Polytasks: A Compressed Task Representation for HPC Runtimes

Daniel Orozco[1,2], Elkin Garcia[1], Robert Pavel[1], Rishi Khan[2] and Guang Gao[1]
orozco@udel.edu, egarcia@udel.edu, rspavel@udel.edu,
rishi@etinternational.com, ggao@capsl.udel.edu

[1]University of Delaware
[2]ET International

Abstract. The increased number of execution units in many-core processors is driving numerous paradigm changes in parallel systems. Previous techniques that focused solely upon obtaining correct results are being rendered obsolete unless they can also provide results *efficiently*.
This paper dives into the particular problem of efficiently supporting fine-grained task creation and task termination for runtime systems in shared memory processors.
Our contributions are inspired by our observation of High Performance Computing (HPC) programs, where it is common for a large number of similar fine-grained tasks to become enabled at the same time.
We present evidence showing that task creation, assignment of tasks to processors, and task termination represent a significant overhead when executing fine-grained applications in many-core processors.
We introduce the concept of the *polytask*, wherein the similarity of tasks created at the same time is exploited to allow faster task creation, assignment and termination. The polytask technique can be applied to any runtime system where tasks are managed through queues.
The main contributions of this work are:

1. The observation that task management may generate substantial overhead in fine-grained parallel programs for many core processors.
2. The introduction of the polytask concept: A data structure that can be added to queue-centric scheduling systems to represent groups of similar tasks.
3. Experimental evidence showing that the polytask is an effective way to implement fine-grained task creation/termination primitives for parallel runtime systems in many-core processors.

We use microbenchmarks to show that queues modified to handle polytasks perform orders of magnitude faster than traditional queues in some scenarios. Furthermore, we use microbenchmarks to measure the amount of time spent executing tasks. We show situations where fine-grained programs using polytasks are able to achieve efficiencies close to 100% while their efficiency becomes only 20% when not using polytasks. Finally, we use several applications with fine granularity to show that the use of polytasks results in average speedups from 1.4X to 100X depending on the queue implementation used.

This research was, in part, funded by the U.S. Government. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the U.S. Government.

1 Introduction

The development of processor chip architectures containing hundreds of execution units has unleashed challenges that span from efficient development to efficient execution of applications.

The idea of partitioning computations into *tasks* (or equivalent concepts) has been used by a number of execution paradigms such as pthreads[3], OpenMP [7], Cilk[2] and others to address the efficiency of execution. Our techniques, results and conclusions focus on the aspect of viewing tasks as units of computation, and they are orthogonal to implementation of tasks in particular execution paradigms.

Execution of tasks varies from paradigm to paradigm, and generally includes the use of queues to produce and consume (or execute) tasks as the program progresses. It is a common occurrence in previous research to assume that tasks that become enabled can be executed immediately, overlooking the fact that enqueueing a task to make it available to other threads takes a nonzero time. This assumption is reasonable in systems with few processors or coarse task granularity because the time taken to enqueue a task is negligible compared to the time taken to execute it. However, the enqueueing process can become a significant source of overhead for systems where a large number of processors participate in fine-grained execution.

A simple observation can be used to illustrate the problem: A system that uses a queue (centralized or distributed) for task management where there are P idle processors requires at least P tasks to be enqueued to allow execution in all processors. Even if enough tasks are available, the time to enqueue them becomes relevant as P grows. Advanced algorithms based on distributed structures or trees softens this problem by providing faster primitives due to lower contention, but ultimately they do not intrinsically reduce the total number of queue operations needed.

The overhead of assigning tasks in systems with many processors can be solved by leveraging on a simple observation: *Tasks that become enabled at the same time are frequently very similar* because, in many cases, all tasks direct processors to execute the same instructions, using the same shared data, and only a few parameters and local variables differentiate tasks from one another.

Our contributions are inspired by the idea that similar tasks can be efficiently *compressed* and represented as a single task, that we refer to as a *polytask*. We address the specific case in which tasks can be written in such a way that only a single integer number can be used to retrieve their task-specific data. Many programs have that property: iterations of parallel loops are differentiated by their iteration index, threads in fork-join applications can be differentiated by their thread identifier and so on.

We show that most queue primitives can support compression of tasks if a small data structure containing two integers is added to the task description. Using the added integers, we show that it is possible to efficiently perform task management on compressed tasks.

The effectiveness of compressing tasks into polytasks is shown in experiments where three traditional queueing techniques are modified to allow compression. Our results show that compression of tasks, when possible, allows much faster queue primitives than their noncompressed counterparts because (1) compressed tasks represent several tasks, making a queue operation on them equivalent to several queue operations on noncompressed tasks and (2) queue operations on compressed tasks are frequently faster than queue operations on regular tasks. We show that queues modified to handle polytasks perform at par with their non-modified versions when compression is not possible.

Our contributions are relevant to queue-centric runtime systems whether they have one queue or many. Other runtime systems that are not necessarily queue centric, such as OpenMP [7] have developed alternatives that are very similar in their implementation and their objective. Our contribution is the presentation of a systematic way to modify queue-centric systems to address task compression.

The advantages of task compression in larger applications are shown in Section 5 using microbenchmarks and applications. All cases present evidence supporting our claims regarding the effectiveness of task compression.

The rest of the paper is organized as follows: Section 2 presents relevant background, Section 3 presents some motivation as well as the specific definition of the problem addressed in this paper, Section 4 represents the core of our paper, presenting our technique for task compression, Section 5 describes our experiments and results, Section 6 presents related work and Section 7 presents conclusions and future work.

2 Background

This section presents background related to our contributions, including a brief summary of the queue algorithms referred throughout the paper, and a description of the processor used for our experiments.

2.1 Queue Algorithms

The central work of this paper tries to enhance the capabilities of existing queue algorithms for the particular case of task management. A large number of queue algorithms exist, for example, Shafiei [16] shows a survey summarizing the relative advantages of many different algorithms. We have chosen three queue algorithms that cover a significant portion of the design space.

The first algorithm, which we refer to as the SpinQueue algorithm, uses a linked list as the basic data structure for the queue and a spinlock to avoid data races on processors accessing the queue. The SpinQueue is a simple implementation, that is easy to understand, easy to program, and that offers excellent performance if there is low contention at the queue. The simple implementation of the SpinQueue makes it suitable for quick development of parallel applications.

The MS-Queue is an advanced non-blocking algorithm, presented by Michael and Scott [12], that uses a Compare and Swap operation to allow concurrent

operations on the queue. The MS-Queue algorithm has become an industry *de facto* standard, being used in the Java Concurrency Constructs, and in many other high-profile implementations.

The MC-Queue is a queue algorithm presented by Mellor-Crummey[11] that distributes queue operations over a group of nonblocking queues to maximize performance. In the MC-Queue, the queue structure is composed of several independent nonblocking queue implementations. When queue operations are requested, an atomic addition is used to select one of the available nonblocking queues, effectively distributing the operations across them. The MC-Queue is an excellent choice for applications where a large number processors attempt to execute operations concurrently.

2.2 Cyclops-64 architecture

Cyclops-64 (C64) is a processor developed by IBM. The architecture and features of C64 have been described extensively in previous publications [6, 5].

Each C64 chip has 80 computational cores, no automatic data cache, and 1GB of addressable memory. Each core contains two single-issue thread units, 60KB of user-addressable memory that is divided into stack space and shared memory, a 64 bit floating point unit, and one memory controller.

One of the main features of the C64 chip is that *memory controllers can execute atomic operations*. In C64, each memory controller contains its own Arithmetic and Logical Unit that allows the memory controller to execute integer and logical atomic operations *in memory* without the intervention of a processor or a thread unit. Atomic operations in C64 take 3 cycles to complete at the memory controller. All memory controllers in C64 have the capability to execute atomic operations.

Under the default configuration, C64 has 16KB of stack space for each thread unit, 2.5MB of shared on-chip memory, and 1GB of DRAM memory.

3 Motivation

The difficulties in traditional task management as well as the possibilities of task compression can be illustrated using the kernel of a simulation of an electromagnetic wave propagating using the Finite Difference Time Domain algorithm in 1 Dimension (FDTD1D), shown in Figure 1.

The parallel loops in Figure 1 can be efficiently executed in a many-core processor such as C64 if the iterations in the parallel loops are expressed as tasks. The granularity of the execution can be varied through the tile size (`TileSize` in Figure 1). A small tile size will result in finer grain and more parallelism, but it will also incur a higher runtime system overhead because of the additional burden in task management.

The main problem is that *fine-grained* execution is difficult: The granularity of tasks is limited by the overhead of task management. In general, fine grain execution is useful only when the overhead associated with the execution

```
// TileSize controls the granularity
#define TileSize 16
void FDTD1D( double *E, double *H, int N,
  int Timesteps,
  const double k1, const double k2 )
{
  int i, t;
  for ( t=0;t<Timesteps;t++ )
  {
    parallel for (i=1; i<N/TileSize; i++)
    {
      E_Tile( i, E, H, N, k1, k2 );
    }

    parallel for (i=1; i<N/TileSize; i++)
    {
      H_Tile( i, E, H, N, k1, k2 );
    }
  }
}
```

Fig. 1. FDTD1D Kernel

```
void E_Tile( int TileID,
  double *E, double *H, int N,
  int Timesteps,
  const double k1, const double k2 ) {
  int i, Start, End;
  Start = TileID * TileSize;
  End = Start + TileSize;
  for ( i = Start; i < End; i++ )
    E[i]=k1*E[i]+k2*(H[i]-H[i-1]);
}

void H_Tile( int TileID,
  double *E, double *H, int N,
  int Timesteps,
  const double k1, const double k2 ) {
  int i, Start, End;
  Start = TileID * TileSize;
  End = Start + TileSize;
  for ( i = Start; i < End; i++ )
    H[i]+=E[i]-E[i+1];
}
```

Fig. 2. FDTD1D Compute Tiles

is acceptable. In contrast, coarse-grained executions decrease the proportional overhead of task management at the cost of reducing parallelism and reducing the opportunities for load balancing in many-core systems [9].

Traces of several executions of FDTD1D, executed using the TIDeFlow runtime system [13] with the SpinQueue algorithm in C64 were obtained to show the limits in granularity:

Process	**Cycles**	**Process**	**Cycles**
Enqueue one task	6200	Enqueue a task for each processor	9.9×10^5
Execute a tile of size 1	180	Execute a tile of size 256	16000
Execute a tile of size 1024	56000	Execute a tile of size 16384	9.0×10^5
Execute a tile of size 65536	3.6×10^6		

Table 1. Task duration and enqueueing overhead for C64 (SpinQueue Algorithm)

Table 1 exposes the problem faced by programs with fine granularity: If the tile size is set to 1, approximately 10^6 cycles will be required to enqueue one task for each one of the 160 processors in C64 while a single processor executes one task in only 180 cycles. The conclusion is that programs where tiles take less than 10^6 cycles to execute result in poor performance because processors will consume tasks faster than they are written to the queue.

The solution that we pursue in this paper is based on the observation that in many programs, tasks are very similar, and they may be compressed into a single task to reduce the total time for task creation. Task compression opens a number of questions that we intend to address.

3.1 Problem Formulation

The following question summarizes our research goals:

> How is it possible to exploit the commonly found similarities between tasks created at the same time to achieve efficient representation of tasks?

The main question opens related questions:

- How can several, similar tasks be efficiently compressed into a polytask?
- Is it possible to concurrently and efficiently extract a task from a polytask?
- Is it possible to efficiently support termination operations such as *join* when using compressed task representations such as polytasks?
- Does task compression introduce additional overheads in applications where tasks are dissimilar making task compression unnecessary?

We intend to fully answer these questions in the following sections. We also provide experimental results to back our claims regarding the usability of polytasks.

4 Polytasks: efficient building blocks for runtime systems.

Creation of similar tasks at the same time is common in scientific programs using execution models that support parallel execution of loops. Figure 1 shows one such example where a parallel loop results in creation of a large number of similar tasks, that execute the same instructions, that access the same global variables and that are distinguished only by their loop index, or rather, by their *execution instance*.

The main idea in our proposed solution is to represent all the tasks related to a parallel loop with a single data structure that we call a polytask. A polytask is a data structure that includes all the information commonly found in a task plus additional information describing the number of tasks it represents (N, the number of iterations in the parallel loop) and their state.

```
typedef struct task_s {
  // Task Information:
  // Environment information
  // Code to be executed
  ...
} task_t
```

Fig. 3. Original task data structure

```
typedef struct polytask_s {
  int TasksAvailable;
  int TasksPending;

  // Task Information:
  // Environment information
  // Code to be executed
  ...
} polytask_t
```

Fig. 4. Polytask data structure

A generic structure that can be used to represent a single, particular task is presented in Figure 3. The structure of Figure 3 has been upgraded (Figure

4) to allow representation of several, similar tasks as a polytask. The polytask structure contains a counter that specifies the number of tasks that are available in the polytask (`TasksAvailable`) and a counter containing the number of tasks that have not finished (`TasksPending`). The `TasksPending` counter facilitates implicit join operations at the end of parallel loops. At initialization, both counters are initialized to N to indicate that N tasks are available and that none of them have finished execution.

Note that any queue algorithm can be used to implement polytasks operations: A polytask can be enqueued into the work queue in the same way as a single task can be enqueued. The algorithm in Figure 6 shows how to upgrade a generic runtime system to support polytasks by introducing three operations for task management: `PolyEnqueue` is used to create N tasks of a particular type. `PolyDequeue` is used to extract the next available task from the queue and `TaskCompleted` is used to count the number of tasks that have finished execution.

A significant advantage of polytasks over individual tasks is that a polytask is enqueued *once* into the work queue. `PolyEnqueue`, when called, initializes the information in the task structure to specify that N tasks must be created and N tasks must complete. `PolyEnqueue` enqueues a single queue item containing the polytask to the queue. In general, only minimum modifications to the data structures and to the original enqueueing algorithm are required to support polytasks.

Extracting individual tasks from a polytask is more challenging because an unmodified dequeue operation will remove the polytask from the queue. Instead, `PolyDequeue` extracts a single task from the polytask at the head of the queue using an atomic decrement on its `TasksAvailable` counter. The value returned by the atomic decrement is used as the *execution instance* for the execution of the task (e.g. iteration index of a parallel loop) if it is positive. It is possible to obtain invalid (non-positive) execution instances during the extraction process because the atomic decrement is a concurrent process. This is not a problem since polytasks are quickly removed from the queue after all their available tasks have been claimed, effectively presenting a new polytask at the head of the queue. Processors that did not obtain valid execution instances during their fist attempt can retry until the polytask at the head of the queue contains enough available tasks.

Execution instances are assigned to processors starting from N and going down to 1. When a processor extracts the last task from the polytask, (*i.e.* when the execution instance obtained is 1), the processor dequeues the polytask from the queue using the original queue algorithm selected.

`TaskCompleted` implements *join* behavior for tasks in a particular polytask. The `TasksPending` counter in the polytask structure can be used as a synchronization point to reliably know the number of tasks that have not finished execution in a particular polytask. Processors atomically decrement the `TasksPending` counter to indicate termination of individual tasks in the polytask. A processor can know if it executed the last task in the polytask (*i.e.* the join operation

```
/* ---  Queue Variables --- */
typedef struct QueueItem_s {
  polytask_t PolyTask;
  // Other Queue-Specific Members
  ...
}
QueueItem_t;

QueueItem_t  *Head, *Tail;

/* --- PolyTask Functions --- */
task_t * PolyDequeue( void ) {
StartDequeue:
  polytask_t PolyTask = Head->PolyTask;
  int ExecutionInstance =
    Atomic_Decrement(
      &( PolyTask->TasksAvailable ) );

  if ( ExecutionInstance == 1 ) {
    // Removes Task from Queue
    Dequeue();
  }

  if ( ExecutionInstance > 0 ) {
    return( PolyTask );
  }

  goto StartDequeue;
}
```

Fig. 5. Polytask Dequeue

```
void PolyEnqueue( PolyTask_t *PolyTask,
      int N )
{
  PolyTask.TasksAvailable = N;
  PolyTask.TasksPending = N;
  Enqueue( PolyTask );
}

void TaskCompleted( PolyTask_t *PolyTask )
{
  int PendingTasks =
    Atomic_Decrement(
      &( PolyTask->TasksPending ) );

  if ( PendingTasks == 1 ) {
    // This is the last task
    // Join operation successful
    ....
  }
}
```

Fig. 6. Polytask Enqueue and Complete Task operations

is complete) by inspecting the return value of the atomic decrement. Figure 6 shows the structure of the `TaskCompleted` function and how it can be used to handle the behavior of join operations by the runtime system.

Polytask operations are faster than traditional task operations on queues. When task compression is possible ($N > 1$), they provide flexible mechanisms for synchronization and continuation of tasks. The speed up will be significant, especially when fine granularity is used and the number of processors increase. The synchronization between tasks of the same type (*i.e.* in a polytask) is easier and allows smooth support for control flow mechanisms such as creation, termination and continuation found in parallel loops. The reasons for faster operations and easier synchronization and continuation are:

- **Task Creation:** N tasks can be written with a single queue operation.
- **Task Assignment:** In most cases, the concurrent part of the algorithm that assigns a single task to a processor is a single atomic decrement, that can be executed in parallel by all processors with very little contention. This is an improvement over a single task-based approach where the algorithm to access the queue is based on locks and it could be better even on queues using Compare-and-Swap operations. However, on C64, atomic decrements are faster than compare and swap operations because atomic decrements can be executed directly *in memory* by the memory controller (see section 2.2).
- **Join and Continuation Operations:** The use of a single counter (`TasksPending`) modified through atomic operations allows fast task syn-

chronization during the termination phase of parallel loops. The decentralized nature of the count, where any processor may be the last one, reduces the overhead of other implementations where a particular, centralized process is responsible for the termination and continuation of the parallel loop.

Section 5 uses microbenchmarks and some applications to show the advantages of polytasks over traditional queue techniques.

5 Experiments

The effectiveness of compressing tasks into polytasks is analyzed in this section. We have designed experiments that show the effect of task compression in an isolated scenario –using microbenchmarks– and in a full production runtime system –using full applications–. Our results show that applications with many similar tasks greatly benefit from the use of polytasks, without adversely affecting applications without good task similarity.

We have chosen C64, a many-core processor architecture (Section 2.2) as the testbed architecture because it is a logical choice to support task-based runtime systems due to its large number of processors in a shared memory environment and its non-preemptive execution model.

All of our experiments where written in C and they were compiled with ET International's C64 C compiler, version 4.3.2, with compilation flags -O3. We ran all of our experiments using FAST[5], a highly accurate C64 simulator.

5.1 Microbenchmarks

In our first study, we analyze the advantages and disadvantages of polytasks in a controlled environment that attempts to show the behavior of the required queue primitive operations without external perturbations.

In our first set of experiments, we isolate the behavior of queue operations by running programs where all processors in a C64 chip continuously produce and consume randomly generated tasks without executing them. In these experiments, each processor produces 512 tasks and consumes 512 tasks. Embedded hardware performance counters are used to collect timing data. To illustrate the effectiveness of polytasks, the number of tasks that are similar in each experiment was modified. A **task similarity** of N (See Figures 7 and 8) indicates that groups of N tasks are similar and can be potentially compressed into a single polytask. Our experiments show results that range from task similarity of 1, where each task is unique, to task similarity of 256, where groups of 256 tasks are compressed into a single polytask. When the results are reported, enqueueing one polytask that contains N tasks is considered equivalent to enqueueing N tasks directly because a runtime system will observe, in both cases, that N tasks have been created.

We have explored the sensitivity of polytasks to the underlying queueing algorithm used. From the many queue algorithms that exist (Shafiei [16] has

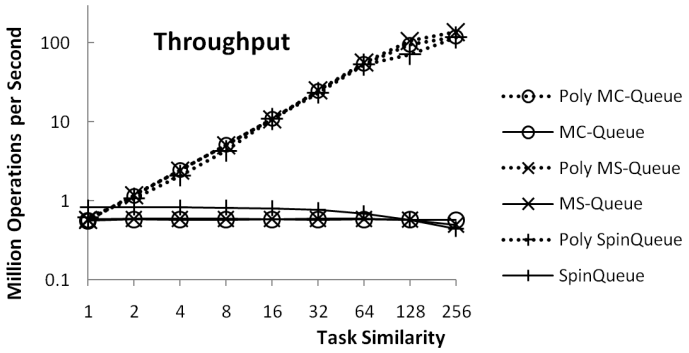


Fig. 7. Effect of polytask compression on throughput

compiled an excellent summary) we have chosen three algorithms that cover a significant portion of the design space, a spinlock-based queue (SpinQueue, the simplest to implement), the MS-Queue algorithm[12] (the most famous non-blocking algorithm) and the MC-Queue algorithm[11] (a distributed queue with very high performance). We implemented polytasks on each one of the selected queue algorithms in an effort to present quality over quantity.

Figures 7 and 8 show the results of our experiments. Inspection of the figures allows us to reach important conclusions:

- Polytasks increase the performance of runtime operations, both in total (aggregated) operations per second in the whole system as well as in terms of reducing the latency of individual operations.
- The overhead of polytask compression is very small. Both in terms of latency and throughput there is not a significant overhead (less than 2% in all but one case) when polytasks are used in situations where tasks are not similar.
- The advantages of polytasks are not dependent on the queue algorithm used to implement the polytask operations: All queue algorithms tested present excellent performance gains, and very low overhead.
- When task similarity is high, the average performance of task operations is improved by up to two orders of magnitude. The reason for the increase in performance is that calls to polytask operations that do not result in calls to queue operations finish quickly.

The results shown in Figures 7 and 8 show that polytasks can benefit applications where task compression is possible without degrading the performance of applications where task compression is not possible.

The second set of experiments was designed to show the advantages of the use of polytasks in applications with varying levels of granularity. To avoid external perturbations, synthetic tasks that execute a delay loop of varying duration were used.

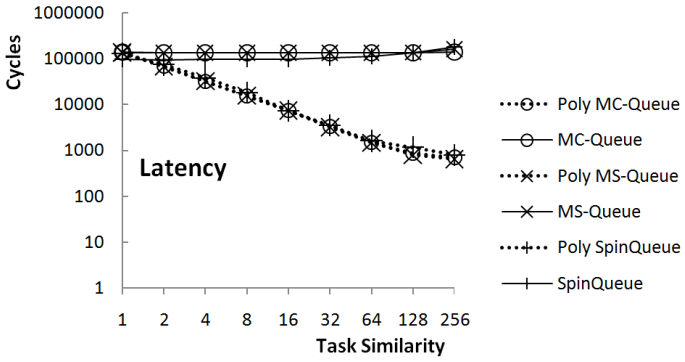


Fig. 8. Effect of polytask compression on latency

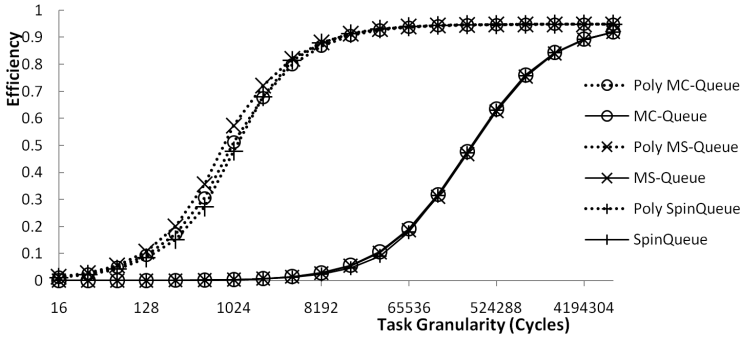


Fig. 9. Advantages of polytasks as a function of program granularity

Figure 9 shows the results of our experiments. In the figure, the efficiency is the fraction of time that the processor spends executing tasks, and it is calculated as the ratio between the time executing tasks and the total execution time, including overheads. The Task Granularity is the duration of the tasks executed. Each data point reported in the figure was obtained by running 40960 tasks that execute a delay loop whose duration is specified in the figure as Task Granularity.

Our results show that fine grained applications greatly benefit from the use of polytasks. Polytasks enable greater runtime system efficiencies at very fine grain synchronization while traditional approaches only allow coarser grain synchronization.

As is to be expected, if the application uses very coarse grained parallelism the burden of task management does not affect the efficiency of the system because the time to perform a queue operation will not be significant when compared to the execution time of a task.

The results of Figure 9 show that the advantages of polytasks under varying task granularity remain, independent of the queue algorithm used. In our experiments, we observe that polytasks provide significant advantages in efficiency for all three of the queue algorithms used (SpinQueue, MC-Queue and MS-Queue) in fine-grained environments.

5.2 Applications

The advantages of polytasks in production systems was tested using scientific applications running in typical environments.

Several applications were tested: Fast Fourier Transforms that use the Cooley-Tukey algorithm with two-point butterflies (FFT) and simulations of electromagnetic waves propagating using the Finite Difference Time Domain algorithm in 1 Dimension (FDTD1D)[15] and 2 dimensions (FDTD2D)[14]. FFT was run with input sizes 2^9 (FFT2P 2^9) through 2^{12} (FFT2P 2^{12}), FDTD1D runs a problem of size 20000 with 3 timesteps and tiles of width 16, and FDTD2D runs a problem of size 128 by 128 with 2 timesteps and tiles of width 4 by 4.

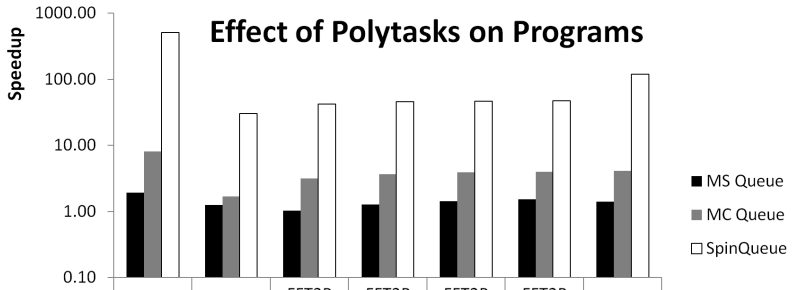
The results reported for all applications reflect the complete program and include memory allocation, initialization, computation and reporting of results.

In all cases, the programs were developed to use the TIDeFlow [13] runtime system. TIDeFlow uses a priority queue to assign work to processors in the system.

We compared the effect of polytasks by running several sets of experiments: The first set of experiments consists of pairing each one of the programs with each one of the versions of the TIDeFlow runtime system that use each one of the possible underlying queue algorithms described (SpinQueue, MC-Queue and MS-Queue) without the advantage of polytasks. For the second set of experiments, we ran all combinations of programs and TIDeFlow implementations using polytasks.

Figure 10 reports the speedup of each program with reference to an execution using the unmodified runtime system (Single Task). The objective of showing a comparison of polytasks against single tasks for each one of several underlying queue algorithms is to show that the advantages of polytasks are not exclusive to a particular queue algorithm. Such advantages are primarily the result of a runtime that operates more efficiently.

The reasons for the excellent speedup in the programs tested are: (1) The applications use very fine grain synchronization and a significant portion of the time is spent in task management. As explained in Section 5.1 and in Figure 9, these results hold for fine-grained applications. The impact of polytasks on other coarse-grained applications not considered here may not be as pronounced as the impact in the fine-grained applications that we tested. (2) The applications exhibit a large degree of task similarity because their main computational kernel consists mainly of `parallel for` loops.



	FDTD1D	FDTD2D	FFT2P 2^9	FFT2P 2^10	FFT2P 2^11	FFT2P 2^12	Average
MS Queue	1.92	1.26	1.03	1.27	1.44	1.53	1.41
MC Queue	8.12	1.69	3.18	3.68	3.93	4.00	4.10
SpinQueue	510.9	30.5	42.6	45.7	46.8	47.2	120.6

Fig. 10. Advantages of polytasks on applications

6 Related Work

Several execution paradigms based in tasks have been presented in the past.

Intel's Concurrent Collections (CnC)[10] is an execution paradigm where programs are expressed in terms of computational tasks (or steps in the CnC terminology), data dependencies and control dependencies. The current generation of CnC implementations represent tasks as individual items in the work pool. However, polytasks could be a promising addition to CnC because it is frequent that several computation steps are similar in everything except in their control tag value, making the addition of polytasks a natural extension to allow higher scalability and performance.

The University of Delaware's Codelet Model[8], part of the ongoing DARPA UHPC project, is an initiative to achieve unprecedented parallelism in programs by expressing computations as dataflow graphs composed of codelets (computational tasks) that can migrate across large systems. Polytasks can potentially impact the effectiveness of the codelet model because it may help migration of tasks to remote locations if several tasks are compressed to a single polytask.

Other execution paradigms that use tasks in a way or another, and queues to manage tasks include X10[4], EARTH[17], Cilk[2] and Habanero C[1]. Polytasks offer interesting opportunities for those execution paradigms, and may potentially be used to improve their performance and scalability.

7 Conclusions and Future Work

We have shown that polytasks are an effective way to exploit the similarity between tasks that is commonly found in scientific programs that use a queue-centric approach for execution. The polytask technique allows queue-centric runtimes to exploit the same parallel loops as the OpenMP *dynamic* construct. Our

research focus on how to develop a systematic technique for task compression rather than addressing particular situations in particular systems. Future work will compare the differences in performance between OpenMP's *dynamic* construct and polytasks.

We have presented a line of thought that concludes that there is a high degree of similarity in tasks that are enabled at the same time in scientific programs, mostly resulting from parallel loops that are expressed as a set of embarrassingly parallel tasks that become enabled.

We have taken advantage of the similarity of tasks and their proximity in time to invent a way to express them in a compressed form, that we call a polytask. We have shown that the data structures and algorithms of runtime systems require only minor modifications to support polytasks.

We have provided evidence, both in our informal analysis and in our experiments of the usability of polytasks for runtime systems.

The polytasks improve the performance of the runtime system when the programs run exhibit high task similarity. In cases where task compression is not possible, polytasks do not introduce significant overhead, and can be used safely.

The effect of polytasks on the speed of the runtime system can only be noticed in applications with fine granularity. In applications where parallelism has been exposed at a coarse-grain level, the issue of task management overhead is not as relevant because most of the application time is spent executing tasks. Nevertheless, if significant parallelism is required in future generations of multiprocessors with a large number of processing units per chip, fine grain parallelism will become a necessity.

We have shown that polytasks are effective for C64, a system that is non-preemptive, that has no cache and that supports atomic operations in-memory. Polytasks are effective for C64 because processors have very little overhead when they start executing a task: There is no cache that needs to be filled, there is no thread-state that needs to be put in place and there is no virtual memory that needs to be made available.

Although future work may analyze the usability of polytasks for preemptive systems, with few cores, caches, and other architectural features, it is the impression of the authors that queue-based runtime systems will primarily benefit systems with massive hardware parallelism where the architecture is directly exposed to the users.

Future work will focus on extending the polytask concept beyond the implementation of runtime systems, including the development of language or compiler extensions to indicate or hint task similarity outside of the trivial case of parallel loops.

References

1. Barik, R., Budimlic, Z., Cave, V., Chatterjee, S., Guo, Y., Peixotto, D., Raman, R., Shirako, J., Tasirlar, S., Yan, Y., Zhao, Y., Sarkar, V.: The habanero multicore

software research project. In: Proceeding of the 24th ACM SIGPLAN conference companion on Object oriented programming systems languages and applications. pp. 735–736. OOPSLA '09, ACM, New York, NY, USA (2009)
2. Blumofe, R.D., Joerg, C.F., Kuszmaul, B.C., Leiserson, C.E., Randall, K.H., Zhou, Y.: Cilk: an efficient multithreaded runtime system. In: Proceedings of the fifth ACM SIGPLAN symposium on Principles and practice of parallel programming. pp. 207–216. PPOPP '95, ACM, New York, NY, USA (1995)
3. Butenhof, D.R.: Programming with POSIX threads. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA (1997)
4. Charles, P., Grothoff, C., Saraswat, V., Donawa, C., Kielstra, A., Ebcioglu, K., von Praun, C., Sarkar, V.: X10: an object-oriented approach to non-uniform cluster computing. SIGPLAN Not. 40, 519–538 (October 2005)
5. del Cuvillo, J., Zhu, W., Hu, Z., Gao, G.: Fast: A functionally accurate simulation toolset for the cyclops-64 cellular architecture. CAPSL Technical Memo 062 (2005)
6. del Cuvillo, J., Zhu, W., Hu, Z., Gao, G.R.: Toward a software infrastructure for the cyclops-64 cellular architecture. In: High-Performance Computing in an Advanced Collaborative Environment, 2006. p. 9 (May 2006)
7. Dagum, L., Menon, R.: Openmp: an industry standard api for shared-memory programming. Computational Science Engineering, IEEE 5(1), 46 –55 (jan-mar 1998)
8. Gao, G., Suetterlein, J., Zuckerman, S.: Toward an execution model for extreme-scale systems -runnemede and beyond. CAPSL Technical Memo 104
9. Garcia, E., Venetis, I.E., Khan, R., Gao, G.: Optimized Dense Matrix Multiplication on a Many-Core Architecture. In: Proceedings of the Sixteenth International Conference on Parallel Computing (Euro-Par 2010), Part II. Lecture Notes in Computer Science, vol. 6272, pp. 316–327. Springer, Ischia, Italy (2010)
10. Knobe, K.: Ease of use with concurrent collections (cnc). In: Proceedings of the First USENIX conference on Hot topics in parallelism. pp. 17–17. HotPar'09, USENIX Association, Berkeley, CA, USA (2009)
11. Mellor-Crummey, J.: Concurrent queues: Practical fetch and phi algorithms. Tech. Rep. 229, Dep. of CS, University of Rochester (1987)
12. Michael, M.M., Scott, M.L.: Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. In: Proc. of the 15th ACM symposium on Principles of distributed computing. pp. 267–275. PODC '96, ACM, New York, NY, USA (1996)
13. Orozco, D., Garcia, E., Pavel, R., Khan, R., Gao, G.: Tideflow: The time iterated dependency flow execution model. CAPSL Technical Memo 107 (2011)
14. Orozco, D., Garcia, E., Gao, G.: Locality optimization of stencil applications using data dependency graphs. In: Proceedings of the 23rd international conference on Languages and compilers for parallel computing. pp. 77–91. LCPC'10, Springer-Verlag, Berlin, Heidelberg (2011)
15. Orozco, D.A., Gao, G.R.: Mapping the fdtd application to many-core chip architectures. In: Proceedings of the 2009 International Conference on Parallel Processing. pp. 309–316. ICPP '09, IEEE Computer Society, Washington, DC, USA (2009)
16. Shafiei, N.: Non-blocking array-based algorithms for stacks and queues. In: Proceedings of the 10th International Conference on Distributed Computing and Networking. pp. 55–66. ICDCN '09, Springer-Verlag, Berlin, Heidelberg (2009)
17. Theobald, K.: EARTH: An Efficient Architecture for Running Threads. Ph.D. thesis (1999)