A Framework for Space and Time Efficient Scheduling of Parallelism

Girija J. Narlikar Guy E. Blelloch

December 1996
CMU-CS-96-197

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Abstract

Many of today's high level parallel languages support dynamic, fine-grained parallelism. These languages allow the user to expose all the parallelism in the program, which is typically of a much higher degree than the number of processors. Hence an efficient scheduling algorithm is required to assign computations to processors at runtime. Besides having low overheads and good load balancing, it is important for the scheduling algorithm to minimize the space usage of the parallel program. In this paper, we first present a general framework to model non-preemptive parallel computations based on task graphs, in which schedules of the graphs represent executions of the computations. We then prove bounds on the space and time requirements of certain classes of schedules that can be generated by an offline scheduler. Next, we present an online scheduling algorithm that is provably space-efficient and time-efficient for multithreaded computations with nested parallelism. If a serial execution requires S_1 units of memory for a computation of depth D and work W, our algorithm results in an execution on p processors that requires $S_1 + O(pD \log p)$ units of memory, and $O(W/p + D \log p)$ time, including scheduling overheads. Finally, we demonstrate that our scheduling algorithm is efficient in practice. We have implemented a runtime system that uses our algorithm to schedule parallel threads. The results of executing parallel programs on this system show that our scheduling algorithm significantly reduces memory usage compared to previous techniques, without compromising performance.

This research is sponsored by the Wright Laboratory, Aeronautical Systems Center, Air Force Materiel Command, USAF, and the Advanced Research Projects Agency (ARPA) under grant F33615-93-1-1330. The U.S. Government is authorized to reproduce and distribute reprints for Government purposes, notwithstanding any copyright notation thereon. Views and conclusions contained in this document are those of the author and should not be interpreted as representing the official policies, either expressed or implied, of ARPA or the U.S. Government.

Keywords: Space efficiency, dynamic scheduling, nested parallelism, parallel language implementation.

1 Introduction

Many of today's high level parallel programming languages provide constructs to express dynamic, fine-grained parallelism. Such languages include data-parallel languages such as NESL [3] and HPF [21], as well as control-parallel languages such as ID [1], Cilk [5], CC++ [12], Sisal [17], and Proteus [26]. These languages allow the user to expose all the parallelism in the program, which is typically of a much higher degree than the number of processors. The language implementation is responsible for scheduling this parallelism onto the processors. If the scheduling is done at runtime, then the performance of the high-level code relies heavily on the scheduling algorithm, which should have low scheduling overheads and good load balancing.

Several systems providing dynamic parallelism have been implemented with efficient runtime schedulers [7, 11, 13, 18, 19, 22, 23, 29, 30, 31], resulting in good parallel performance. However, in addition to good time performance, the memory requirements of the parallel computation must be taken into consideration. In an attempt to expose a sufficient degree of parallelism to keep all processors busy, schedulers often create many more parallel threads than necessary, leading to excessive memory usage [15, 28, 32]. Further, the order in which the threads are scheduled can greatly affect the total size of the live data at any instance during the parallel execution, and unless the threads are scheduled carefully, the parallel execution of a program may require much more memory than its serial execution. Since the price of the memory is a significant portion of the price of a parallel computer, and parallel computers are typically used to run big problem sizes, reducing memory usage is often as important as reducing running time. Many researchers have addressed this problem in the past. Early attempts to reduce the memory usage of parallel computations were based on heuristics that limited the parallelism [10, 15, 28, 32], and are not guaranteed to be space-efficient in general. These were followed by scheduling techniques that provide proven space bounds for parallel programs [6, 8, 9]. If S_1 is the space required by the serial execution, these techniques generate executions for a multithreaded computation on p processors that require no more than $p \cdot S_1$ space. A scheduling algorithm that significantly improved this bound was recently proposed [2], and was used to prove time and space bounds for the implementation of NESL [4]. It generates a schedule that uses only $S_1 + O(p \cdot D \cdot \log p)$ space on a standard p-processor EREW PRAM, where D is the depth of the parallel computation (i.e., the longest sequence of dependencies or the critical path in the computation). This bound is lower than the previous bound of $p \cdot S_1$ when $D < S_1$, which is true for parallel programs that have a sufficient degree of parallelism. The low space bound is achieved by ensuring that the parallel execution follows an order of computations that is as close as possible to the serial execution. However, the algorithm has scheduling overheads that are too high for it to be practical. Since it is synchronous, threads need to be rescheduled after every unit computation to guarantee the space bounds. Moreover, it ignores the issue of locality — a thread may be moved from processor to processor at every timestep.

In this paper, we describe a general framework to model parallel computations, in which portions of a thread can be executed non-preemptively. In this framework, a parallel computation is represented by a task graph, and executions of the computation are represented by non-preemptive schedules of the task graph. We define two classes of efficient schedules that can be generated offline for any task graph, and prove upper bounds on their space and time requirements. We then present an online, asynchronous scheduling algorithm called Async-Q, which generates a schedule with the same space bound of $S_1 + O(p \cdot D \cdot \log p)$ (including scheduler space) that is obtained in [2]; the algorithm assumes an EREW PRAM with a unit time fetch-and-add operation. As with their scheduling algorithm, our Async-Q algorithm applies to task graphs representing nested parallelism. However, our algorithm overcomes the above problems with their scheduling algorithm: it allows a thread to execute non-preemptively on the same processor until it reaches a point that requires a big allocation of space. This results in better locality, and since threads are suspended and rescheduled less often, it has low scheduling overheads. The asynchronous nature of our algorithm results in an execution order that differs from the order generated by the algorithm in [2]. This requires a different approach to prove space-efficiency. In addition to proving space-efficiency, as with [2], we bound the time required to execute the schedule generated by our algorithm in terms of the total work (number of operations) W and depth D of the parallel computation. If the space dynamically allocated within the computation is $O(W)$, the generated schedule runs in $O(W/p + D \cdot \log p)$ time. The algorithm assumes a shared memory programming model, and applies to languages providing nested parallelism. These include nested data-parallel languages, and control-parallel languages with a fork-join style of parallelism. The central data structure used in the algorithm is a shared scheduling queue, in which parallel threads are prioritized according to their serial execution order. Large allocations of space in the parallel computation are delayed (lowered in priority), so that other, higher priority threads can be executed instead.

We have built a runtime system that uses our algorithm to schedule parallel threads on the SGI Power Challenge. To test its effectiveness in reducing memory usage, we have executed a number of parallel programs on it, and compared their space and time requirements with previous scheduling techniques. The experimental results show that, compared to previous scheduling techniques, our system significantly reduces the maximum amount of live data at any time during the execution of the programs. In addition, good single-processor performance and high parallel speedups indicate that the scheduling overheads in our system are low, that is, memory requirements can be effectively reduced without compromising performance.

1.1 An example

The following pseudocode illustrates the main ideas behind our scheduling algorithm, and how they reduce memory usage compared to previous scheduling techniques.

```
In parallel for i = 1 to n
   Temporary B[n]
   In parallel for j = 1 to n
      F(B,i,j)
   Free B
```

This code has two levels of parallelism: the `i`−loop at the outer level, and the `j`-loop at the inner level. In general, the number of iterations in each loop may not be known at compile time. Space for an array `B` is allocated at the start of each `i`-iteration, which is freed at the end of the iteration. Assuming that `F(B,i,j)` does not allocate any space, the serial execution requires $O(n)$ space, since the space for array `B` is reused for each `i`-iteration. Now consider the parallel implementation of this function on p processors, where $p < n$. Previous scheduling systems [6, 8, 9, 19, 23, 28, 32], which include both heuristic-based and provably space-efficient techniques, would schedule the outer level of parallelism first. This results in all the p processors executing one `i`-iteration each, and hence the total space allocated is $O(p \cdot n)$. Our scheduling algorithm also starts by scheduling the outer parallelism, but stalls big allocations of space. Moreover, it gives a greater priority to the inner parallelism. As a result, the processors suspend the execution of their respective `i`-iterations before they allocate $O(n)$ space each, and execute `j`-iterations belonging to a single `i`-iteration instead. This allows the parallel computation to run in just $O(n + p \cdot D \cdot \log p)$ space, where D depends on the depth of the function `F`. Note that giving a higher priority to the inner parallelism keeps the order of computations close to that of the serial execution, which results in low memory usage. As an example, `F(B,i,j)` could be a simple function that is used to sort an array `A`:

```
F(B,i,j) :
   B[j] = if (A[i]>A[j]) then 1 else 0
```

so that summing the elements of `B` just before it is freed in iteration `i` gives the rank of element `A[i]` (the summing of the elements of `B` is not shown in the above code).

As another related example, consider n users of a parallel machine, each running parallel code. The user programs allocate a large block of space as they start and deallocate it when they finish. In this case the outer parallelism is across the users and the inner parallelism is within each user's program. A scheduler that schedules the outer parallelism would schedule p user programs to run simultaneously, requiring a total memory equal to the sum over the memory requirements of p programs. On the other hand, our scheduling algorithm would schedule one program at a time, as long as there is sufficient parallelism within each program to keep the processors busy. In this case, the total memory required is just the maximum over the memory requirement of each user's program.

A disadvantage of our scheduling technique is that if the inner parallelism is fine-grained, then scheduling it with a higher priority may lead to poor locality and high scheduling overheads. We overcome this disadvantage by grouping the fine-grained iterations of innermost loops into chunks. Our experimental results demonstrate that this approach is sufficient to yield good performance in time and space (see Section 7). In the results reported in this paper we have blocked the iterations into chunks by hand, but in Section 8 we discuss some ongoing work on automatically chunking the iterations.

1.2 Outline of the paper

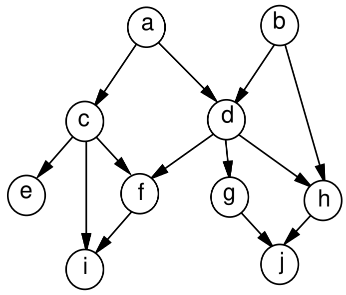
The remainder of this paper is organized as follows. In Section 2 we describe an existing model of parallel computations based on DAGs and schedules, and extend it to represent computations consisting of non-preemptive, variable-length tasks, with latencies between the tasks. The upper bound for the space required by a certain class of schedules that are based on serial schedules, is presented in Section 3. We define a new class of schedules called Q-prioritized schedules and bound their space requirements in Section 4. Our Async-Q scheduling algorithm generates Q-prioritized schedules. The next three sections describe the Async-Q algorithm and its implementation: Section 5 outlines the programming model that the algorithm implements; the Async-Q algorithm, along with proofs for upper bounds on the space and time requirements of the schedules generated by it, are presented in Section 6; Section 7 briefly describes our implementation of a runtime system based on the Async-Q algorithm and presents the results of implementing five parallel programs on it. Finally, we summarize and discuss future work in Section 8.

2 Modeling parallel computations: graphs and schedules

In this section, we describe a model for parallel computations based on directed acyclic graphs (DAGs), similar to the one defined in [2]. In this model, a parallel computation can be represented by a DAG called a *computation graph*. In a computation graph $G = (V, E)$, each node $v \in V$ corresponds to a *unit computation*, which requires a single timestep to be executed on one processor. Each edge $(u, v) \in E$ is a dependence edge, which imposes the condition that the computation represented by v can be executed only after the computation represented by u. For every edge $(u, v) \in E$, we call u the *parent* of v, and v the *child* of u. Any node with no parents is called a *root node*. The *work* W (the total number of unit computations) performed in the parallel computation is the total number of nodes in G. A computation with W work requires W timesteps to execute on one processor. The *depth* D of the parallel computation is the number of nodes on the longest path in G. A computation with depth D requires D timesteps to execute on an infinite number of processors.

A *schedule* for a computation graph $G = (V, E)$ is a sequence $(V_1, V_2, \ldots, V_T)$ that satisfies the following conditions.

1. For $i = 1, \ldots, T$, $V_i \subseteq V$,



V_1= {a, b}　　V_3= {f, g, h}

V_2= {c, d}　　V_4= {e, i, j}

Figure 1: An example computation graph and a p-schedule of length 4 for it (with $p \geq 3$). For $i = 1, \ldots, 4$, V_i is the set of nodes scheduled at timestep i. Each node can be scheduled only after all its parents have been scheduled.

2. $V = \bigcup_{i \in \{1,\ldots,T\}} V_i$, and $\forall i, j \in \{1, \ldots, T\}$, $V_i \cup V_j = \phi$, and,
3. $\forall (u, v) \in E$, $u \in V_i$ and $v \in V_j \Rightarrow j > i$.

A *p-schedule* is a schedule $\mathcal{S}_p = (V_1, V_2, \ldots, V_T)$ such that $\forall i \in \{1, \ldots, T\}$, $|V_i| \leq p$. For example, Figure 1 shows a simple computation graph and a p-schedule for it. The execution of a parallel computation on p processors can be represented by a p-schedule of its computation graph. Thus the set of nodes V_i represents the unit computations executed by the processors in timestep i. The *length* $|\mathcal{S}_p|$ of a p-schedule $\mathcal{S}_p$ is the number of timesteps required for its execution on p processors. Note that a 1-schedule represents a serial execution. We say a node v is *scheduled* in timestep t if $v \in V_t$. At the end of any timestep in a schedule, each node can be in one of three states: dormant, ready, and computed. A node v is *computed* at the end of t timesteps, if $v \in \bigcup_{i \in \{1,\ldots,t\}} V_i$, that is, if it has been scheduled in the first t steps. Any node with at least one parent that is not computed is said to be *dormant*. When all the parents of a node are computed, but the node itself is yet to be computed, we say it is *ready*. Thus, at the beginning of a schedule, all root nodes are ready, and all other nodes are dormant. At the end, all nodes are computed. If $t_{ready}(v)$ is the timestep at the beginning of which any node $v \in V$ becomes ready, and $t_{sched}(v)$ is the timestep in which it is scheduled, then condition 3 above implies that $t_{sched}(v) \geq t_{ready}(v) = \max\{t_{sched}(u) + 1 \mid (u, v) \in E\}$.

Greedy schedules. Let R_t be the set of ready nodes at the beginning of timestep t. At most p nodes from R_t are scheduled in timestep t of a p-schedule. R_{t+1} contains the remaining nodes of R_t plus nodes which became ready at the end of step t. A p-schedule is a *greedy schedule* if it satisfies the additional condition:

4. $\forall t = 1, \ldots, T$, $|V_t| = \min(p, |R_t|)$.

Thus if p or more nodes are ready at the beginning of timestep t, p nodes will be scheduled, that is, $|V_t| = p$; otherwise all the (fewer than p) ready nodes will be scheduled. For example, the schedule in Figure 1 is a greedy p-schedule for $p = 3$.

Modeling space allocations. To model memory allocations in the parallel computation that a computation graph $G = (V, E)$ represents, we add a *weight function* w that maps nodes $v \in V$ to integers. $w(v)$ represents the units of memory allocated by the unit computation corresponding to v. We assume that a unit computation can perform either a single allocation or a single deallocation; if the computation deallocates memory, $w(v) < 0$. The space requirement $space(\mathcal{S})$ of any schedule $\mathcal{S} = (V_1, \ldots, V_T)$ of G is the maximum space allocated during the execution represented by $\mathcal{S}$, and is defined as follows.

$$space(\mathcal{S}) = \max_{i=1,\ldots,T} \left(\sum_{j=1,\ldots,i} \sum_{v \in V_j} w(v) \right) \tag{1}$$

Let the *excess weight* W_e of G be the sum of all node weights that are greater than one, that is, $W_e = \sum\{w(v) \mid v \in V \text{ and } w(v) > 1\}$. Blelloch, Gibbons and Matias proved the following theorem in [2].

Theorem 2.1 *Let G be a computation graph with weight function w and depth D, and let $\mathcal{S}_1$ be any 1-schedule of G. Then for all $p \geq 1$ and all $m \geq 1$, there is a p-schedule $\mathcal{S}_p$ such that $space(\mathcal{S}_p) \leq space(\mathcal{S}_1) + m \cdot p \cdot D$, and $|\mathcal{S}_p| \leq |\mathcal{S}_1|/p + D + W_e/(m \cdot p)$.* ∎

2.1 Latency-weighted computation graphs

In this section, we extend the definition of a computation graph by allowing weights on the edges. We will call such a graph a *latency-weighted computation graph*, or simply a latency-weighted DAG. Let $G = (V, E)$ be a latency-weighted DAG representing a parallel computation. Each edge $(u, v) \in E$ has a nonnegative weight $l(u, v)$ which represents the *latency* between the computations of the nodes u and v. The *latency-weighted length* of a path in G is the sum of the total number of nodes in the path $plus$ the sum of the latencies on the edges along the path. We define *latency-weighted depth* D_l of G to be the maximum over the latency-weighted lengths of all paths in G. Since all latencies are nonnegative, $D_l \geq D$. Note that a computation graph is a special case of a latency-weighted DAG in which the latencies on all the edges are zero.

Schedules for latency-weighted DAGs. The definition of a schedule for a latency weighted DAG G is identical to the definition of a schedule for a computation graph given earlier in Section 2, except that condition 3 is replaced by

3′. $\forall(u, v) \in E$, $u \in V_i$ and $v \in V_j \Rightarrow j > i + l(u, v)$.

Thus a latency $l(u, v)$ implies that v cannot be scheduled until at least $l(u, v)$ timesteps after u is scheduled, that is, $t_{sched}(v) \geq t_{ready}(v) = \max\{t_{sched}(u) + l(u, v) + 1 \mid (u, v) \in E\}$. A latency-weighted DAG can be used to model a parallel computation with latencies between its unit computations, and a p-schedule for such a DAG represents an execution of the parallel computation on p processors. Any schedule of G will have a length of at least D_l, that is, the parallel computation represented by G will require at least D_l timesteps to execute on any number of processors. Similarly, even if G has work W, a serial execution may require more than W time due to the latencies.

2.2 Time bound for greedy schedules

The definition of a greedy scheduling algorithm can be easily extended to latency-weighted DAGs. A node v is added to the set of ready nodes after all its parents have been computed, *and* all the latencies imposed by the edges into v are satisfied (i.e., at timestep $t_{ready}(v)$). Blumofe and Leiserson [6] showed that any greedy p-schedule for a computation graph with depth D and work W has a length of at most $W/p + D$. In this section, we prove a similar upper bound on the length of any greedy p-schedule for latency-weighted DAGs.

Lemma 2.2 *Given a latency-weighted computation graph G with W nodes and latency-weighted depth D_l, any greedy p-schedule of G will require at most $W/p + D_l$ timesteps.*

Proof: We transform G into a DAG G' by replacing each edge (u, v) with a chain of $l(u, v)$ *dummy nodes*. The dummy nodes do not represent real work, but require a timestep to be executed. Any dummy node that becomes ready at the end of timestep $t - 1$ is automatically scheduled in timestep t. Therefore, replacing each edge (u, v) with $l(u, v)$ dummy nodes imposes the required condition that v becomes ready $l(u, v)$ timesteps after u is scheduled. The depth of G' is D_l.

Let $\mathcal{S}_p = (V_1, \ldots, V_T)$ be a greedy p-schedule for G'. For $i = 1, \ldots, T$, V_i may contain at most p real nodes, and any number of dummy nodes. This schedule corresponds to a parallel execution where the dummy nodes do not require processors to be executed, and hence all the dummy nodes ready at a timestep are scheduled in that timestep. The greedy p-schedule $\mathcal{S}_p$ of G' can be converted into a greedy p-schedule of G having the same length, by simply deleting the dummy nodes from $\mathcal{S}_p$.

We now prove that any greedy p-schedule $\mathcal{S}_p$ of G' will have a length of at most $W/p + D_l$. Let G'_i denote the subgraph of G' containing nodes that have not yet been computed at the beginning of timestep i. Note that $G'_1 = G'$. Let n_i be the number of real nodes (not including dummy nodes) scheduled in timestep i. Since $\mathcal{S}_p$ is a p-schedule, $n_i \leq p$. If $n_i = p$, there can be at most W/p such timesteps, since there are W nodes in the graph. If $n_i < p$, consider the set of nodes R_i that are ready at the beginning of timestep i, that is, the set of root nodes in G'_i. Since this is a greedy schedule, there are less than p real nodes in R_i. Hence all the real nodes in R_i get scheduled in timestep i. In addition, all the dummy nodes in R_i get scheduled in this step, since they are ready, and do not require processors. Since all the nodes in R_i are scheduled, the depth of G'_{i+1} is one less than the depth of G'_i. Since D_l is the depth of G'_1, there are at most D_l such timesteps. Thus $\mathcal{S}'_p$ (and hence $\mathcal{S}_p$) can have at most $W/p + D_l$ timesteps. ∎

Space requirements for schedules of latency-weighted DAGs. As with computation graphs, the space allocations of the computations can be modeled by a weight function w. Given a latency-weighted DAG $G = (V, E)$, for all $v \in V$, the integer weight $w(v)$ represents the units of memory allocated by the unit computation corresponding to v. The definition of the space requirement $space(\mathcal{S})$ of a schedule $\mathcal{S}$ for a latency-weighted DAG is the same as that in equation 1. Let the *maximum weight* M of G be defined as $M = \max\{w(v) \mid v \in V\}$. We state the following theorem, which is a special case of a theorem we prove later in the paper.

Theorem 2.3 *Let G be a latency-weighted DAG with maximum weight M and latency-weighted depth D_l, and let $\mathcal{S}_1$ be any 1-schedule of G. Then for all $p \geq 1$, there is a greedy p-schedule $\mathcal{S}_p$ such that $space(\mathcal{S}_p) \leq space(\mathcal{S}_1) + M \cdot p \cdot D_l$.* ∎

2.3 Task graphs

In this section, we define a class of DAGs called *task graphs*. A task graph $G = (V, E)$ is a DAG with weights on the nodes and edges. We will call each node of G a *task*, which can require multiple timesteps to be executed. Each task $v \in V$ is labeled with a nonnegative integer weight $t(v)$, which is the *duration* of task v (the time required to execute v, or the number of serial, unit-sized computations in v). Each edge $(u, v) \in E$ has a weight $l(u, v)$, which represents the latency between tasks u and v.

We will use task graphs to represent parallel computations in which variable-length tasks are executed non-preemptively: once a task v is scheduled on a processor, it executes to completion in $t(v)$ timesteps. The work W of a task graph G is defined as $W = \sum_{v \in V} t(v)$. The length of a path in G is the sum of the durations of the tasks along the path. Similarly, the latency-weighted length is the sum of the durations of the tasks plus the sum of the latencies on the edges along the path. The depth D of G is the maximum over the lengths of all the paths in G, and the latency-weighted depth D_l is the maximum over the latency-weighted lengths of all the paths in G. A latency-weighted DAG is a special case of a task graph in which $t(v) = 1$ for every task v.

Schedules for task graphs. A non-preemptive p-schedule $\mathcal{S}_p$ for a task graph is a sequence $(V_1, V_2, \ldots, V_T)$ which satisfies the following conditions.

1. $V = \bigcup_{i=\{1,\ldots,T\}} V_i$

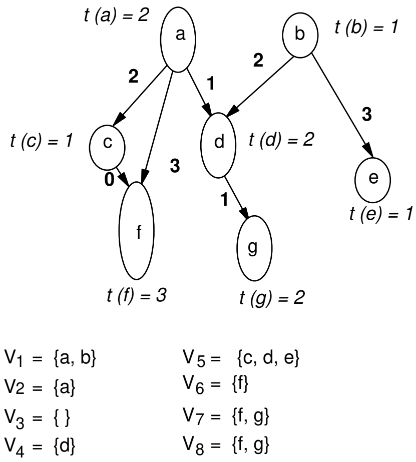


Figure 2: An example task graph and a p-schedule for it (with $p \geq 3$). Each oval represents a variable-length task. For each task v, $t(v)$ denotes the duration of the task, that is, the number of consecutive timesteps for which v must execute. Each edge is labeled (in bold) with its latency. For $i = 1, \ldots, 8$, V_i is the set of tasks being executed during timestep i of the p-schedule.

2. We say v is *scheduled* in timestep i if $v \notin V_{i-1}$ and $v \in V_i$. If v is scheduled at timestep i, then $v \in V_j \iff i \leq j < i + t(v)$.

3. $\forall i \in 1, \ldots, T,\ |V_i| \leq p$.

4. A task is *completed* in timestep i if $v \in V_i$ and $v \notin V_{i+1}$. $\forall (u, v) \in E$, if u is completed in timestep i, and v is scheduled in timestep j, then $j > i + l(u, v)$.

Since $\mathcal{S}_p$ is a non-preemptive schedule, condition 2 ensures that every task v appears in exactly $t(v)$ consecutive steps of $\mathcal{S}_p$. Figure 2 shows an example task graph and a non-preemptive p-schedule for it. Let $t_{ready}(v)$ be the timestep in which task v becomes *ready*, that is, when all the parents of v have been completed, and all the latencies imposed by edges into v are satisfied. If $t_{sched}(v)$ is the timestep in which v is scheduled, then the above conditions imply that $t_{sched}(v) \geq t_{ready}(v) = \max\{t_{sched}(u) + t(u) + l(u, v) \mid (u, v) \in E\}$. We say a task is being *executed* during timestep i if $v \in V_i$. Henceforth, we will restrict our discussion of schedules for task graphs to non-preemptive schedules, since we use task graphs to represent non-preemptive computations.

Greedy schedules for task graphs. Let R_t be the set of tasks that are ready at the beginning of timestep t of a p-schedule for a task graph. Since the schedule represents the non-preemptive execution of variable-length tasks on p processors, fewer than p processors may become idle at any timestep. Let n_t of the p processors become idle at the end of timestep $t - 1$ (i.e., $p - n_t$ continue executing tasks scheduled previously). A p-schedule that satisfies the following additional condition is a greedy p-schedule.

5. At every timestep t, min($n_t, |R_t|$) tasks are scheduled, that is, $|V_t| = min(n_t, |R_t|) + (p - n_t)$.

For example, the p-schedule in Figure 2 is greedy for $p = 3$. We prove the following lemma about the upper bound on the length of a greedy p-schedule.

Lemma 2.4 *Given a task graph G with W work and latency-weighted depth D_l, any greedy p-schedule of G will require at most $W/p + D_l$ timesteps.*

Proof: We convert G into a latency-weighted DAG G' by replacing each task of duration $t(v)$ into a chain of $t(v)$ nodes, each representing a unit computation. We will call these new nodes $v_1, v_2, \ldots, v_{t(v)}$. The new edges in the chain are assigned zero latencies. The work, depth and latency-weighted depth of G' are equal to those of G.

We now show that any greedy p-schedule $\mathcal{S}_p = (V_1, \ldots, V_T)$ for G can be converted into an equivalent greedy p-schedule $\mathcal{S}'_p = (V'_1, \ldots, V'_T)$ of G': we simply replace the i^{th} occurrence of task v in $\mathcal{S}_p$ with the node v_i. Thus if V_j contains the i^{th} occurrence of task v, then $v_i \in V'_j$. The resulting schedule $\mathcal{S}'_p$ is a p-schedule for G' of the same length as $\mathcal{S}_p$. Let R_t be the set of ready tasks of G at timestep t, and let R'_t be the set of ready nodes of G' at timestep t. Note that $\forall v \in R_t,\ v_1 \in R'_t$. In addition, for every task v, $\forall i \in 2, \ldots, t(v)$, if the $v_{i-1} \in V'_{t-1}$ then $v_i \in R'_t$ (i.e., when a node in the chain representing a task is scheduled, its child is ready in the next timestep).

Let n_t be the number of processors that become idle at any timestep t of $\mathcal{S}_p$. Since $\mathcal{S}_p$ is a greedy schedule, $min(n_t, |R_t|)$ tasks are scheduled at timestep t of $\mathcal{S}_p$. This implies that $(p - n_t)$ processors were busy executing unit-sized computations of previously scheduled tasks. Therefore, $|R'_t| = |R_t| + (p - n_t)$. Hence in $\mathcal{S}'_p$, the number of nodes scheduled in timestep i of $\mathcal{S}'_p$ are $min(n_t, |R_t|) + (p - n_t) = min(p, |R'_t|)$. Hence $\mathcal{S}'_p$ is a greedy schedule for G', and by Lemma 2.2, has a length of at most $W/p + D_l$. Since $\mathcal{S}_p$ has the same length as $\mathcal{S}'_p$, $\mathcal{S}_p$ has a length of at most $W/p + D_l$. ∎

Prioritized schedules. At every step of execution of a schedule, the scheduling algorithm must decide which of the ready tasks to schedule in that step. For example, it may pick tasks at random, or according to some predetermined priorities. Let R_t be the set of ready tasks at the beginning of timestep t, and let $r_t \subseteq R_t$ be the set of tasks that are scheduled in timestep t. Let the tasks in R_t be assigned priorities. If at every step t, r_t is a subset of R_t with the highest priorities (i.e., $\forall u \in r_t$ and $v \in (R_t - r_t)$, $priority(u) \geq priority(v)$), we call the resulting schedule a *prioritized schedule*. This definition is similar to the class of list schedules described in [16]. We call a prioritized schedule that is greedy a *greedy prioritized schedule*. Let $\mathcal{S}_1 = (\{v_1\}, \{v_2\}, \ldots, \{v_n\})$ be a 1-schedule for a task graph with n tasks. As defined in [2], we say a prioritized p-schedule is *based* on $\mathcal{S}_1$ if the relative priorities of tasks are based on their serial execution order: $\forall i, j \in \{1, \ldots, n\},\ i < j \Rightarrow priority(v_i) > priority(v_j)$. Thus v_1 gets the highest priority and v_n gets the lowest priority. If the task graph G and the serial schedule $\mathcal{S}_1$ for G are known beforehand, a simple, offline scheduling algorithm can generate the prioritized p-schedule for G based on $\mathcal{S}_1$. However, if the task graph G is revealed at runtime, an online scheduling algorithm that can dynamically find the ready tasks with the highest priorities is required to generate a prioritized p-schedule.

3 Modeling space allocations with task graphs

As with computation graphs, we add integer weights to the tasks to model space allocations. However, since a task may consist of multiple unit computations, each of which may perform a space allocation or deallocation, a single weight is insufficient to model all the space allocations taking place in a task. Therefore, we introduce two new integer weights for each task v in a task graph G: the *net memory allocation* $n(v)$, and the *memory requirement* $h(v)$. $n(v)$ is the difference between the total memory allocated and the total memory deallocated in v, and may be negative. $h(v)$ is the high-water mark of memory allocation, that is, the maximum memory allocated throughout the execution of task v. The task v can be executed on a processor given at least $h(v)$ units of memory to allocate and deallocate from. For every task v in G, $h(v) \geq n(v)$.

As before, the space requirement $space(\mathcal{S})$ of a schedule $\mathcal{S} = (V_1, \ldots, V_T)$ for G is the maximum memory allocated during the execution represented by $\mathcal{S}$. We now calculate an upper bound for $space(\mathcal{S})$. Let E_i be the set of tasks that have been completed at or before timestep i of S, that is,

$E_i = \{v \in V_j \mid (j \leq i)\ and\ (v \notin V_{i+1})\}$. Let s^i be the total units of memory allocated across all processors at the end of timestep i. Then

$$s^i \leq \sum_{v \in E_i} n(v) + \sum_{v \in V_i - E_i} h(v) \tag{2}$$

since every task v that has been completed at or before timestep i has allocated $n(v)$ units of memory, and every other task u being executed at timestep i may have allocated at most $h(u)$ units of memory. The space requirement of $\mathcal{S}$ is defined as $space(\mathcal{S}) = \max(s^1, s^2, \ldots, s^T)$. Note that this is a conservative estimate of the space required by the actual execution, because even if multiple tasks are executed at any timestep t, each such task v may not have allocated all the $h(v)$ units of memory it requires in that timestep. However, if $\mathcal{S}$ is a serial schedule, then the above expression for $space(\mathcal{S})$ is an exact value for the space requirement of the execution it represents. This is true because at most one task v is executed in each timestep of $\mathcal{S}$, and it is executed to completion; therefore it will allocate $h(v)$ units of memory at some timestep before it is completed.

3.1 Space bounds for prioritized p-schedules

Consider a task graph $G = (V, E)$ with latency-weighted depth D_l. Let the *maximum task space* M of G be the maximum over the memory requirements of all tasks in G; that is, $M = \max_{v \in V}\{h(v)\}$. Let $\mathcal{S}_p$ be a prioritized p-schedule for G based on a 1-schedule $\mathcal{S}_1$. In this section, we will prove that $space(\mathcal{S}_p) \leq space(\mathcal{S}_1) + M \cdot (p-1) \cdot (D_l - 1)$.

To prove the space bound, we first convert G into an equivalent computation graph G' as follows.

1. Replace each task $v \in G$ with a chain $(v_1, \ldots, v_{t(v)})$ of $t(v)$ nodes, each of which represents a unit computation. We call the first node of this chain, v_1, a *heavy node*, and the remaining nodes *light nodes*.

2. Replace each edge $(u, v) \in E$ with a chain of $l(u, v)$ *dummy nodes*. Each dummy node takes a timestep to be computed, and all the dummy nodes ready at the start of any timestep t of any schedule for G' are scheduled in that timestep.

All the edges in G' are assigned zero latencies, making G' a computation graph. Note that the depth of G' is D_l. We define $nodes(v)$ to be the set of nodes in the chain $(v_1, \ldots, v_{t(v)})$ that replaces a task v. For $i = 1, \ldots, t(v)$, the node v_i in G' represents the i^{th} unit computation of the task v in G; we define $task(v_i) = v$. Let $\mathcal{H}(G')$ be the set of heavy nodes in G'. Figure 3 shows a sample task graph G, and its equivalent computation graph G'.

A valid p-schedule $\mathcal{S}'_p = (V'_1, \ldots, V'_T)$ for G' is obtained from $\mathcal{S}_p$ by replacing the i^{th} occurrence of task v with the node v_i, and adding all the dummy nodes ready at the beginning of any timestep t to V'_t. Note that V'_t contains at most p real nodes, and an arbitrary number of dummy nodes. A heavy node $v_1 \in V'_t$ if the task v is scheduled in timestep t of $\mathcal{S}_p$. Every light or dummy node is scheduled exactly one timestep after its only parent is scheduled. We obtain a 1-schedule $\mathcal{S}'_1$ for G' from $\mathcal{S}_1$ in a similar manner.

For $i = 1, \ldots, T$, let $C_t = (V'_1 \cup V'_2 \cup \ldots \cup V'_t)$ be the *configuration* of $\mathcal{S}'_p$ after step t, that is, the set of nodes that have been computed at the end of step t. Similarly, let C^1_t be the configuration of $\mathcal{S}_1$ after step t. Given a configuration C_i of $\mathcal{S}'_p$, we call C^1_j the *largest contained serial configuration* of C_i if $C^1_j \subseteq C_i$, and either j is the last step of $\mathcal{S}_1$, or $C^1_{j+1} \nsubseteq C_i$. Thus C^1_j is the largest configuration of $\mathcal{S}_1$ contained within C_i. We define $\mathcal{P}(C_i) = C_i - C^1_j$ to be the set of *premature nodes* in C_i, and $\mathcal{P}_h(C_i) = \mathcal{P}(C_i) \cap \mathcal{H}(G')$ to be the set of *heavy premature nodes* C_i. Figure 4 shows an example computation graph with a prioritized p-schedule, and the set of premature nodes in a configuration of the p-schedule.

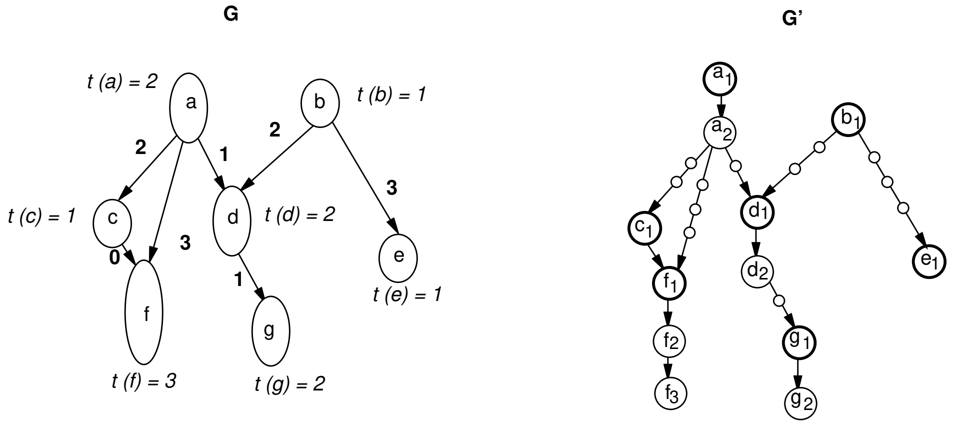
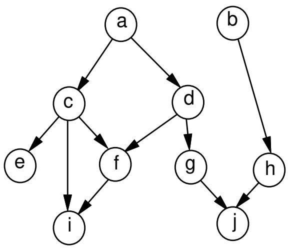


Figure 3: A transformation of a task graph G into a computation graph G'. Each edge (u, v) in G is labeled with the latency $l(u, v)$. The heavy nodes in G' are shown as bold, and the dummy nodes are shown as small circles added along the edges (the arrows on edges into dummy nodes are not explicitly drawn here).



S_1 = [{a}, {c}, {e}, {d}, {f}, {i}, {g}, {b}, {h}, {j}]

S_p = [{a, b}, {c, d, h}, {e, f, g}, {i, j}]

C_3 = {a, b, c, d, e, f, g, h}

C_5^1 = {a, c, e, d, f }

Figure 4: A simple computation graph with a prioritized p-schedule $\mathcal{S}_p$ based on a 1-schedule $\mathcal{S}_1$, where $p \geq 3$. C_3 is the configuration of $\mathcal{S}_p$ after 3 timesteps, and C_5^1 is its largest-contained serial configuration. The set of premature nodes in C_3 is $\mathcal{P}(C_3) = C_3 - C_5^1 = \{b, g, h\}$.

Lemma 3.1 *The number of heavy premature nodes in any configuration C_i of $\mathcal{S}'_p$ is at most $(p-1)\cdot(D_l-1)$, that is,*

$$\max\{|\mathcal{P}_h(C_i)| : 1 \le i \le T\} \le (p-1)\cdot(D_l-1)$$

Proof: The nodes in any configuration C^1 of $\mathcal{S}_1$ can be partitioned by level[1]. For $l = 1,\ldots,D_l$, let $C^1[l]$ be the set of nodes in C^1 at level l. We say the timestep t of $\mathcal{S}'_p$ *completes* a level l' of C^1 if it is the first timestep at the end of which all nodes in $C^1[1],\ldots,C^1[l']$ are computed; that is, if

1. $\exists l \in \{1,\ldots,l'\},\ C^1[l] \not\subseteq C_{t-1}$, and,
2. $\forall l \in \{1,\ldots,l'\},\ C^1[l] \subseteq C_t$.

Let C_i be any configuration of $\mathcal{S}'_p$, and let C^1_j be the largest contained serial configuration of C_i. Let v be the first node in $\mathcal{S}'_1$ that is not in C^1_j (if no such node exists, then C^1_j is the set of all nodes in G', and $|\mathcal{P}(C_i)| = 0$). This implies that $v \notin C_i$, or else it would be in C^1_j. Since $task(v)$ is scheduled in $\mathcal{S}_1$ before $task(u)$ for any $u \in \mathcal{P}_h(C_i)$, $task(v)$ has a higher priority than $task(u)$.

We show that any timestep t, such that $V'_t \cap \mathcal{P}_h(C_i) \neq \phi$, that is, any timestep t in which heavy premature nodes are scheduled, must complete a level $l < D_l$ of C^j_1. Consider such a timestep t. Let $l' < D_l$ be the smallest level such that $C^1_j[l'] \not\subseteq C_{t-1}$, that is, the smallest level of C^1_j that is not yet complete at the end of timestep $t-1$. If there is no such level, that is, if all the levels $1,\ldots,D_l-1$ are complete, then all parents of node v, which must be in these levels, are computed, and v is ready. If v is a light or dummy node, it is must be scheduled at or before timestep t, which is a contradiction. If v is a heavy node, it must be scheduled with or before any heavy premature node $u \in \mathcal{P}_h(C_i)$ (since $task(v)$ has a higher priority than $task(u)$), which is also a contradiction. Therefore, at least one level $l' < D_l$ exists, which is not complete at the end of timestep $t-1$.

Let U be the set of nodes in $C^1_j[l']$ that are not yet scheduled at the end of timestep $t-1$. All the nodes in U are ready, since all their parents in lower levels are computed. Therefore all the light and dummy nodes in U are scheduled in timestep t. Remember that $\mathcal{S}_p$ is a prioritized schedule; therefore at every step of $\mathcal{S}_p$ the ready tasks with the highest priorities are scheduled. But tasks with heavy nodes in $\mathcal{P}_h(C_i)$ have lower priorities than tasks with heavy nodes in U. Since at least one of the heavy nodes in $\mathcal{P}_h(C_i)$ gets scheduled in timestep t, all the heavy nodes in U must be scheduled with it. Therefore all the nodes in U are scheduled in timestep t, that is, t completes level l'. There can be at most (D_l-1) timesteps that complete a level of C^1_j less than D_l. At least one node from that level is scheduled in each such timestep t, and hence at most $(p-1)$ heavy premature nodes may be scheduled with it. Therefore, the total number of heavy premature nodes $|\mathcal{P}_h(C_i)|$ scheduled in the first i timesteps of $\mathcal{S}'_p$ is at most $(p-1)\cdot(D_l-1)$. ∎

Lemma 3.2 *Let $\mathcal{P}(C_i)$ be the set of premature nodes in any configuration C_i of $\mathcal{S}'_p$. Then $v_t \in \mathcal{P}(C_i) \Rightarrow (t = 1$ or $v_{t-1} \in \mathcal{P}(C_i))$.*

Proof: If $t = 1$ we are done. Assume that $t > 1$, and $v_{t-1} \notin \mathcal{P}(C_i)$. Since v_{t-1} is a parent of v_t and $v_t \in C_i$, we must have $v_{t-1} \in C_i$. Since $v_{t-1} \notin \mathcal{P}(C_i)$, $v_{t-1} \in C^1_j$, where C^1_j is the largest contained serial configuration of C_i. But v_t is scheduled in $\mathcal{S}_1$ in the timestep after v_{t-1}, and hence must also belong to C^1_j; otherwise C^1_j would not be the largest contained serial configuration of C_i. This is a contradiction. Therefore, for every node $v_t \in \mathcal{P}(C_i)$, all the nodes in its chain up to v_t (including the heavy node v_1) must be in $\mathcal{P}(C_i)$. ∎

[1]The length of the longest path from root nodes to a node v in a graph is the level of v. Level 1 consists of all the root nodes.

Corollary 3.3 *The number of distinct tasks v such that $nodes(v) \cap \mathcal{P}(C_i) \neq \phi$ is $|\mathcal{P}_h(C_i)|$.*

Lemma 3.4 *Let $\mathcal{S}'_p$ be the p-schedule for the computation graph G', obtained from the prioritized p-schedule $\mathcal{S}_p$ for G based on $\mathcal{S}_1$.. Let C_i be the configuration of $\mathcal{S}'_p$ after timestep i, and let $\mathcal{P}_h(C_i)$ be the set of heavy premature nodes in C_i. Then, $space(\mathcal{S}_p) \leq space(\mathcal{S}_1) + M \cdot (\max_{i=1,\ldots,T} |\mathcal{P}_h(C_i)|)$*

Proof: For a set of nodes C from G', let $F(C)$ be the set of tasks v of G such that $nodes(v) \subseteq C$, and let $A(C)$ be the tasks v such that $nodes(v) \cap C \neq \phi$ and $nodes(v) \not\subseteq C$. Thus $F(C)$ is the set of tasks that have all their nodes in C, and $A(C)$ is the set of tasks that have some, but not all of their nodes in C. Note that for every configuration C_i^1 of $\mathcal{S}_1$, $|A(C_i^1)| \leq 1$. Let s_p^i be the upper bound on the total space allocated at the end of timestep i of $\mathcal{S}_p$, and let s_1^i be the exact space allocated at the end of timestep i of $\mathcal{S}_1$, as defined in equation 2 in Section 3. Note that dummy nodes are not associated with any task, and do not allocate any space; hence we ignore them while measuring the space requirement.

Then

$$\begin{aligned} s_p^i &\leq \sum_{v \in F(C_i)} n(v) + \sum_{v \in A(C_i)} h(v) \\ &= \sum_{v \in F(C_j^1)} n(v) + \sum_{v \in A(C_j^1)} h(v) + \\ &\quad \sum_{v \in F(\mathcal{P}(C_i))} n(v) + \sum_{v \in A(\mathcal{P}(C_i))} h(v) \end{aligned}$$

If $|A(C_j^1)| = 0$, let $k = j$; otherwise, let k be the timestep of $\mathcal{S}_1$ at the end of which the single task $v \in A(C_j^1)$ has allocated $h(v)$ units of memory[2]. Then $s_k^1 = \sum_{v \in F(C_j^1)} n(v) + \sum_{v \in A(C_j^1)} h(v)$. Therefore

$$\begin{aligned} s_p^i &\leq s_k^1 + \sum_{v \in F(\mathcal{P}(C_i))} n(v) + \sum_{v \in A(\mathcal{P}(C_i))} h(v) \\ &\leq s_k^1 + M \cdot |\mathcal{P}_h(C_i)| \end{aligned}$$

since $\forall v,\ n(v) \leq h(v) \leq M$, and using Corollary 3.3.
Therefore, $space(\mathcal{S}_p) = \max(s_p^1, \ldots, s_p^T) \leq space(\mathcal{S}_1) + M \cdot \max_{i=1,\ldots,T} |\mathcal{P}_h(C_i)|$. ∎

Theorem 3.5 *Let G be a task graph with latency-weighted depth D_l, and maximum task space M. Let $\mathcal{S}_p$ be the prioritized p-schedule based on any 1-schedule $\mathcal{S}_1$ for G. Then,*

$$space(\mathcal{S}_p) \leq space(\mathcal{S}_s) + M \cdot (p-1) \cdot (D_l - 1).$$

Proof: We transform G into an equivalent computation graph G' of depth D_l, and $\mathcal{S}_p$ into a schedule $\mathcal{S}'_p$ for G', as described in this section. The required result follows using Lemmas 3.1 and 3.4. ∎

[2] Since the maximum requirement of v is $h(v)$, there must be at least one such timestep.

4 Q-prioritized schedules

Later in this paper, we describe a scheduling algorithm that generates a class of p-schedules that are based on serial schedules, but are not completely prioritized: at every timestep t, the tasks scheduled may not be the tasks with the highest priority. This deviation from the serial execution order allows a simple and efficient implementation of the scheduling algorithm. To define the generated class of schedules more precisely in this section, we introduce the concept of a *work queue*. The work queue is a FIFO queue of tasks, which are a subset of the ready tasks. When any processor becomes idle, that is, when the current task executing on the processor is completed, the task at the head of the work queue is scheduled on that processor. If $n > 1$ processors become idle in the same timestep, we assume that they all simultaneously pick a task each from the first n tasks in the work queue to begin executing in that timestep. If the number of tasks in the work queue is less than the number of processors that are idle, the tasks are assigned to an arbitrary subset of the idle processors, while the remaining busy wait. A scheduling algorithm inserts ready tasks into the work queue.

Let $G = (V, E)$ be a task graph with a 1-schedule $\mathcal{S}_1$. As with prioritized schedules, the tasks of G are assigned priorities based on their order in $\mathcal{S}_1$. Let the *maximum size* q_{max} of the work queue be the maximum number of tasks that it can hold. At the start of any timestep t, let R_t be the set of ready tasks, let Q_t be the subset of these tasks that are in the work queue, and let $R'_t = R_t - Q_t$ be the set of remaining ready tasks. Thus $|R_t| = |R'_t| + |Q_t|$ is the total number of ready tasks. We define a *queuing step* as a timestep in which nodes are added to the work queue. If a timestep t is a queuing step, then at most $\min(q_{max} - |Q_t|, |R'_t|)$ tasks are moved from R'_t to Q_t; otherwise no tasks are moved to Q_t. In addition, the set of tasks moved to Q_t in the queuing step t must be the tasks with the highest priorities in R'_t. Figure 5 shows the movement of tasks in the system. The resulting schedule $(V_1, \ldots, V_T)$, where V_i is the set of tasks executing at timestep i, is called a *Q-prioritized schedule* based on $\mathcal{S}_1$. On p processors (i.e., when for $i = 1, \ldots, T$, $|V_i| \leq p$), the resulting schedule is a Q-prioritized p-schedule. Note that for a given q_{max}, several Q-prioritized p-schedules may exist for the same $\mathcal{S}_1$, since we do not specify which timesteps are queuing steps, and how many tasks must be moved to the work queue in each queuing step. Figure 6 shows one possible p-schedule (using $p = 2$) of a simple task graph. Tasks in a Q-prioritized schedule may be scheduled several timesteps after they are inserted into the work queue; therefore, by the time a task is scheduled, it may no longer be among the highest priority ready tasks.

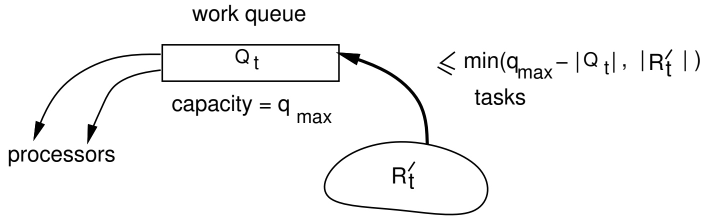
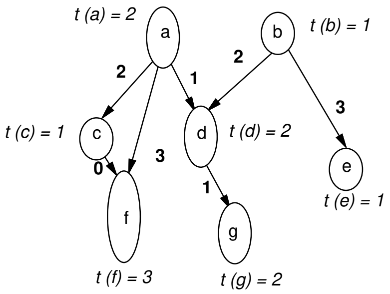


Figure 5: The movement of tasks in the system generating a Q-prioritized schedule. Q_t is the set of ready tasks in the FIFO work queue at the start of timestep' t; R'_t is the set of remaining ready tasks. R'_t includes tasks that become ready at the start of timestep t. In each queuing step t, at most $\min(q_{max} - |Q_t|, |R'_t|)$ tasks are moved from R'_t to the work-queue. Thus the work queue can never have more than q_{max} tasks.

We call u the *last parent* of v in a schedule, if it is the last of v's parents to be completed in the schedule. For any task $v \in V$, let $q(v)$ be the number of queuing steps that take place after the last parent of v has been completed and before task v is ready. We define the *queuing frequency* δ of a Q-prioritized schedule as $\delta = \max_{v \in V}\{q(v)\}$.



(a)

t	R'_t	Q_t	R'^{*}_t	Q^*_t	V_t
1	a, b	–	b	a	a
2	b	–	–	b	a, b
3	–	–	–	–	–
4	–	–	–	–	–
5	c, d	–	d	c	c
6	d, e, f	–	f	d, e	d, e
7	f	–	–	f	d, f
8	–	–	–	–	f
9	g	–	g	–	f
10	g	–	–	g	g
11	–	–	–	–	g

(b)

Figure 6: (a) An example task graph G. (b) One of the many possible Q-prioritized p-schedules $\mathcal{S}_p = (V_1, \ldots, V_{11})$ for G, using $p = 2$ and $q_{max} = 2$, assuming an alphabetical priority on the tasks. Q_t is the set of tasks in the work queue at the start of timestep t, while R'_t is the set of remaining ready tasks. Q^*_t and R'^{*}_t are the respective sets after tasks are moved from R'_t to Q_t at the start of timestep t. The set of tasks moved from R'_t to Q_t must be the ones with the highest priorities in R'_t. If either of the two processors is idle at timestep t, it picks up a task from Q^*_t to execute in that timestep. The queueing steps are $t = 1, 2, 5, 6, 7, 10$.

4.1 Space requirements of Q-prioritized schedules

In this section, we prove an upper bound on the space requirements of Q-prioritized schedules. Let $G = (V, E)$ be any task graph. Consider any Q-prioritized p-schedule $\mathcal{S}_p$ based on a 1-schedule $\mathcal{S}_1$, generated using a work queue of capacity q_{max}, and having a queuing frequency δ.

We first convert G into a latency-weighted DAG G', by replacing every task v with a chain of nodes $(v_1, \ldots, v_{t(v)})$. As described in Section 3.1, we call the first node of this chain a heavy node, and the remaining nodes light nodes; every new edge in this chain is assigned a latency of zero. Note that unlike the transformation in Section 3.1, we do not replace each edge of G with dummy nodes in G'; since G' is a latency-weighted DAG, we leave the latencies on the edges unchanged.

The p-schedule $\mathcal{S}_p$ for G is converted into a p-schedule $\mathcal{S}'_p$ for G' by replacing the i^{th} occurrence of task v with the node v_i. Similarly, the 1-schedule $\mathcal{S}_1$ is converted into a 1-schedule $\mathcal{S}'_1$ for G'. Remember that the scheduling algorithm uses a work queue to store ready tasks. We will view this as a queue of nodes; if it holds the task v, we will instead say it holds the heavy node v_1, which is the first node of v to be scheduled when v is removed from the queue. Thus the work queue may contain up to q_{max} heavy nodes.

The *level* for each node in G' is the maximum number of nodes along any path from a root node to that node. As in Section 3.1, for every configuration C^1 of $\mathcal{S}'_1$, we denote the nodes at level l in C^1 by $C^1[l]$. Let C_i be the i^{th} configuration of $\mathcal{S}'_p$, and let C^1_j be the largest contained serial configuration of C_i. Remember that the set of premature nodes in C_i is defined as $\mathcal{P}(C_i) = C_i - C^1_j$, and the set of heavy premature nodes is $\mathcal{P}_h(C_i) = \mathcal{P}(C_i) \cap \mathcal{H}(G')$, where $\mathcal{H}(G')$ is the set of heavy nodes in G'.

Lemma 4.1 *Let C_i be the configuration of $\mathcal{S}'_p$ after timestep i, and let $\mathcal{P}_h(C_i)$ be the number of heavy premature nodes in C_i. Then,*

$$\max\{|\mathcal{P}_h(C_i)| : 1 \leq i \leq T\} \leq ((\delta + 1) \cdot q_{max} + p - 1) \cdot D$$

Proof: Let C_j^1 be the largest contained serial configuration of C_i. Let t_l be the timestep that completes level l of C_j^1. Since there are no nodes in G' at level 0, we define t_0 to be 0. For $l = 1, \ldots, D$, let $I_l = (t_{l-1}, t_l]$ be the interval beginning after timestep t_{l-1} and ending with timestep t_l.

We will bound the number of heavy premature nodes scheduled in any such interval of $\mathcal{S}'_p$. Consider any arbitrary interval I_l. By the end of timestep t_{l-1}, all the nodes in $C_j^1[l-1]$ are computed. Hence, all the light nodes in $C_j^1[l]$, which must be scheduled one timestep after their parent, are computed by the end of timestep $t_{l-1}+1$. However, due to latencies on the edges, all the heavy nodes in $C_j^1[l]$ may not even be ready in timestep $t_{l-1}+1$. At the start of this timestep, the work queue may contain at most q_{max} heavy premature nodes. Let $t_r > t_{l-1}$ be the timestep in which the last node in $C_j^1[l]$ becomes ready[3]. Since δ is the queuing frequency of $\mathcal{S}_p$, at most δ queuing steps can take place after t_{l-1} and before t_r. There may result in at most $\delta \cdot q_{max}$ heavy nodes being added to the work queue, and all of these nodes may be premature.

Let U be the subset of heavy nodes in $C_j^1[l]$ that are ready but are not yet in the work queue at the start of timestep t_r; the remaining nodes in $C_j^1[l]$ are either computed or in the work queue. For any $u \in U$ and $v \in \mathcal{P}_h(C_i)$, $task(u)$ has a higher priority than $task(v)$. Therefore, timestep t_r onwards, all nodes in U must be added to the work queue before any premature nodes. Since the work queue is a FIFO, a total of at most $((\delta+1)\cdot q_{max})$ heavy premature nodes get scheduled before all the nodes in U are scheduled. By definition, the last node(s) in U gets scheduled at timestep t_l; in this timestep, at most $(p-1)$ premature nodes may be scheduled along with the last node in U. Thus, a total of $((\delta+1)\cdot q_{max} + p - 1)$ heavy premature nodes may get scheduled in the interval I_l. Since there are at most D such intervals, at most $((\delta+1)\cdot q_{max} + p - 1)\cdot D$ heavy premature nodes may be scheduled in the first i timesteps of $\mathcal{S}'_p$, that is, for any $i = 1, \ldots, T$, $|\mathcal{P}_h(C_i)| \leq ((\delta+1)\cdot q_{max} + p - 1)\cdot D$. ∎

Lemma 4.2 *Let C_i be the configuration of $\mathcal{S}'_p$ after timestep i, and let $\mathcal{P}_h(C_i)$ be the set of heavy premature nodes in C_i. Then, $space(\mathcal{S}_p) \leq space(\mathcal{S}_1) + M \cdot (\max_{i=1,\ldots,T} |\mathcal{P}_h(C_i)|)$*

Proof: The proof is identical to the proof of Lemma 3.4. ∎

Theorem 4.3 *Let G be a task graph with depth D and maximum task space M. Let $\mathcal{S}_1$ be any 1-schedule of G, and let $\mathcal{S}_p$ be any Q-prioritized greedy p-schedule for G based on $\mathcal{S}_1$ and generated using a work queue of capacity q_{max}. If δ is the queuing frequency of $\mathcal{S}_p$, then $space(\mathcal{S}_p) \leq space(\mathcal{S}_1) + ((\delta+1)\cdot q_{max} + p - 1)\cdot M \cdot D$.*

Proof: We transform G into a latency-weighted DAG G' as described in this section. Then the required result follows from Lemmas 4.1 and 4.2. ∎

Consider a Q-prioritized p-schedule $\mathcal{S}_p = (V_1, \ldots, V_T)$ of a task graph G based on a 1-schedule $\mathcal{S}_1$. Let G' be the corresponding latency-weighted computation graph, with a corresponding p-schedule $\mathcal{S}'_p$. We will call the set of tasks $TC_i = \bigcup_{j=1,\ldots,i} V_j$ the *task configuration* of $\mathcal{S}_p$ after i timesteps. If C_i is the corresponding configuration of $\mathcal{S}'_p$, and $\mathcal{P}(C_i)$ is the set of premature nodes in C_i, the set of tasks that have their nodes in $\mathcal{P}(C_i)$ are the *premature tasks* in TC_i.

5 Programming model for online scheduler

So far we have presented results for efficient schedules that can be generated offline for any parallel computation represented by a task graph. In the next section, we present an efficient, online scheduling

[3]If $C_j^1[l]$ has no nodes that are not ready, then $t_r = t_{l-1}+1$.

algorithm for executing a class of parallel computations. We describe the class of parallel computations in this section.

The algorithm is applicable to languages that support nested parallelism, which include data-parallel languages (with nested parallel loops and nested parallel function calls), control-parallel languages (with fork-join constructs), and any mix of the two. The system assumes a shared memory programming model, in which parallel programs can be described in terms of threads. The computation starts with one initial (root) thread. On encountering a parallel loop (or fork), a thread forks one child thread for each iteration and then suspends itself. Each child thread may, in turn, fork more threads. We assume that the child threads do not communicate with each other. They synchronize at the end of the iteration and terminate. The last child thread to reach the synchronization point awakens the suspended parent thread. A thread may fork any number of child threads, and this number need not be known at compile time. When a thread forks child threads, we assume there is a known ordering among the child threads, which is the order in which they are forked in a serial execution.

5.1 Parallel computation as a task graph

The entire multithreaded computation can be represented by a task graph G, as defined in Section 2.3. This task graph is formed at runtime by splitting each thread into a series of tasks, such that each task v has a memory requirement $h(v) \leq K$ units, where $K \geq 1$ is a constant. *We assume for now that no unit computation allocates more than K space*; this assumption will be relaxed later. Thus a task v in G represents a series of $t(v)$ consecutive unit computations of a thread, requires no more than K space, and is executed non-preemptively on one processor. For the model of nested parallelism described above, the parallel computation results in a *series-parallel* task graph. A series-parallel task graph, similar to a series-parallel computation graph [2], can be defined inductively: the graph G_0 consisting of a single task (which is both its source and sink) and no edges, is series parallel. If $G_1, \ldots, G_n$, $n \geq 1$ are series parallel, then the graph obtained by adding to $G_1 \cup \ldots \cup G_n$ a new source task u, with edges from u to the source tasks of $G_1, \ldots, G_n$, and a new sink task v, with edges from the sink tasks of $G_1, \ldots, G_n$ to v is series-parallel.

For a series parallel task graph G, a *depth-first* 1-schedule or 1DF-*schedule* is defined as follows. In the first step of a 1DF-schedule, schedule the root task. At every subsequent step, schedule the first ready child of the most recently scheduled task with a ready child[4]. Let $\mathcal{S}_1$ be a 1DF-schedule. We say the i^{th} task to be scheduled in $\mathcal{S}_1$ has a 1DF-*number* i. Figure 7 shows an example series parallel task graph; each task is labeled with its 1DF-number. In most implementations of the above programming model, the serial execution can be represented by a 1DF-schedule. In the next section, we describe an online scheduling algorithm that dynamically creates the series parallel task graph G for a nested parallel computation, and generates a Q-prioritized schedule for G based on its 1DF-schedule $\mathcal{S}_1$. The scheduling algorithm is an online algorithm because neither the task graph G, nor the 1DF-schedule $\mathcal{S}_1$ are given as input; they are revealed as the parallel computation proceeds[5]. We show how the tasks in a series-parallel task graph can be ordered by their priorities (serial execution orders) at runtime. Note that a similar scheduling algorithm will apply to any task graph for which such an order can be maintained at runtime.

6 Async-Q: An efficient scheduling algorithm

In this section, we describe an asynchronous scheduling algorithm called Async-Q. Let $\mathcal{S}_1$ be the schedule representing the serial depth-first execution of a parallel computation of depth d. Later in the paper we show that the Async-Q algorithm results in an execution on p processors that requires at most

[4]Recall that the children of a task have a known ordering.

[5]This differs from offline algorithms where the task graph is known beforehand.

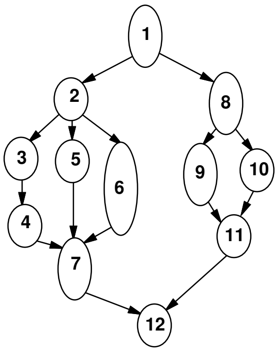


Figure 7: A series-parallel task graph, with variable-length tasks. Each task is labeled with its 1DF-number, that is, the order in which it is scheduled in a 1DF-schedule. We have assumed a left-to-right ordering among child tasks.

$space(\mathcal{S}_1) + O(pd \log p)$ memory; we also bound the number of timesteps required for the resulting execution.

As described in Section 5.1, each thread can be represented by a series of tasks. At any timestep during the execution, we define the *header task* of a thread as the first task of the thread that is not yet computed, that is, the task representing the series of computations that are either being executed or are to be executed next. The Async-Q scheduling algorithm maintains the *ready set* R, which is a set of threads with ready header tasks. The threads in R are ordered by the priorities of their header tasks. We will explain later how the relative ordering of the threads is maintained, even though the task graph is revealed at runtime. For simplicity, we will call the priority of the header task of a thread the priority of the thread itself[6]. In addition to R, there are two FIFO queues called Q_{in} and Q_{out}. The threads with the highest priorities are moved from R to Q_{out}. Processors pick threads from the head of Q_{out} and start executing their computations. Let m be the running count of the net space allocated by a processor after it picks a thread from Q_{out}; m may change with every timestep. Since each task v represents a series of computations of a single thread, and is allowed to have a memory requirement $h(v)$ of at most K units, each processor executes a thread until the thread terminates, or until it reaches a unit computation that is either a fork, or one that would cause m to exceed K units. At this point, we say the header task of the thread has been computed. Threads are thus split into a series of tasks at runtime. After a processor computes the header task of the thread, it inserts the thread into Q_{in}.

The *scheduler* is a computation responsible for moving threads from Q_{in} to R, creating child threads for a thread that forks, and moving threads with the highest priorities from R to Q_{out}. Figure 8 shows the migration of threads between the three data structures. The scheduler computation consists of a while loop; we will call each iteration of the loop a *scheduling iteration.* We will say a task v is in Q_{in} (Q_{out} or R) if the thread with v as its header task is in Q_{in} (Q_{out} or R). If the last parent of a task v is in Q_{in} when a scheduling iteration starts, then we say that v becomes *ready* at the end of that scheduling iteration; if it is among the highest priority tasks in R and is moved to the work queue, it is now available to be scheduled.

The pseudocode for the Async-Q algorithm is given in Figure 9. $|Q_{out}|$ is the number of tasks in Q_{out}. The computation starts with R containing the root thread, and Q_{in} and Q_{out} being empty. We will call the processors executing the worker computation the *workers.*A parent thread that suspends after forking is not stored explicitly in R; instead, we delete the thread after it forks, and allow the last child thread that reaches the synchronization point to continue as the parent thread. Initially, the system contains only the root thread, which is in the *ready* state. We say a thread is ready if it has a ready header task. When the parent of a thread τ reaches the point where it forks τ and its siblings, τ is created as a ready thread in R. When it is

[6]Since a thread may have a different header task at different timesteps, the priority of a thread may change with time.

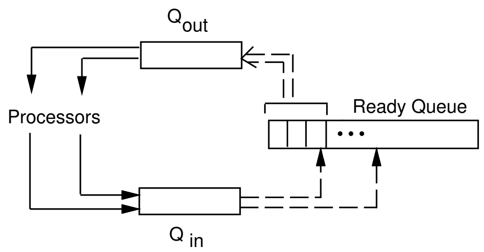


Figure 8: The movement of threads between the processors and the scheduling queues. Q_{in} and Q_{out} are FIFOs, and R is a set of threads with ready header tasks, ordered by the priorities of the header tasks. The dashed lines indicate the movement of threads by the scheduler, whereas the solid lines indicate the movement of threads by the worker processors (the processors that execute the threads).

worker:
begin
  while *(there exist threads in the system)*
    τ *:= remove-thread(*Q_{out}*);*
    execute the header task of τ*;*
    insert-thread(τ*,* Q_{in}*);*
end

scheduler:
begin
  while *(there exist threads in the system)*
    move all threads from Q_{in} *to* R*;*
    for every terminated thread τ *in* R *that is the last to synchronize,*
      retain τ *in* R *to execute as its parent;*
    delete all other terminated threads from R*;*
    for every thread in R *that forks, replace it with its child threads;*
    move $min(|R|, q_{max} - |Q_{out}|)$ *threads with the highest priority*
      from R *to* Q_{out}.
end

Figure 9: The Async-Q scheduling algorithm. The worker and scheduler computations execute asynchronously in parallel. q_{max} is the maximum size of Q_{out}, and $|Q_{out}|$ is the number of tasks in Q_{out}. A parent thread that suspends after forking is not stored explicitly; its last child to synchronize continues the computations of the parent. The function *remove-thread(*Q_{out}*)* busy waits until a thread becomes available and is removed from Q_{out}.

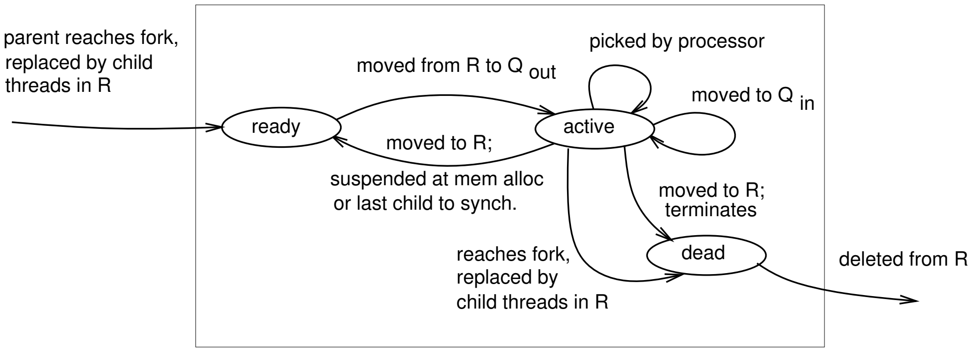


Figure 10: The state transition diagram for threads in the scheduling system. All threads inside the rectangular box exist in the system; initially only the root thread exists as a ready thread. The dead threads in R are deleted by the scheduler in every scheduling iteration. All threads are deleted from the system by the end of the execution.

moved from R to Q_{out}, it becomes *active*, and remains active until it is returned to R. If it has terminated, it becomes *dead*; if it has reached a fork, it is replaced by its child threads in R and hence becomes dead; if it is suspended due to a memory allocation, or is the last child to reach a synchronization point, it becomes ready on insertion into R[7]. A dead thread is removed from the system on deletion from R. The state transition diagram for threads is shown in Figure 10.

6.1 Scheduler implementation

To ensure efficient implementation of the scheduler, we set the maximum size q_{max} of Q_{out} to be $p \log p$, where p is the total number of processors. In this section, we describe how the scheduler can maintain the ordered set of ready threads R, prioritized by the 1DF-numbers of their header tasks. Let a constant fraction αp of the processors ($0 < \alpha < 1$) be devoted to executing the scheduler. The remaining $(1 - \alpha)p$ processors execute as workers. We assume a unit time fetch-and-add operation in our system. This makes the access to the queues Q_{in} and Q_{out} unit time for all processors.

The ready set R can be implemented as an array of threads, stored in decreasing order of priorities, say, from left to right. Initially R contains the root thread. When a thread is moved from R to Q_{out}, a *stub* for the thread is kept in its original position, and a pointer to the stub is stored in the thread. The header task of this thread is subsequently executed by a worker processor which then inserts the thread into Q_{in}. When the scheduler moves the thread from Q_{in} to R, it puts it back in the position of its stub. Thus every active thread has one stub in R. A thread returned to R may either terminate, or reach a computation that allocates more memory or performs a fork. If the thread terminates (is not the last child to reach a synchronization point), it is deleted from R; if it reaches a fork, it is marked as a *seed* thread. Otherwise it simply remains in R with a new header task[8]. For each seed representing an n-way fork, the scheduler replaces it with its n child threads, which are stored from left to right in decreasing order of the 1DF-numbers of their header tasks[9].

Lemma 6.1 *The threads in R are always in decreasing order of priorities.*

Proof: This can be proved by induction; the proof is similar to the one given in [2], and we present the outline here. Before the execution begins, R contains just the root thread, and therefore it is ordered by the

[7]The last child thread to reach the synchronization point becomes ready to continue as the parent thread.

[8]The new header task starts with the unit computation that would have caused the memory requirement of the previous header task to exceeded K, or is the first header task of the parent after a synchronization point.

[9]This is just the predefined order in which the child threads are forked in $\mathcal{S}_1$.

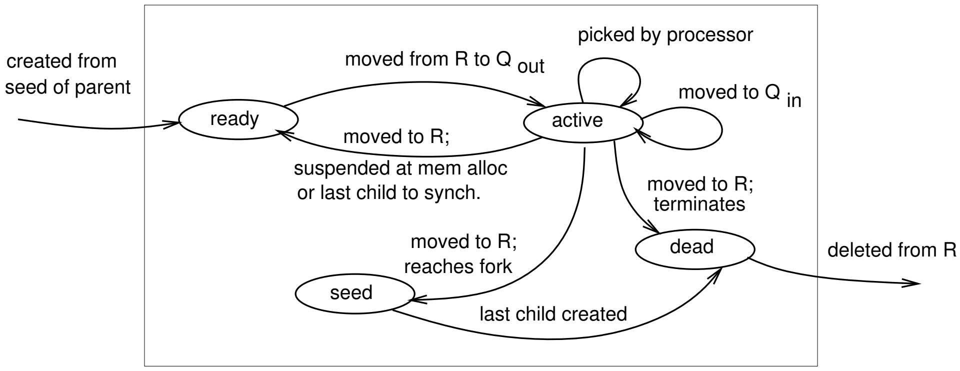


Figure 11: The new state transition diagram for threads using a lazy implementation of the scheduler. This differs from the previous diagram in Figure 10 in two ways: first, a thread, on reaching a fork, becomes a seed that implicitly represents its child threads, instead of being eagerly replaced by them; second, all dead threads in R need not be deleted in every scheduling iteration.

1DF-numbers. Assume that at some timestep the threads in R are in decreasing order of the 1DF-numbers of their header tasks. For a thread that forks, replacing it with its child threads in the order of their 1DF-numbers, and allowing the last child thread that synchronizes to continue as the parent thread, maintains the ordering by 1DF-numbers. A thread whose header task has been executed, but does not fork, is returned to its original position with a new header task. The new header task, which is the child of the old header task, has the same 1DF-number as the parent, *relative* to header tasks of other threads. Deleting threads from R does not affect the ordering of threads in R. Therefore the ordering of threads in R by 1DF-numbers is preserved after every operation on R. ∎

In every iteration, the scheduler needs to move the leftmost $min(|R|, q_{max} - |Q_{out}|)$ threads to Q_{out}, since by Lemma 6.1, these are the threads with the highest priorities.

6.2 Lazy scheduler

Eagerly creating all the child threads from a seed may cause the size of R to exceed the space bounds stated at the beginning of Section 6. In this section, we describe a lazy implementation of the scheduler, using which we later prove that the size of R is within the stated space bounds. When a thread forks child threads, we lazily create the child threads; all the child threads that are still to be created are represented by the seed thread. Similarly, we perform all deletions lazily: every thread that terminates is simply marked in R as a *dead* thread, to be deleted in some subsequent scheduling iteration. The new state-transition diagram for the threads is shown in Figure 11. We will call all threads that are ready, or are seed or active threads as *live* threads. Note that each live thread has one entry in R. A seed is simply a compact representation for one or more ready sibling threads. For every (explicitly represented) ready or seed thread τ in R, we use a nonnegative integer $c(\tau)$ to describe its state: $c(\tau)$ equals the number of ready threads τ represents. Then for every seed τ, $c(\tau)$ is the number of child threads still to be created from v. For every ready thread in R, $c(\tau) = 1$, since it represents itself.

Consider any scheduling iteration of the lazy scheduler. Let r be the total number of ready threads represented in R after threads from Q_{in} are moved to R at the beginning of the iteration. Let $q_o = min(r, p \log p - |Q_{out}|)$ be the number of threads the scheduling iteration will move to Q_{out}. The scheduling iteration of a lazy scheduler is defined as follows.

1. Collect all the threads from Q_{in} and move them to R, that is, update their states in R.

2. Delete all the dead threads up to the leftmost $\max(2,\ q_o + 1)$ ready or seed threads.

3. Perform a prefix-sums computation on the $c(\tau)$ values of the leftmost q_o ready or seed threads to find the set C of the leftmost q_o ready threads represented by these threads. For every thread in C that is represented implicitly by a seed, create an entry for the thread in R, marking it as a ready thread. Mark the seeds for which all child threads have been created as dead.

4. Move the threads in the set C from R to Q_{out}, leaving stubs in R to mark their positions.

Consider a thread τ that is the last child thread to reach the synchronization point in a fork, and is not the rightmost child. Some of τ's siblings, which have terminated, may be represented as dead threads to its right. Since τ now represents the parent thread after the synchronization point, it has a higher 1DF-number than the dead threads to its immediate right. Thus due to lazy deletions, dead threads may be out of order in R. However, since the scheduler deletes all dead threads to the immediate right of any ready thread (or seed representing a ready thread) before it is scheduled, no descendents of the ready thread may be created until these dead threads are deleted. Thus a thread may be out of order with only the dead threads to its immediate right.

The synchronization (join) between child threads of a forking thread can be easily implemented using fetch-and-decrement operation on a synchronization counter associated with the fork. Each child that reaches the synchronization point decrements the counter by one. The last child to reach the synchronization point (the one that decrements the counter to zero) marks itself as ready, and continues as the parent; the remaining child threads mark themselves as dead.

Recall that a thread is active when it is either in Q_{out} or Q_{in}, or when it is being executed on a processor. Once a scheduling iteration empties Q_{in}, at most $p \log p + (1-\alpha)p$ threads are active. The iteration creates at most another $p \log p$ active threads before it ends, and no more threads are made active until the next scheduling step. Therefore at most $2p \log p + (1-\alpha)p$ threads can be active at any timestep, and each has one stub in R. We now prove the following bound on the time required to execute a scheduling iteration.

Lemma 6.2 *A scheduling iteration that deletes n dead threads runs in $O(n/p + \log p)$ time on αp processors.*

Proof: Let $q_o \leq p \log p$ be the number of threads the scheduling iteration must move to Q_{out}. At the beginning of the scheduling iteration, Q_{in} contains at most $2p \log p + (1-\alpha)p$ threads. Since each of these threads has a pointer to its stub in R, αp processors can move the threads to R in $O(\log p)$ time. Let τ be the $\max(2, q_o + 1)^{th}$ ready or seed thread in R (starting from the left end). The scheduler needs to delete all dead threads to the left of τ. In the worst case, all the stubs are to the left of τ in R. However, the number of stubs in R is at most $2p \log p + (1-\alpha)p$. Since there are n dead threads to the left of τ, they can be deleted from $n + 2p \log p + (1-\alpha)p$ threads in $O(n/p + \log p)$ timesteps on αp processors. After the deletions, the leftmost $q_o \leq p \log p$ ready threads are among the first $3p \log p + (1-\alpha)p$ threads in R; therefore the prefix-sums computation will require $O(\log p)$ time. Finally, q_o new child threads can be created and added in order to the left end of R in $O(\log p)$ time. Note that all deletions and additions are at the left end of R, which are simple operations in an array[10]. Thus the entire scheduling iteration runs in $O(n/p + \log p)$ time. ∎

6.3 Space and time bounds

In this section, we prove an upper bound on the space and time required by the Async-Q scheduling algorithm and the resulting schedules. We first prove the following result about the generated task graph and schedule.

[10]The additions and deletions must skip over the stubs to the left of τ, which can add at most a $O(\log p)$ delay.

Lemma 6.3 *For a parallel computation with work w and depth d, in which every unit computation allocates at most K space, the Async-Q algorithm, with $q_{max} = p \log p$, creates a task graph G with work $W = O(w)$, depth $D = O(d)$, latency-weighted depth $D_l = O(w/p + d \log p)$, and maximum task space K. The algorithm generates a Q-prioritized schedule for G with a queuing frequency of at most 2.*

Proof: As described in Section 6, the Async-Q algorithm splits each thread into a series of tasks at runtime. Let G be the resulting task graph. Since each thread is executed non-preemptively as long as it does not terminate or fork, and allocates at most a net of K units of memory, each resulting task v has a memory requirement $h(v)$ of at most K units. For each task v in G, a processor performs two unit-time accesses to the queues Q_{out} and Q_{in}. Thus the total work performed by the processor for task v is $t(v) + 2$, where $t(v) \geq 1$ and $\sum t(v) = w$. Therefore the total work of G is $O(w)$, and similarly, the depth is $O(d)$.

Next, we show that the schedule generated is a Q-prioritized schedule with a queuing frequency δ of at most 2. If r is the number of ready threads represented in R, the scheduler puts $\min(r, p \log p - |Q_{out}|)$ tasks into Q_{out}. Moreover, these tasks are the tasks with the highest priorities in R. Therefore, using Q_{out} as the FIFO work queue of maximum size $q_{max} = p \log p$, and the ordered set R to represent the set of ready tasks that are not in the work queue, the Async-Q algorithm is a Q-prioritized scheduling algorithm. With $(1-\alpha)p$ processors executing the worker computation, the resulting schedule is a Q-prioritized $(1 - \alpha)p$-schedule. The first timestep after any scheduling iteration ends is a queuing step, since all the tasks moved to Q_{out} in that iteration are available in this step. Consider any task v in G. Let t be the timestep in which the last parent u of v is completed. u is placed in Q_{in} at timestep $t + 1$. In the worst case, a scheduling iteration may end in timestep t, making $t + 1$ a queuing step. Further, the next scheduling iteration may begin in timestep $t+1$, and will result in the second queuing step. However, the next scheduling iteration must find u in Q_{in}; therefore, the third queuing step, which takes place in the timestep following the end of the this next scheduling iteration, is the timestep in which v becomes ready. Thus only two queuing step can take place after u is computed and before v becomes ready, that is, $\delta \leq 2$.

Finally, we show that $D_l = O(W/p + D \log p)$. Consider any path in G. Let l be its length. For any edge $e = (u, v)$ along the path, if u is the last parent of v, then v becomes ready at the end of at most two scheduling iterations after u is computed. Therefore the latency $l(u, v)$ is the duration of the two scheduling iterations. Let n_e and n'_e be the number of dead threads deleted by these two scheduling iterations. Then, using Lemma 6.2, $l(u, v) = O(n_e/p + n'_e/p + \log p)$. Since each thread is deleted by the scheduler at most once, a total of $O(w)$ deletions take place. Since any path in G has at most $(d - 1)$ edges, the latency weighted depth of the path is the sum of the latencies on at most d edges, which is $O(w/p + d \log p)$. ∎

We can now bound the length of the resulting schedule.

Lemma 6.4 *Let G be the task graph created by the Async-Q algorithm for a parallel computation with work w and depth d, and let $\mathcal{S}_p$ be the $(1 - \alpha)p$-schedule generated for G. If $q_{max} = p \log p$, then $|\mathcal{S}_p| = O(w/p + d \log p)$.*

Proof: We will show that $\mathcal{S}_p$ is a greedy schedule, with $O(w/p)$ additional timesteps in which the workers may be idle. Consider any scheduling iteration. Let t_i be the timestep at which the i^{th} scheduling iteration ends. After tasks are inserted into Q_{out} by the i^{th} scheduling iteration, there are two possibilities:

1. $|Q_{out}| < p \log p$. This implies that all the ready tasks are in Q_{out}, and no new tasks become ready until the end of the next scheduling iteration. Therefore, at every timestep j such that $t_i < j \leq t_{i+1}$, if m_j processors become idle and r_j tasks are ready, $\min(m_j, r_j)$ tasks are scheduled.

2. $|Q_{out}| = p \log p$. Since $(1 - \alpha)p$ worker processors will require at least $\log p/(1 - \alpha)$ timesteps to execute $p \log p$ tasks, none of the processors will be idle for the first $\log p/(1 - \alpha)$ steps after t_i.

However, if the $(i+1)^{th}$ scheduling iteration, which is currently executing, has to delete n_{i+1} dead threads, it may execute for $O(n_{i+1}/p + \log p)$ timesteps. Thus in the worst case, the processors will be busy for $\log p/(1-\alpha)$ steps and then remain idle for another $O(n_{i+1}/p + \log p)$ steps, until the next scheduling iteration ends. We call such timesteps *idling* timesteps. If the worker processors execute tasks for $\log p/(1-\alpha)$ timesteps, and then remain idle for another $O(\log p)$ timesteps, the number of such idling steps is within a constant factor of the number of steps when all workers are busy. There can be at most $w/(1-\alpha)p = O(w/p)$ steps in which the all worker processors are busy; therefore the additional $O(\log p)$ idling steps caused by the scheduling iteration can add up to at most $O(w/p)$. In addition, since each thread is deleted only once, at most w threads can be deleted. Therefore, if the $(i+1)^{th}$ scheduling iteration results in an additional $O(n_{i+1}/p)$ idle steps, they add up to $O(w/p)$ idle steps over all the scheduling iterations. Therefore, a total of $O(w/p)$ idling steps can result due the scheduler.

All timesteps besides the idling steps caused by the scheduler obey the conditions required to make it a greedy schedule, and therefore add up to $O(w)/p + D_l = O(w/p + d\log p)$ (using Lemmas 6.3 and 2.4). Along with the idling steps, the schedule requires a total of $O(w/p + d\log p)$ timesteps. ∎

We now prove an upper bound on the space requirement of the resulting schedule.

Lemma 6.5 *Let G be the task graph created by the Async-Q algorithm for a parallel computation with work w and depth d, and let $\mathcal{S}_p$ be the Q-prioritized schedule generated for G. If $q_{max} = p\log p$, then $space(\mathcal{S}_p) \leq space(\mathcal{S}_1) + O(dp\log p)$*

Proof: By Lemma 6.3, G has a depth $O(d)$, and a maximum task space K; $\mathcal{S}_p$ is a Q-prioritized schedule on $(1-\alpha)p$ processors with queuing frequency $\delta = 2$. Therefore, using Theorem 4.3, the space requirement is given as $space(\mathcal{S}_p) \leq space(\mathcal{S}_1) + ((\delta+1)\cdot q_{max} + (1-\alpha)p - 1)\cdot K \cdot D$. The required result follows using $q_{max} = p\log p$ and $\delta = 2$. ∎

We still need to prove that the space required by the scheduler for R, Q_{in} and Q_{out} does not exceed the above space requirement of the generated Q-prioritized schedule.

Lemma 6.6 *The total space required for storing threads in Q_{in}, Q_{out}, and R while executing a parallel computation of depth d on p processors is $O(dp\log p)$.*

Proof: Q_{out} may hold at most $q_{max} = p\log p$ threads at any time. Similarly, Q_{in} may hold at most $2p\log p + (1-\alpha)p$ threads, which is the maximum number of active threads. Each thread can be represented using a constant amount of space. Therefore the space required for Q_{in} and Q_{out} is $O(p\log p)$.

We now bound the space required for R. Recall that a thread is live if it is a stub, ready or seed thread. Thus R consists of live threads and dead threads. Consider any task configuration TC_i (after i timesteps) of $\mathcal{S}_p$. Since $\mathcal{S}_p$ is a Q-prioritized schedule on $(1-\alpha)p$ processors with a queuing frequency of 2, and uses $q_{max} = p\log p$, TC_i may have at most $O(dp\log p)$ premature tasks (Lemma 4.1). We call a thread a *premature thread* if at least one of its tasks that was scheduled or put on Q_{out} is premature. The number of live threads is at most the number of premature threads, plus the number of stubs (which is $O(p\log p)$), plus the number of live threads that are not premature (which is bounded by the maximum number of live threads in $\mathcal{S}_1$). The 1DF-schedule $\mathcal{S}_1$ may have at most d live threads at any timestep. Therefore, the total number of live threads after timestep i is at most $O(dp\log p)$.

The scheduler performs lazy deletions of dead threads; therefore, the number of dead threads in R must also be counted. Recall that in every scheduling iteration, the scheduler deletes at least all the dead threads up to the second ready or seed thread. There can be at most $O(dp\log p)$ premature threads in R, all of which may be dead. We will show that after the current scheduling iteration performs deletions, all dead threads in R must be premature. Let τ_1 and τ_2 be the first two ready (or seed) threads in R. Since the scheduler always deletes all dead threads up to τ_2, there can be no dead threads to the immediate right of τ_1 that may have a lower 1DF-number than τ_1, that is, τ_1 has a lower 1DF-number number than all the dead threads in R. Since all the dead threads have been executed before the ready threads represented by τ_1, they must be premature. Therefore, all dead threads in R must be premature, and are $O(dp \log p)$ in number. Note that when the next scheduling iteration begins, moving threads from Q_{in} to R does not increase the total number of entries in R, even though some live threads may be marked as dead. After this iteration performs deletions of dead threads and before it creates new entries in R, there are once again at most $O(dp \log p)$ dead threads in R. Thus at any time, the space required for R is $O(dp \log p)$. ∎

We say an algorithm *executes* a schedule if it results in an execution of a parallel computation that is represented by the schedule. We can now state the following theorem using Lemmas 6.4, 6.5 and 6.6.

Theorem 6.7 *Let $\mathcal{S}_1$ be the* 1DF*-schedule for a parallel computation with w work and d depth, in which at most a constant amount of memory is allocated at each timestep. The Async-Q algorithm, with $q_{max} = p \log p$, generates a schedule for the parallel computation and executes it on p processors in $O(w/p + d \log p)$ timesteps, requiring a total of $space(\mathcal{S}_1) + O(dp \log p)$ units of memory.* ∎

Handling arbitrarily big allocations.

So far we had assumed that consecutive unit computations can be grouped together into tasks such that no task has a memory requirement of more than a constant K units. However, there may be some unit computations that allocate more than K units; they are handled in the following manner, similar to the technique suggested in [2]. The key idea is to delay the big allocations, so that if tasks with lower 1DF-numbers become ready, they will be executed instead. Consider a unit computation of a thread that allocates m units of space ($m > K$), in a parallel computation with work w and depth d. We transform the computation by inserting a fork of m/K parallel threads before the memory allocation (see Figure 12). These new child threads do not allocate any space, or perform any work; each of them consists of a dummy task. By the time the last of these new threads gets executed, and the execution of the original parent thread is resumed, we have scheduled m/K tasks. Since m/K tasks may allocate a total of m space (which they do not in this case), the original parent thread may now proceed with the allocation of m space without exceeding our space bound. This transformation of the parallel computation increases its depth by at most a constant factor. Let S_K be the *excess allocation* in the parallel computation, defined as the sum of all memory allocations greater than K units. Then the work in the transformed computation is at most $w + S_K/K$.

Theorem 6.7 can now be generalized to allow arbitrarily big allocations of space. Note that the space and time bounds include the overheads of the scheduler.

Theorem 6.8 *Let $\mathcal{S}_1$ be the* 1DF*-schedule for a parallel computation with w work and d depth. For any constant $K \geq 1$, let S_K be the excess allocation in the parallel computation. The Async-Q algorithm, with $q_{max} = p \log p$, generates a schedule for the parallel computation and executes it on p processors in $O(w/p + S_K/pK + d \log p)$ timesteps, requiring a total of $space(\mathcal{S}_1) + O(dp \log p)$ units of memory.* ∎

7 Implementation and results

We have built a runtime system that uses the Async-Q algorithm to schedule parallel threads, and have run several experiments to analyze both the time and the memory required by parallel computations. In this section we briefly describe the implementation of the system, followed by the experimental results.

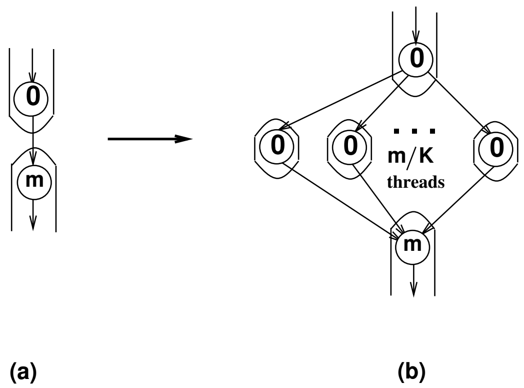


Figure 12: A transformation of the parallel computation to handle a large allocation of space within a unit computation without violating the space bound. The nodes represent unit computations within threads; the threads are sequences of such nodes. Consecutive nodes of a thread are grouped into tasks, shown as outlines around the nodes. Each node is labeled with the amount of memory its computational step allocates. When a thread needs to allocate m space ($m > K$), we insert m/K parallel child threads before the allocation. Each child thread consists of a dummy node that does not allocate any space. After these child threads complete execution, the original allocation can be performed.

The runtime system has been implemented on the SGI Power Challenge, which has a shared memory architecture with processors and memory connected via a fast shared-bus interconnect. Since the number of processors on this architecture is not very large, a serial implementation of the scheduler does not create a bottleneck in our runtime system. Therefore, for simplicity, we have implemented the scheduler as a serial computation, and the worker processors take turns executing it. The parallel programs executed on this system have been explicitly hand-coded in the continuation-passing style, similar to the code generated by the Cilk preprocessor [5]. Each continuation points to a C function representing the next computation of a thread, and a structure containing all its arguments. These continuations are created and moved between the queues. A worker processor takes a continuation off the out-queue, and simply applies the function pointed to by the continuation, to its arguments. The high-level program is broken into such functions at points where it executes a parallel fork, a recursive call, or a memory allocation.

We implemented five parallel programs on our runtime system: blocked recursive matrix multiplication, Strassen's matrix multiplication [33], the Fast Multipole Method for the n-body problem [20], sparse matrix vector multiplication and the ID3 algorithm for building decision trees [27]. The implementation of these programs is described in appendix A, along with the problem sizes we used in our experiments.

7.1 Time performance

The above programs were executed on an 8-processor Power Challenge. Figure 13 shows the speedups for the above programs for up to 7 processors (this was the maximum number of processors available for dedicated use). The speedup for each program is with respect to its efficient serial C version, which does not use our runtime system. Since the serial C program runs faster than our runtime system running on a single processor, the speedup shown for one processor is less than 1. However, for all the programs, it is close to 1, implying that the overheads in our system are low. The timings on our system include the delay introduced before large allocations, in the form of m/K dummy threads for an allocation of m bytes. We have used a value of $K = 1000$ bytes in our experiments. Figure 13 also shows the speedups for the same programs running on an existing space-efficient system Cilk [5]; the time performance on our system is comparable with that on Cilk.

Figure 14 shows the breakup of the running time for one of the programs, blocked recursive matrix

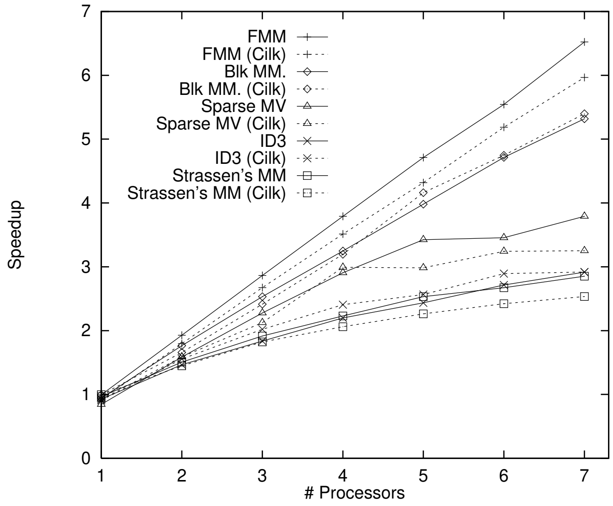


Figure 13: The speedups achieved on up to 7 processors of a Power-Challenge machine, using a value of K=1000 bytes. The speedup on p processors is the time taken for the serial C version of the program divided by the time for our runtime system to run it on p processors. For each application, the solid line represents the speedup on our system, while the dashed line represents the speedup on the Cilk [5] system. The low overheads of our runtime system account for the speedups for $p = 1$ being marginally lower than 1.

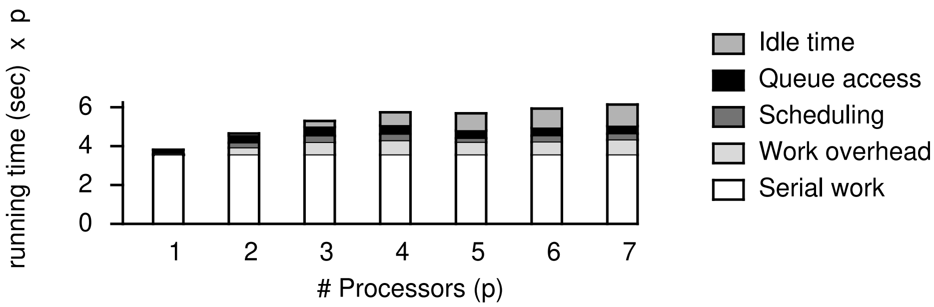


Figure 14: The total processor time (the running time multiplied by the number of processors p) for blocked recursive matrix multiplication. "Serial work" is the time taken by a single processor executing the equivalent serial C program. For ideal speedups, all the other components would be zero. The other components are overheads of the parallel execution and the runtime system. "Idle time" is the total time spent waiting for threads to appear in the out-queue; "queue access" is the total time spent by the worker processors inserting threads into Q_{in} and removing them from Q_{out}. "Scheduling" is the total time spent as the scheduler, and "work overhead" includes overheads of creating continuations, building structures to hold arguments, executing dummy threads, and (de)allocating memory from a shared pool of memory, as well as the effects of cache misses and bus contention.

multiplication. The results show that the percentage of time spent waiting for threads to appear in the out-queue increases as the number of processors increases, as should be expected. A parallel implementation of the scheduler will be more efficient for a larger number of processors.

For nested parallel loops, we group iterations of the innermost loop into equal-sized chunks, provided it does not contain calls to any recursive functions[11]. Scheduling a chunk at a time improves performance by reducing scheduling overheads and providing good locality, especially for fine-grained iterations. We are currently working on ways to incorporate coarsening in the algorithm itself, instead of relying on compiler support.

7.2 Space performance

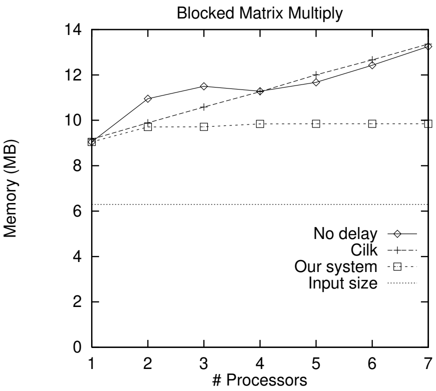
Figure 15 shows the memory usage for each application. Here we compare three implementations for each program. One implementation is on the existing Cilk [5] system. The other two implementations are on our scheduling system; in one case, large memory allocations are delayed by inserting dummy nodes, and in the other case they are allowed to proceed immediately. For the programs we have implemented, the version without the delay results in approximately the same space requirements as would result from scheduling the outermost level of parallelism. For example, in Strassen's matrix multiplication, our algorithm without the delay would allocate temporary space required for p branches at the top level of the recursion tree before reverting to the execution of the subtree under the first branch. On the other hand, scheduling the outer parallelism would allocate space for the p branches at the top level, and then execute each subtree serially. Hence we use our algorithm without the delay to estimate the memory requirements of previous techniques like [14, 23], which schedule the outer parallelism with higher priority. Cilk uses less memory than this estimate due to its use of randomization: an idle processor steals the topmost thread (representing the outermost parallelism) from the private queue of a randomly picked processor; this thread may not represent the outermost parallelism in the entire computation. Previous techniques like [8, 9, 19] use a strategy similar to that of Cilk. The results show that when big allocations are delayed with dummy nodes, our algorithm results in a significantly lower memory usage in all the programs. We have not compared it to naive scheduling techniques, such as breadth-first schedules, which have much higher memory requirements.

As mentioned above, the number of dummy threads introduced depends on the number of bytes K that a thread is allowed to allocate without being preempted. Hence there is a trade-off between memory usage and running time. For example, Figure 16 shows how the running time and memory usage for blocked recursive matrix multiplication are affected by K. For all the programs we implemented, the trade-off curves looked similar, and a value of $K = 1000$ bytes resulted in a good balance between space and time performance. The curve may vary for other parallel programs; however, the user can experimentally choose a suitable value to satisfy her space or time requirements.

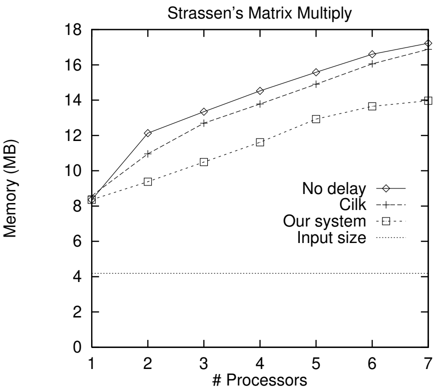
8 Summary and discussion

We have presented a model for non-preemptive parallel computations based on a class of DAGs called task graphs, and described how executions of the computations can be represented by schedules of such graphs. We defined two classes of schedules: prioritized schedules and Q-prioritized schedules, and proved a bound on their space requirements. The Async-Q scheduling algorithm presented in this paper generates Q-prioritized schedules; we bounded the space and time requirements of the schedules generated by the algorithm. Finally, we described an implementation of a runtime system that uses the Async-Q algorithm to schedule threads in a multithreaded computation. The results of implementing five parallel programs on

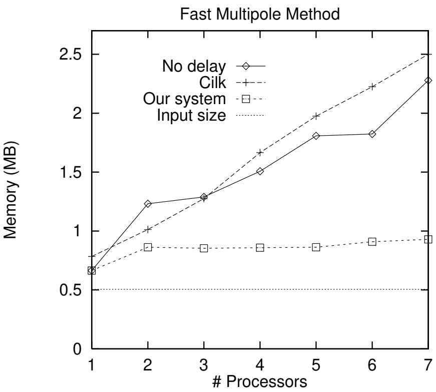
[11] It should be possible to automate such coarsening with compiler support.



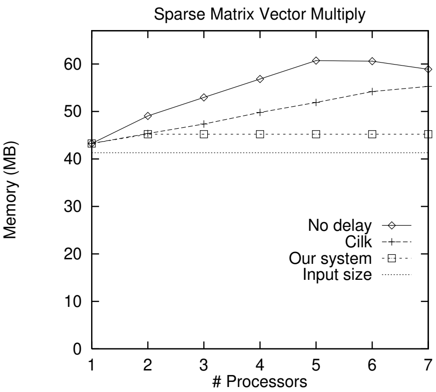
(a)



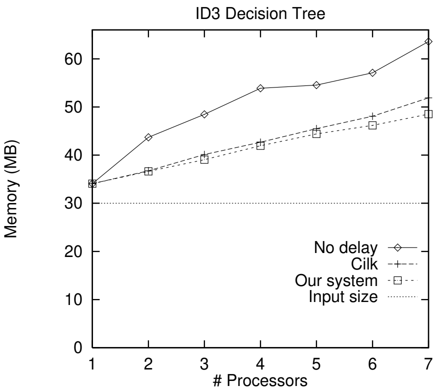
(b)



(c)



(d)



(e)

Figure 15: The memory requirements of the parallel programs. For $p = 1$ the memory usage shown is for the serial C version. We compare the memory usage of each program when the big memory allocations are delayed by inserting dummy threads (using $K = 1000$), with when they are allowed to proceed without any delay, as well as with the memory usage on Cilk [5]. The version without the delay on our system (labeled as "No delay") is an estimate of the memory usage resulting from previous scheduling techniques. These results show that delaying big allocations significantly changes the order of execution of the threads, and results in much lower memory usage, especially as the number of processors increases.

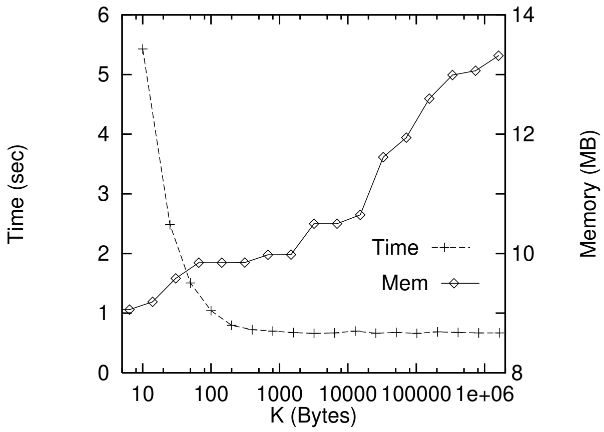


Figure 16: The variation of running time and memory usage with K (in bytes) for multiplying two 512×512 matrices using blocked recursive matrix multiplication on 7 processors. For very small K, many dummy threads are inserted, which results in a high running time. For very large K, very few dummy threads are inserted, causing only a small delay before a big memory allocation, resulting in high memory usage. K=500-2000 bytes results in both good performance and low memory usage.

this system demonstrate that our approach is more effective in reducing space usage compared to previous scheduling techniques, and at the same time, yields good parallel performance.

We are currently considering methods to further improve the scheduling algorithm, particularly to provide better support for fine-grained computations. At present, fine-grained iterations of innermost loops are statically grouped into fixed-size chunks. A dynamic, decreasing-size chunking scheme such as [24, 25, 34] can be used instead. We are considering ways of automatically introducing such coarsening at runtime through the scheduling algorithm; for example, by allowing the execution order to differ to a limited extent from the order dictated by the 1DF-numbers. We also plan to reduce contention due to a shared queue by introducing multiple queues; the processors will access separate queues at any given time. We finally plan to implement faster parallel calls to efficiently execute fine-grained computations.

References

[1] Arvind, Rishiyur S. Nikhil, and Keshav K. Pingali. I-structures: Data structures for parallel computing. *ACM Transactions on Programming Languages and Systems*, 11(4):598–632, October 1989.

[2] G. E. Blelloch, P. B. Gibbins, and Y. Matias. Provably efficient scheduling for languages with fine-grained parallelism. In *Proceedings of the 1995 ACM Symposium on Parallel Algorithms and Architectures*, Santa Barbara, July 1995. ACM.

[3] Guy E. Blelloch, Siddhartha Chatterjee, Jonathan C. Hardwick, Jay Sipelstein, and Marco Zagha. Implementation of a portable nested data-parallel language. *Journal of Parallel and Distributed Computing*, 21(1):4–14, April 1994.

[4] Guy E. Blelloch and John Greiner. A provable time and space efficient implementation of nesl. In *ACM SIGPLAN International Conference on Functional Programming*, pages 213–225, May 1996.

[5] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou. Cilk: An efficient multithreaded runtime system. In *Proceedings of the Fifth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 207–216, November 1995.

[6] R. D. Blumofe and C. E. Leiserson. Space-efficient scheduling of multithreaded computations. In *Proc. 25th ACM Symp. on Theory of Computing*, pages 362–371, May 1993.

[7] R. D. Blumofe and C. E. Leiserson. Scheduling multithreaded computations by work stealing. In *Proc. 35th IEEE Symp. on Foundations of Computer Science*, pages 356–368, November 1994.

[8] F. W. Burton. Storage management in virtual tree machines. *IEEE Trans. on Computers*, 37(3):321–328, 1988.

[9] F. W. Burton and D. J. Simpson. Space efficient execution of deterministic parallel programs. Manuscript, December 1994.

[10] F. W. Burton and M. R. Sleep. Executing functional programs on a virtual tree of processors. In *Conference on Functional Programming Languages and Computer Architecture*, October 1981.

[11] Rohit Chandra, Anoop Gupta, and John Hennessy. COOL: An object-based language for parallel programming. *IEEE Computer*, 27(8):13–26, August 1994.

[12] K. M. Chandy and C. Kesselman. Compositional c++: Compositional parallel programming. In *Proc. 5th. Intl. Wkshp. on Languages and Compilers for Parallel Computing*, pages 124–144, New Haven, CT, August 1992.

[13] J. S. Chase, F. G. Amador, and E. D. Lazowska. The amber system: Parallel programming on a network of multiprocessors. In *Proceedings of the Twelfth ACM Symposium on Operating Systems Principles*, December 1989.

[14] J. H. Chow and W. L. Harrison III. Switch-stacks: A scheme for microtasking nested parallel loops. In *Proceedings of Supercomputing*, pages 190–199, New York, NY, November 1990.

[15] D. E. Culler and Arvind. Resource requirements of dataflow programs. In *Proceedings of the 15th Annual International Symposium on Computer Architecture*, May 1988.

[16] E.G.Coffman, Jr., editor. *Computer and job-shop scheduling theory*. John Wiley & Sons, New York, 1976.

[17] John T. Feo, David C. Cann, and Rodney R. Oldehoeft. A report on the Sisal language project. *Journal of Parallel and Distributed Computing*, 10(4):349–366, December 1990.

[18] V. W. Freeh, D. K. Lowenthal, and G. R. Andrews. Distributed filaments: efficient fine-grain parallelism on a cluster of workstations. In *First Symposium on Operating Systems Design and Implementation*, pages 201–212, Monterey, CA, November 1994.

[19] S. C. Goldstein, D. E. Culler, and K. E. Schauser. Enabling primitives for compiling parallel languages. In *Third Workshop on Languages, Compilers, and Run-Time Systems for Scalable Computers*, Rochester, NY, May 1995.

[20] L. Greengard. *The rapid evaluation of potential fields in particle systems*. The MIT Press, 1987.

[21] High Performance Fortran Forum. *High Performance Fortran language specification*, May 1993.

[22] W. E. Hseih, P. Wang, and W. E. Weihl. Computation migration: enhancing locality for distributed memory parallel systems. In *Proceedings of the Fourth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, San Francisco, California, May 1993.

[23] S. F. Hummel and E. Schonberg. Low-overhead scheduling of nested parallelsim. *IBM Journal of Research and Development*, 35(5-6):743–65, 1991.

[24] S. F. Hummel, E. Schonberg, and L. E. Flynn. Factoring: a method for scheduling parallel loops. *Communications of the ACM*, 35(8):90–101, Aug 1992.

[25] C. D. Polychronopoulos; D.J. Kuck. Guided self-scheduling: a practical scheduling scheme for parallel supercomputers. *IEEE Transactions on Computers*, C-36(12):1425–39, Dec 1987.

[26] Peter H. Mills, Lars S. Nyland, Jan F. Prins, John H. Reif, and Robert A. Wagner. Prototyping parallel and distributed programs in Proteus. Technical Report UNC-CH TR90-041, Computer Science Department, University of North Carolina, 1990.

[27] J. R. Quinlan. Induction of decision trees. *Machine learning*, 1(1):81–106, 1986.

[28] Jr. R. H. Halstead. Multilisp: A language for concurrent symbolic computation. *ACM Trans. on Programming Languages and Systems*, 7(4):501–538, 1985.

[29] M. C. Rinard, D. J. Scales, and M. S. Lam. Jade: A high-level, machine-independent language for parallel programming. *IEEE Computer*, June 1993.

[30] A. Rogers, M. Carlisle, J. Reppy, and L. Hendren. Supporting dynamic data structures on distributed memory machines. *ACM Transactions on Programming Languages and Systems*, 17(2):233–263, March 1995.

[31] R.S.Nikhil. Cid: A parallel, shared-memory c for distributed memory machines. In *Proc. 7th. Ann. Wkshp. on Languages and Compilers for Parallel Computing*, pages 376–390, August 1994.

[32] C. A. Rugguero and J. Sargeant. Control of parallelism in the manchester dataflow machine. In *Functional Programming Languages and Computer Architecture*, volume 174 of *Lecture Notes in Computer Science*, pages 1–15. Springer-Verlag, 1987.

[33] V. Strassen. Gaussian elimination is not optimal. *Numerische Mathematik*, 13:354–356, 1969.

[34] T. H. Tzen and L. M. Ni. Trapezoid self-scheduling: a practical scheduling scheme for parallel compilers. *IEEE Transactions on Parallel and Distributed Systems*, 4(1):87–98, Jan 1993.

A Parallel programs implemented on our runtime system

In this section, we describe the implementation of the programs used to demonstrate the effectiveness of our scheduling algorithm. The five programs are:

- *Blocked recursive matrix multiply.* This program multiplies two dense $n \times n$ matrices using a simple recursive divide-and-conquer method, and performs $O(n^3)$ work. The recursion stops when the blocks are down to the size of 64x64, after which the standard row-column matrix multiply is executed serially. This algorithm significantly outperforms the row-column matrix multiply for large matrices (e.g., by a factor of over 4 for 1024×1024 matrices) because its use of blocks results in better cache locality. At each step, the eight recursive calls are made in parallel. Each recursive call needs to allocate temporary storage, which is deallocated before returning from the call. The results reported are for the multiplication of two 512×512 matrices of double-precision floats.

- *Strassen's matrix multiply.* The DAG for this algorithm is very similar to that of the blocked recursive matrix multiply, but performs only $O(n^{2.807})$ work and makes seven recursive calls at each step [33]. Once again, a simple serial matrix multiply is used at the leaves of the recursion tree. The sizes of matrices multiplied were the same as for the previous program.

- *Fast multipole method (FMM).* This is an n-body algorithm that calculates the forces between n bodies in $O(n)$ work [20]. We have implemented the most time-consuming phases of the algorithm, which are a bottom-up traversal of the octree followed by a top-down traversal. In the top-down traversal, the forces on the cells in a level of the octree, due to their neighboring cells are calculated in parallel. For each cell, the forces over all its neighbors are also calculated in parallel, for which temporary storage needs to be allocated. This storage is freed when the forces over the neighbors have been added to get the resulting force on that cell. With two levels of parallelism, the structure of this code looks very similar to the pseudocode described in Section 1. We executed the FMM on a uniform octree with 4 levels (8^3 leaves), using 5 multipole terms to calculate the forces.

- *Sparse matrix-vector multiplication.* This multiplies an $m \times n$ sparse matrix with a $n \times 1$ vector. The dot product of each row of the matrix with the vector is calculated to get the corresponding element of the resulting vector. There are two levels of parallelism: over each row of the matrix, and over the elements of each row multiplied with the corresponding elements of the vector to calculate the dot product. For our experiments, we used $m = 12$ and $n = 800000$, and 30% of the elements were non-zeroes. (Using a large value of n provides sufficient parallelism within a row, but using large values of m leads to a very large size of the input matrix, making the amount of dynamic memory allocated in the program negligible in comparison.)

- *ID3.* This algorithm by Quinlan [27] builds a decision tree from a set of training examples in a top-down manner, using a recursive divide-and-conquer strategy. At the root node, the attribute that best classifies the training data is picked, and recursive calls are made to build subtrees, with each subtree using training examples with a particular value of that attribute. Each recursive call is made in parallel, and the computation of picking the best attribute at a node, which involves counting the number of examples in each class for different values for each attribute, is also parallelized. Temporary space is allocated to store the subset of training examples used to build each subtree, and is freed once the subtree is built. In our experiments we built a tree from 1.5 million test examples, each with 4 multi-valued attributes.