Executing Task Graphs Using Work-Stealing

Kunal Agrawal
Washington University in St Louis
St Louis, MO 63130, USA

Charles E. Leiserson Jim Sukha
Massachusetts Institute of Technology
Cambridge, MA 02139, USA

***Abstract*— NABBIT is a work-stealing library for execution of task graphs with arbitrary dependencies which is implemented as a library for the multithreaded programming language Cilk++. We prove that NABBIT executes static task graphs in parallel in time which is asymptotically optimal for graphs whose nodes have constant in-degree and out-degree. To evaluate the performance of NABBIT, we implemented a dynamic program representing the Smith-Waterman algorithm, an irregular dynamic program on a two-dimensional grid. Our experiments indicate that when task-graph nodes are mapped to reasonably sized blocks, NABBIT exhibits low overhead and scales as well as or better than other scheduling strategies. The NABBIT implementation that solves the dynamic program using a task graph even manages in some cases to outperform a divide-and-conquer implementation for directly solving the same dynamic program. Finally, we extend both the NABBIT implementation and the completion-time bounds to handle dynamic task graphs, that is, graphs whose nodes and edges are created on the fly at runtime.**

***Keywords*-Cilk, dag, dynamic multithreading, parallel computing, work/span analysis.**

I. INTRODUCTION

Many parallel-programming problems can be expressed using a ***task graph***: a directed acyclic graph (dag) $\mathcal{D} = (V, E)$, where every node $A \in V$ represents some task with computation $\textsc{Compute}(A)$, and a directed edge $(A, B) \in E$ represents the constraint that B's computation depends on results computed by A. ***Executing*** a task graph means assigning every node $A \in V$ to a processor to execute at a given time and executing $\textsc{Compute}(A)$ at that time such that every predecessor of A has finished its computation beforehand. A ***schedule*** of $\mathcal{D}$ is the mapping of nodes of V to processors and execution times.

Task graphs come in two flavors. A ***static***[1] task graph is one where the structure of the task graph is given, and a ***dynamic*** task graph means that the nodes and edges are created on the fly at runtime. For example, Johnson *et al.* [12] describe an interface for a dynamic task graph, where new task nodes can be added or deleted dynamically as the task graph is being executed.

The problem of finding a minimum-time schedule on P processors is known to be NP-complete [20], even for static task graphs where the compute times are known in advance. As Kwok and Ahmad [14] describe in their survey of scheduling of static task graphs with known compute times, however, many efficient approximation algorithms and heuristics exist in a variety of computational models.

For task graphs where compute times are not known in advance, one must use a ***dynamic scheduler*** — one that makes decisions at runtime — to efficiently load-balance the computation. Most dynamic schedulers for generic task graphs rely on a ***task pool***, a data structure that dynamically maintains a collection of ***ready*** tasks whose predecessors have completed. Processors remove and work on ready tasks, posting new tasks to the task pool as dependencies are satisfied. Using task pools for scheduling avoids the need for accurate time estimates for the computation of each task, but maintaining a task pool may introduce runtime overheads.

One way to reduce the runtime overhead of task pools is to impose additional structure on the task graphs so that one can optimize the task-pool implementation. For example, Hoffman, Korch, and Rauber [10], [13] describe and empirically evaluate a variety of implementations of task pools in software and hardware. They focus on the case where tasks have hierarchical dependencies, i.e., a parent task depends only on child tasks that it creates. In their evaluation of software implementations of task pools, they observe that distributed task pools based on dynamic "task stealing" perform well and provide the best scalability.

Dynamic task-stealing is closely related to the ***work-stealing*** scheduling strategy used in parallel languages such as Cilk [3], [9], Cilk++ [11], [15], Fortress [1], X10 [7], and parallel runtime libraries such as Hood [5] and Intel Threading Building Blocks [18]. A work-stealing scheduler maintains a distributed collection of ready queues where processors can post work locally. Typically, a processor finds new work from its own work queue, but if its work queue is empty, it ***steals*** work from the work queue of another processor, typically chosen

This work was supported in part by the National Science Foundation under Grant 0615215.

[1] Some of the existing literature on task-graph scheduling uses the term "static" to mean task graphs that are static in our sense, but also where the time needed to execute each task is known before the computation begins.

at random. Blumofe and Leiserson [4] provided the first work-stealing scheduling algorithm coupled with an asymptotic analysis showing that their algorithm performs near optimally.

These languages and libraries all support fork-join constructs for parallelism, allowing a programmer to express series-parallel task graphs easily. They do not support task graphs with *arbitrary* dependencies, however. To do so, the programmer must maintain additional state to enforce dependencies that are not captured by the fork-join control flow of the program. Furthermore, depending on how the programmer enforces these dependencies, the theorems that guarantee the theoretical efficiency of work-stealing no longer apply.

In this paper, we explore how to schedule task graphs in work-stealing environments. Our contributions are as follows:

- The NABBIT library for Cilk++, which provides a programming interface for specifying arbitrary static task graphs, and which can execute these task graphs using conventional work-stealing. Since NABBIT does not modify the Cilk++ language or runtime system, it can be adapted to work with any fork-join language that uses work-stealing.
- Theoretical bounds on the time NABBIT requires to execute task graphs.
- An extension of the NABBIT implementation and its foundational theory to dynamic task graphs.

For both static and dynamic task graphs, we prove that on P processors, NABBIT executes task graphs in time which is asymptotically optimal when nodes have constant in-degree and out-degree.

There are four important advantages to using NABBIT for executing task graphs:

Low contention: Since work-stealing is a distributed scheduling strategy, NABBIT exhibits lower contention than centralized task-pool schedulers.

Economy of mechanism: Many languages and libraries already implement work-stealing. By using NABBIT, these mechanisms can be used directly to schedule arbitrary task graphs.

Interoperability: Each node in the task graph can represent an arbitrary computation, including a parallel computation. Since Cilk++ can automatically schedule these computations, NABBIT makes it easy to exploit not only the parallelism among the different dag nodes, but also possible fork-join parallelism within the COMPUTE function of each dag node.

Robustness: Work-stealing schedulers "play nicely" in multiprogrammed environments. Fork-join languages such as Cilk++ usually execute computations on P worker threads, with one thread assigned to each processor. If the operating system deschedules a worker, the worker's work is naturally stolen away to execute on active workers. Arora *et al.* [2] provide tight asymptotic bounds on the performance of work-stealing when workers receive different amounts of processor resource from the operating system.

The remainder of this paper is organized as follows. Section II describes the interface and the implementation of NABBIT. Section III provides a theoretical work/span analysis of performance of NABBIT. Section IV describes an irregular dynamic-programming application and presents experimental results indicating that NABBIT performs well on this kind of application. Section V presents extensions to NABBIT to support dynamic task graphs, and Section VI presents a synthetic benchmark on randomly generated dags that evaluates the library's performance for both static and dynamic task graphs.

II. THE NABBIT TASK-GRAPH LIBRARY

NABBIT is a library for executing task graphs with arbitrary dependencies. This section introduces NABBIT by describing the interface and implementation of ***static* NABBIT**, a library optimized to execute static task graphs. We discuss extending NABBIT to handle dynamic task graphs later in Section V.

Interface

In NABBIT, programmers specify task graphs by creating nodes that extend from a base DAGNODE object, specifying the dependencies of each node, and providing a COMPUTE method for each node.

As a concrete example, consider a dynamic program on an $n \times n$ grid, which takes an $n \times n$ input matrix s and computes the value $M(n, n)$ based on the following recurrence:

$$M(i,j) = \max \begin{cases} M(i-1,j) + s(i-1,j) \\ M(i,j-1) + s(i,j-1) \end{cases} \tag{1}$$

Figure 1 illustrates how one can formulate this problem as a task graph. The code constructs a node for every cell $M(i, j)$, with the node's class extending from a base DAGNODE class. The programmer uses two methods of this base class: ADDDEP specifies a predecessor node on which the current node depends, and EXECUTE tells NABBIT to execute a task graph using the current node A (with no predecessors) as a source node. The EXECUTE method calls COMPUTE on the current node A and recursively on all the successors of A that are enabled.

In the example from Figure 1, the COMPUTE method for each task graph node is a short, serial section of code. Since we implemented NABBIT using Cilk++ without modifying the Cilk++ runtime, programmers can use `cilk_spawn` and `cilk_for` to expose additional parallelism within a node's COMPUTE method.

```
class DPDag {
  int n; int* s; MNode* g;
  DPDag(int n_, int* s_): n(n_), s(s_) {
    g = new MNode[n*n];
    for (int i = 0; i < n; ++i) {
      for (int j = 0; j < n; ++j) {
        int k = n*i+j;
        g[k].pos = k; g[k].dag = (void*) this;
        if (i > 0) {g[k].AddDep(&MNode[k-n])};
        if (j > 0) {g[k].AddDep(&MNode[k-1])};
  } } }
  int Execute() { g[0]->Execute(); }
};

class MNode: public DAGNode {
  int res;
  void Compute() {
    this->res = 0;
    for (int i = 0; i < predecessors.size(); i++) {
      MNode* pred = predecessors.get(i);
      int pred_val = pred->res + s[pred->pos];
      res = MAX(pred_val, res);
} } };
```

Figure 1. Cilk++ code that uses NABBIT to solve the dynamic program in Equation (1). The code constructs a task-graph node for every cell $M(i, j)$.

COMPUTEANDNOTIFY(A)
1 COMPUTE(A)
2 **parallel for** all $B \in A.successors$
3 $\quad val =$ ATOMDECANDFETCH$(B.join)$
4 $\quad$ **if** $val == 0$
5 $\quad\quad$ COMPUTEANDNOTIFY(B)

Figure 2. Pseudocode for NABBIT operating on a static task graph. COMPUTEANDNOTIFY computes a node A and then greedily spawns the computation for any immediate successors of A which are enabled by A's computation. The iterations of line 2 are spawned in binary-tree fashion, and all can potentially run in parallel.

Implementation

Our implementation of static NABBIT maintains the following fields for each task-graph node A:

- **Successor array:** An array of pointers to A's immediate successors in the task graph.
- **Join counter:** A variable whose value tracks the number of A's immediate predecessors that have not completed their COMPUTE method.
- **Predecessor array:** An array of pointers to A's immediate predecessors in the task graph, i.e., the nodes on which A depends.[2]

To execute a task graph $\mathcal{D}$, NABBIT calls the COMPUTEANDNOTIFY method in Figure 2 on the source node of $\mathcal{D}$.

[2] NABBIT maintains a predecessor array for each node A only to allow users to conveniently access the nodes on which A depends inside A's COMPUTE method (e.g., so that A can aggregate results from its predecessors). Maintaining this array is not always necessary. For example, for the code in Figure 1, one can also find the predecessors of a node through pointer and index calculations.

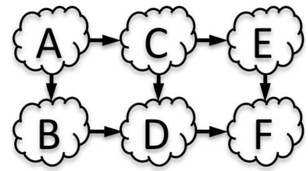

Figure 3. A task graph for computing $M(2, 3)$ using Recurrence (1). The execution of COMPUTEANDNOTIFY(A) is nondeterministic. The method COMPUTEANDNOTIFY(A) recursively calls two methods COMPUTEANDNOTIFY(B) and COMPUTEANDNOTIFY(C), but in a particular execution, only one, but not both of these methods recursively calls COMPUTEANDNOTIFY(D).

III. ANALYSIS OF PERFORMANCE

This section provides a theoretical analysis of the performance of the NABBIT library when executing a static task graph on multiple processors. To analyze the runtime of NABBIT, we employ a work/span analysis (as in [6], Chapter 27), and calculate upper bounds on the work and span[3] of the executions of the code in Figure 2. Then, we translate these bounds into completion-time bounds for NABBIT using known theoretical bounds on the completion time of fork-join parallel programs scheduled with randomized work-stealing [2], [4].

Definitions

Consider a task graph $\mathcal{D} = (V, E)$. Conceptually, each node $A \in V$ maintains a list $in(A)$ of immediate predecessors and a list $out(A)$ of immediate successors. Let $outDeg(A) = |out(A)|$ and $inDeg(A) = |in(A)|$ be the out- and in-degrees of A, respectively. For simplicity in stating the results, assume that for a task graph $\mathcal{D}$, every node is a successor of a unique ***source*** node s with no incoming edges and a predecessor of a unique ***sink*** node t with no outgoing edges. Let $paths(A, B)$ be the set of all paths in $\mathcal{D}$ from node A to node B.

Every execution of a task graph invokes COMPUTEANDNOTIFY(A) for each $A \in V$ exactly once. For many task graphs, such as the one in Figure 3, the execution of COMPUTEANDNOTIFY can be nondeterministic, since COMPUTE(A) may be invoked by a different predecessor depending on the underlying scheduling. Each possible execution can be represented as a computation dag [6, p. 777] $\mathcal{E}$, which should not be confused with the task graph itself. The nodes of the computation dag are serial chains of executed instructions, and the edges represent dependencies between them.

We define several notations for subgraphs of a computation dag. For a particular computation dag $\mathcal{E}$ and a task-graph node A, let $CN^{\mathcal{E}}(A)$ be the subgraph corresponding to the call COMPUTEANDNOTIFY(A),

[3] "Span" is sometimes called "critical-path length" and "computation depth" in the literature.

and let $com^{\mathcal{E}}(A)$ be the subgraph corresponding to COMPUTE(A). For any subgraph $\mathcal{E}'$ of a computation dag, we define the ***work*** of $\mathcal{E}'$ to be the sum of the execution times of all the nodes in $\mathcal{E}'$, which we denote by $W(\mathcal{E}')$. We define the ***span*** of $\mathcal{E}'$ to be the longest execution time along any path $\mathcal{E}'$, which we denote by $S(\mathcal{E}')$. We overload notation so that when the superscript $\mathcal{E}$ is omitted, we mean the maximum of the quantity over all computation dags. For example, $W(CN(A))$ denotes the maximum work for COMPUTEANDNOTIFY(A) over all possible computation dags.

To analyze NABBIT's running time on a task graph $\mathcal{D}$ with source s, let us examine the execution of COMPUTEANDNOTIFY(s). The total work done by a computation $\mathcal{E}$ of $\mathcal{D}$ is $W(CN^{\mathcal{E}}(s))$, and the span is $S(CN^{\mathcal{E}}(s))$. Since the computation dag is nondeterministic, we shall analyze the maximum of these values — namely, $W(CN(s))$ and $S(CN(s))$ — over all possible computation dags and use them as upper bounds in our analyses.

Work analysis

Lemma 1: Any execution of $\mathcal{D}$ using NABBIT has work $W(CN(s))$ at most

$$\sum_{A \in V} W(com(A)) + O(|E|) + O(C_W) \ ,$$

where

$$C_W = \sum_{B \in V} inDeg(B) \cdot \min\{inDeg(B), P\} \ .$$

Proof: The first term arises from the work of the COMPUTE functions. The second term bounds the work of traversing $\mathcal{D}$, assuming no contention. The third term covers the contention cost on the join counter. For each node B, its join counter is decremented $inDeg(B)$ times, and each decrement waits at most $O(\min\{inDeg(B), P\})$ time. ■

Span analysis

The nondeterministic nature of the computation complicates the direct calculation of $S(CN(s))$. Consequently, our strategy is to construct a new, deterministic computation dag $\mathcal{E}^*$, whose span is an upper bound on the span of COMPUTEANDNOTIFY(s), and analyze that. We define the method COMPUTEANDNOTIFY$^*(A)$ to be the same as the original method, except that line 4 in Figure 2 is omitted. In other words, COMPUTEANDNOTIFY$^*(A)$ always makes recursive calls on all of A's successors. Let $CN^*(A)$ be the computation dag corresponding to this modified method, and let $\mathcal{E}^*$ be the computation dag for COMPUTEANDNOTIFY$^*(s)$. Figure 4 shows $\mathcal{E}^*$ for the task graph shown in Figure 3. Since any computation

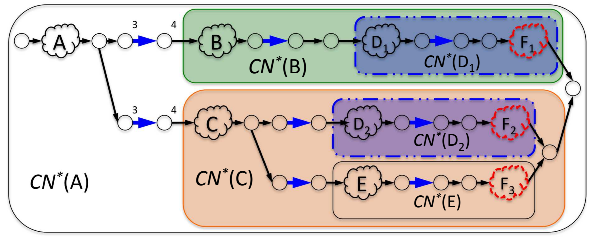

Figure 4. The computation dag $CN^*(A)$ for the execution of the task graph in Figure 3. Clouds represent COMPUTE methods for nodes, numbers correspond to line numbers from the code in Figure 2, and large arrows represent atomic decrements of join counters. An actual execution of COMPUTEANDNOTIFY(A) generates a computation dag which is a subdag of $CN^*(A)$. Four subdags are possible: (A, B, C, D_1, E, F_1), (A, B, C, D_1, E, F_3), (A, B, C, D_2, E, F_2), or (A, B, C, D_2, E, F_3). Each possible subdag contains exactly one of $\{D_1, D_2\}$, and one of $\{F_1, F_2, F_3\}$.

$\mathcal{E}$ forms a subdag of $\mathcal{E}^*$, we have $S(CN^*(A)) \geq S(CN^{\mathcal{E}}(A))$. Then, we bound $S(CN^*(s))$ using Lemma 2.

Lemma 2: Any execution of $\mathcal{D}$ using NABBIT has span $S(CN^*(s))$ at most

$$\max_{p \in paths(s,t)} \left\{ \sum_{X \in p} n(X) + \sum_{(X,Y) \in p} C_S(X,Y) \right\} \ ,$$

where

$$\begin{aligned} n(X) &= S(com(X)) + O\left(\lg(outDeg(X))\right) \ , \\ C_S(X,Y) &= O\left(\min\{inDeg(Y), P\}\right) \ . \end{aligned}$$

Proof: For each node X, the method COMPUTEANDNOTIFY$^*(X)$ enables all of X's immediate successors, with the recursive calls to COMPUTEANDNOTIFY* operating in parallel. As Figure 4 illustrates, any path through the computation dag $CN^*(X)$ contains the COMPUTE of only those nodes along a corresponding path through $\mathcal{D}$.

The term $n(X)$ accounts for the span $S(com(X))$ of X itself plus the additional span $O(\lg(outDeg(X)))$ required to spawn recursive calls along X's outgoing edges using a **parallel for** loop. In Cilk++, a **parallel for** loop[4] spawns iterations in the form of a balanced binary tree, and thus the depth of the tree is logarithmic.

The term $C_S(X,Y)$ accounts for the contention cost of decrementing the join counter for Y, where Y is a successor of X. In the worst case, this decrement might wait for $\min\{inDeg(Y), P\}$ other decrements. ■

Completion-time bounds

We have bounded the work and span of the computation dag using the characteristics of the task graph. Now, let us relate these bounds back to the time it takes

[4]The Cilk++ keyword is actually `cilk_for`.

to execute a task graph $\mathcal{D} = (V, E)$ on an ideal parallel computer. Let T_1 be the work of $\mathcal{D}$, i.e., the time it takes to execute $\mathcal{D}$ on a single processor. We have that

$$T_1 = \sum_{A \in V} W(com(A)) + O(|E|) \ ,$$

since any execution of $\mathcal{D}$ executes the COMPUTE method of every node once and must traverse every edge. Similarly, let T_∞ be the span of $\mathcal{D}$, i.e., the time it takes to execute $\mathcal{D}$ on an ideal parallel computer with an infinite number of processors. Define M as the number of nodes on the longest path in $\mathcal{D}$ from the source s to the sink t. We have

$$T_\infty = \max_{p \in paths(s,t)} \left\{ \sum_{X \in p} S(com(X)) \right\} + O(M) \ ,$$

since nodes along any path through $\mathcal{D}$ can not execute in parallel. By the work and span laws [6, p. 780], the completion time on P processors for a task graph is at least $\max\{T_1/P, T_\infty\}$.

Using Lemmas 1 and 2 and the analysis of a Cilk-like work-stealing scheduler [4], we obtain the following completion time bound for NABBIT.

Theorem 3: Let $\mathcal{D} = (V, E)$ be a task graph with maximum in-degree Δ_i and maximum out-degree Δ_o. With probability at least $1 - \epsilon$, NABBIT executes $\mathcal{D}$ on P processors in time

$$O\left(T_1/P + T_\infty + \lg(P/\epsilon) + M \lg \Delta_o + C(\mathcal{D})\right) \ ,$$

where $C(\mathcal{D}) = O\left((|E|/P + M) \min\{\Delta_i, P\}\right)$.

Proof: From [4], Cilk++'s work-stealing scheduler completes a computation with work W and span S in time $O(W/P + S + \lg(P/\epsilon))$ on P processors with probability at least $1 - \epsilon$. To bound the completion time, we relate the work and span of the computation dag $CN(s)$ to T_1 and T_∞. Bounding the contention term in Lemma 1 using Δ_i, we have

$$W(CN(s)) = T_1 + O(|E| \cdot \min\{\Delta_i, P\}) \ ,$$

since the sum of the in-degrees of the nodes in a graph is the cardinality of the edge set. Similarly, one can use Δ_i and Δ_o in Lemma 2 to show that

$$S(CN(s)) = T_\infty + O\left(M \lg \Delta_o + M \cdot \min\{\Delta_i, P\}\right) \ .$$

The theorem follows directly. ∎

The $M \lg \Delta_o$ term accounts for the additional span required to visit all the successors of a node in parallel. Whereas NABBIT allows a programmer to specify task graphs whose nodes have large degrees, fork-join languages such as Cilk++ produce computation dags where every node has constant out-degree, in which case this term is absorbed in the T_∞ term. Even when the out-degree is not constant, one would expect this term to be dominated by the T_∞ term if the task-graph nodes contain a reasonable amount of work.

The $C(\mathcal{D})$ term in Theorem 3 is an upper bound on the contention due to synchronization during the task-graph execution. The term $|E|/P + M$ is a bound on the P-processor execution time needed for a parallel traversal of $\mathcal{D}$, including updating the join counters on every edge. The extra factor of $\min\{\Delta_i, P\}$ appears, because we assume worst-case contention, i.e., that processors wait as long as possible on every decrement of a join counter. In the case where every node has constant degree, the term $C(\mathcal{D})$ is absorbed by $T_1/P + T_\infty$, and thus in this case the running time in Theorem 3 is asymptotically optimal. Even when the degree is more than constant, worst-case contention is unlikely to occur in practice for every decrement.

Although the contention term in Theorem 3 grows linearly with the maximum out-degree, in principle, one can modify the scheduler to asymptotically eliminate the contention term $C(\mathcal{D})$ from the completion-time bound.

Corollary 4: For any static task graph $\mathcal{D}$ with maximum degree Δ, there exists a work-stealing scheduler that can execute $\mathcal{D}$ in $O(T_1/P + T_\infty + M \lg \Delta + \lg(P/\epsilon))$ time on P processors with probability at least $1 - \epsilon$.

Proof: Given $\mathcal{D} = (V, E)$, one can create an equivalent task graph $\mathcal{D}'$ in which every node has constant degree by adding dummy nodes to $\mathcal{D}$. This construction adds only $O(|E|)$ dummy nodes and extends the longest path by $O(M \lg \Delta)$ nodes. By Theorem 3, executing $\mathcal{D}'$ with NABBIT gives us the desired bound. ∎

We did not implement this modification in NABBIT, since in practice for relatively small values of P, we expect that the overheads of this modification would be more expensive than simply suffering the contention.

IV. AN IRREGULAR DYNAMIC PROGRAM

One common application for the task-graph execution is dynamic-programming computations with irregular structure. This section describes experiments showing that NABBIT can be used to efficiently parallelize an irregular dynamic program based on the Smith-Waterman [19] dynamic-programming algorithm used in computational biology. This empirical study indicates that an implementation of the dynamic program using NABBIT is competitive with and can often outperform other Cilk++ implementations of the same dynamic program. Thus, in this example, the ability to execute a task graph with arbitrary dependencies improves overall performance and scalability, despite the added overhead that NABBIT requires to track dependencies during work-stealing that are not series-parallel.

Consider an irregular dynamic program on a 2-dimensional grid that computes a value $M(i, j)$ based on the following set of recursive equations:

$$\begin{aligned} E(i,j) &= \max_{k\in\{0,1,\ldots,i-1\}} M(k,j)+\gamma(i-k) \ ; \\ F(i,j) &= \max_{k\in\{0,1,\ldots,j-1\}} M(i,k)+\gamma(j-k) \ ; \\ M(i,j) &= \max \begin{cases} M(i-1,j-1)+s(i,j) \ , \\ E(i,j) \ , \\ F(i,j) \ . \end{cases} \end{aligned} \tag{2}$$

As described in [16], this particular dynamic program models the computation used for the Smith-Waterman [19] algorithm with a general penalty gap function γ. The functions $s(i,j)$ and $\gamma(z)$ can be computed in constant time. This dynamic program is irregular because the work for computing the cells is not the same for each cell. Specifically, $\Theta(i+j)$ work must be done to compute $M(i,j)$. Therefore, in total, computing $M(m,n)$ using Equation (2) requires $\Theta(mn(m+n))$ work ($\Theta(n^3)$, when $m=n$).

Parallel algorithms

We explored three parallel algorithms for this dynamic program. The first algorithm creates and executes a task graph using NABBIT. The second algorithm performs a wavefront computation, and the third algorithm uses a divide-and-conquer approach. Although these two approaches work for this particular dynamic program, neither can handle general task graphs. For each of the three algorithms, in order to improve cache locality and to amortize overheads, we blocked the cells into $B\times B$ blocks, where block (b_i,b_j) represents the block with upper-left corner at cell (b_iB,b_jB).

The first algorithm expresses the dynamic program in Equation (2) as a task graph by creating a task graph $\mathcal{D}$ similar to the code in Figure 1,[5] except that the cells are blocked and each node of the task graph represents a $B\times B$ block of cells. Block (b_i,b_j) depends on (at most) two blocks (b_i-1,b_j) and (b_i,b_j-1). The COMPUTE method for each node computes the values of M for the entire block serially.

The second algorithm performs a wavefront computation. The computation is divided into about n/B phases, where phase i handles the ith block antidiagonal of the grid. Within each phase, computation of each block along the antidiagonal is spawned, since blocks on an antidiagonal can be computed independently.

The third algorithm is a divide-and-conquer algorithm for the dynamic program that divides the grid into 4 subgrids and then computes the cells in each subgrid recursively. This algorithm computes the upper-left subgrid first, then the lower-left and upper-right subgrids in parallel next, and then finally the lower-right subgrid.

[5]Although $M(i,j)$ depends on the entire row i to the left of the cell and the entire column j above the cell, it is sufficient to create a task graph with dependencies to $M(i,j)$ only from $M(i-1,j)$ and $M(i,j-1)$. Transitivity ensures the other dependencies are satisfied.

One can show that asymptotically, if $n>B$, the parallelism (work divided by span) of both the task-graph and the wavefront algorithms is $\Theta(n/B)$, since both algorithms have $O(n^3)$ work and $\Theta(n^2B)$ span. The span of $\mathcal{D}$ consists of $\Theta(n/B)$ blocks, with a least half the blocks requiring $\Theta(nB^2)$ work.

One can also show that the divide-and-conquer algorithm has a span of $\Theta(n^{\lg 6}B^{3-\lg 6})\approx\Theta(n^{2.585}B^{0.415})$, and it therefore has a lower theoretical parallelism of $\Theta((n/B)^{\lg 6})\approx\Theta((n/B)^{0.415})$. This algorithm incurs lower synchronization overhead than the other two, however. One can asymptotically increase the parallelism of a divide-and-conquer algorithm by dividing M into more subproblems, but the code becomes more complex. In the limit, the resulting algorithm is equivalent to the wavefront computation.

Implementations

In our experiments, we compared four parallel implementations of Equation (2), based on (1) a task graph and NABBIT, (2) a wavefront algorithm, (3) a divide-and-conquer algorithm, dividing each dimension of the matrix by $K=2$, and (4) a divide-and-conquer algorithm, dividing each dimension by $K=5$. For a fair comparison, all implementations use the same memory layout and reuse the same code for core methods, e.g., computing a single $B\times B$ block. Each implementation looks up values for s and γ from arrays in memory.

Since memory layout impacts performance significantly for large problem sizes, we stored both $M(i,j)$ and $s(i,j)$ in a cache-oblivious [8] layout. The computations of $E(i,j)$ and $F(i,j)$ require scanning along a column and row, respectively, and thus, simply storing M in a row-major or column-major layout would be suboptimal for one of the computations. To support efficient iteration over rows and columns, we used dilated integers as indices into the grid [21] and employed techniques for fast conversion between dilated and normal integers from [17].

Experiments

We ran two different types of experiments on our implementations of the dynamic program. The first experiment measures the parallel speedups for the four different algorithms on various problem sizes. The second experiment measures the sensitivity of the algorithms to different choices in block size. We ran all experiments on a multicore machine with 8 total cores.[6]

[6]The machine contained two chip sockets, with each socket containing a quad-core 3.16 GHz Intel Xeon X5460 processor. Each processor had 6 MB of cache, shared among the four cores, and a 1333 MHz FSB. The machine had a total of 8 GB RAM and ran a version of Debian 4.0, modified for MIT CSAIL, with Linux kernel version 2.6.18.8. All code was compiled using the Cilk++ compiler (based on GCC 4.2.4) with optimization flag `-O2`.

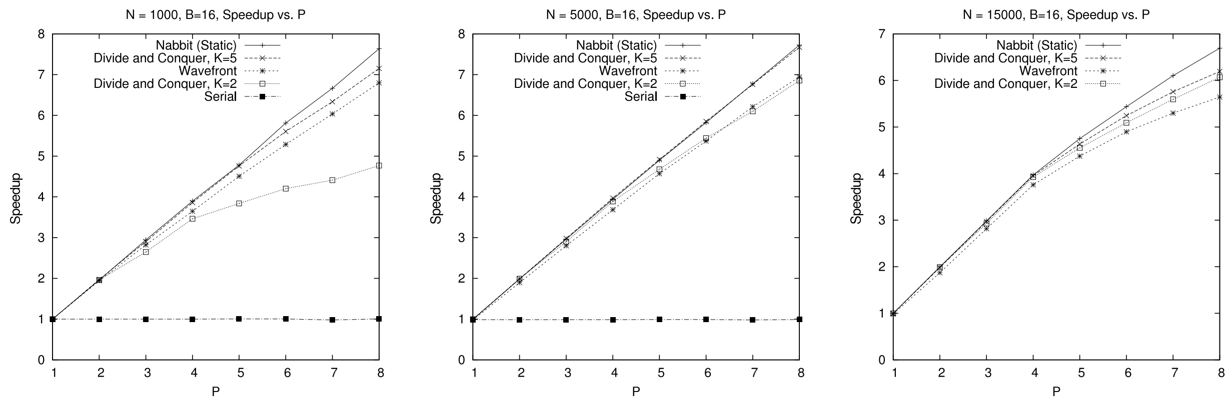

Figure 5. The performance of the dynamic program on an $N \times N$ grid. Speedups are normalized against the fastest run with $P = 1$. For $N = 1000$, the baseline is 2.05 s for serial execution. For $N = 5000$, the baseline is 263 s using divide-and-conquer. For $N = 15,000$, the baseline is 8279 s using NABBIT.

Speedups for the various implementations

In this experiment, we compared the speedup provided by the four algorithms, with a fixed block size at $B = 16$. Each task-graph node was responsible for computing a 16×16 block of the original grid, and the wavefront and divide-and-conquer algorithms operated on blocks of size 16×16 in the base case.

Figure 5 shows the speedup on P processors for $N \in \{1000, 5000, 15000\}$. NABBIT outperforms the other implementations in all these experiments. For example, at $N = 1000$, the divide-and-conquer algorithm achieves speedup of 5 on 8 processors, while the NABBIT implementation exhibits a speedup of about 7. This result is not surprising, since the task-graph execution has a higher asymptotic parallelism than the divide-and-conquer algorithm. Even though the wavefront algorithm has the same asymptotic parallelism as the task-graph execution, however, it performs worse than the divide-and-conquer algorithm for $K = 5$, which is slightly worse than the NABBIT implementation. As N increases to 5000, all the algorithms improve in scalability, and the gap between NABBIT and the other algorithms narrows. As N increases even more to $15,000$, however, the speedup starts to level off, and eventually decrease. We conjecture that this slowdown is due to increased data bus traffic and a lack of locality when computing the terms $E(i, j)$ and $F(i, j)$. In Equation (2), if we replace the γ term with indices which are independent of k, then we see a significant improvement in speedup on $N = 15000$ (graph not shown).

Effect of block size

To study the sensitivity of the algorithms to block size, we fixed N and varied B. Figure 6 shows the results for $N = 4000$. For small block sizes, we see that the task-graph algorithm using NABBIT performs worse than the divide-and-conquer algorithm with $K = 5$.

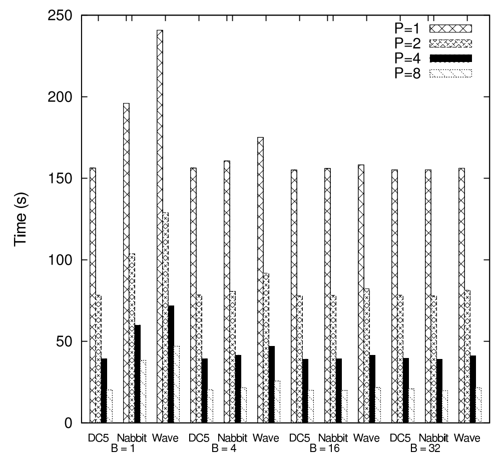

Figure 6. Running time for the dynamic program with $N = 4000$, varying the block size B for the base case.

For example, for $P = 1$ and $B = 1$, both divide-and-conquer algorithms require about 156 seconds, as opposed to 196 seconds for the task-graph algorithm. This result is not surprising, since for small block sizes, each node does not do enough work to amortize NABBIT's overhead for each node. In addition NABBIT also has significant space overhead for each node.

As B increases, however, at $P = 1$, the runtime using NABBIT approaches the runtime for divide-and-conquer with $K = 5$, and it begins to slightly outperform the other algorithms when $B \geq 16$. The wavefront algorithm at small B appears to have overhead which is even higher than the task-graph algorithm. In particular, for $P = 1$ and $B = 1$, the wavefront algorithm required about 241 seconds. Some of the wavefront algorithm's overhead is likely due to the cost of spawning computations on small blocks on each antidiagonal.

In summary, our experiments indicate that while NABBIT may suffer from high overheads when each node does little work, for this dynamic program, NABBIT generally exhibits relatively small overheads and is at least competitive with — and sometimes faster than — both divide-and-conquer and wavefront implementations for blocks of reasonable sizes.

V. DYNAMIC TASK GRAPHS

This section presents some extensions to NABBIT for supporting the execution of dynamic task graphs, whose nodes and edges are created on the fly at runtime. We first explain how dynamic NABBIT extends static NABBIT's interface. Then, we describe the modifications to the static NABBIT implementation required to support this interface. Finally, we summarize the theoretical guarantees provided by dynamic NABBIT.

Interface

Dynamic NABBIT provides an interface to programmers for executing a task graph $\mathcal{D}$ whose nodes and edges are created on the fly at runtime. As in static NABBIT, for each node A in $\mathcal{D}$, programmers must specify a COMPUTE method, which performs A's computation only after all of A's predecessors in $\mathcal{D}$ have been computed. Unlike static NABBIT, however, before executing the computation of a node A, dynamic NABBIT first "discovers" the nodes on which A depends. This interface reflects the notion that in some task-graph applications, a node A knows which nodes it requires values from, but A may not be aware of all nodes that may use its value.

In dynamic NABBIT, the programmer refers to nodes using hashable keys, rather than to nodes directly, which allows the space of possible nodes to be much larger than those that are actually created. A node with key k discovers its immediate predecessors by executing a programmer-specified INIT method. In the INIT method, the programmer specifies any other key k' on which k depends using the (library-provided) function ADDDEP(k'). Although multiple keys may depend on the same key k, NABBIT creates only one node object for each key, thus guaranteeing that INIT and COMPUTE execute exactly once per key.

NABBIT *implementation*

Dynamic task graphs are more complicated to support than static task graphs because a new node B that is a successor of A can be created at any time with respect to A's creation. Specifically, B can be created (1) before A has been created, (2) after A has been created but before A has completed its computation and notified its successors, or (3) after A has completed its notification. Thus, dynamic NABBIT requires additional bookkeeping.

Dynamic NABBIT maintains the following fields for each task-graph node A:

- **Key**: A unique 64-bit integer identifier for A.
- **Predecessor-key array**: The keys of A's immediate predecessors.
- **Status**: A field which changes monotonically, from UNVISITED to VISITED, then to COMPUTED, and finally to COMPLETED.
- **Notification array**: An array of A's successor nodes that need to be notified when A completes.
- **Join counter**: A's join counter reaches 0 when the COMPUTE for A is ready to be executed.

```
STARTEXECUTION(f)
  inserted = INSERTTASKIFABSENT(f)
  A = GETTASK(f)
  if inserted
      INITANDCOMPUTE(A)
```

Figure 7. Subroutine for dynamic NABBIT starting execution of a task graph at a node with key f.

To compare with the implementation of static NABBIT described in Section II, the predecessor-key array replaces the predecessor array in static NABBIT, and the notification array replaces the successor array. In dynamic NABBIT, the notification array for A need not contain all of A's successors, since some successors may be created after A finishes its notifications.

Since dynamic NABBIT works with keys instead of pointers to node objects, NABBIT maintains a hash table for task-graph nodes and guarantees that a node with a particular key is never created more than once. NABBIT uses a hash-table implementation that supports two functions: INSERTTASKIFABSENT(k), and GETTASK(k). The first atomically adds a new node object for a specified key k to the hash table if none exists, and the second looks up a node for a key k.[7] The atomic insertion of a node into the hash table also changes the node's status from UNVISITED to VISITED.

Dynamic NABBIT generally tries to execute a task graph in a depth-first fashion. Execution begins with a call to STARTEXECUTION(f) (shown in Figure 7), where f is the key of $\mathcal{D}$'s sink node t. NABBIT assumes that only a single call to STARTEXECUTION is active at any time. As its first step, NABBIT attempts to create a new node A for key f and atomically insert A into its hash table. Then, if this insertion is successful, NABBIT calls INITANDCOMPUTE(A).[8] The INITANDCOMPUTE method (shown in Figure 8) creates A by calling INIT(A) and recursively creates any dependencies (i.e., predecessors) of A. When this recursion reaches any node B with no dependencies, NABBIT calls COMPUTEANDNOTIFY(B) to compute B and any

[7] NABBIT could be easily modified to use any user-provided hash table that supports these two functions. This functionality would allow programmers to optimize by using an application-specific hash table.

[8] If this insertion fails, then a task with key f has already been created and/or computed, and thus the method does nothing.

successors of B that are subsequently enabled. When NABBIT uses multiple processors, these methods still attempt to execute in a depth-first fashion if possible, but the execution is not strictly depth-first because of Cilk++'s work-stealing strategy.

Synchronization in NABBIT

For static task graphs, synchronization occurs primarily through changes to join counters. The dynamic protocol is slightly more complicated, however, because the number of other nodes on which A depends is unknown before INIT is executed. Instead, A's join counter is atomically incremented when the user calls ADDDEP(k) inside INIT(A). In order to prevent the join counter from reaching 0 before all the dependencies have been created, the join counter for every node A is initialized to 1 and is decremented atomically after INIT(A) has completed.

During execution, A's join counter gets decremented once for every edge from Y to A. If NABBIT tries to traverse the edge (Y, A) after Y has been COMPUTED, then A's join counter is decremented in line 12 of TRYINITCOMPUTE. If NABBIT tries to traverse the edge (Y, A) before Y has been COMPUTED, then A is added to the list $Y.notifyArray$ of nodes that Y's notifies upon completion. Eventually, A's join counter is decremented in line 1 of COMPUTEANDNOTIFY(Y). To avoid race conditions, both the addition of a node to A's notification array and the change of A's status to COMPLETED (line 13 in COMPUTEANDNOTIFY) are performed while holding A's lock.

Discussion of theory

One can prove a completion time bound analogous to Theorem 3 for dynamic task graphs executed using NABBIT. The proof requires some additional definitions. For a task graph $\mathcal{D} = (V, E)$, and any node $A \in V$, let $loops(A)$ be the set

$$loops(A) = \bigcup_{X \in V} paths(X, A) \times paths(X, A),$$

that is, the set of all pairs of paths (p_1, p_2) in $\mathcal{D}$ from any node X to A. Conceptually, (p_1, p_2) represents a loop that traverses A to X along edges in p_1 backwards, and then back from X to A along forward edges in p_2. For a given computation dag $\mathcal{E}$ and a node A, let $init^{\mathcal{E}}(A)$ be the subgraph corresponding to INIT(A), and let $IC^{\mathcal{E}}(A)$ be the subgraph corresponding to INITANDCOMPUTE(A). As before, let $IC^*(A)$ be the computation dag representing an execution where all potential recursive calls occur.

For a dynamic task graph $\mathcal{D}$, the values of T_1 and T_∞ are greater than those for a static version of $\mathcal{D}$, since any execution must traverse $\mathcal{D}$ backwards from t to discover

```
TRYINITCOMPUTE(A, pkey)
 1  inserted = INSERTTASKIFABSENT(pkey)
 2  B = GETTASK(pkey)
 3  if inserted
 4      spawn INITANDCOMPUTE(B)
 5  finished = TRUE
 6  lock(B)
 7  if B.status < COMPUTED
 8      add A to B.notifyArray
 9      finished = FALSE
10  unlock(B)
11  if finished
12      val = ATOMDECANDFETCH(A.join)
13      if val = 0
14          COMPUTEANDNOTIFY(A)
15  sync
```

```
INITANDCOMPUTE(A)
1  assert(A.status == VISITED)
2  assert(A.join ≥ 1)
3  INIT(A)
4  for pkey ∈ A.predecessors
5      spawn TRYINITCOMPUTE(A, pkey)
6  val = ATOMDECANDFETCH(A.join)
7  if val == 0
8      COMPUTEANDNOTIFY(A)
9  sync
```

```
DECCOMPUTENOTIFY(X)
1  val = ATOMDECANDFETCH(X.join)
2  if val == 0
3      COMPUTEANDNOTIFY(X)
```

```
COMPUTEANDNOTIFY(A)
 1  COMPUTE(A)
 2  A.status = COMPUTED
 3  n = SIZEOF(A.notifyArray)
 4  A.notified = 0
 5  while A.notified < n
 6      for i ∈ [A.notified, n)
 7          X = A.notifyArray[i]
 8          spawn DECCOMPUTENOTIFY(X)
 9      A.notified = n
10      lock(A)
11      n = SIZEOF(A.notifyArray)
12      if A.notified == n
13          A.status = COMPLETED
14      unlock(A)
15  sync
```

Figure 8. Pseudocode for executing dynamic task graphs. For a node A, the TRYINITCOMPUTE(A) method attempts to create a predecessor (i.e., dependency) of A with the key $pkey$. INITANDCOMPUTE(A) spawns calls to try to create all of A's predecessors. Eventually, this method or one its spawned calls triggers COMPUTEANDNOTIFY(A), which executes A and all successors of A enabled by the completion of A.

the dependencies of each node. More precisely, we have

$$T_1 = \sum_{A \in V} \left(W(init(A)) + W(com(A)) \right) + O(|E|) \ ,$$

$$T_\infty = \max_{(p_1, p_2) \in loops(t)} \left\{ \sum_{X_1 \in p_1} S(init(X_1)) + \sum_{X_2 \in p_2} S(com(X_2)) \right\} + O(M) \ .$$

Theorem 5 states the completion time bound for dynamic task graphs. This bound matches the bound in Theorem 3, except for an $O(M\Delta)$ term instead of $O(M \lg \Delta)$. This difference arises since the successors of a node A might be created and added to $A.\mathit{notifyArray}$ sequentially and may be notified one by one, instead of in parallel (as in static NABBIT).

Theorem 5: Let $\mathcal{D} = (V, E)$ be a dynamic task graph with maximum degree Δ. With probability at least $1-\epsilon$, NABBIT executes $\mathcal{D}$ in time

$$O\left(T_1/P + T_\infty + \lg(P/\epsilon) + M\Delta + C(\mathcal{D})\right)$$

where $C(\mathcal{D}) = O\left(\left(|E|/P + M\right)\min\{\Delta, P\}\right)$.

Proof sketch: NABBIT's execution of $\mathcal{D}$ is modeled by the computation dag $IC(t)$. As for Theorem 3, we bound the completion time by calculating $W(IC(t))$ and $S(IC(t))$ and applying the analysis for Cilk.

We have $W(IC(t)) = T_1 + O(|E| \cdot \min\{\Delta, P\})$, since INIT and COMPUTE for each node A happens exactly once, and $O(1)$ synchronization operations happen for every edge $(A, B) \in E$, with each operation waiting at most $O(\min\{\Delta, P\})$ time due to contention.

We now argue the span $S(IC(t))$ is bounded by $S(IC^*(t))$, and then we show that $S(IC^*(t)) = T_\infty + O(M\Delta)$. From Figure 9, we can see that any path through $IC^*(t)$ travels along a single loop $(p_1, p_2) \in \mathit{loops}(t)$, which is to say that it walks backward along p_1 calling INIT and then forward along p_2 calling COMPUTE. Thus, INIT and COMPUTE contribute at most T_∞ to the span. For every edge (A, B) along this loop, the added overhead due to bookkeeping and contention on synchronization is $O(\Delta)$: in the worst case, Δ iterations of the loop in line 5 of COMPUTEANDNOTIFY occur, each notifying one successor of A. Since each loop contains $O(M)$ edges, the total overhead along the span is at most $O(M\Delta)$. ■

Strongly dynamic task graphs

Although dynamic NABBIT discovers the nodes and edges of a task graph at runtime, it can not create new task nodes based on the result of the COMPUTE of any task nodes. One might wish to extend NABBIT to handle this more general class of ***strongly dynamic*** task graphs, i.e., task graphs for which the COMPUTE for a node A can trigger the creation of new task nodes. For example, in Figure 8, in COMPUTE(A), the user might specify a list of keys of new tasks that A should generate, and then after finishing COMPUTE(A), NABBIT might begin executing a new task graph $\mathcal{D}_i$ for each generated key f_i by calling STARTEXECUTION. Although we assumed NABBIT has only one call to STARTEXECUTION active at a time, the implementation does correctly support concurrent calls to STARTEXECUTION, even when the task graphs $\mathcal{D}_i$ overlap (i.e., share nodes). In this case,

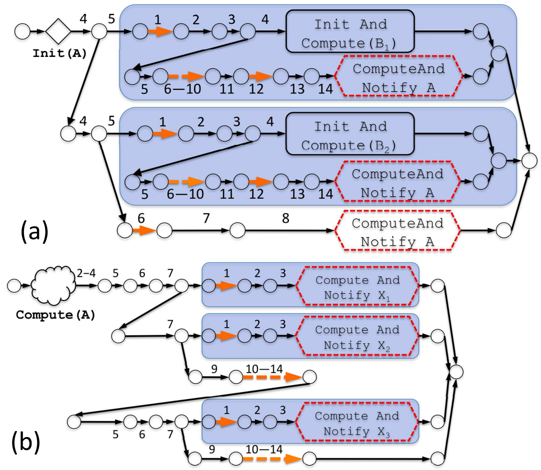

Figure 9. Example computation dags generated by calls to (a) INITANDCOMPUTE(A), and (b) COMPUTEANDNOTIFY(A). Arrows are labeled with line numbers from the code in Figure 8. Thick arrows correspond to synchronization operations that may experience contention. In INITANDCOMPUTE(A), shaded rectangles correspond to calls to TRYINITCOMPUTE for A for two predecessors B_1 and B_2. Dashed hexagons correspond to potential calls to COMPUTEANDNOTIFY. In (a), exactly one call to COMPUTEANDNOTIFY(A) occurs. In (b), the computation of A enables up to 3 successors of A, namely, X_1, X_2, and X_3.

NABBIT guarantees only that all $\mathcal{D}_i$ are computed after the last call to STARTEXECUTION finishes and the system quiesces.

The creation of new task nodes complicates the theoretical analysis of runtime, however, because different task graphs may begin executing at different times. Theorem 5 does not hold for strongly dynamic task graphs in part because it does not handle the dependencies and interactions between task graphs $\mathcal{D}_i$ that overlap. An interesting direction for future work is to extend the theory to handle strongly dynamic task graphs.

VI. RANDOM TASK-GRAPH BENCHMARK

This section studies and compares the overheads associated with static and dynamic versions of NABBIT. To do so, we constructed a microbenchmark that executes randomly constructed task graphs. We generate a random task graph $\mathcal{D}$ based on three parameters: Δ_i, the maximum in-degree of any node; U, the size of the universe from which keys are chosen; and W, the work in the compute of each node. The task graph $\mathcal{D}$ has a single sink node A_0 with key 0. Then, iterating over k from 0 to $U-1$, we repeat the following process:

- If $\mathcal{D}$ has a node A_k, choose an integer d_k uniformly at random from the closed interval $[1, \Delta_i]$.
- Create a multiset S_k of d_k integers, with each element chosen uniformly at random from $[k+1, U]$.
- Remove any duplicates from S_k, and for all $k' \in S_k$, add an edge $(A_{k'}, A_k)$ to the task graph (creating $A_{k'}$ if it doesn't already exist).

In $\mathcal{D}$, each task-graph node A_k performs W work, computing $k^W \bmod p$ using repeated multiplication, where p is a fixed 32-bit prime number. The benchmark provides the option of either performing this work serially, or in parallel (dividing the work in half, spawning each half, and recursing down to a base case of $W = 25$).

Experiments

We used the random task-graph benchmark in three experiments: (1) to measure the overhead of parallel execution, (2) to compare the overheads of the static and dynamic NABBIT, and (3) to evaluate the benefits of allowing parallelism inside the computes of nodes.

For the static task-graph benchmarks (static NABBIT), we allocated the memory for nodes and created nodes with pointers to its dependencies before executing the task graph. For the dynamic benchmarks (dynamic NABBIT), we constructed the same nodes as for the static benchmark, and then inserted these nodes into a hash table. The implementation of dynamic NABBIT atomically "inserts" a node for a key by looking it up in a hash table and marking it as VISITED.

To measure the approximate overhead for manipulating node objects and for parallel bookkeeping, we constructed a medium-sized random task graph and varied W. We compared static and dynamic NABBIT against corresponding serial algorithms. These serial algorithms perform the same computation as NABBIT with $P = 1$, except that all lock acquires are removed and all atomic decrements are changed to normal updates.

In Figure 10, we see that when $W = 1$ (each node does small work), the overhead of bookkeeping for static NABBIT is about 20% more than the serial version of the same algorithm. For dynamic NABBIT, the slowdown is about 16% over the serial algorithm. This baseline overhead shows that one would not want to use NABBIT for task graphs where each node does little work, since the overheads of bookkeeping dominate. As each node does more and more work and W increases to 1000, however, the difference becomes less than 5%.

From this data, we also see that our implementation of dynamic NABBIT exhibits a factor of 5 overhead over static NABBIT when $W = 1$. This difference is not surprising, since dynamic NABBIT ends up traversing a dag twice — from the final node to the root and then back — while static NABBIT only traverses the dag from root to final node. Also, in our benchmark, dynamic NABBIT performs additional look-ups in a hash table that the static version avoids. We observe that each node generally requires W to be on the order of at least $1,000$ to $10,000$ before the dynamic NABBIT attains performance comparable to the static version.

Our next experiment compares the speedups of static and dynamic NABBIT (graph not shown). We first created a large random task graph with small work ($W = 1$) per node. Even in the case when each node has small work and NABBIT has large overheads, observe that static NABBIT provides a speedup of up to 4.5 on 8 processors. Dynamic NABBIT scales and achieves a speedup of about 3.7 on 6 processors over the serial dynamic NABBIT execution. Compared to the static versions, however, dynamic NABBIT is overwhelmed due to the overheads.

	Static		*Dynamic*	
W	NABBIT	*Serial*	NABBIT	*Serial*
1	0.010	0.008	0.051	0.044
10	0.010	0.009	0.052	0.046
100	0.022	0.022	0.064	0.057
1000	0.137	0.134	0.178	0.177
$10,000$	1.267	1.265	1.306	1.301

Figure 10. Time in seconds for serial execution of $\mathcal{D}$ with $|V| = 14259$, $|E| = 78434$, and $M = 99$ nodes. The task graph $\mathcal{D}$ was randomly generated with $\Delta_i = 10$, $U = 100,000$, and $W = 1$.

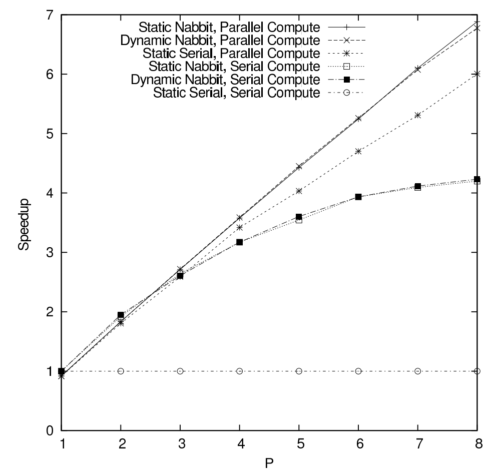

Figure 11. Comparison of static and dynamic NABBIT with and without parallelism in the COMPUTE function. For the random dag, $|V| = 127$, $|E| = 614$, $M = 29$, and $W = 10^6$. Speedup is normalized over the time (1.12 s) for the static serial execution.

On the other hand, we can see from Figure 11 that when each node has a large amount of work to do, the performances of static and dynamic NABBIT are nearly identical. In this case, the task graph contains relatively few nodes (only 127). If we examine the version where each node is computed serially, the theoretical parallelism is only about $127/29 = 4.4$. The static and dynamic versions of NABBIT both exploit most of this parallelism, providing a speedup of up to 4.2.

More importantly, however, Figure 11 demonstrates that to attain the best performance, one needs to exploit parallelism both at the task-graph level and within the COMPUTE functions. When only the dag-level parallelism is exploited, we obtain a speedup of 4.2. On the other hand, when NABBIT is not used and nodes are visited serially — only exploiting parallelism within the compute function — the speedup is about 6. The best case occurs by exploiting the parallelism both between nodes and within nodes, in which case both static and dynamic versions of NABBIT provide a speedup of 7.

The experiments on these random dags indicate that although NABBIT exhibits significant overhead on dynamic task graphs, this overhead can be amortized when each node does enough work. We also see that to get the best speedup, it pays to exploit both the dag-level parallelism and the parallelism within each task. NABBIT allows a programmer to exploit both seamlessly.

VII. CONCLUDING REMARKS

The dynamic-programming benchmark indicates that the performance of a task-graph execution may be limited by memory bandwidth. For graphs with regular structure, it is sometime possible to coarsen the dag—treating multiple nodes as a single node — so as to enhance locality. An interesting research direction is to investigate how one can best take advantage of locality in task graphs with irregular structure.

The space used by NABBIT is proportional to the size of the task graph. Once a node has executed and its successors have computed, however, it should be possible to garbage-collect the node and reuse it later in the computation, thereby saving space. We are currently exploring how to specify such a task-graph computation and how the garbage collection might best be implemented.

REFERENCES

[1] E. Allen, D. Chase, J. Hallett, V. Luchango, J.-W. Maessen, S. Ryu, G. L. Steele Jr., and S. Tobin-Hochstadt. The Fortress language specification, version 1.0. Technical report, Sun Microsystems, Inc., March 2008.

[2] N. S. Arora, R. D. Blumofe, and C. G. Plaxton. Thread scheduling for multiprogrammed multiprocessors. In *ACM Symposium on Parallel Algorithms and Architectures*, pages 119–129, Puerto Vallarta, Mexico, 1998.

[3] 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 *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 207–216, Santa Barbara, California, July 1995.

[4] R. D. Blumofe and C. E. Leiserson. Scheduling multithreaded computations by work stealing. *Journal of the ACM*, 46(5):720–748, 1999.

[5] R. D. Blumofe and D. Papadopoulos. Hood: A user-level threads library for multiprogrammed multiprocessors. Technical report, University of Texas at Austin, 1999.

[6] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*. The MIT Press, third edition, 2009.

[7] K. Ebcioglu, V. Saraswat, and V. Sarkar. X10: An experimental language for high productivity programming of scalable systems. In *Workshop on Productivity and Performance in High-End Computing (P-PHEC)*, 2005.

[8] M. Frigo, C. E. Leiserson, H. Prokop, and S. Ramachandran. Cache-oblivious algorithms. In *40th Annual Symposium on Foundations of Computer Science*, pages 285–297, New York, New York, Oct. 17–19 1999.

[9] M. Frigo, C. E. Leiserson, and K. H. Randall. The implementation of the Cilk-5 multithreaded language. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 212–223, 1998.

[10] R. Hoffmann, M. Korch, and T. Rauber. Performance evaluation of task pools based on hardware synchronization. page 44, Washington, DC, 2004. IEEE Computer Society.

[11] Intel Corporation. *Intel Cilk++ SDK Programmer's Guide*, October 2009. Document Number: 322581-001US.

[12] T. Johnson, T. A. Davis, and S. M. Hadfield. A concurrent dynamic task graph. *Parallel Computing*, 22(2):327–333, 1996.

[13] M. Korch and T. Rauber. A comparison of task pools for dynamic load balancing of irregular algorithms. *Concurrency and Computation: Practice & Experience*, 16(1):1–47, 2003.

[14] Y.-K. Kwok and I. Ahmad. Static scheduling algorithms for allocating directed task graphs to multiprocessors. *ACM Computing Surveys*, 31(4):406–471, 1999.

[15] C. E. Leiserson. The Cilk++ concurrency platform. In *DAC '09: Proceedings of the 46th Annual Design Automation Conference*, pages 522–527, New York, NY, 2009. ACM.

[16] M. Y. H. Low, W. Liu, and B. Schmidt. A parallel BSP algorithm for irregular dynamic programming. In *7th International Symposium on Advanced Parallel Processing Technologies*, pages 151–160. Springer, 2007.

[17] R. Raman and D. Wise. Converting to and from dilated integers. *IEEE Transactions on Computers*, 57(4):567–573, April 2008.

[18] J. Reinders. *Intel Threading Building Blocks: Outfitting C++ for Multi-Core Processor Parallelism*. O'Reilly, 2007.

[19] T. F. Smith and M. S. Waterman. Identification of common molecular subsequences. *Journal of Molecular Biology*, 147:195–197, 1981.

[20] J. D. Ullman. NP-complete scheduling problems. *Journal of Computer and System Sciences*, 10:384–393, 1975.

[21] D. S. Wise and J. D. Frens. Morton-order matrices deserve compilers' support. Technical Report TR533, Indiana University, 1999.