ELSEVIER

Parallel Computing 22 (1996) 327–333

PARALLEL COMPUTING

Short communication

A concurrent dynamic task graph †

Theodore Johnson [a,*], Timothy A. Davis [a], Steven M. Hadfield [b]

[a] *Department of Computer and Information Sciences, University of Florida, Gainesville, Fl 32611-6120, USA*

[b] *Department of Mathematical Sciences, US Air Force Academy, CO, USA*

Received 5 August 1994; revised 30 December 1994, 28 July 1995, 15 September 1995

Abstract

Task graphs are used for scheduling tasks on shared memory parallel processors when the tasks have dependencies. If the program's task graph is known ahead of time, then the tasks can be statically and optimally allocated to the processors. If the tasks and task dependencies are not known ahead of time (the case in some sparse matrix algorithms), then task scheduling must be performed dynamically. We present a simple algorithm 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 dynamically.

Keywords: Shared memory multiprocessors; Task graph scheduling; Multifrontal method; Sparse matrices; Lu factorisation

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 [2,7,9,10,11,15,16] has assumed that the task graph is specified ahead of time. This is often a reasonable assumption, since the task graph can often be generated by a parallelizing compiler, or by an a-priori analysis of the problem to be solved.

If 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,

* Corresponding author. Email: ted@cis.ufl.edu

† This project is supported the National Science Foundation (ASC-9111263, DMS-9223088).

SSDI 0167-8191(95)00061-5

the tasks are allocated to the processors before the computation starts [2,9,15,16]. In *dynamic* scheduling, the tasks are allocated to processors during the computation [7,11]. 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 the unsymmetric-pattern multifrontal method for the LU factorization of unsymmetric sparse matrices [3,5,6].

2. Concurrent dynamic-task graph

A *dynamic-task graph* $\mathscr{D} = (\mathscr{V}, \mathscr{A})$, or DTG, consists of a set of labeled vertices $\mathscr{V}$ and a set of arcs on the vertices $\mathscr{A}$. The arcs in $\mathscr{A}$ are the dependencies among the tasks. If the arc (t_1, t_2) is in $\mathscr{A}$, then task t_1 must complete execution before task t_2 can start execution. We call t_1 the *prerequisite* task, and t_2 the *dependent* task. Obviously, the DTG must be an acyclic directed graph. 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 yet defined.

A task t_0 is *eligible* for execution only if all tasks t_i such that $(t_i, t_0) \in \mathscr{A}$ are labeled **F** (similar to a *mature* node in [15]).

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. 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 multifrontal sparse matrix solver the name of the task can be the row index of initial pivot of the frontal matrix [3].

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 yet defined* 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-yet-defined 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 cannot find a task $t_d \in D$ in the DTG, then t_d has finished.

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`.

A DTG operation needs to find the tasks in the DTG without an explicit search. 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 binary tree. The bucket data structure does not 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 = `tanslate_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 (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.

We store the dependency pointers in the record of the prerequisite task. 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

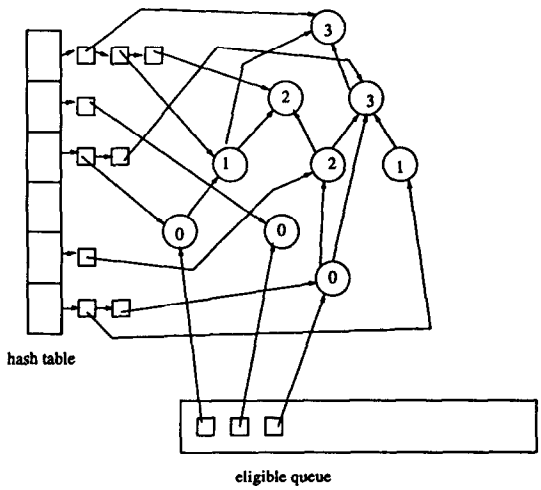

Fig. 1. Concurrent task graph data structures.

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 alternatives.

The pseudo-code for the `add_task`, `get_task`, and `delete_task` operations follows. The lock can be a simple busy-wait lock, or the contention-free MCS lock [13]. 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 a `finished_task` operation). The data structures for the concurrent dynamic-task graph are shown in Fig. 1.

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

```
    t.ND-=number_finished
    if t.ND==0
      add t to the eligible queue
    unlock(t)
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 |
    s=i 'th task in t.dependent
    lock(s)
    s->ND--
    if s->ND==0
      add s to the eligible queue
  reclaim the space used by t
```

2.1 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. While this assumption is usually safe, it might not hold if new tasks are generated in parallel. Two tasks, t_1 and t_2 might be created in concurrently, where t_2 depends on t_1, but t_2 is added to the DTG first.

To distinguish between not-yet-defined 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 $s \in D(t)$ is accessed in the for loop of the `add_task` operation s does not exist in the hash table, a task record for s is created, its state is set to **N**, and a pointer to t is added. If a record for s 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-yet-defined 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 unsymmetric multifrontal sparse matrix algorithm [3]. The task can be named by the row of the upper left hand pivot, so there are 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.

3. Eligible queue

The DTG operations require a scheduler to select among different eligible tasks; making the best scheduling decision is NP-complete. The best heuristics are based on global information about the task graph. Since our task graph is constructed dynamically, we are constrained to select among limited-knowledge schedulers, which are not based on global information. Our studies indicate that a wide range of limited-knowledge schedulers provide sufficient performance.

We simulated the execution of the DTG with several limited-knowledge schedulers, including FIFO, Maximum task weight, and Maximum dependencies. To drive the simulator, we used both synthetic traces and traces derived from the DAG of a parallel sparse matrix factorization algorithm [6]. We found that the best DTG scheduler algorithms yielded speedups within 5% of each other, and within 10% of limited-knowledge schedulers on static task graphs. We therefore recommend a method with low overhead such as a simple lock-free FIFO queue [14], or Manber's concurrent pools [12].

We also examined the effect of varying the number of buckets in the hash table. We found that setting the number of buckets to the number of processors always gave good performance. Further increasing the number of buckets did not increase performance substantially.

4. Conclusion

In this paper, we present a concurrent data structure suitable for performing *dynamic-task graph scheduling*. We present the *concurrent dynamic-task graph*, which 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 LU factorization of unsymmetric sparse matrices.

References

[1] R. Ayani, LR-algorithm: Concurrent operations on priority queues, in: *Proc. Second IEEE Symp. Parallel and Distributed Processing*, (1990) 22–25.

[2] E.G. Coffman, *Computer and Job-Shop Scheduling* (John Wiley, 1976).

[3] T.A. Davis and I.S. Duff, An unsymmetric-pattern multifrontal method for sparse LU factorization, to appear in *SIAM J. Matrix Analysis and Applications*. See also University of Florida, Dept. of CIS report TR-94-038.

[4] I.S. Duff, R.G. Grimes and J.G. Lewis, Sparse matrix test problems, *ACM Trans. Math. Softw.* 15 (1989) 1–14.

[5] S.M Hadfield, On the LU factorization of sequences of identically structured sparse matrices within a distributed memory environment, PhD thesis, Computer and Information Sciences Department, University of Florida, Gainesville, FL (also TR-94-019), 1994. Available via anonymous ftp to ftp.cis.ufl.edu:cis/tech-reports.

[6] S.M. Hadfield and T.A. Davis, Potential and achievable parallelism in the unsymmetric-pattern multifrontal LU factorization method for sparse matrices, in: *Proc. Fifth SIAM Conf. on Applied Linear Algebra* (1994) 387–391.

[7] J. Ji and M. Jeng, Dynamic task allocation on shared memory multiprocessor systems, in: *ICPP* (1990) I: 17–21.

[8] T. Johnson, A concurrent dynamic task graph, Technical Report TR93-011, Computer and Information Sciences Department, University, of Florida, Gainesville, FL 1994. Available via anonymous ftp to ftp.cis.ufl.edu:cis/tech-reports.

[9] H. Kasahara and S. Narita, Practical multiprocessor scheduling algorithms for efficient parallel processing, *IEEE Trans. Comput.* C-33 (1984) 1023–1029.

[10] D. Klappholz and S. Narita, Practical multiprocessor scheduling algorithms for efficient parallel processing, *EEE Trans. Comput.* C-33 (1984) 315–321.

[11] D. Klappholz and H.C. Park, Parallelized process scheduling for a tightly-coupled mimd machine, in: *Int. Conf. on Parallel Processing* (1984) 315–321.

[12] U. Manber, On maintaining dynamic information in a concurrent environment, *SIAM J. Comput.* 15(4) (1986) 1130–1142.

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

[14] S. Prakash, Y. H. Lee and T. Johnson, A non-blocking algorithm for shared queues using compare-and-swap; in: *Proc. Int. Conf. on Parallel Processing* (1991) II68–II75.

[15] Shirazi, Wang and Pathak, Analysis and evaluation of heuristic methods of static task scheduling, *J. Parallel and Distributed Comput.* 10 (1990) 222–232.

[16] Z. Yin, C. Chui, R. Shu and K. Huang, Two precedence-related task-scheduling algorithms, *Int. J. High Speed Comput.* 3(3) (1991) 223–240.