tools support, Cilk Plus, CUDA, and OpenMP are three implementations that provide a dedicated tool interface or software. Many of the host-only models can use standard system profiling tools such as Linux perf.

B. Runtime Systems

The fork-join execution model and workstealing of dynamic tasks are the two main scheduling mechanisms used in the threading programming systems. The complexity of runtime system varies depending on the features to support for a programming model. Using OpenMP as example, which provides more features than any other model, it fundamentally employs the fork-join execution model and worksharing runtime for OpenMP worksharing loops. In fork-join model, a master thread is the single thread which begins execution until it reaches a parallel region. Then, the master thread forks a team of worker threads and all threads execute the parallel region concurrently. Upon exiting parallel region, all threads synchronize and join, and only the master thread is left after the parallel region. In data parallelism, the iterations of a parallel loop are distributed among threads, which is called worksharing. For tasking in OpenMP, a workstealing scheduler is normally used within the fork-join runtime [15, 5]. Using Intel OpenMP runtime library as example [2], the runtime employs a hybrid schedulers of fork-join, worksharing, and workstealing.

The Cilk Plus and TBB use random work-stealing scheduler [11] to dynamically schedule tasks on all cores. In Cilk Plus, each worker thread has a double-ended queue (deque) to keep list of the tasks. The work-stealing scheduler of a worker pushes and pops tasks from one end of the queue and a thief worker steals tasks from the other end of the queue. In this technique, if the workload is unbalanced between cores, the scheduler dynamically balance the load by stealing the tasks and executing them on the idle cores. Cilk Plus uses the workstealing run-

TABLE III: Comparison of Mutual Exclusions and Others

	Mutual exclusion	Language or library	Error handling	Tool support
Cilk Plus	containers, mutex, atomic	C/C++ elidable language extension	x	Cilkscreen, Cilkview
CUDA	atomic	C/C++ extensions	x	CUDA profiling tools
C++11	std::mutex, atomic	C++	C++ exception	System tools
OpenACC	atomic	directives for C/C++ and Fortran	x	System/vendor tools
OpenCL	atomic	C/C++ extensions	exceptions	System/vendor tools
OpenMP	locks, critical, atomic, single, master	directives for C/C++ and Fortran	omp cancel	OMP Tool interface
PThread	pthread_mutex, pthread_cond	C library	pthread_cancel	System tools
TBB	containers, mutex, atomic	C++ library	cancellation and exception	System tools

time for scheduling data parallelism specified using cilk_for. Achieving load balancing across cores when there are more tasks than the number of cores is known as composability problem [19]. In Cilk Plus, the composition problem has been addressed through the workstealing runtime. In OpenMP, the parallelism of a parallel region is mandatory and static, i.e., system must run parallel regions in parallel, so it suffers from the composability problem when there is oversubscription.

A work-stealing scheduler can achieve near-optimal scheduling in a dedicated application with a well-balanced workload [1]. OpenMP uses work-stealing only in task parallelism. For data parallelism, it uses work-sharing scheduler, in which users are required to specify the granularity of assigning tasks to the threads. In OpenMP, task schedulers are based on work-first and breadth-first schedulers. In work-first, tasks are executed once they are created, while in breadth-first, all tasks are first created, and the number of threads is limited by the thread pool size. In C++11, task can be generated by using std:async and a new thread is created by using std::thread. In task level parallelism, runtime library manages tasks and load balancing, while in thread level parallelism programmers should take care of load balancing.

The runtime systems for low-level programming models (C++ std::thread, CUDA, OpenCL, and PThreads) could be simpler than that for more comprehensive models such as OpenMP, Cilk Plus, OpenACC, C++ std::future and TBB. The C++11 standard enables users to make the most use of the available hardware directly using the interfaces that are similar to the PThread library [14]. The implementation of the std::thread interfaces could be simple mapping to PThread APIs, thus has minimum scheduling in the runtime. It leaves to users to make mapping decisions between threads and cores.

The support for offloading in models such as OpenACC and OpenMP also varies depending how much the offloading features should be integrated with the parallelism support from CPU side, e.g. whether it allows each of the CPU threads to launch an offloading request. The support for data/event-driven parallelism also adds another dimension of complexity in the runtime systems, to both CPU parallelism and the offloading, as shown in our previous work for implementing OpenMP task dependency [12].

IV. Performance Comparison

For performance comparison, we only choose OpenMP, Cilk Plus and C++ which we believe are the three most used models for CPU parallelism. We also concentrate on data and task parallelism patterns in comparisons which have been used widely.

Two set of applications have been developed for experimental evaluation, which are simple computation kernels, and applications from the Rodinia benchmark [7]. For each application, six versions have been implemented using the three APIs. The OpenMP data parallel makes use of the parallel and for directives, task version uses the task and taskwait directives. For Cilk Plus versions, cilk_for statement has been used for data parallelism, while cilk_spawn and cilk_async have been used to implement the task version. The two versions for C++11 use std::thread and std::async APIs. For data parallelism support in C++, we use a for loop and manual chunking to distribute loop iterations among threads and tasks. In principle, OpenMP static schedule is applied to all the three models for data parallelism, allowing us to have fair comparison of the runtime performance. The experiments were performed on a machine with two-socket Intel Xeon E5-2699v3 CPUs and 256 GB of 2133 MHz DDR4 ECC memory forming a shared-memory NUMA system. Each socket has 18 physical cores (36 cores in the system) clocked at 2.3 GHz (turbo frequency of 3.6 GHz) with two-way hyper-threading. The host operating system is CentOS 6.7 Linux with kernel version 2.6.32-573.12.1.el6.x86_64. The code was compiled with the Intel icc compiler version 13.1.3. The Rodinia version is 3.1.

A. Benchmark Kernels

Small kernels provide insights of runtime scheduling overhead between different programming models and runtime. We used the following scientific kernels.

Axpy: Axpy solves the equation y = a * x + y where x and y are vectors of size N and a is scalar. The vector size used in evaluation is 100 Million(M). Fig.1 shows the performance results. The C++ implementation has two different versions for std::thread and std::async: recursive and iterative versions. In recursive versions, in order to avoid creating a large number

of small tasks, a cut-off BASE is used [9], which is calculated as N divided by the number of threads. This helps to control task creation and to avoid oversubscription of tasks over hardware threads.

Referring to Fig.1, it is obvious that cilk_for implementation has the worst performance, while other versions almost show the similar performance that are around two times better than cilk_for except for 32 cores. The reason is that workstealing operations in Cilk Plus serialize the distributions of loop chunks among threads, thus incurring more overhead than worksharing approach. Also, if the function in the loop is not big enough, the stealing costs could degrade the performance.

Sum: Sum calculates sum of a * X[N], where X is a vector of size N and a is scalar. The vector size is 100M. Fig.2 shows the performance of different implementations for this application. cilk_for performs the worst while omp_task has the best performance and performs around five times better than cilk_for as Sum is the combination of worksharing and reduction, showing that workstealing for worksharing+reduction is not the right choice.

Matvec: Matvec is matrix vector multiplication of problem size 40k. The performance of this kernel is presented in Fig.3 which shows that cilk_for performs around 25% worse than the other versions.

Matmul: Matmul is matrix multiplication of 2k problem size. The results in Fig.4 show cilk_for has the worst performance for this kernel as well, and other versions perform around 10% better than cilk_for. The performance trend of these three kernels (Axpy, Matvec, Matmul) are similar because they have the same nature and the function in the loop is small. However, as the computation intensity increases from AXPY to Matvec and Matmul, we see less impact of runtime scheduling to the performance.

Fibonacci: Fibonacci uses recursive task parallelism to compute the nth Fibonacci number, thus cilk_for and omp_for are not practical. In addition, for recursive implementation in C++, when problem size increases to 20 or above, the system hangs because huge number of threads is created. Thus, for this application, only the performance of cilk_spawn and omp_task for problem size 40 are provided. As it is shown in Fig 5. cilk_spawn performs around 20% better than omp_task except for 1 core, because the workstealing for omp_task in Intel compiler uses lock-based deque for pushing, popping and stealing tasks in the deque, which increases more contention and overhead than the workstealing protocol in Cilk Plus [11].

Overall, for Axpy, Matvec and Matmul, cilk_for has the worst performance while other versions perform similarly. For Sum, cilk_for performs worst while omp_task has the best performance. For Fibonacci, cilk_spawn performs better than omp_task. It demonstrates that worksharing for task parallelism may incur more overhead, while for data parallelism workstealing creates more overhead. Thus, worksharing mostly shows better performance for data parallelism and workstealing has better performance for task parallelism. Even though performance trends for different implementations of each application have been varied, but the algorithms perform similarly with regard to execution time, as more threads of execution participated in the execution of work. However, the rate of decrease is slower as more threads are added. This can be explained by the overhead involved in the creation and management of those threads.

B. Benchmarks from Rodinia

BFS: Breadth-first traversal is an algorithm that starts searching from root of graph and search neighbor nodes before moving to the next level of tree. There are two parallel phases in this application. Each phase must enumerate all the nodes in the array, determine if the particular node is of interest for the phase and then process the node. The first phase visits the discovered nodes and discovers new nodes to visit. The second phase marks the newly discovered nodes as discovered for the next run of the first phase. Each phase is parallelized on its own.

For parallel version of BFS, each thread processes the same number of tasks while the amount of work that they handle might be different. This algorithm does not have contiguous memory access, and it might have high cache miss rates.

A data set was generated that describes a graph consisting of 16 million inter-connected nodes. The Fig.6 represents the test runs of this application. Overall, this algorithm scales well up to 8 cores. The comparative execution time of the different implementations shows cilk_for has the worst performance while others perform closely. This happens because workstealing creates more overhead for data parallelism, while worksharing for data parallelism is able to have close performance to other implementations.

HotSpot: HotSpot is a tool to estimate processor temperature based on an architectural floorplan and simulated power measurements [13] using a series of differential equations solver. It includes two parallel loops with dependency to the row and column of grids. Each thread receives the same number of tasks with possible different workload. The memory access is not sequential for this algorithm that might result in more execution time because of more cache miss rates. The problem size used for the evaluation was 8192.

For this application, data parallelism of both Cilk Plus and OpenMP show poor performance, which most likely happen because of the dynamic nature of this algorithm and dependency in different compute intensive parallel loop phases. Task version of OpenMP also shows weak performance for small number of threads because of more overhead costs, but a slightly stronger correlation can indicate that as more threads are added, the task parallel implementations are gaining more than the worksharing parallel implementations. The execution time of the other implementations, on the other hands, are close and scale well specially when more thread is adding. However, the rate of decrease gets slower for higher number of threads.

LUD: LU Decomposition (LUD) accelerates solving linear equation by using upper and lower triangular products of a matrix. Each sub-equation is handled in separate parallel region, so the algorithm has two parallel loops with dependency to an outer loop. In each parallel loop, thread receives the same number of tasks with possible different amount of workload.

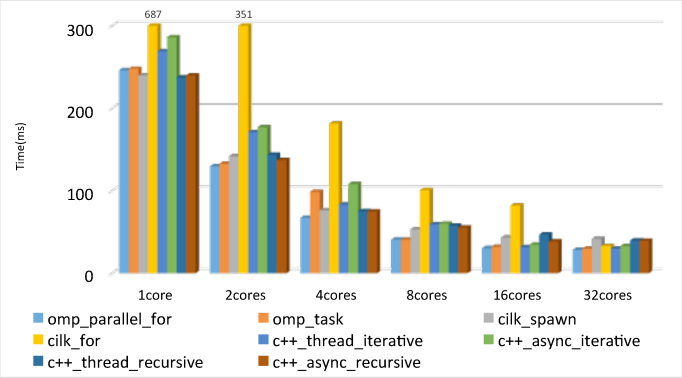

Fig. 1: Axpy performance

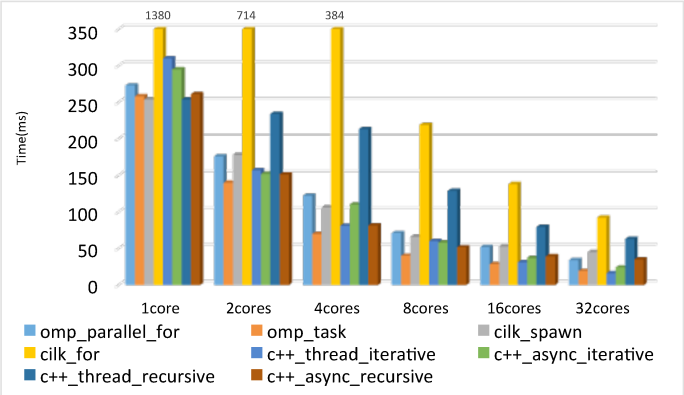

Fig. 2: Sum performance

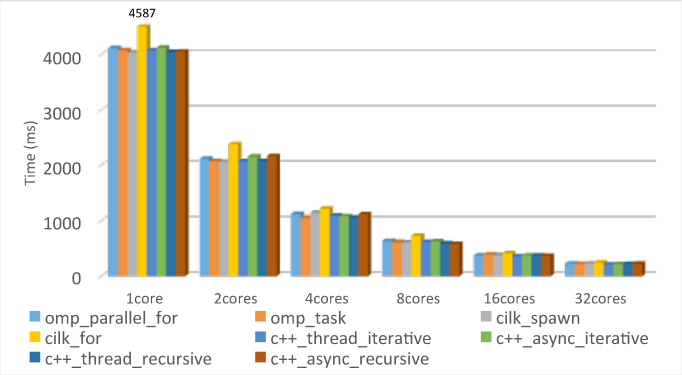

Fig. 3: Matvec performance

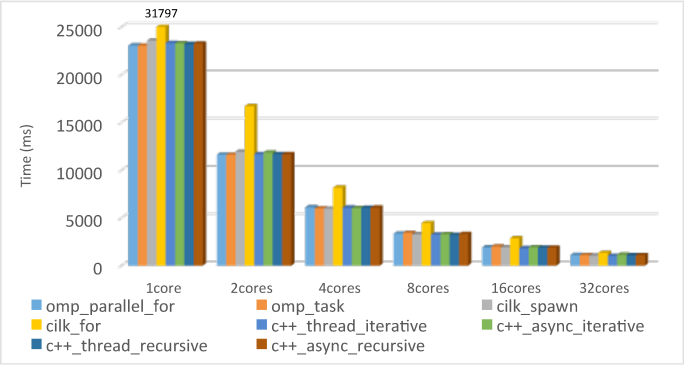

Fig. 4: Matmul performance

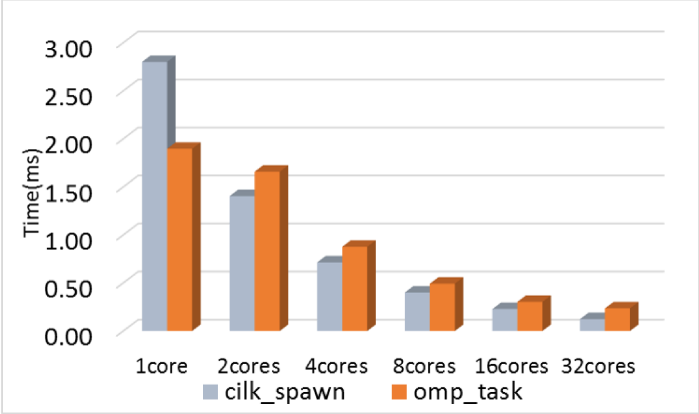

Fig. 5: Fib performance

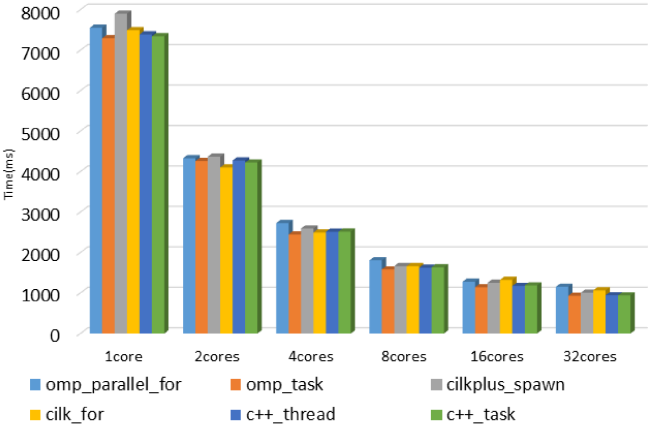

Fig. 6: BFS performance

This algorithm has good memory locality feature because it sequentially accesses to the memory block that leads to low cache miss rates. The problem size for this evaluation was 2048. Refereeing to Fig.8, it is shown that OpenMP data parallelism and Cilk Plus task parallelism achieve the good performance, while task version of OpenMP and data parallel version of Cilk Plus have the worst performance. Because for data parallelism workstealing creates more overhead. Thus, worksharing shows better performance for data parallelism and workstealing has better performance for task parallelism. OpenMP task version also is not able to achieve good performance, because of the high overhead costs in OpenMP tasking implementation. Referring to Fig.8, results show that although the execution time of C++11 task and thread versions are very close, thread version performs better than task version especially when the number of core increases because task version doesn't guarantee low overhead and good load balancing with different amount of workload that each task might receive. Thus, for higher number of threads C++11 task implementation shows an upward trend.

LavaMD: LavaMD calculates particle potential and relocation due to mutual forces between particles within a large 3D

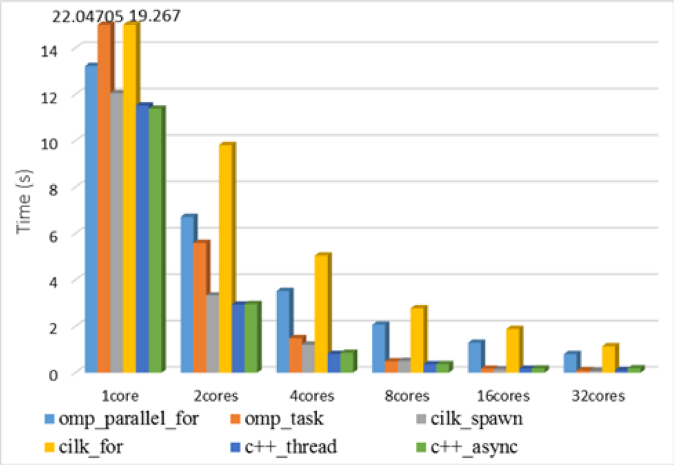

Fig. 7: HotSpot performance

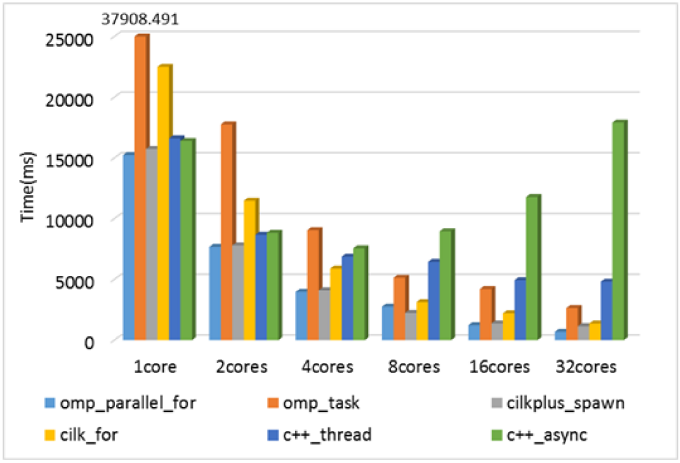

Fig. 8: LUD performance

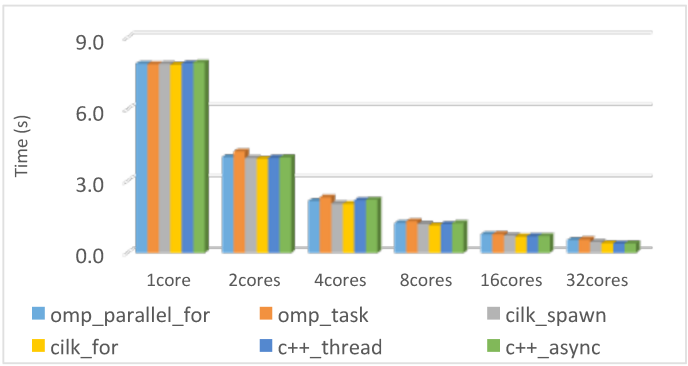

Fig. 9: LavaMD performance

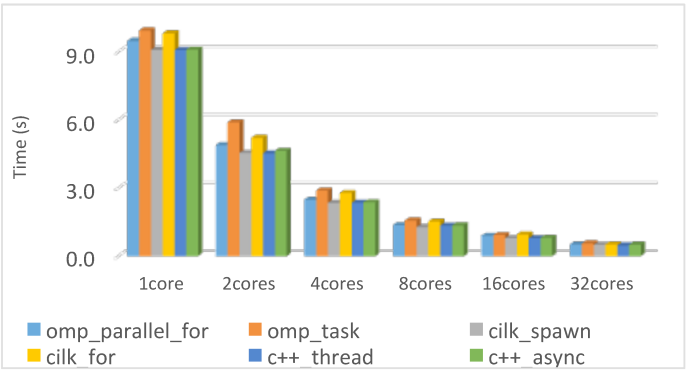

Fig. 10: SRAD performance

space [20]. In this application, there is a large space of particles, which is divided into large boxes. Each large box itself is divided into small boxes that are processed concurrently. These processes have been handled in one parallel loop where each thread receives the same amount of work. The memory access for this loop is ordered, which leads to better cache hit rates. The problem size for the evaluation was 10 boxes. Referring to Fig.9, the comparison of data and task implementations show that different implementations behave almost the same and this most likely happens because this application has good load balancing. However, data parallel versions of Cilk Plus still behave a bit better because LavaMD is a compute-intensive algorithm, in which each thread receives the same amount of work.

Regarding OpenMP implementations, for the small number of threads, the results are very similar. But by adding more number of threads, OpenMP data-parallelism version outperforms task-parallelism version, because task version implementation creates more overhead as the number of cores increases. The trend of both task and thread versions of C++11 are almost the same and these two implementations show better results for higher number of threads (16 and 32 cores).

SRAD: Speckle Reducing Anisotropic Diffusion (SRAD) is a method for removing noise in ultrasonic/radar imaging. This application has two different parallel loops. The tasks between threads are distributed equally with the same amount of work. Each loop has contiguous memory access, which helps to increase the cache hit rates and to reduce the possible overhead. The considered problem size for this application was 8192. As shown in Fig.10, all the implementations performed similarly with regard to the execution time and speedup. The task implementations of Cilk Plus and future/thread versions of C++11 behave similarly and have the best performance, while the OpenMP task version delivers the worst performance due to high tasking overhead in the implementation. cilk_for still is not among the best performance but its performance is closer to other implementations in comparison with the rest applications except LavaMD. This more likely happens because in SRAD and LavaMD applications, each thread receives task with possible same amount of work which results in better workload uniformity and consequently, different implementations behave more closely. Overall, the results of the Benchmark kernels show that the performance of different implementations could be varied with respect to some factors such as load balancing and task workload uniformity, scheduling and loop distribution overhead, as well as the runtime systems. Generally, worksharing may create less overhead for data parallelism and workstealing has better performance for task parallelism as it is shown in LUD and BFS applications. However, if the application has adequate load balancing and uniform task workloads the effect might be less in the result and different implementations would per-

form more closely such as LavaMD and SRAD applications. Lastly, when application has a dynamic nature and there is dependency in different parallel loop phases, tasking might outperform worksharing such as Hotspot application.

V. Related Work

A comparative study of different task parallel frameworks has been done by Podobas et al [18] using BOTS benchmark [9]. Different implementations of OpenMP from Intel (ICC), Oracle (SunCC), GCC (libgomp), Mercurium/Nanos++[8], OpenUH [3] and two runtime implementations including Wool [10] and Cilk Plus, had been evaluated. Speedup performance, power consumption, caches and load-balancing properties had been considered for the evaluation. They however did not compare and evaluate other language features.

Shen et al [7] did a performance evaluation for eleven different types of applications in Rodinia benchmark on three multi-core CPUs using OpenMP. This work examined the results by scaling the dataset and the number of threads for each application. They found out OpenMP generally performs and scales well for most applications reaching the maximum performance around the number of hardware cores/threads.

Olivier et al evaluated performance of task parallelism in OpenMP using Unbalanced Tree Search (UTS) benchmark [17]. They also compared expressiveness and scalability for OpenMP, Cilk, Cilk Plus, Intel Thread Building Blocks, as well as an OpenMP implementation for this benchmark. They concluded that only the Intel compiler illustrates good load balancing on UTS, but it still does not have an ideal speedup.

Leist et al [16] also did a comparison of parallel programming models for C++ including Intel TBB, Cilk Plus and OpenMP. This work considered three common parallel scenarios: recursive divide-and-conquer, embarrassingly parallel loops and loops that update shared variables. Their results indicated that all of the models perform well across the chosen scenarios, and OpenMP is more susceptible to a loss of performance in comparison with other frameworks, which perform similarly.

Ajkunic et al [4] did a similar comparison to Leist et al's work, but they considered two additional APIs: OpenMPI and PThreads. They only chose the matrix multiplication for comparison. The authors concluded that OpenMP performs poorly when compared to Cilk Plus and TBB. With regard to effort required for implementation, OpenMP and C++ require the least effort, whereas TBB and PThreads require more effort.

Our work differs from previous research in that we summarized an extensive list of features for threading models and compared three most widely used programming languages, OpenMP , Cilk Plus and C++11 for both task and data parallelism. We also provided a comparison of programming interfaces and runtime systems for these programming languages.

VI. Conclusion

In this paper, we report our extensive study and comparison of state-of-the-art threading parallel models for the existing and emerging computer systems. We provide a summary of language features that are considered in a programming model and highlight runtime scheduling techniques to implement a programming model. We have compared the performance of three popular parallel programming models: OpenMP, Cilk Plus and C++11 with regard to task parallelism and data parallelism. Overall, the results show that the execution times for different implementations vary because of strategies of load balancing in the runtime and task workload uniformity of applications, and scheduling and loop distribution overhead.

For example, workstealing runtime could incur high overhead for data parallel programming of applications because of the serialization of the distribution of loop chunks among parallel threads. However, if the application has adequate load balancing and efficient task workload the effect of overhead costs created by runtime could be less. Thus, dynamic nature of task creation can influence the performance.

Acknowledgment

This material is based upon work supported by the National Science Foundation under Grant No. 1409946 and 1652732.

References

[1] CilkPlus . https://www.CilkPlus.org/cilk-plus-tutorial. Accessed: July 03, 2016.

[2] Intel OpenMP runtime library. https://www.openmprtl.org/.

[3] Cody Addison, James LaGrone, Lei Huang, and Barbara Chapman. OpenMP 3.0 tasking implementation in openuh. In *Open64 Workshop at CGO*, volume 2009, 2009.

[4] Ensar Ajkunic, Hana Fatkic, Emina Omerovic, Kristina Talic, and Novica Nosovic. A comparison of five parallel programming models for c++. In *MIPRO, 2012 Proceedings of the 35th International Convention*, pages 1780–1784. IEEE, 2012.

[5] Eduard Ayguadé, Nawal Copty, Alejandro Duran, Jay Hoeflinger, Yuan Lin, Federico Massaioli, Xavier Teruel, Priya Unnikrishnan, and Guansong Zhang. The design of OpenMP tasks. *IEEE Transactions on Parallel and Distributed Systems*, 20(3):404–418, 2009.

[6] Hans-J Boehm and Sarita V Adve. Foundations of the c++ concurrency memory model. In *ACM SIGPLAN Notices*, volume 43, pages 68–78. ACM, 2008.

[7] Shuai Che, Michael Boyer, Jiayuan Meng, David Tarjan, Jeremy W. Sheaffer, Sang-Ha Lee, and Kevin Skadron. Rodinia: A benchmark suite for heterogeneous computing. In *Proceedings of the 2009 IEEE International Symposium on Workload Characterization (IISWC)*, IISWC '09, pages 44–54, Washington, DC, USA, 2009. IEEE Computer Society.

[8] Alejandro Duran, Eduard Ayguadé, Rosa M Badia, Jesús Labarta, Luis Martinell, Xavier Martorell, and Judit Planas. Ompss: a proposal for programming heterogeneous multi-core architectures. *Parallel Processing Letters*, 21(02):173–193, 2011.

[9] Alejandro Duran González, Xavier Teruel, Roger Ferrer, Xavier Martorell Bofill, and Eduard Ayguadé Parra. Barcelona OpenMP tasks suite: A set of benchmarks targeting the exploitation of task parallelism in OpenMP. In *38th International Conference on Parallel Processing*, pages 124–131, 2009.

[10] Karl-Filip Faxén. Wool-a work stealing library. *ACM SIGARCH Computer Architecture News*, 36(5):93–100, 2008.

[11] Matteo Frigo, Charles E Leiserson, and Keith H Randall. The implementation of the cilk-5 multithreaded language. In *ACM Sigplan Notices*, volume 33, pages 212–223. ACM, 1998.

[12] Priyanka Ghosh, Yonghong Yan, Deepak Eachempati, and Barbara Chapman. *A Prototype Implementation of OpenMP*

Task Dependency Support, pages 128–140. Springer Berlin Heidelberg, Berlin, Heidelberg, 2013.

[13] Wei Huang, Shougata Ghosh, Sivakumar Velusamy, Karthik Sankaranarayanan, Kevin Skadron, and Mircea R Stan. Hotspot: A compact thermal modeling methodology for early-stage vlsi design. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 14(5):501–513, 2006.

[14] Cameron Hughes and Tracey Hughes. *Parallel and distributed programming using C++*. Addison-Wesley Professional, 2004.

[15] James LaGrone, Ayodunni Aribuki, Cody Addison, and Barbara Chapman. A runtime implementation of OpenMP tasks. In *International Workshop on OpenMP*, pages 165–178. Springer, 2011.

[16] Arno Leist and Andrew Bilman. A comparative analysis of parallel programming models for c++. In *The Ninth International Multi-Conference on Computing in the Global Information Technology*, 2014.

[17] Stephen L Olivier and Jan F Prins. Comparison of OpenMP 3.0 and other task parallel frameworks on unbalanced task graphs. *International Journal of Parallel Programming*, 38(5-6):341–360, 2010.

[18] Artur Podobas, Mats Brorsson, and Karl-Filip Faxén. A comparative performance study of common and popular task-centric programming frameworks. *Concurrency and Computation: Practice and Experience*, 27(1):1–28, 2015.

[19] Arch D Robison. Composable parallel patterns with intel cilk plus. *Computing in Science and Engineering*, 15(2):66–71, 2013.

[20] Lukasz G Szafaryn, Todd Gamblin, Bronis R De Supinski, and Kevin Skadron. Experiences with achieving portability across heterogeneous architectures. *Proceedings of WOLFHPC, in Conjunction with ICS, Tucson*, 2011.