Application Characteristics of Many-tasking Execution Models

T. Gilmanov, M. Anderson, M. Brodowicz and T. Sterling
Center for Research in Extreme Scale Technologies, Indiana University, Bloomington, IN, USA

Abstract— *Performance gain for computer systems through Moore's Law is jeopardized by the limitations of clock rate growth due to power considerations and the limitations in instruction-level parallelism improvement from processor core computer architecture experienced over the last decade. High performance computer architectures are addressing this challenge through multicore processors that combine many processing units on a single chip. For this class of machines, conventional programming and execution practices, while still effective for some application algorithms, are suffering in scalability for many others. Multithreaded runtime systems offer convenient many-tasking execution model implementations that show increased efficiency and scalability through exploiting medium-grained thread parallelism. This paper provides an overview of the existing runtime systems and libraries used for many-tasking computation. They are empirically examined in terms of overheads and the potential impact resulting on application performance. Intra-node performance aspects are investigated using synthetic benchmarks.*

Keywords: runtime systems, thread parallelism, benchmarks

1. Introduction

Sustaining performance growth of high performance computing systems through device technology improvements reflected by Moore's Law is becoming increasingly challenging using conventional static programming and execution methods. Due to power limitations, processor core clock rates are constrained and in some cases even reduced. Also, opportunities for instruction level parallelism (ILP) within core architectures have been exhausted as well. Thus, the two principal means of performance growth of microprocessor cores that have served high performance computing (HPC) for more than two decades have ceased to be a significant factor. Alternative architecture strategies based on multi-core processors accelerators to deliver continued performance growth have introduced the need for abundant and continually increasing medium-grained parallelism to achieve scalability.

While the concept of runtime systems is not new, their application to HPC is extremely constrained due to the problems of additional overheads being introduced. But runtimes offer the possibility of dynamic adaptive resource management and task scheduling employing information during time of execution that is not available at time of programming or compilation. It is postulated that access to runtime information may provide the means to dramatically improve operational efficiency while achieving significant increases in scalability. Such runtime systems must reduce typical sources of work starvation, latency effects, overhead, and waiting for contention resolution to shared resources (physical or logical).

In a multiprocessor setting, task schedulers can implicitly load balance computations and more easily enable fine grain computations. Asynchrony management constructs remove global barriers and enable codes to overlap phases of computation to improve performance. In a distributed setting, this introduces the capability to hide network latency and better overlap computation and communication. Both Charm++ [1] and Unified Parallel C (UPC) [2] have for a long time exhibited the major benefits of task parallelism along with a shared address space. Other more recent approaches include the Intel Threading Building Blocks library (TBB) [3], the High Performance ParalleX (HPX) runtime system [4], Cilk plus [5], Chapel [6], Qthreads [7]. Even MPI and OpenMP [8] have been used to accomplish task parallelism.

This paper provides an overview of the existing runtime systems and libraries used for many-tasking computation. They are empirically examined in terms of overheads and the potential impact resulting on application performance. The approach taken starts by characterizing cumulative overhead of task initialization, execution, and end-synchronization for several many-tasking runtime systems and libraries, namely: TBB, Cilk plus (both Intel and gcc versions), OpenMP (both Intel and gcc versions), Charm++, HPX, and Qthreads. We explore the impact of NUMA awareness and the TCMalloc allocator [9] separately for each runtime system and library. Two micro benchmarks are presented: homogeneous task spawn and heterogeneous task spawn.

2. Related Work

In spite of the significant interest in task parallelism, there exists relatively few side by side comparisons of the various overheads in implemented many-tasking execution models. This is partially explained by the fact that the functional capabilities of a fundamental task vary widely from implementation to implementation and depend upon the particular many-tasking execution model that the runtime system intends to implement. This makes direct comparisons difficult. However, some key similarities are shared among the various execution models and some studies have been conducted to compare these overheads.

Burkhart et al. [10] examined Chapel, Unified Parallel C, OpenMP, and MPI using a generic stencil computation as the unified performance test. system. The work by Olivier et al. [11] discusses an approach to compare the performance of a

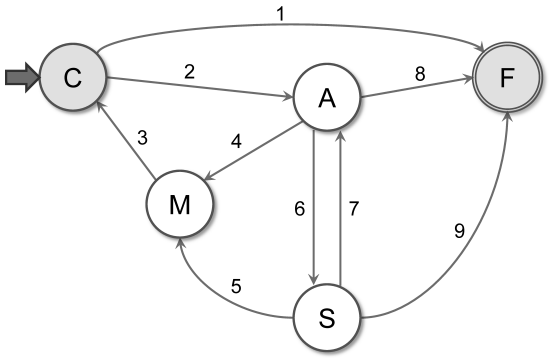

Fig. 1: General task state diagram. The nodes (marked with letters) represent states; the edges (with number labels) signify transitions between the states. The life cycle of a task starts in the creation state C identified by the arrow and ends in finished state F; both are represented by shaded circles to emphasize the transition endpoints. The remaining states include active (or execution) state A, suspended state S, and migration state M.

few runtime systems, including OpenMP, Cilk, Cilk++, and Intel TBB using a highly unbalanced task graph in order to introduce challenges arising in work balancing, scheduling, and termination detection. Other comparison works include comparing Cilk with MPI for finite differencing [12].

3. Task benchmarks

Central to a many-tasking execution model performance quantification is the threading characteristics of a simulation as it scales to larger core counts. Some implementations support thread asynchrony management thereby enabling activation, migration, and suspension of tasks. Other implementations support a much smaller subset of possible thread states. Quantification of these overheads is fundamental to the development of a performance model for design and deployment of many-tasking based applications.

Figure 1, even though substantially simplified, illustrates the significant number of possible task states and transitions, each with an associated latency and implementation overhead, in a many-tasking execution model. Note that while some runtime systems do not explicitly use the notion of tasks, this term is often equivalent to a work unit executed by a user level thread (*cf.* HPX, Qthreads). Among the possible task states are creation and initialization (C), migration (M), active (A), suspended (S), and finished (F). Every task's life cycle begins in the creation state C. The creation state involves a potentially complex sequence of operations, ranging from creating an entry in the runtime system's task table or queue, allocating the memory to hold the necessary description of task state, function to execute and its arguments, hooks to the related runtime system data structures and functions, *etc*. An initialized task typically transitions along edge 2 from the creation state C to the active execution state A. In rare cases when the work in the application has been already completed but an abundance of tasks has been created to increase the parallelism or conditionally explore additional branches of program execution, they will be automatically terminated after moving to state F. The active execution state A is a superposition of internal states in which a task may actively execute on a processor, may await the assignment of a physical execution resource, or be queued in scheduler's run queue. These internal states share a common characteristics: all data dependencies required by the task to perform the next logical operation must be satisfied. If this condition is not true, for example, the task needs data returned by a pending I/O operation or data that has to be conveyed by a message from a remote node, the runtime system places the task in a suspended state S. Similarly to the active state, the suspended state may internally include a number of variants. The efficient utilization of the hardware resources depends on the runtime system's ability to identify tasks blocked due to data dependencies and removing them from the active state; note that this is not always straightforward as such knowledge may be available only at the operating system level. Over the lifetime of a complex task, the transitions 6 and 7 between active and suspended state may happen many times. Other transitions from the active and suspended states lead to either termination F or migration M. Currently, only few runtime systems support migration of task state at a global (distributed machine) scale; moreover, additional restrictions may be placed on which tasks are permitted to migrate to remote execution locales. Most general implementations will support migration for both suspended and executing tasks (although the latter will have to be preempted from the execution resource and removed from the active queue before the state migration sequence is initiated). Since recreation of task structures on the remote end shares many similarities with creation of a new task (with possible differences in initialization phase), transition 3 from migration leads to the initial state C. All tasks entering the finished state F are terminated. Their representation is removed from the runtime system data structures and all related resources are reclaimed. Entering this state may not always cause the immediate release of the resources used by the task. In cases where the final value computed by a task has to be communicated to other tasks (akin to thread join), the related portion of task state may linger long after its execution phase is over. Additionally, other dedicated system entities, such as garbage collectors, may ultimately decide when to reclaim the terminated task's storage.

We first consider the most fundamental task phases in a many-tasking implementation shared by all aforementioned implementations: task initialization, task execution, and final synchronization. They are directly related to the task states described above. The task initialization corresponds to the creation state C, perhaps with the added overhead of an external loop generating the task instances. Task execution is a superposition of active (A) and suspended state (S), since migration

was not explored in our tests. Given that the single work grain does not include any blocking operations, once a task is moved to the active state, it will remain there until completion. The synchronization at the end of workload certainly involves transition of individual tasks to the F state, but also includes transition between states A and S of the master task (spawner) that monitors the execution. This may be caused, for example, by the readout of an atomically updated counter representing the number of completed tasks, or by checking the state of a future object which hasn't been updated yet. Note that such updates may be performed either by the code of individual tasks directly, or by a background system thread, depending on the implementation. For the former, additional A↔S transitions (possibly multiple) may occur at the very end of task execution during periods of high contention when competing for access to the shared resource.

The cumulative overhead of these task phases is crucial in determining the optimal task granularity for a specific application simulation. The tests described in the next two subsections analyze the task spawn performance on various systems using two modes of operation; homogeneous and heterogeneous.

3.1 Homogeneous Task Spawn

In the homogeneous task benchmark presented in this section, we spawn a fixed number of tasks, 5×10^5 in this case, where each task has the same work unit defined by a function given in Listing 1. The grain size given the work unit is specified by the number of iterations in the for loop in Listing 1. It is varied across several orders of magnitude from 0, 10, 100, 1000, and 10,000 in order to reflect fine to medium grain application simulations. Code listings for this benchmark are given for OpenMP (Listing 2), HPX (Listing 3), Cilk+ (Listing 4), Charm++ (Listing 5), Qthreads (Listing 6), and TBB (Listing 7).

Listing 1: Work function used in conjunction with the various task spawn tests [13]

```
void worker(){
  double volatile d = 0.;
  for (uint64_t i=0; i<delay; ++i)
      d += 1. / (2. * i + 1.);
}
```

Listing 2: OpenMP task spawn [13]

```
#pragma omp parallel
#pragma omp single
{
  for (uint64_t i=0; i<tasks; ++i)
    #pragma omp task untied
    worker();

  #pragma omp taskwait
}
```

Listing 3: HPX task spawn [13]

```
for (uint64_t i=0; i<tasks; ++i)
  register_work(HPX_STD_BIND(&worker));

do suspend();
while (get_thread_count() > 1);
```

Listing 4: Cilk+ task spawn [14]

```
for (uint64_t i=0; i<tasks; ++i)
  cilk_spawn worker();

cilk_sync;
```

Listing 5: Charm++ task spawn

```
main::main(CkArgMsg *m){
  mainProxy = thishandle;
  count = tasks;

  for (uint64_t i=0; i<tasks; ++i)
    CProxy_worker::ckNew();
}

void main::results(){
  if (0 == --count) CkExit();
}

worker::worker(){
  double volatile d = 0.;
  for (uint64_t i=0; i<delay; ++i)
    d += 1. / (2. * i + 1.);

  mainProxy.results();
}
```

Listing 6: Qthreads task spawn [14]

```
for (uint64_t i=0; i<tasks; i++) {
  qthread_fork(worker_task, NULL, NULL);

  do qthread_yield();
  while (donecount != tasks);
}
```

Listing 7: TBB task spawn [14]

```
// body of worker::execute()
// identical to Listing 1

tbb::task *spawner::execute(){
  set_ref_count(tasks+1);

  for (uint64_t i=0; i<tasks; ++i){
    worker& a =
      *new(tbb::task::allocate_child())
        worker();

    if (i != tasks-1) spawn(a);
    else spawn_and_wait_for_all(a);
  }
}

int tbb_main(){
  spawner& a =
    *new(tbb::task::allocate_root())
      spawner();
  tbb::task::spawn_root_and_wait(a);
  return 0;
}

int main(int argc, char **argv){
  tbb::task_scheduler_init init(tasks);
  return tbb_main();
}
```

The task spawn tests were benchmarked on a computer equipped with dual 8-core E5-2670 Intel processors running at 2.6 GHz and 32 GB of memory. This machine is representative of the type of nodes commonly found in recent supercomputer configurations. The hyper-threading was explicitly disabled in BIOS to avoid issues related to inconsistent enumeration of

Package	Version
Compilers	
GNU binutils	2.23.1
gcc	4.6.3 release
gcc Cilk+ patch	4.8.0 (20121210)
Intel Composer	2013.1.117
Support libraries	
boost	1.52.0
hwloc	1.6
Google Performance Tools	2.0
Runtime systems	
Charm++	6.4.0
HPX	git hash a0786ed (12/12/2012)
Qthreads	git hash f2ae33d (01/02/2013)
TBB	4.1.20121003

Table 1: Versions of software packages used in task spawn tests.

Iterations	0	10	100	1,000	10,000
Duration	2.5 ns	103 ns	1.059 μs	10.61 μs	106.2 μs

Table 2: Actual duration of the work units used in the tests.

logical processors between different libraries used for NUMA affinity management. Red Hat Enterprise Linux version 6.3 provided the basic operating environment. It was augmented with the prerequisite libraries and compilation tools required by runtime systems tested; they are listed in Table 1 for reference.

The work unit durations for various grain sizes collected on a minimally loaded system are listed in Table 2. To obtain them, many runs were performed in sequence to alleviate the effects of finite timer resolution and ensure full cache initialization. Since the E5 Xeon processors feature Turbo Boost technology (temporary increase of clock speed when executing workload on a limited number of cores), the presented numbers are very close to the minimal possible execution times per task on that architecture.

Unless stated otherwise (in particular for Cilk+ and OpenMP tests), all runtime system libraries and test codes were compiled using gcc-4.6.3. Using more recent 4.7 series resulted in unstable code in some instances. Whenever possible, production or release configurations were used, with optimization flag of "-O2" and no debugging information. Since certain implementations exhibit measurable performance improvements when using additional external libraries, separate tests were performed for these configurations. They include NUMA affinity management (typically using `hwloc` library), and thread-aware memory allocation (e.g., `tcmalloc` from Google Performance Tools suite). The impact of these packages in a typical case can be seen in Figure 2. The NUMA-aware thread assignment improves the performance for lower numbers of cores by limiting the degree of freedom in distributing the OS threads across available NUMA domains; these benefits are gone when the number of underlying OS threads matches the core count. Contrariwise, `tcmalloc` effects are greatest for high OS thread counts, as these cases exhibit the highest access contention to shared resources used by conventional memory allocation routines. To avoid excessive clutter in charts comparing the performance of multiple runtime systems, only the best performing configurations were selected, with relevant details embedded in plot labels.

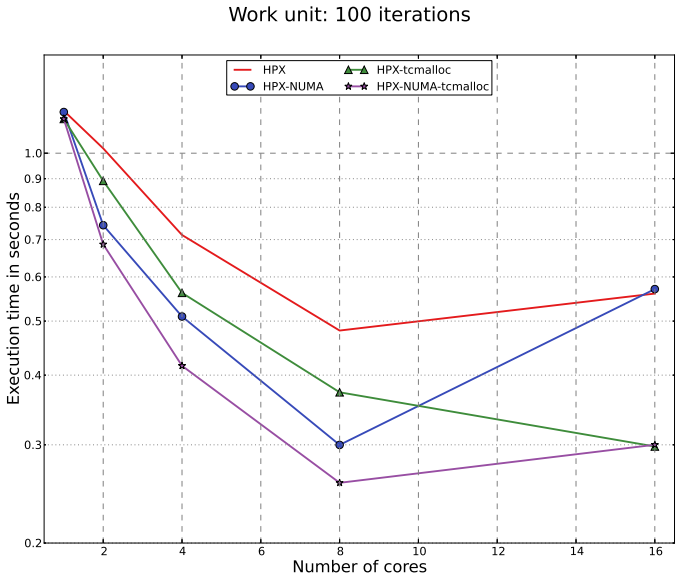

Fig. 2: Impact of NUMA-aware OS thread placement and thread caching memory management on many-tasking performance in HPX using a medium granularity where each tasks performs $\sim 1\mu$s of workload.

Special care was taken to ensure that the simulated work unit is identical between the benchmarks and no unfair optimizations are taking place. For this reason, the work unit function was placed in a separate compilation unit and inter-procedural analysis in the linkage phase was suppressed. The memory management calls, required by several systems to create and release required data structures representing individual tasks were removed outside the timed region, or low-overhead equivalents provided by the runtime systems were used where possible. Similarly, detection of the completion of task execution phase relied on optimized techniques suggested by the programming manuals or was leveraged from code examples distributed with the runtime systems.

Figures 3–7 present the results of homogeneous task spawn for various runtime systems and libraries. The data are generated for the plots using the following methodology: each data point is independently computed ten times; the four largest outliers in the data are then discarded and the remaining six points are averaged to produce the data point. Figure 3 shows the results for the lightest weight tasks with only $\sim 2-3$ ns of work per task. Nearly all problems scale poorly in this test, with Cilk+ showing the best performance. Figure 4 presents comparisons using ~ 103 ns per task resulting in a much closer result between the flavors of Cilk+ and Qthreads. Figures 5 and 6 use $\sim 1\mu$s and $\sim 10\mu$s tasks, respectively. We begin to observe scaling for most systems in these medium grain

cases with Qthreads optimal in both the 100 and 1,000 iteration cases and TBB along with HPX closely following the Qthreads performance in the 1,000 iteration case. A peculiar performance drop is observed for OpenMP after 8 cores in Figure 6, which may be attributed to insufficient NUMA support in the OpenMP runtime libraries, coupled with the default affinity policies enforced by the OS. In Figure 7 we explore a medium-heavy grain size with each task taking $\sim 106\mu$s. Here all systems show sustained scaling while the best performance is achieved for Cilk+ and OpenMP using Intel compilers. Many of the remaining runtime systems show very similar performance across the processor counts tested at this granularity.

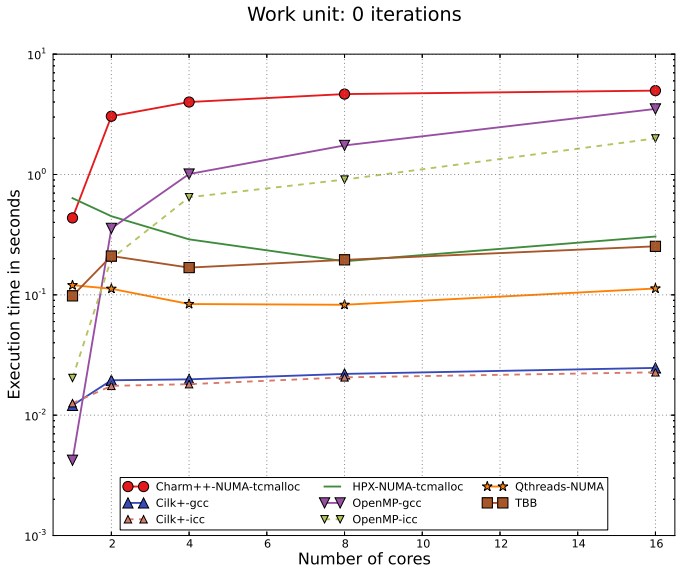

Fig. 3: Homogeneous task spawning with 500,000 tasks where each task has a work unit of approximately 2-3 ns. Poor scaling is generally observed throughout as the task overhead significantly exceeds the amount of work done per task. The two flavors of Cilk+, gcc and icc, significantly outperform the other alternatives in this regime.

In large applications, we do not expect all tasks to have the exact same work unit. This naturally leads us to explore a benchmark where tasks have various sized work units selected from a stable probability distribution.

3.2 Heterogeneous Task Spawn

The heterogeneous task benchmark spawns 5×10^5 tasks just as in the homogeneous task benchmark; however, this time the work unit per task is varied. The code used for heterogeneous tests is almost identical to that of the homogeneous tests (Listing 1- 7), except that each task was assigned a delay parameter (the number of iterations from Listing 1) taken from a probability distribution. The delay parameters in the heterogeneous test were drawn from three stable distributions: the Normal distribution, the CMS distribution, and the Lévy

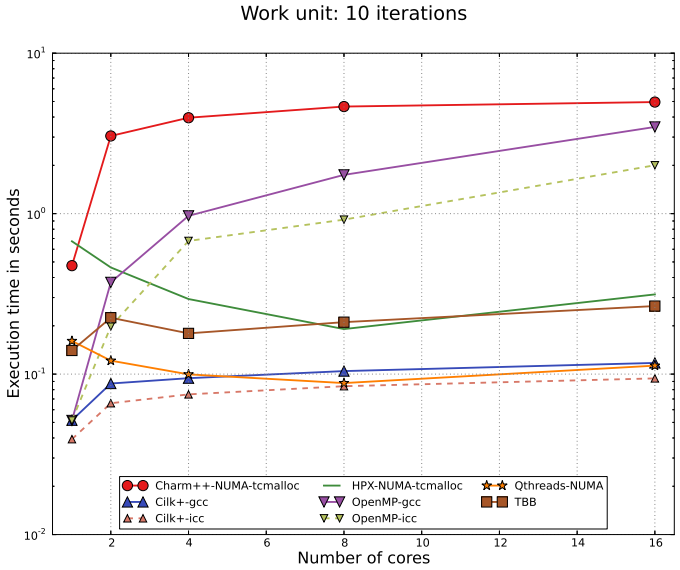

Fig. 4: Homogeneous task spawning with 500,000 tasks where each task has a work unit of ~ 103 ns. The results here share much in common with those of the null tasks in Figure 3 except that the flavors of Cilk+ and Qthreads are much more similar in terms of performance.

distribution. These distributions are stable and are also attractors in probability space making them physically relevant for performance modeling.

The probability density function of a Normal distribution is defined as follows:

$$f(x) = \frac{1}{\sigma\sqrt{2\pi}} \exp\left[-\frac{1}{2}\left(\frac{x-\mu}{\sigma}\right)\right]^2,$$

where μ is the expectation, σ is the standard deviation, and $x \in (-\infty, \infty)$. The standard deviations of $\sigma = 25$ and $\sigma = 75$ were used in the experiments presented in Table 3; an expectation of $\mu = 100$ was selected for the heterogeneous experiments in order to compare with the homogeneous results presented in Figure 5.

The CMS distribution [15] provides much fatter tails than the Normal distribution along with a parameter to control skewness. A generic characteristic function for this distribution is as follows:

$$f(t;\alpha,\beta) = \exp\left[-|t|^\alpha \exp\left(-\frac{1}{2}\pi\beta k(\alpha)\,\mathrm{sign}(t)\right)\right],$$

with $0 < \alpha \leq 2, \alpha \neq 1$, where parameter β controls skewness, function $k(\alpha) = 1 - |1-\alpha|$, α controls the leptokurtosis, and t is the Fourier transform variable. This formula assumes a location parameter of zero and dispersion parameter of unity. Random variables from this distribution are translated to have a location parameter of 100 in order to better compare results with those in Figure 5. The other parameters selected were skewness $\beta = -1$ and $\alpha = 1.0744$ producing a near-Cauchy distribution. These parameters reflect the observation that tasks

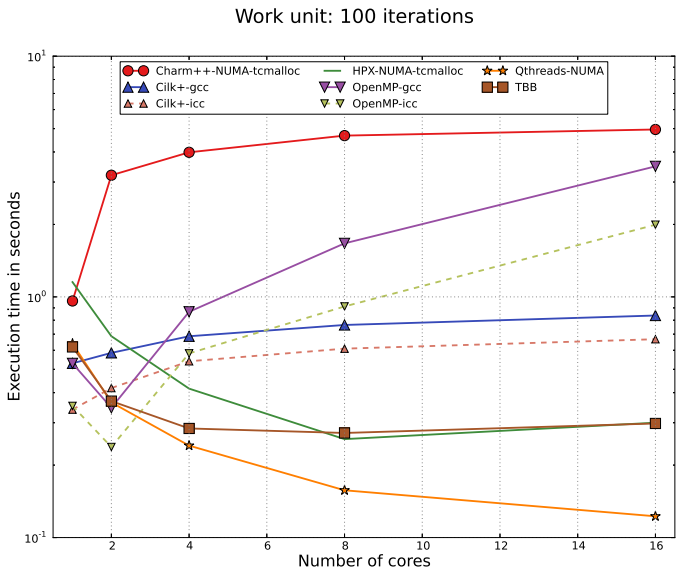

Fig. 5: Homogeneous task spawning with 500,000 tasks where each task has a work unit of $\sim 1\mu$s. The flavors of Cilk+ perform worse at this grain size than they did in Figures 3 and 4. Qthreads substantially outperforms the other packages at this grain size while the flavors of OpenMP still do not show any sustained scaling.

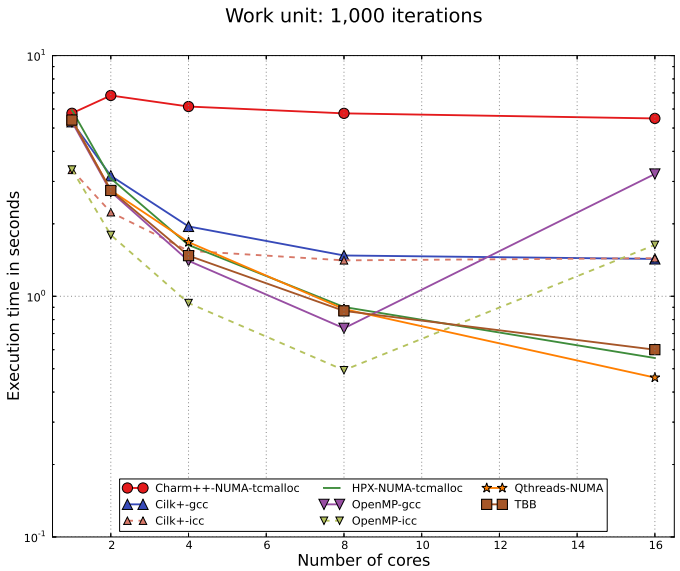

Fig. 6: Homogeneous task spawning with 500,000 tasks where each task has a work unit of $\sim 10\mu$s. OpenMP code generated by the Intel compiler performs best on up to 8 cores after which its performance degrades significantly likely due to insufficient NUMA support in the implementation. Qthreads, HPX, and TBB scale well with the number of cores, showing similar characteristics at this grain size. This similarity increases even further for larger task sizes, as in Figure 7.

on occasion take minimally faster than the average time but often much longer than that; hence a non symmetric fat tail towards longer work units.

The probability density function of the Lévy stable distribution [16] is given by

$$L(z) = \frac{1}{\pi}\int_0^\infty \exp(-\gamma q^\alpha)\cos(qz)dq,$$

with parameters γ and α, and $z \in (-\infty, \infty)$. Like CMS, the location of the extremum in this probability density function was selected to be 100, with $\alpha = 1.5$ and $\gamma = 1$.

A comparison between the homogeneous and heterogeneous measurements is presented in Table 3. Within each probability distribution, each test used the exact same set of random numbers describing the work unit sizes. The largest work unit sizes in the task set for the CMS and Lévy distributions were 3×10^5 and 1×10^6, respectively. In the table, the largest percentage difference in the heterogeneous run from the comparable homogeneous run is presented when run across 1 – 16 cores. The maximum difference in execution time of heterogeneous vs homogeneous task spawn did not exceed 2% for any of the distributions, runtime systems or libraries, or core counts explored. The runtime systems and libraries tested are fairly robust against variations in task work unit size. The results suggest that Cilk and OpenMP may be slightly more sensitive to both Lévy-distributed and CMS task spawn experiments; in case of Normally-distributed delays, HPX, OpenMP, and Cilk were the most sensitive systems found.

RTS	CMS, 3e5	Lévy, 1e6	Normal, 25	Normal, 75
Charm++	0.0456	0.9871	0.0260	0.0404
Cilk+-gcc	0.0358	1.2718	0.0355	-0.1197
Cilk+-icc	0.1104	1.2603	0.1452	0.0805
HPX	0.1214	0.8918	0.1955	0.2526
OpenMP-gcc	0.2523	1.2677	0.2977	0.3510
OpenMP-icc	0.0814	1.2440	-0.4012	-0.2708
Qthreads	0.0260	1.1783	-0.0091	0.0272
TBB	0.0995	1.1943	0.0700	0.0599

Table 3: The largest percentage difference in runtime between the heterogeneous and homogeneous benchmarks is presented here for three different probability distributions and eight runtime systems (RTS) or libraries when run on 1 – 16 cores. Within each probability distribution, each test used the exact same set of random numbers describing the work unit sizes. The largest work unit sizes in the task set for the CMS and Lévy distributions were 3×10^5 and 1×10^6, respectively. The standard deviations used in the Normal distribution ($\sigma = 25$ and $\sigma = 75$)) are given in the Normal column header. The largest difference of heterogeneous vs homogeneous runs across all the runtime systems or libraries, probability distributions, and core counts is lower than 2%. These results suggest that the runtime systems explored here are fairly robust in terms of execution times even when there is a broad mix of both lighter and heavier weight tasks in the task scheduler.

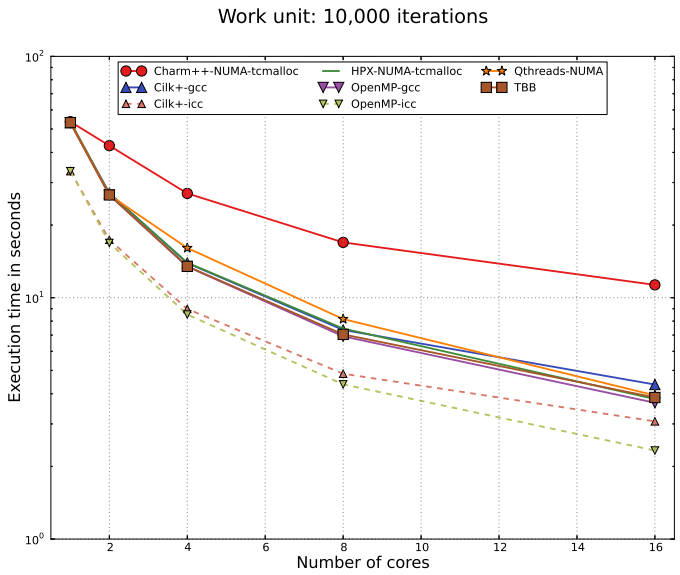

Fig. 7: Homogeneous task spawning with 500,000 tasks where each task has a work unit of $\sim 106\mu$s. The Intel OpenMP and Cilk+ flavors outperform the other runtime systems and libraries at this grain size. The remaining systems, with the exception of Charm++, begin to behave in almost exactly the same way.

4. Conclusions

With the continued dominance of multi-core processors in HPC systems and the emergence of a number of many-tasking capable runtime systems and libraries, some performance comparisons and characterization of overheads are critical in order to better understand and leverage dynamic execution capabilities with balancing overhead costs. These overheads have important implications for determining the grain size at which an application using one of these runtime systems behaves optimally. We have examined a number of many-tasking runtime systems in the context of task initialization, execution and finalization, both for uniform and varying workload sizes in order to better characterize the overheads introduced by the runtime system itself.

In the synthetic benchmarks examined, we found that Cilk plus behaves better at close-to-zero-workload tasks while OpenMP outperforms the other runtime systems for the largest workloads examined. We did not find a significant sensitivity to the fat tails in the heterogeneous task spawn benchmark. The results also revealed a few surprises. The first is that there is no "one size fits all" solution that performs uniformly well for all workload granularities. The second is that despite the old age, some runtime systems still manage to stay competitive to relatively modern implementations in a number of scenarios.

The synthetic benchmarks presented here contain a minimal number of synchronization types compared to what a full application would contain. Such synchronizations would introduce potentially expensive context switches with a different overhead characteristic to each runtime system. Future work in characterizing the runtime system overheads will include context switches and synchronization primitives. Future work will also incorporate these characteristic overheads into a discrete event simulator for application performance modeling.

5. Acknowledgments

The authors wish to thank Hartmut Kaiser, Dylan Stark and Daniel Kogler for their technical assistance.

References

[1] L. Kalé and S. Krishnan, "CHARM++: A Portable Concurrent Object Oriented System Based on C++," in *Proceedings of OOPSLA'93*, A. Paepcke, Ed. ACM Press, September 1993, pp. 91–108.

[2] W. W. Carlson, J. M. Draper, D. E. Culler, K. Yelick, E. Brooks, and K. Warren, "Introduction to UPC and Language Specification," *CCS-TR-99-157*, May 1999.

[3] J. Reinders, *Intel Threading Building Blocks: Outfitting C++ for Multi-Core Processor Parallelism*, 1st ed. O'Reilly Media, July 2007. [Online]. Available: http://www.amazon.com/exec/obidos/redirect?tag=citeulike07-20&path=ASIN/0596514808

[4] C. Dekate, M. Anderson, M. Brodowicz, H. Kaiser, B. Adelstein-Lelbach, and T. Sterling, "Improving the scalability of parallel N-body applications with an event-driven constraint-based execution model," *International Journal of High Performance Computing Applications*, vol. 26, no. 3, pp. 319–332, 2012. [Online]. Available: http://hpc.sagepub.com/content/26/3/319.abstract

[5] "Intel Cilk Plus," http://cilkplus.org/, 2012.

[6] B. Chamberlain, D. Callahan, and H. Zima, "Parallel programmability and the Chapel language," *Int. J. High Perform. Comput. Appl.*, vol. 21, no. 3, pp. 291–312, Aug. 2007. [Online]. Available: http://dx.doi.org/10.1177/1094342007078442

[7] K. Wheeler, R. Murphy, and D. Thain, "Qthreads: An API for Programming with Millions of Lightweight Threads," in *International Parallel and Distributed Processing Symposium. IEEE Press*, 2008.

[8] L. Dagum and R. Menon, "OpenMP: An industry-standard API for shared-memory programming," *IEEE Computational Science and Engineering*, vol. 5, no. 1, pp. 46–55, 1998.

[9] "Tcmalloc: Thread-caching malloc," available from http://goog-perftools.sourceforge.net/doc/tcmalloc.html.

[10] H. Burkhart, M. Christen, M. Rietmann, M. Sathe, and O. Schenk, "Run, Stencil, Run! A Comparison of Modern Parallel Programming Paradigms," in *PARS-Mitteilungen*, 2011.

[11] S. Olivier and J. F. Prins, "Comparison of OpenMP 3.0 and other task parallel frameworks on unbalanced task graphs." *International Journal of Parallel Programming*, vol. 38, no. 5-6, pp. 341–360, 2010. [Online]. Available: http://dblp.uni-trier.de/db/journals/ijpp/ijpp38.html

[12] S. Tham, "Cilk vs MPI: comparing two very different parallel programming styles," in *International Conference on Parallel Processing*, 2003, pp. 143–152.

[13] Stellar group, "HPX GIT repository," 2012, https://github.com/STEllAR-GROUP/hpx.git.

[14] Available from https://code.google.com/p/qthreads/.

[15] J. M. Chambers, C. L. Mallows, and B. W. Stuck, "A method for simulating stable random variables," *Journal of the American Statistical Association*, vol. 71, no. 354, pp. 340–344, 1976. [Online]. Available: http://www.jstor.org/stable/2285309

[16] R. N. Mantegna, "Fast, accurate algorithm for numerical simulation of Lévy stable stochastic processes," *Phys. Rev. E*, vol. 49, no. 5, pp. 4677–4683, May 1994. [Online]. Available: http://dx.doi.org/10.1103/PhysRevE.49.4677