A Concurrent Dynamic Task Graph *

Theodore Johnson
Dept. of Computer and Inf. Science
University of Florida
Gainesville, Fl 32611-2024
ted@cis.ufl.edu

Abstract

Task graphs are used for scheduling tasks on parallel processors when the tasks have dependencies. If the execution of the program is known ahead of time, then the tasks can be statically and optimally allocated to the processors. If the tasks and task dependencies aren't known ahead of time (the case in some analysis-factor sparse matrix algorithms), then task scheduling must be performed on the fly. We present simple algorithms for a concurrent dynamic-task graph. A processor that needs to execute a new task can query the task graph for a new task, and new tasks can be added to the task graph on the fly. We present several alternatives for allocating tasks for processors and compare their performance.

1 Introduction

A common method for expressing parallelism is through a task graph. Each node in the task graph represents a unit of work that needs to be performed, and edges represent dependencies between tasks. If there is an edge from task T_1 to task T_2 in the task graph, then T_1 must complete before T_2 can begin. Previous work [3, 17, 11, 10, 20, 9, 12] has assumed that the task graph is specified ahead of time. This is often a reasonable assumption, since the task graph can be generated by a parallelizing compiler, or by a symbolic analysis of the problem to be solved (i.e, analysis-only LU decomposition algorithms).

Since the task graph is specified ahead of time, it can be analyzed for static scheduling purposes. The scheduling can be static or dynamic. In *static* scheduling, the tasks are allocated to the processors before the computation starts [3, 17, 10, 20]. In *dynamic* scheduling, the tasks are allocated to processors on the fly [9, 12]. If good task execution time estimates can be made in advance, static scheduling will outperform dynamic scheduling, but dynamic scheduling will adjust to the actual execution conditions.

In this paper, we propose a scheduling structure, the *dynamic-task graph*, or *DTG*, that allows the task graph to be specified during the program execution. A DTG is useful when the structure of the problem instance is determined at execution time. This work was motivated by the problem of parallelizing analysis-factor LU decomposition algorithms for asymmetric sparse matrices. A common approach is the multi-frontal method, in which portions of the matrix are gathered into fronts for factoring, and these fronts make contributions to other fronts. The tasks in the DTG represent the fronts, and the links represent the contributions passed between fronts. In some analysis-factor multifrontal algorithms [4, 5], the tasks and their dependencies are determined during execution.

In section 2, we present the basic concurrent DTG algorithm, and some extensions. In section 3, we explore algorithms for scheduling eligible tasks, and in section 4 we examine some performance issues. Finally, in section 5 we draw our conclusions.

2 Concurrent Dynamic-Task Graph

A *dynamic-task graph*, or DTG, consists of a set of labeled vertices V and a set of arcs on the vertices A. The arcs in A are the dependencies among the tasks. If the arc $(t1, t2)$ is in A, then task $t1$ must complete execution before task $t2$ can start execution. We call $t1$ the *prerequisite* task, and $t2$ the *dependent* task. Obviously, the DTG must be an acyclic digraph. The nodes correspond to tasks, and are labeled:

U : if the task is unexecuted,
E : if the task is executing,
F : if the task has finished execution, or
N : if the task is not ready.

*In the 1993 Int'l Conf. on Parallel Processing

A task t_0 is *eligible* for execution only if all tasks t_i such that $(t_i, t_0) \in A$ are labeled F (similar to a *mature* node in [17]).

There are three operations on a DTG:

1 **add_task(T,D)**: The add_task operation adds task T to the DTG, and specifies that the set of tasks $D = \{t_1, \ldots, t_n\}$ must finish execution before T can start execution. The set D is task T's *dependency set*.

2 **t=get_task()**: The get_task operation returns a task that is eligible for execution. If there is no eligible task, the processor blocks until a task becomes eligible.

3 **finished_task(T)**: The finished_task operation declares that task T has completed its execution.

When a task is added to the DTG, it must be uniquely named. This requirement is not a problem, since a processor can name the task it adds to the DTG with a sample from a local counter appended to its processor id, or with a pointer to a description of the task. The application might naturally provide a unique task name. For example, in a sparse matrix solver the name of the task can be the row of initial pivot of the frontal matrix.

When a task is added to the DTG (via the `add_task` operation), it is labeled **U**. When a task is selected for execution, its label is changed to **E**. When a task completes its execution, it performs the `finished_task` operation, which changes the task state to **F**. If in the `add_task` operation, task t specifies that it is dependent on task t' but t' has not yet been added to the DTG, then task t must create an entry for t' and specify that t' is *not ready* by setting the state of t' to **N**. When `add_task(t',D)` is executed, the state of t' changes to **U**.

We initially assume that all tasks in a task's dependency set have already been added to the DTG, and later extend our algorithms to handle not-ready tasks. Since all tasks in the DTG have been determined, this assumption is reasonable. Furthermore, it lets us reclaim tasks from the DTG. Whenever a task finishes execution, it is dropped from the DTG. If an `add_task` operation can't find a task $t_d \in D$ in the DTG, then t_d has finished.

Figure 1 illustrates a sample execution sequence. Tasks $T1$, $T2$, $T3$, and $T4$ are added to the DTG, and task $T4$ depends on tasks $T1$ and $T2$. Next, tasks $T1$ and $T2$ are selected for execution in response to `get_task` requests. Task $T1$ finishes, and its state changes to **F**. Task $T3$ is selected for execution in response to a `get_task` request. Task $T2$ finishes, and task $T4$ becomes eligible.

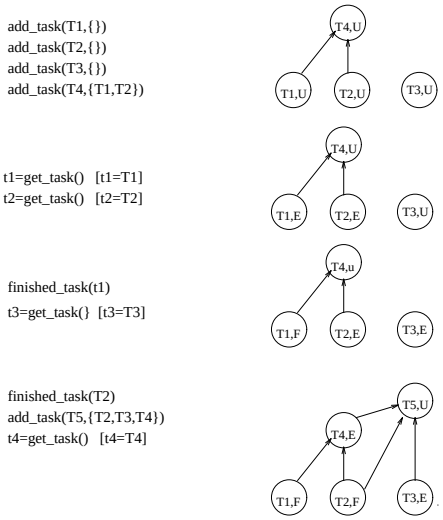

Figure 1: Example execution sequence

A task is represented by a *task record* in the DTG. The task record contains a field for the *name* of the task, information necessary for executing the task, the number of unfinished prerequisite tasks `ND`, and a list of the dependent tasks `dependent`.

The first issue to address is the problem of finding the tasks in the DTG without explicitly searching the DTG. Since the queries that need to be made are simple look-up queries, a hash table is the best data structure to maintain the translation table. We use a static-sized open hash table. The primary purpose of the hash table is to permit parallel access to the tasks in the DTG, so the number of hash table buckets only needs to be proportional to the number of processors (as opposed to the number of tasks). If the DTG has few buckets, the buckets of the DTG might be required to store many task records, so the records should be stored in some fast access structure, such as a binary tree. The bucket data structure doesn't need to be a concurrent data structure, instead the entire bucket can be locked. The hash table operations are:

1 **enter_task(T)**: put a new task in the hash table.

```
a lock hash table bucket
b insert T into bucket.
c unlock hash table bucket.
```

2 **p=translate_task(T)**: search the hash table for the task, and return a pointer to the task.

```
a lock hash table bucket.
b find and lock T.
  i If T is not found, release all locks and
    return NIL.
c unlock hash table bucket.
```

3 **delete_task(T)**: remove T from the hash table.

```
a lock hash table bucket.
b find and lock T.
c remove T from bucket.
d unlock T and hash table bucket.
```

The next issue is where to store the dependency pointers: in the prerequisite task record or in the dependent task record. Storing the dependency pointer in the dependent task record simplifies the operation of adding a task, but greatly complicates the operation of getting an eligible task. We choose the option of storing the pointers in the prerequisite task records. This choice requires that when a task is added to the DTG, all tasks in the dependency set must be modified. Fortunately, the hash table permits a fast lookup.

The last issue is finding tasks that are eligible for execution. We assume that pointers to these tasks are stored in a separate data structure, the *eligible queue*. A task can be inserted into the eligible queue, and the eligible queue can be queried for an operation to execute. We leave the implementation and the semantics of the eligible queue unspecified for now, since there are many possible allocation heuristics.

The psuedo-code for the add_task, get_task, and delete_task operations follows. We assume that the tasks and the hash table buckets can be locked (we later discuss a non-locking implementation). The lock can be a simple busy-wait lock, or the contention-free MCS lock [15]. Each task graph entry t has three fields: a field for the lock, a count of the number of unfinished prerequisite tasks (`ND`), and a list of tasks that depend on t (`dependent`). When a `translate_task` operation is performed, the lock on the hash table entry is retained. This ensures that the task remains in the task graph until it is modified (in spite of the concurrent execution of `finished_task` operation). The data structures for the concurrent dynamic-task graph are shown in Figure 2.

```
add_task(T,D)
  T.ND=0
  enter_task(T)
  for i=1 to |D| do
    t=translate_task(i'th task in D)
    if t is null  // i.e., finished
      number_finished++
    else
      add T to t->pointers
      unlock(t)
  if number_finished > 0
    lock(T)
    T.ND -= number_finished
    if T.ND == 0
      add T to the eligible queue
```

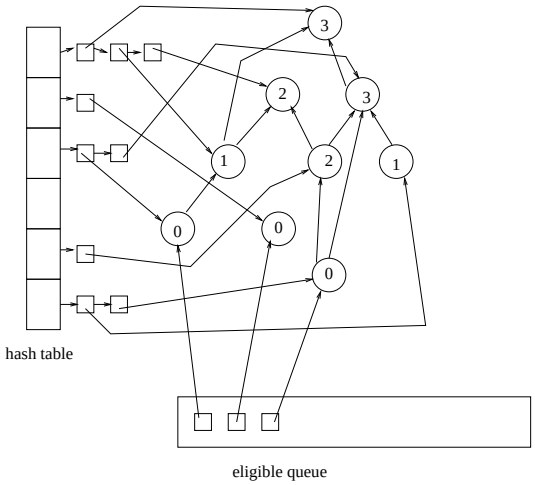

Figure 2: Concurrent task graph data structures

```
get_task()
  get a task t from the eligible queue
  return(t)

finished_task(T)
  delete_task(T) // from hash table only
  for i = 1 to |T.dependent|
    t = i'th task in T.dependent
    lock(t)
    t->ND- -
    if t->ND == 0
      add t to the eligible queue
  reclaim the space used by T
```

2.1 Correctness

In this section, we make some correctness arguments. In particular we show that a task is added to the eligible queue before all of its dependent tasks complete the `finished_task` operation, but after all of its dependent tasks start the `finished_task` operation.

We will say that task T *exists in* DTG $\mathcal{D}$ between the times that `add_task(T)` and `delete_task(T)` are executed. This period of time is well defined because of the locks which make the hash table operations atomic. When T executes the `translate_task` operation for t in the `for` loop of the `add_task` operation, we say that T *accesses* t.

Lemma 1 *The* `add_task(T,D)` *inserts a dependency link from* t *to* T *if and only if* $t \in$ D *and* t *exists in* $\mathcal{D}$ *when* T *accesses* t.

Proof: The `for` loop in the `add_task` operation scans through the tasks $t \in$ D and adds a dependency pointer to t if and only if t is in the hash table when T accesses t. Since t is in the hash table if and only if t exists in $\mathcal{D}$, the lemma follows.□

Lemma 2 *When a task completes, it decrements the dependency count of all tasks that add a dependency pointer to it.*

Proof: When a task completes, it executes the `finished_task` procedure. The `finished_task` operation removes the completed task's record from the hash table, so no further pointers are added to it. The finished_task operation is atomic due to the locks it sets, so after the `finish_task` operation, a tasks record will contain all dependency pointers that are inserted. The remainder of the `finish_task` operation decrements the dependent tasks. □

Theorem 1 *Every task is added to the eligible queue exactly once, and only after all prerequisite tasks are marked F.*

Proof: Consider a new task T executing the `add_task` operation. Task T accesses every $t \in$ D(T). By lemma 1, T adds a pointer to t if $t \in \mathcal{D}$. If $t \notin \mathcal{D}$, then t must be finished, since we have assumed that all tasks in D(T) have been added to $\mathcal{D}$ already. Therefore, T correctly counts the number dependent tasks that are finished when accessed, and the remaining tasks receive a pointer to T.

The `add_task` operation decrements T's dependency count by the number of finished dependent tasks. By Lemma 2, the unfinished dependent tasks decrement t's dependency count by one. The use of locks makes the decrement atomic. Since the task graph is a DAG, all prerequisite tasks of T will finish. The last task to perform the decrement will find that the dependency count is zero, and will add T to the eligible queue. □

2.2 Extensions

In this section, we discuss some possible extensions and optimizations of the concurrent DTG.

Not-Ready Tasks The algorithms that we presented depend on the assumption that all tasks in a new task's dependency set exist in the DTG. One can imagine that a new task might depend on tasks that have not yet been added to the DTG. For example, a parallel algorithm for the LU factorization of sparse asymmetric matrices might assign the task of adding pivots to frontal matrices to one set of processors, and assign the task of composing and factoring the frontal matrices to a different set [4]. Processors p and q might build frontal matrices A and B concurrently, where elements of B depend on the factorization of A. Since A and B are built concurrently, B might be added to the DTG before A.

To distinguish between not-ready and finished tasks the state of a task is explicitly stored in the task. A finished task is retained in the DTG and is marked **F**. When a task $t \in D(T)$ is accessed in the `for` loop of the `add_task` operation, the following protocol is used: If t doesn't exist in the hash table, a task record for t is created, its state is set to N, and a pointer to T is added. If a record for t exists in the hash table, its state is tested to determine whether or not the task has finished. The `enter_task` hash table operation must be modified to account for the possibility that the task T already exists as a not-ready task.

Dense Task Names The concurrent DTG requires a hash table if the range of task names is large and the names of the actual tasks is sparse. In some applications, the tasks in the DTG are relatively dense in their name space. An example are frontal matrices in an asymmetric sparse matrix algorithm. The task can be named by the row of the upper left hand pivot, so there are $O(n)$ possible task names. Sparse matrix algorithms contain several $O(n)$ supplementary data structures, so allocating a bucket for each possible task name does not create an excessive space overhead. Allocating a bucket for each task greatly simplifies the implementation of the DTG, since the hash table operations become simple $O(1)$ procedures. In addition, the bucket lock serves as the task record lock, so only half the number of locks need to be set as would otherwise be needed.

Non-locking Algorithms A non-locking algorithm uses atomic read-modify-write instead of locking to ensure correctness in spite of concurrent accesses. Non-locking algorithms have the attractive property that they avoid busy-waiting, which can degrade performance [1, 6]. These algorithms typically use the *compare_and_swap* or the *compare_and_swap double* instruction to commit modifications [8, 16, 18, 19], although some algorithms use the fetch-and-add instruction [7]. The correctness of the DTG algorithms depends on the atomicity of the hash table operations. Fortunately, many practical non-locking list and search structure algorithms exist in the literature [16, 19].

One place where care must be taken involves access to a tasks list of dependent tasks. The `finished_task`

operation should declare that a task is finished with a decisive operation, so that the correctness of Lemma 2 is maintained. We can modify the technique of Prakash, Lee and Johnson by maintaining the list of dependent tasks in a task record as a non-locking stack, and use one bit of the pointer to the head of the stack as a *deleted* bit. The `finished_task` operation sets the deleted bit as its decisive operation, and a `add_task` operation that reads a set deleted bit is a task's dependent list considers the task to be finished.

3 Eligible Queue

The eligible queue is responsible for scheduling tasks for execution. The goal in designing the eligible queue is to maximize the speedup of the parallel computation. Maximizing the speedup requires that we minimize the blocking that occurs at the `get_task` procedure, and that we minimize the overhead of the scheduler. However, minimizing blocking and minimizing overhead are conflicting goals. As previous works have shown, careful scheduler design can reduce response time by reducing the amount of time that a processor is blocked waiting for a task to become eligible for execution. However, optimal scheduling is NP-complete, and the best heuristics require that the entire task graph be known in advance and require considerable overhead. Scheduling overhead also reduces performance, and should be kept to a minimum.

In order to provide guidance on implementing an eligible queue, we ran simulation experiments to compare scheduling algorithm performance. We wrote a simulation of a parallel dynamic-task graph execution. The simulation is initialized with 80 initial tasks. An additional 80 tasks are created, each with dependencies on the preceding tasks. These tasks have a random number of dependencies (a truncated normal distribution with a mean of 4.0 and standard deviation of 5.2), and a prerequisite task is chosen by subtracting a randomly chosen *backwards distance* from the new task's number. The backwards distance has a truncated Erlang distribution with a mean of 80. Given this initial task graph, p processors execute by repeatedly getting an eligible task, finishing the task, and adding a random number of new tasks. The number of new tasks added has a binomial distribution with mean 2 for the first 2000 tasks to execute, and mean .5 for the remainder. We set the task execution time so that the scheduling overhead has little effect.

We tested six eligible queue algorithms:

FIFO: The eligible queue is managed as a FIFO queue.

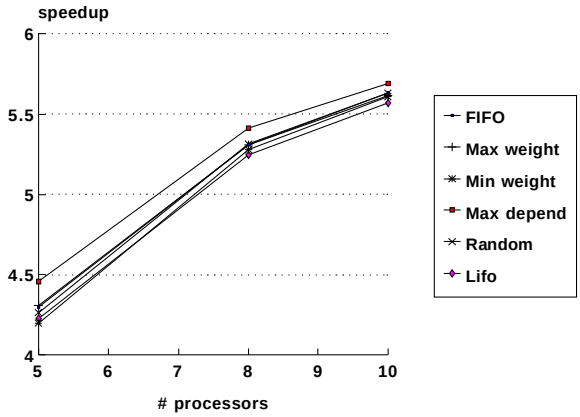

Figure 3: Comparison of DTG scheduler performance.

Max weight: The eligible queue always returns the task with the greatest execution time in response to a `get_task` operation.

Min weight: The eligible queue returns the task with the minimum execution time.

Max depend: The eligible queue returns the task with the maximum number of dependent tasks.

Random: The eligible queue returns a uniformly randomly chosen task.

LIFO: The eligible queue is maintained as a LIFO queue.

Figure 3 shows a comparison of the speedup produced by each algorithm given 5, 8, and 10 processors. To collect a data point, we used the average of ten runs. Figure 3 shows that Max depend is consistently the best algorithm, while LIFO is consistently the worst. However, no algorithm is significantly better than another. The summary in in Table 1 shows that there is only about a 5% difference in the speedup offered by the best and the worst algorithm.

In Table 2, we list the speedup from static-task graph algorithms. We collected the dynamic-task graph by recording the dependency edges, and added a dependency link from task T to all tasks that T creates. We simulated the following algorithms on the equivalent static-task graphs using eight processors:

CP: Critical path method [17]. An eligible node's priority is the weight of the heaviest weighted path to any exit node.

FIFO: Schedule tasks in the order that they become available.

Largecalc: Schedule the heaviest task first [2].

Scheduler	5 Proc.	8 Proc.	10 Proc.
FIFO	4.297	5.311	5.632
Maxweight	4.309	5.309	5.616
Minweight	4.198	5.280	5.610
Maxdepend	4.456	5.414	5.690
Random	4.263	5.315	5.631
LIFO	4.227	5.246	5.631

Table 1: Comparison of DTG scheduler performance.

Scheduler	speedup
Max	7.072
CP	7.036
FIFO	5.470
Largecalc	5.470
Heavy	5.456
Levelfifo	5.474
Levellarge	5.492
Maxdep	5.440

Table 2: Comparison of static-task graph scheduler performance (eight processors).

Heavy: An eligible node's priority is the sum of its weight and the weight of its immediate successors [2].

Levelfifo: Assign BFS levels to tasks, and schedule the tasks within a level by FIFO.

Levellarge: Assign BFS levels to tasks, and schedule the heaviest task in a level first [17].

Maxdep: Schedule the task with the greatest number of dependent tasks first [2].

Comparing Tables 1 and 2, we note that there is little difference among most of the dynamic and static scheduling policies. While the best speedup from the static-task graphs heuristics is considerably greater than that provided by the DTG scheduling algorithms, the only static-task graph scheduler that provides this performance is CP. The remaining static-task graph scheduling policies provide a speedup similar to that of the DTG scheduling policies. Thus, there is little performance difference between local scheduling on a static-task graph and scheduling on a dynamic-task graph. We note that the CP method requires $O(n^2)$ time for processing, and may not always be reasonable.

Since there is little difference in DTG scheduler performance, we recommend a simple or low-overhead method. For example, [16] presents a simple lock-free FIFO queue which can serve as the eligible queue. Manber [14] has proposed *concurrent pools* as a low overhead method for implementing shared queues that doesn't require a FIFO ordering.

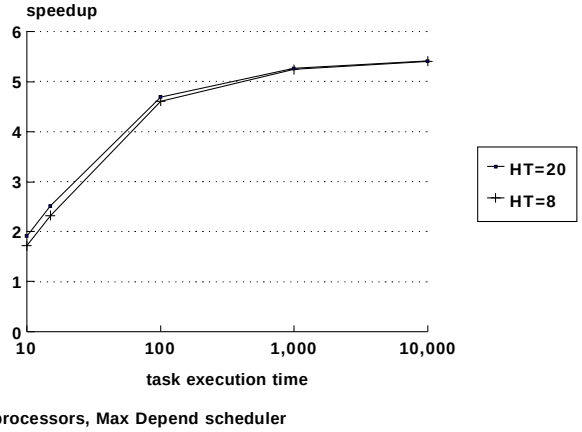

Figure 4: Speedup vs. task execution time.

4 Performance

In this section, we investigate the effect of DTG overhead on speedup. Figure 4 shows a plot of speedup versus task execution time. The simulator and parameters are the same as those used in the previous section, but 8 processors are used and the task execution time varied. In addition, we ran one set of experiments with 8 buckets in the hash table, and another set with 20 buckets. The leftmost point on the chart is a close approximation to the speedup obtainable without scheduler overhead. If S_{max} is the speedup without scheduler overhead, each task executes for E seconds, and the total time to process a task in the DTG is O, then the speedup is approximately

$$S_{actual} = S_{max} \frac{E}{E+O}$$

In the simulations, each task was dependent on about 4.0 prerequisite tasks, and adding each task pointer required 2 time units. Each finished task was required to access about 4.0 tasks, each access requiring 2 time units. The `add_task` procedure required 3 time units to modify the tasks's own record, and the `finished_task` procedure required 1. Finally, each task was added to and later removed from the eligible queue, and both of these actions required 1 time unit. In total, processing each task required about 22 time units of DTG overhead. In addition, there was a certain mbount of overhead due to lock contention.

When the task execution time is 10 units, the formula for S_{actual} predicts a speedup of about 1.70. The

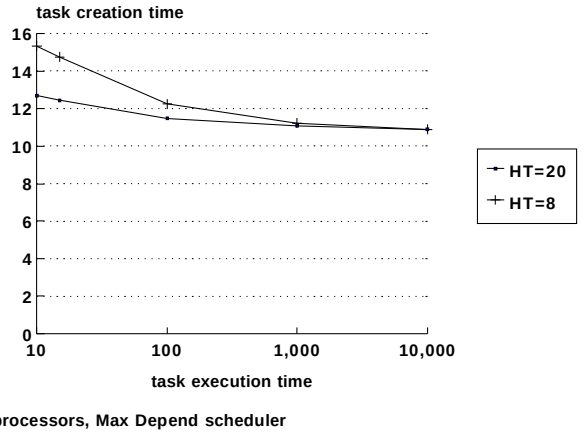

Figure 5: Add_task execution time vs. task execution time.

observed speedup is somewhat greater, because the increased scheduler overhead replaces some of the task blocking. In general the formula holds, and we see that the DTG is appropriate for medium to coarse grain parallelism, but not for fine grain parallelism.

In Figure 4, we show the speedup for two cases: when the hash table contains 8 entries and when the hash table contains 20 entries. When the task execution time is small, the smaller hash table shows a slightly smaller speedup. The reduction in speedup is due to the increased lock contention, As a result, the overhead for processing a task in the DTG is greater. Figure 5 shows a plot of the time to execute the `add_task` operation against the task execution time. When the task size becomes moderately large, Figure 4 shows that the difference in speedup becomes negligible.

We develop a performance model to determine an appropriate size for the hash table. Let

D: be the average number of task dependencies,
E: be the average task execution time,
T_q: be the average time to access the eligible queue,
T_a: be the average time to access a task,
S: be the speedup, and
H: be the number of hash table entries.

Suppose that we model each hash table bucket as an M/M/1 queue (a crude but workable approximation). We first calculate the arrival rate. Each dependency link causes two task accesses: one in the `add_task` operation and one in the `finished_task` operation. Since every task is added once and finishes once, we assume that a processor cycles between executing and accessing the DTG. After executing a task, a processor declares that the task is finished (requiring $D+1$ task accesses) and adds one task to the eligible queue, adds a new task to the DTG ($D+1$ task accesses), and gets a new task from the eligible queue. On average, S processors are executing, and the task graph accesses are hashed among H buckets. Therefore, the arrival rate at a bucket is:

$$\lambda_b = \frac{2S(D+1)}{E + 2((D+1)T_a + T_q)} * \frac{1}{H}$$

The time to execute a task access is T_a, so

$$\mu_b = 1/T_a$$

The waiting time at an M/M/1 queue is [13]

$$W = \lambda/(1-\rho)$$

where $\rho = \lambda/\mu$. Therefore

$$W_b = \frac{2(D+1)S}{H(E + 2(D+1)T_a + 2t_q) - 2(D+1)ST_a}$$

To find the minimum number of hash table buckets to support a speedup of S, we solve $\rho = 1$ for H and find

$$H_{min} = \frac{2(D+1)ST_a}{H(E + 2(D+1)T_a + 2T_q)}$$

and a rule of thumb is to use at least enough buckets so that $\rho < .5$, or

$$H_{half} = \frac{4(D+1)ST_a}{H(E + 2(D+1)T_a + 2T_q)}$$

We used the parameters from the example in Figure 5 to generate Table 3. The analytical model is uniformly optimistic. The DTG accesses in the simulation are very non-uniform, being much heavier in the initial part of the simulation. For $E = 10$, the analytical model recommends at least 6.85 buckets, and for $E = 100$, the model recommends at least 1.83 buckets.

	analytical	simulation
8 buckets, E=10	.374	.755
20 buckets, E=10	.103	.267
8 buckets, E=100	.065	.197
20 buckets, E=100	.024	.0723

Table 3: comparison of analytical and simulation predictions of bucket waiting times.

5 Conclusion

In this paper, we present algorithms for performing *dynamic-task graph scheduling*. We present a concurrent data structure, the *concurrent dynamic-task graph* that allows the task graph of the computation to be specified while the parallel computation proceeds. Such a capability is useful for certain classes of parallel computations, such as analysis-factor LU factorizations of asymmetric sparse matrices. We study some aspects of the performance of the DTG algorithms. We conclude that simple scheduling strategies work well, that the DTG is appropriate for medium to coarse grained computations, and provide a rule of thumb for determining the number of hash table buckets for the DTG.

Acknowledgements We'd like to than Steve Hadfield for his help and for the use of his static-task graph scheduler.

References

[1] R. Anani. LR-algorithm: Concurrent operations on priority queues. In *Proceedings of the Second IEEE Symposium on Parallel and Distributed Processing*, pages 22–25, 1990.

[2] C.D. Polychronopolous U. Bannerjee. Processor allocation for horizontal and vertical parallelism and related speedup bounds. *IEEE Transactions on Computers*, C-36:410–420, 1987.

[3] E.G. Coffman. *Computer and Job-Shop Scheduling*. John Wiley and Sons, 1976.

[4] T. A. Davis and P. C. Yew. A nondeterministic parallel algorithm for general unsymmetric sparse LU factorization. *SIAM J. Matrix Anal. Appl.*, 11(3):383–402, 1990.

[5] T.A. Davis and I.S. Duff. Unsymmetric-pattern multifrontal methods for parallel sparse LU factorization. Technical report, University of Florida, Dept. of CIS TR-91-23, 1991. Available at anonymous ftp site cis.ufl.edu:cis/tech-reports.

[6] R.R. Glenn, D.V. Pryor, J.M. Conroy, and T. Johnson. Characterizing memory hotspots in a shared memory mimd machine. In *Supercomputing '91*. IEEE and ACM SIGARCH, 1991.

[7] A. Gottlieb, B. D. Lubachevsky, and L. Rudolph. Basic techniques for the efficient coordiantion of very large numbers of cooperating sequential processors. *ACM Trans. on Programming Languages and Systems*, 5(2):164–189, 1983.

[8] M. Herlihy. A methodology for implementing highly concurrent data structures. In *Proceeding of the Second ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 197–206. ACM, 1989.

[9] J. Ji and M. Jeng. Dynamic task allocation on shared memory multiprocessor systems. In *ICPP*, pages I:17–21, 1990.

[10] H. Kasahara and S. Narita. Practical multiprocessor scheduling algorithms for efficient parallel processing. *IEEE Trans. on Computers*, C-33:1023–1029, 1984.

[11] D. Klappholz and S. Narita. Practical multiprocessor scheduling algorithms for efficient parallel processing. *IEEE Transactions on Computers*, C-33:315–321, 1984.

[12] D. Klappholz and H.C. Park. Parallelized process scheduling for a tightly-coupled mimd machine. In *Int'l Conf. on Parallel Processing*, pages 315–321, 1984.

[13] L. Kleinrock. *Queueing Systems*, volume 1. John Wiley, New York, 1975.

[14] W. Massey. A probabilistic analysis of a database system. In *ACM SIGMETRICS Conference on Measuring and Modeling of Computer Systems*, pages 141–146, Aug. 1986.

[15] J.M. Mellor-Crummey and M.L. Scott. Algorithms for scalable synchronization on shared-memory multiprocessors. *ACM Trans. Computer Systems*, 9(1):21–65, 1991.

[16] S. Prakash, Y.H. Lee, and T. Johnson. A non-blocking algorithm for shared queues using compare-and-swap. In *Proc. Int'l Conf. on Parallel Processing*, pages II68–II75, 1991.

[17] Shirazi, Wang, and Pathak. Analysis and evaluation of heuristic methods of static task scheduling. *Journal of Parallel and Distributed Computing*, 10:222–232, 1990.

[18] J. Turek, D. Shasha, and S. Prakash. Locking without blocking: Making lock based concurrent data structure algorithms nonblocking. In *ACM Symp. on Principles of Database Systems*, pages 212–222, 1992.

[19] J.D. Valois. Towards Practical Lock-free Data Structures. Submitted for publication, 1993.

[20] Z. Yin, C. Chui, R. Shu, and K. Huang. Two precedence-related task-scheduling algorithms. *Int'l Journal of High Speed Computing*, 3(3):223–240, 1991.