Executing Dynamic 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 executing dynamic task graphs with arbitrary dependencies. We prove that Nabbit achieves asymptotically optimal performance for task graphs whose nodes have constant in-degree and out-degree. We have implemented Nabbit in the multithreaded programming language Cilk++. Since the implementation of Nabbit required no modification to the Cilk++ runtime system, it should not be hard to port it to other fork-join languages and libraries.**

In order 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 which is used in computational biology. 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. In some cases, the Nabbit implementation even manages to outperform a divide-and-conquer implementation.

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.

The existing literature on task-graph scheduling tends to focus on ***static task graphs***, where the structure of the task graph and the amount of time it takes to execute each task are known before the computation begins. Typically, the scheduling algorithms themselves are also ***static***, meaning that nodes are mapped to processors in advance. For arbitrary task graphs, finding an optimal schedule on P processors is known to be NP-complete [19]. As Kwok and Ahmad describe in their survey of results on static scheduling [14], however, many efficient approximation algorithms and heuristics exist for static scheduling of static task graphs in a variety of computational models.

Dynamic task graphs come in two flavors. A ***weakly dynamic*** task graph means that the compute times of the nodes are not known in advance and can generally only be determined at runtime by executing the COMPUTE function of a node. A ***strongly dynamic*** task graph is not only weakly dynamic, but the nodes and edges themselves are determined at runtime. For example, Johnson *et al.* in [12] describe an interface for a strongly dynamic task graph, where new task nodes can be added or deleted dynamically as the task graph is being executed.

For dynamic task graphs, 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 — those 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 describe and empirically evaluate a variety of implementations of task pools in software [13] and hardware [11]. 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 can be considered as a special case of a ***work-stealing*** scheduling strategy such as is used in parallel languages such as Cilk [4], [10], Cilk++ [1], Fortress [2], X10 [8], and parallel runtime libraries such as Hood [6] and Intel Threading Building Blocks [17]. 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 at random. Blumofe and Leiserson [5] 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 dynamic task graphs in work-stealing environments. Our contributions are as follows:

- The Nabbit library for Cilk++, which provides an interface for programmers to execute weakly dynamic task graphs and uses conventional work stealing augmented with automatic reference counting and synchronization to schedule these dynamic task graphs. 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 required to execute weakly dynamic task graphs using Nabbit. For an arbitrary task graph $\mathcal{D}$, Nabbit can be used to execute $\mathcal{D}$ on P processors in $O(T_1/P + T_\infty \lg d)$ time in expectation, where T_1 is the work of $\mathcal{D}$, T_∞ is the span of $\mathcal{D}$, and d is the maximum out-degree of $\mathcal{D}$.
- An extension to Nabbit that supports strongly dynamic task graphs. The theoretical bounds for this extension are slightly weaker than those for the weakly dynamic case.

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 weakly dynamic 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.* [3] 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 shows the theoretical analysis of performance of Nabbit. Section IV describes a dynamic programming application, and presents experimental results evaluating the library's performance on this application. Section V presents extensions to Nabbit to support strongly dynamic task graphs, and Section VI presents a synthetic benchmark on randomly generated dags that evaluates the library's performance for both weakly and strongly dynamic task graphs.

II. THE NABBIT TASK GRAPH LIBRARY

Nabbit is a library for executing arbitrary weakly dynamic task graphs. Nabbit employs a straightforward scheduler based on reference counting and Cilk-like work-stealing. In this section, we describe how Nabbit operates for weakly dynamic task graphs.

Interface: In Nabbit, programmers specify task graphs by creating nodes that extend from a base `DAGNode` object and specifying the dependencies of each node. Nabbit executes the task graph by invoking a COMPUTE method on the root of the task graph.

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:

```
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].init_node(k, (void*)this);
    if (i > 0) {g[k].add_dep(&MNode[k-n])};
    if (j > 0) {g[k].add_dep(&MNode[k-1])};
   } } }
 int execute() { g[0]->execute(); }
};

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

Fig. 1. Code using Nabbit to solve the dynamic program in Equation (1). This code constructs a task graph node for every cell $M(i, j)$.

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

Figure 1 illustrates how one can formulate this problem as a task graph. The code constructs a node for every cell $M(i, j)$ by extending from a base node class. The programmer uses two methods of the base `DAGNode` class: `init_node` initializes each node with a specified key and a pointer to a structure wrapping the computation's global parameters, and `add_dep` specifies a predecessor node on which the current node depends.

In the example from Figure 1, the COMPUTE method for each node in the task graph is a short, serial section of code. In general, however, programmers can use the Nabbit interface in conjunction with a fork-join parallel language such as Cilk++, and expose nested fork-join parallelism inside the COMPUTE method of each node. For example, one might modify Figure 1 to have each node correspond to a block of cells in the matrix M instead of

```
ComputeAndNotify($A$)
    Compute($A$)
    for all $B \in$ successors($A$)
        do $val \leftarrow$ AtomDecAndFetch(join($B$))
            if $val = 0$
                then spawn ComputeAndNotify($B$)
```

Fig. 2. Cilk pseudocode for Nabbit operating on a weakly dynamic task graph. In an implementation, the iterations of line 2 are spawned in a binary tree fashion, and all can potentially run in parallel.

a single cell, and then use a parallel divide-and-conquer algorithm to evaluate the dynamic program for cells within each block.

Implementation: Our implementation of Nabbit maintains the following fields for each task-graph node A:

- **Key:** A unique 64-bit integer identifier for 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.
- **Dependency array:** An array of pointers to the nodes on which A depends, i.e., A's immediate predecessors in the task graph.

To execute a task graph $\mathcal{D}$, Nabbit calls the ComputeAndNotify method in Figure 2 on the root(s) of $\mathcal{D}$. Intuitively, ComputeAndNotify computes a node A, and then greedily spawns the computation for any immediate successors of A which are enabled by A's computation.

In our implementation, each node A maintains its dependency array only to allow users to conveniently access the nodes on which A depends inside A's Compute method, possibly to allow A to collect or aggregate results from its predecessors. Maintaining this array is not always necessary. For example, in Figure 1, one could also compute the dependencies of the current node through index calculations.

Finally, since we implemented Nabbit using Cilk++ without any modifications to the Cilk++ runtime, programmers can automatically use `cilk_spawn` to spawn arbitrary functions inside a node's Compute method.

III. Analysis of Performance

In this section, we provide a theoretical analysis of the runtime on P processors of Nabbit library described in Section II. To analyze the runtime of Nabbit, we employ the methodology of [7] and calculate upper bounds on the work and span[1] of the executions of the code in Figure 2. Then, we translate these bounds into completion-time bounds of Nabbit using known theoretical bounds for the completion time of fork-join parallel programs using randomized work-stealing [3], [5].

Definitions: Consider a task graph $\mathcal{D} = (V, E)$. Conceptually, each node $A \in V$ has a list $\texttt{in}(A)$ of immediate predecessors and a list $\texttt{out}(A)$ of immediate successors. Let $\texttt{outDeg}(A) = |\texttt{out}(A)|$ and $\texttt{inDeg}(A) = |\texttt{in}(A)|$ be the out- and in-degrees of A, respectively. For simplicity in stating the results, we assume that every node is a successor of a unique node $\texttt{root}(\mathcal{D})$ with no incoming edges and a predecessor of a unique node $\texttt{final}(\mathcal{D})$ 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. The computation performed by the recursive calls to ComputeAndNotify is nondeterministic, however, and may vary from execution to execution. Each possible execution can be represented as an execution graph (more specifically, DAG) $\mathcal{E}$.

We define several notations for subgraphs of an execution graph $\mathcal{E}$. For a particular execution graph $\mathcal{E}$ and a DAG node A, let $\texttt{comNot}^{\mathcal{E}}(A)$ be the subgraph corresponding to the call ComputeAndNotify(A), and let $\texttt{com}^{\mathcal{E}}(A)$ be the subgraph corresponding to Compute(A). For any subgraph $\mathcal{E}'$ of an execution DAG, we denote the work of the

[1]Sometimes called "critical-path length" and "computation depth" in the literature.

subgraph as $W(\mathcal{E}')$ and the span as $S(\mathcal{E}')$. We overload notation so that when the superscript $\mathcal{E}$ is omitted, we mean the maximum of the quantity over all execution graphs $\mathcal{E}$. For example, $W(\texttt{comNot}(A))$ denotes the maximum work for ComputeAndNotify(A) over all possible executions $\mathcal{E}$.

To analyze Nabbit's running time, we must analyze executions of ComputeAndNotify$(\texttt{root})$. The total work done by an execution $\mathcal{E}$ of $\mathcal{D}$ is $W(\texttt{comNot}^{\mathcal{E}}(\texttt{root}))$, and the span is $S(\texttt{comNot}^{\mathcal{E}}(\texttt{root}))$. Since the execution graph is nondeterministic, we shall analyze the maximum of these values — namely, $W(\texttt{comNot}(\texttt{root}))$ and $S(\texttt{comNot}(\texttt{root}))$ — over all possible execution graphs and use them as upper bounds in our analyses.

Work analysis:

Lemma 1: Any execution of $\mathcal{D}$ using Nabbit has work at most

$$W(\texttt{comNot}(\texttt{root})) = \left(\sum_{A\in V} W(\texttt{com}(A))\right) + O(|E|) + O(C_W).$$

where

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

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 $\texttt{inDeg}(B)$ times, and each decrement may wait at most $\min\{\texttt{inDeg}(B), P\}$ time. ■

Span analysis: The nondeterministic nature of the computation complicates a direct calculation of $S(\texttt{comNot}(A))$. Instead, we construct a new, deterministic execution DAG $\mathcal{E}^*$, whose span is an upper bound on the span of ComputeAndNotify$(\texttt{root})$. We define the method ComputeAndNotify$^*(A)$ to be the same as the original method, except that all possible recursive calls always occur. In other words, ComputeAndNotify$^*(A)$ always makes recursive calls for all of its successors. Let $\texttt{comNot}^*(A)$ be the execution subgraphs corresponding to this modified method, and let $\mathcal{E}^*$ be the execution DAG for ComputeAndNotify$^*(\texttt{root})$. Since any execution $\mathcal{E}$ forms a subdag of $\mathcal{E}^*$, we know $S(\texttt{comNot}^*(A)) \geq S(\texttt{comNot}^{\mathcal{E}}(A))$.

Lemma 2 bounds $S(\texttt{comNot}^*(A))$.

Lemma 2: The span of the computation, $S(\texttt{comNot}^*(\texttt{root}))$ is at most

$$\max_{p\in paths(\texttt{root},\texttt{final})} \left\{ \sum_{X\in p} n(X) + \sum_{(X,Y)\in p} C_S(X,Y) \right\}$$

where

$$n(X) = S(\texttt{com}(X)) + O(\lg(\texttt{outDeg}(X)))$$
$$C_S(X,Y) = O(\min\{\texttt{inDeg}(Y), P\})$$

Proof sketch: From the ComputeAndNotify code in Figure 2, we see that for a node X, $\texttt{comNot}^*(X)$ will enable all immediate successors Y of a node X, with each recursive ComputeAndNotify* happening in parallel.

The term $n(X)$ accounts for the span of X itself, $S(\texttt{com}(X))$, plus the additional span required to spawn recursive calls along X's outgoing edges using a parallel for loop, $O(\lg(\texttt{outDeg}(X)))$.

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

Completion-time bounds: We have bounded the work and span of the execution graph using the characteristics of the task graph. Now, we relate these bounds back to the execution time using a Cilk-like work-stealing scheduler.

For any task graph $\mathcal{D}$, define T_1 as the time it takes to execute $\mathcal{D}$ on a single processor. Define T_∞ as the time it takes to execute $\mathcal{D}$ on an infinite number of processors, assuming no synchronization overhead. We have

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

and

$$T_\infty = \max_{p \in paths(\texttt{root},\texttt{final})} \left\{ \sum_{X \in p} S(\texttt{com}(X)) \right\}.$$

One can prove that 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 [5], we obtain the following bound for the completion time for Nabbit.

Theorem 3: Consider a task graph $\mathcal{D}$ with maximum in-degree d_i, maximum out-degree d_o, and maximum path length (number of nodes on the longest path in the task graph from `root` to `final`) M. With probability at least $1 - \epsilon$, Nabbit executes $\mathcal{D}$ on P in time

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

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

Proof sketch: From [5], a Cilk-like work-stealing scheduler completes a computation with work W and span S in time $O(W/P + S + \lg(P/\epsilon))$ time on P processors with probability at least $1 - \epsilon$. To prove the completion time, we relate the work W and span S of the method `comNot(root)` to T_1 and T_∞.

The proof follows from Lemmas 1 and 2. We bound the in- and out-degrees of nodes by d_i and d_o, respectively, and bound expressions which compute maximum over paths p in terms of M.

Bounding the contention term in Lemma 1 using the maximum in-degree, we know that

$$W(\texttt{comNot}(\texttt{root})) = T_1 + |E| \min\{d_i, P\} \ .$$

Similarly, we can use Lemma 2 to show that $S(\texttt{comNot}(\texttt{root}))$ is not more than

$$O\left(T_\infty + M \lg d_o + M \min\{d_i, P\}\right) \ .$$

■

The $M \lg d_o$ term accounts for the additional span required to visit all the successors of a node in parallel. Normally, fork-join languages such as Cilk automatically generate execution DAGs where every node has constant out-degree. For Nabbit, however, the programmer can specify a task graph $\mathcal{D}$ whose nodes have large degree. This term is absorbed in the T_∞ term if the nodes have bounded out-degrees. Even when the out-degree is not bounded, however, we do not generally expect this term to significantly impact the running time.

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 time on P processors needed for a parallel traversal of $\mathcal{D}$, updating the join counters on every edge. The extra factor of $\min\{d_i, P\}$ appears because we assume worst-case contention: that processors wait as long as possible on every decrement of a join counter. Such a scenario seems unlikely to occur in practice. In the case where every node has bounded in- and out-degrees, the term $C(\mathcal{D})$ is absorbed by $T_1/P + T_\infty$, and thus, the running time we prove is asymptotically optimal for task graphs with bounded degrees.

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. We do not implement this modification, since in practice, we expect this modification to be more expensive than the current Nabbit implementation.

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

Proof sketch: Given $\mathcal{D} = (V, E)$, one can construct an equivalent task graph $\mathcal{D}'$ by adding dummy nodes such that every node $X \in \mathcal{D}'$ has constant in-degree. This construction adds at most $O(|E|)$ dummy nodes to $\mathcal{D}$, and increases the span by at most $M \lg d$. By Theorem 3, Nabbit on $\mathcal{D}'$ gives us the desired bound. ■

IV. A Dynamic-Programming Application

One common application for the dynamic scheduling of static task graphs is in the computation of irregular dynamic programs. In this section, we describe experiments showing that Nabbit can be used to efficiently parallelize an irregular dynamic program by presenting empirical results which show that an implementation of the dynamic program using Nabbit is competitive with, and in some cases outperforms other Cilk++ implementations of the same dynamic program. Thus, in this example, the ability to execute task graph with arbitrary dependencies improves overall performance and scalability, despite the additional overhead that Nabbit requires to track non series-parallel dependencies during work-stealing.

Our primary application is an irregular dynamic programming computation on a 2D grid. In particular, we consider the dynamic program which computes a value $M(i,j)$ based on the following set of recursive equations:

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

The functions $s(i,j)$ and $\gamma(z)$ can be computed in constant time; in our actual implementation, we look up values for s and γ from tables in memory. This dynamic program is irregular because the work for computing the cells is not the same for each cell; $O(i+j)$ 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 $m=n$). As described in [15], this particular dynamic program models the computation used for the Smith-Waterman [18] algorithm with a general penalty gap function γ.

Parallel algorithms: We explored three types of 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. For each of these algorithms, in order to improve cache-locality and to amortize overheads, we block the cells into $B \times B$ blocks, where block (b_i, b_j) represents the block with upper left corner at cell $(b_i B, b_j B)$.

For the first algorithm, we can express the dynamic program in Equation (2) as a task graph by creating a task graph $\mathcal{D}$ similar to the code in Figure 1,[2] 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 sequentially.

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.

Finally we can consider a divide-and-conquer algorithm for the dynamic program, as shown in Figure 3. This algorithm divides the grid into 4 sub-grids, and then computes the cells in each sub-grid recursively. The computations for the two sub-grids along the antidiagonal can be performed in parallel.

It is not difficult to show that asymptotically, if $n > B$, the parallelism of both the task graph algorithm and the wavefront algorithm is $\Theta(n/B)$. Both algorithms have $O(n^3)$ work. Both algorithms also have span $\Theta(n^2 B)$, since the span of $\mathcal{D}$ consists of $\Theta(n/B)$ blocks, with a least half the blocks requiring $\Theta(nB^2)$ work.

[2]Although $M(i,j)$ depends on the entire row i to the left of the cell and the entire column j above the cell, when creating a task graph, it is sufficient to create dependencies to $M(i,j)$ only from $M(i-1,j)$ and $M(i,j-1)$; other dependencies are ensured by transitivity.

```
ComputeM(n) { ComputeMHelp(0, 0, n); }

// Computes M for an n by n grid,
// with upper left corner at (i, j)
ComputeMHelp(i, j, n) {
  if (n <=B) { ComputeMBase(i, j, n); }
  else {
    ComputeMHelp(i, j, n/2);
    cilk_spawn ComputeMHelp(i+n/2, j, n/2);
    cilk_spawn ComputeMHelp(i, j+n/2, n/2);
    cilk_sync;
    ComputeMHelp(i+n/2 j+n/2, n/2);
  } }
```

Fig. 3. Pseudocode for a parallel divide-and-conquer algorithm to compute $M(n, n)$ for the dynamic program in Equation (2). For simplicity, we only show the code when $m = n$ is a power of 2.

With a little more work, one can show that the divide and conquer algorithm has the span of $\Theta(n^{\lg 6} B^{3-\lg 6}) \approx \Theta(n^{2.585} B^{0.415})$ (proof omitted due to space constraints) and therefore it has a lower theoretical parallelism of $\Theta((n/B)^{\lg 6}) \approx \Theta((n/B)^{0.415})$. This algorithm has 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 would be equivalent to the wavefront computation.

Implementation: 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 $K = 2$ as described in Figure 3, 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.

Since memory layout impacts performance significantly for large problem sizes, we stored both $M(i, j)$ and $s(i, j)$ in a cache-oblivious [9] layout. The computations of $E(i, j)$ and $F(i, j)$ require scanning along a column and row, respectively; thus, simply storing M in a row-major or column-major layout would be suboptimal for one of the computations.[3] To support efficient iteration over rows and columns, we use dilated integers as indices into the grid [20], and techniques for fast conversion between dilated and normal integers from [16].

Experiments: We ran two different types of experiments on our implementations of the dynamic program. The first experiment measures the parallel speedup of the four different algorithms for various problem sizes N. The second experiment measures the sensitivity of the algorithms to different choices in block size B.

We ran our experiments on a multicore machine with 8 total cores; the machine contains two sockets, with each socket containing a quad-core 3.16 GHz Intel Xeon X5460 processor. Each processor has 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`.

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

Figures 4, 5 and 6 shows the speedup on P processors for $N \in \{1000, 5000, 15000\}$. Nabbit outperforms all 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 evaluation has a higher asymptotic parallelism than the divide-and-conquer algorithm. The divide-and-conquer algorithm

[3] As a point of comparison, the divide-and-conquer algorithm for $N = 2000$ took about 300 s using a Morton-order layout, but 460 s using a row-major layout.

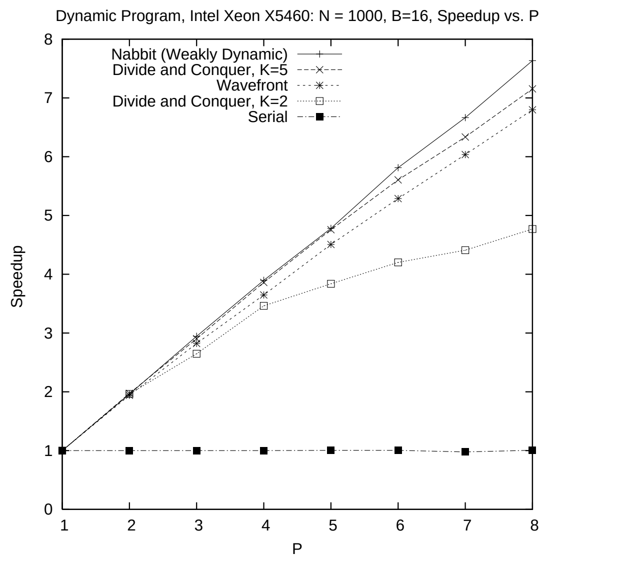

Fig. 4. Dynamic program on an $N \times N$ grid ($N = 1000$ and $B = 16$). Speedup is normalized to fastest run $P = 1$ (2.05 s for serial execution).

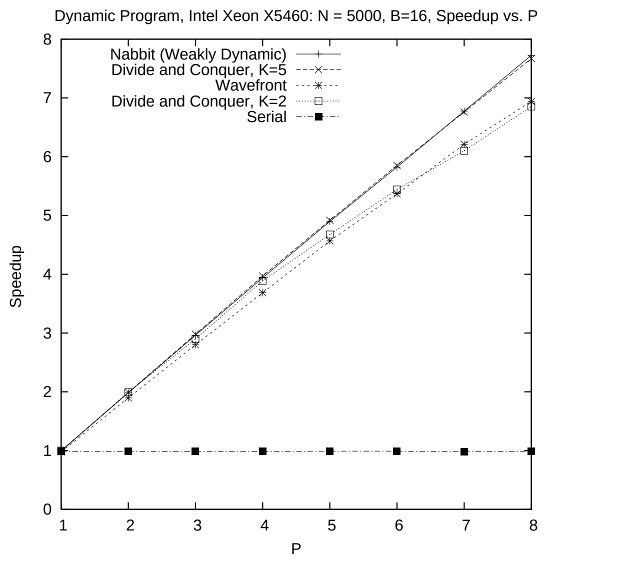

Fig. 5. $N = 5000$ and $B = 16$. Speedup is normalized to the fastest run with $P = 1$ (263 s for divide-and-conquer, $K = 2$).

for $K = 5$ outperforms the wavefront algorithm, even though the wavefront algorithm theoretically has asymptotically the same parallelism as the task graph evaluation. As N increases to 5000, all the algorithms improve in scalability, and the gap between Nabbit and the other algorithms narrows. Finally, as N increases even more, 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 $N = 15000$.

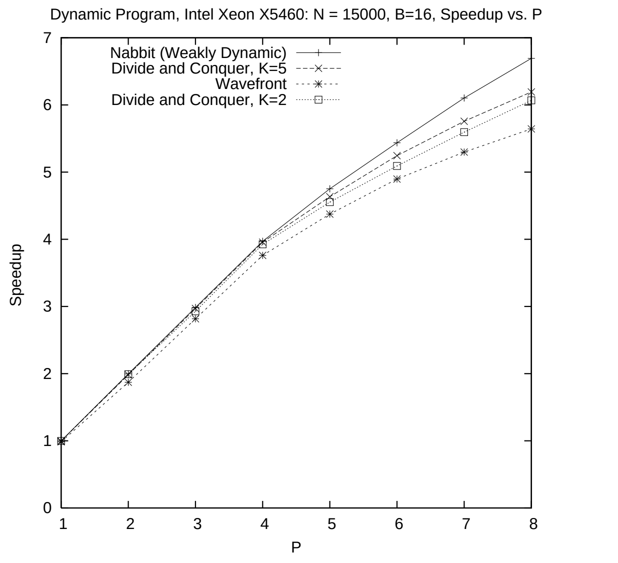

Fig. 6. $N = 15000$ and $B = 16$. Speedup is normalized to the fastest run with $P = 1$ (8279 s Nabbit).

Effect of block size: To measure the sensitivity of eager traversal to block size, we fix N and vary B. Figure 7 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 $K = 5$. 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 Nabbit has overhead for each node, and for small block sizes, each node does not do enough work to amortize this overhead. In addition Nabbit also has significant space overhead for each node.

As the block size 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 block sizes appears to have overhead which is even higher than the task-graph

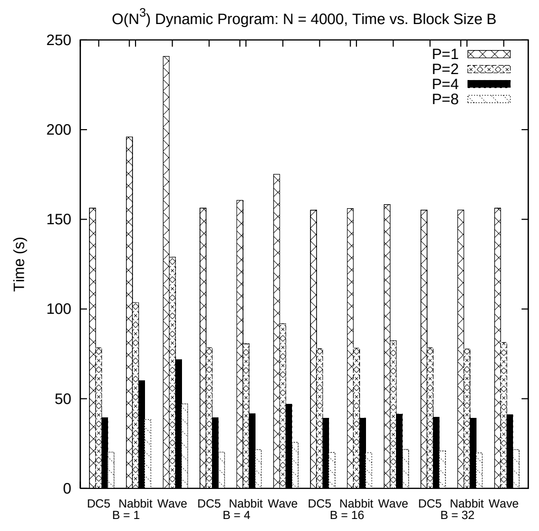

Fig. 7. Running time for $O(N^3)$ dynamic program for $N = 4000$, varying block size for base case B.

algorithm; for $P = 1$ and $B = 1$, the wavefront algorithm required about 241 seconds. Some of this overhead for the wavefront algorithm 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, in general for this dynamic program, Nabbit has relatively small overheads and is competitive with (and sometimes faster than) both divide-and-conquer and wavefront implementations for reasonably sized blocks.

V. STRONGLY DYNAMIC TASK GRAPHS

In this section, we present some extensions to Nabbit for supporting the execution of a strongly dynamic task graph. We first describe an extended interface for Nabbit that permits the addition of new tasks to a task graph $\mathcal{D}$ while $\mathcal{D}$ is being executed. Then, we describe the modifications to Nabbit implementation required to support the extended interface. Finally, we briefly describe the theoretical guarantees provided by this strongly dynamic version of Nabbit.

Interface: Nabbit provides an interface to allow the programmer to add nodes and dependencies to a task graph during runtime in order to create strongly dynamic task graphs. For strongly dynamic task graphs, however, programmers must identify nodes according to some hashable key. Nabbit creates a unique node object for each key, and this object is initialized, and executed exactly once.

For strongly dynamic task graphs, in addition to providing a COMPUTE method for each node, the programmer also specifies additional functions for generating new nodes and for calculating the dependencies of a given node. The programmer may specify the following two functions:

- INIT is called when initializing a node with a particular key k. In the INIT method, the programmer must specify the keys k' which k depends on, using the (library provided) function `add_task`(k'). The programmer may also specify any application-specific initialization for key k inside the INIT method.
- GENERATE is called after the node has been computed and can be used to create new nodes in the task graph by calling a (library provided) function `create_task`(k), which creates a new node with key k (if it doesn't already exist).

The library guarantees that each of these functions — COMPUTE, INIT, and GENERATE — is executed exactly once for every key.

The GENERATE method is the reason that the task graph is strongly dynamic, since this method can access the results of COMPUTE method and make decisions about which new nodes to create depending on this result. Therefore, the task graph is now data dependent. In contrast, programmers can use the INIT method only to statically specify that a key k depends on another key k', since Nabbit provides no guarantee that any nodes have been computed when INIT gets executed. Said differently, one can use the strongly dynamic version of Nabbit to automatically create a weakly dynamic task graph, by having an empty GENERATE method for every node, and specifying the rules for creating dependencies in the INIT method.

Finally, to begin execution of a task graph, the programmer invokes a method INITFINALCOMPUTE$(fKey)$, where $fKey$ is the key of the final node of the task graph.

Nabbit implementation: Strongly dynamic task graphs are more complicated to support than weakly dynamic task graphs because a new node B which is successor of A can be created at any time with respect to A. In general, B can be initialized (1) before A has been initialized, (2) after A has been initialized but before A has completed its computation and notified its successors, or (3) after A has completed its notification. Thus, Nabbit requires additional bookkeeping in nodes.

Nabbit maintains 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 predecessors.
- **Status**: A field which changes monotonically, from `UNVISITED` to `VISITED`, then to `COMPUTED`, and finally to `COMPLETED`.
- **Notification array**: A (possibly partial) array of A's successor nodes that need to be notified when A completes.
- **Join counter**: A's join counter reaches 0 when A is ready to be executed.

To compare with Section II, the notification array replaces the successor array used for the weakly dynamic version of Nabbit, and the predecessor key array replaces the dependency array.

The implementation of the strongly dynamic version of Nabbit works with keys instead of pointers to node objects. For strongly dynamic task graphs, Nabbit maintains a hash table for the nodes of the task graph to guarantee that a node with a particular key is never initialized more than once. Nabbit uses a hash table implementation which supports two functions: `insert_if_absent`(k), and `get_task`(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.[4] The atomic insertion of a node into the hash table also changes the node's status from `UNVISITED` to `VISITED`.

```
INITFINALCOMPUTE($fKey$)
1  $inserted \leftarrow$ INSERTTASKIFABSENT($H, fKey$)
2  $A \leftarrow$ GETTASK($H, fKey$)
3  if $inserted$
4      then INITANDCOMPUTE($A$)
```

Fig. 8. Subroutine for Nabbit task generation.

In general, Nabbit tries to execute a task graph in a depth-first fashion. Execution begins with a call INITFINALCOMPUTE$(fKey)$. First, Nabbit attempts to create a new node A for key $fKey$ and atomically insert A into the task graph's hash table H. Second, if this insert is successful, Nabbit initializes A by calling INIT(A). Third, Nabbit recursively initializes any dependencies (i.e., predecessors) of A. Finally, when this recursion reaches a node B with no dependencies, Nabbit calls COMPUTEANDNOTIFY(B) to compute B, any tasks generated by B, and any successors of B which are subsequently enabled. When Nabbit uses multiple processors, these methods still attempt to execute in a depth-first fashion if possible; however, the execution is not strictly depth-first because of parallelization. Figure 9 shows the pseudocode for the primary methods of a node in Nabbit.

Task generation is handled by the code in Figure 8. Inside the GENERATE(A) method, the user calls `create_task`(k) to try to create a new task with key k. If a task with that key already exists, this action does nothing. Otherwise, this action inserts a new task X with key k into the task graph's hash table, and then tries to execute the task graph starting at X using INITFINALCOMPUTE(k), the same method that the programmer calls on the final key to start the computation.

[4]Nabbit could be easily modified to use any user-provided hash table which supports these two functions. This functionality would allow programmers to optimize the hash table for the keys used by the application.

Synchronization in Nabbit : As for weakly dynamic task graphs, synchronization occurs primarily through changes of join counters. The strongly dynamic protocol is, however, slightly more complicated, 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 `add_dep`(k) inside INIT(A). In order to prevent the join counter from reaching 0 before all the dependencies have been initialized, the join counter for every node A is initialized to 1 and is decremented atomically once 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 `notifyArray`(Y), i.e., the list of nodes that Y's notifies upon completion. Eventually, A's join counter is decremented in line 11 of COMPUTEANDNOTIFY(Y). To avoid race conditions, both the addition of a node to A's notification array the change of A's status to COMPLETED (line 18 in COMPUTEANDNOTIFY) must be done while holding A's lock.

Discussion of Theory: The online generation of tasks for strongly dynamic task graphs complicates the analysis of the execution time. As an extreme example, consider a task graph $\mathcal{D}$ which has N independent tasks, $A_1, A_2, \ldots A_N$, each with $O(1)$ work. Using Nabbit, it is possible for each task A_i to generate a task A_{i+1}, even though A_{i+1} does not *depend* on A_i. This task graph will execute serially, since Nabbit does not discover the existence of A_{i+1} until A_i generates it. In terms of the task graph $\mathcal{D}$, there is no dependency edge from A_{i+1} to A_i, and the span of $\mathcal{D}$ is $O(1)$, even though the "effective span" is $\Omega(N)$, since the nodes are created one after the other. To provide meaningful results for Nabbit when nodes are generated on the fly, we consider the simplified case when a node k can only generate another node k' if k' depends on k, i.e., there is an edge from k to k' in the task graph $\mathcal{D}$.

```
TRYINITCOMPUTE($A, pkey$)
 1  $inserted \leftarrow$ INSERTTASKIFABSENT($H, pkey$)
 2  $B \leftarrow$ GETTASK($H, pkey$)
 3  if $inserted$
 4      then spawn INITANDCOMPUTE($B$)
 5  $finished \leftarrow$ true
 6  lock($B$)
 7  if status($B$) $<$ COMPUTED
 8      then add $A$ to notifyArray($B$)
 9           $finished =$ false
10  unlock($B$)
11  if $finished$
12      then $val \leftarrow$ ATOMDECANDFETCH(join($A$))
13           if $val = 0$
14              then spawn COMPUTEANDNOTIFY($A$)

INITANDCOMPUTE($A$)
1  assert(status($A$) $=$ VISITED)
2  assert(join($A$) $\geq 1|$)
3  INIT($A$)
4  $val \leftarrow$ ATOMDECANDFETCH(join($A$))
5  for ($pkey \in$ predecessors($A$))
6       do spawn TRYINITCOMPUTE($A, pkey$)
7  if join($A$) $= 0$
8     then spawn COMPUTEANDNOTIFY($A$)

COMPUTEANDNOTIFY($A$)
 1  COMPUTE($A$)
 2  status($A$) $=$ COMPUTED
 3  GENERATE($A$)
 4  for $genKey \in$ generatedTasks($A$)
 5        do spawn INITFINALCOMPUTE($genKey$)
 6  $n \leftarrow$ SIZE(notifyArray($A$))
 7  notified($A$) $\leftarrow 0$
 8  while notified($A$) $< n$ do
 9      for $i \in$ [notified($A$), $n$)
10        do $X \leftarrow$ elmt $i$ of notifyArray($A$)
11           $val \leftarrow$ ATOMDECANDFETCH(join($X$))
12           if $val = 0$
13              then spawn COMPUTEANDNOTIFY($X$)
14      notified($A$) $\leftarrow n$
15      lock($A$)
16      $n \leftarrow$ SIZE(notifyArray($A$))
17      if notified($A$) $= n$
18           then status($A$) $\leftarrow$ COMPLETED
19      unlock($A$)
```

Fig. 9. Pseudocode for executing strongly dynamic task graphs. For a node A, TRYINITCOMPUTE(A) method attempts to initialize a predecessor (i.e., dependency) of A with the key *pkey*. INITANDCOMPUTE(A) spawns calls to try to initialize all of A's predecessors; eventually this method or one its spawned calls will trigger COMPUTEANDNOTIFY(A), which executes A and all successors of A enabled by the completion of A.

Theorem 5: Consider a strongly dynamic task graph $\mathcal{D}$ with maximum degree d and maximum path length (number of nodes on the longest path in the task graph from `root` to `final`) M. With probability at least $1-\epsilon$, Nabbit executes $\mathcal{D}$ on P in time

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

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

Proof sketch: The proof is similar to that described in Section III. The main difference is in the term Md (instead of $M \lg d$ in the weakly dynamic case). In the weakly dynamic case, we can get $\lg d$ since all the successors are known and are notified in parallel. In the strongly dynamic case, since nodes are initialized online, successors B of A might be added to A's notification array one at a time, forcing the COMPUTEANDNOTIFY method to serialize the notification process. Also, there could be $O(d)$ contention on the lock for each A. ■

VI. RANDOM TASK GRAPH MICROBENCHMARK

In this section, we will evaluate the overheads associated with both the weakly and the strongly dynamic version of Nabbit. In order to do so, we construct a microbenchmark which evaluates randomly constructed task graphs. We generate a random task graph $\mathcal{D}$ based on three parameters: d – the maximum indegree of each node, U – the size of the universe from which keys are chosen, and W – the work in the compute of each node. $\mathcal{D}$ has a single final 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 $[1, d]$.
- 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 $k^W \texttt{mod}\ p$ using repeated multiplication, where p is a fixed 32-bit prime number. The microbenchmark 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 $W = 25$).

For a benchmark for a strongly dynamic task graph, for every node A_k that gets created, we also choose 3 (possibly duplicate) keys j_1, j_2, j_3 at random from $[1, U]$ as keys to be generated by A_k, create (up to) 3 nodes B_1, B_2, B_3, and recursively choose 3 keys for each B_i to generate.

Experiments: We use the random task graph benchmark in three experiments: (1) to measure the overhead of parallel execution, (2) to compare the overheads of the weak and strong versions of Nabbit and (3) to evaluate the benefits of allowing parallelism inside the computes of nodes.

For the weakly dynamic task graph benchmarks (weak Nabbit), we allocate the memory for nodes and initialize nodes with pointers to its dependencies before executing the task graph. For the strongly dynamic benchmarks (strong Nabbit), we construct the same nodes as for the static benchmark, and then insert these nodes into a hash table. The implementation of strong 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 construct a medium-sized random task graph and vary W. We compare weak and strong 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, the we see that when $W = 1$ (each node does very little work), the overhead of bookkeeping for weak Nabbit is about 20% more than a serial execution of the same al-

	Weak		Strong	
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
10^4	1.267	1.265	1.306	1.301

Fig. 10. Execution of $\mathcal{D}$ with $|V| = 14259$, $|E| = 78434$, and span of 99 nodes, for $P = 1$. $\mathcal{D}$ was randomly generated with $d = 10$, $U = 100000$ and $W = 1$.

gorithm when $W = 1$. For strong Nabbit, the slowdown is about 16% over the serial algorithm. This is the baseline overhead, and 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 strong Nabbit has a factor of 5 overhead over weak Nabbit when $W = 1$. This difference is not surprising, since strong Nabbit ends up traversing a DAG twice, from the final node to the root, and then back, while weak Nabbit only traverses the DAG from root to final node. Also, in our benchmark, strong Nabbit is performing additional lookups in a hash table that the weak version can avoid. We observe that each node generally requires a W on the order of at least 1,000 to 10,000, before the strong Nabbit has comparable performance comparable to the weak version.

Our next experiment compares the speedups of strong and weak Nabbit. We first create with a large random task graph with very little work per node $W = 1$. Even in the case when each node has very little work and Nabbit has large overheads, we see that weak Nabbit provides speedup of up to 4.5 on 8 processors. Strong Nabbit does scale and achieves a speedup of about 3.7 on 6 processors, compared to the corresponding strong serial execution. However, compared to the weak versions, strong Nabbit is overwhelmed due to the overheads. We omit the graph due to space constraints.

On the other hand, we can see from Fig-

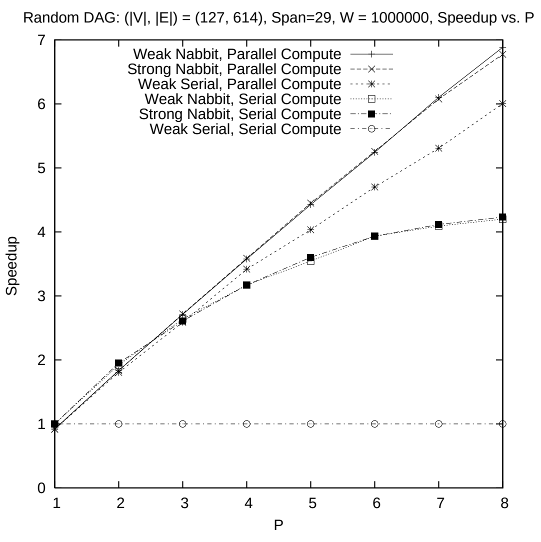

Fig. 11. Comparison of strong and weak Nabbit, with and without parallelism in the COMPUTE function. Speedup is normalized over time for weak serial execution, with $P = 1$ (1.12 s).

ure 11, when each node has a substantial amount of work to do, then the strong and weak Nabbit are both equally scalable. In this case, the task graph has relatively few nodes (only 127). Therefore, if we look at the version where each node is computed sequentially, the theoretical parallelism is only about $127/29 = 4.4$. The strong and weak versions of Nabbit both exploit most of this parallelism, providing speedup upto 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 get a speedup of 4.2. On the other hand, when Nabbit is not used, and nodes are visited sequentially, only exploiting parallelism within the compute function, the speedup is about 6. The best case occurs when both the parallelism between nodes and within nodes is exploited and both strong and weak versions of Nabbit provide the speedup of 7.

The experiments on these random dags indicate that even though Nabbit exhibits significant overhead on strongly dynamic task graphs, this overhead can be amortized when each node does a significant amount of work. In addition, in order to get the best speedup, one should take advantage of both the dag level parallelism and the parallelism within each task. Nabbit allows a programmer to do this seamlessly.

VII. CONCLUDING REMARKS

Nabbit is a Cilk++ library that allows programmers to specify dynamic task graphs with arbitrary dependences and executes these task graphs using work stealing. It allows programmers to exploit both task-graph parallelism and parallelism within COMPUTE functions for dynamic task graphs. We proved that Nabbit executes a task graph in asymptotically optimal time when nodes of the task graph have constant degree. We also benchmarked our library on an irregular dynamic-programming application, showing that Nabbit is competitive with (and sometimes better than) divide-and-conquer algorithms for the same problem.

We would like to find more applications which could benefit from Nabbit. In particular, we want to benchmark Nabbit on real-world strongly dynamic task graphs. Moreover, from our dynamic-programming benchmark, we can see that the performance of a task-graph execution may be limited by locality and memory bandwidth issues. An interesting research direction is to understand whether one can take advantage of locality in a task-graph execution, particularly for graphs with irregular structure.

REFERENCES

[1] Cilk++. http://www.cilk.com, April 2009.

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

[3] N. S. Arora, R. D. Blumofe, and C. G. Plaxton. Thread scheduling for multiprogrammed multiprocessors. In *SPAA*, pages 119–129, Puerto Vallarta, Mexico, 1998.

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

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

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

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

[8] K. Ebcioglu, V. Saraswat, and V. Sarkar. X10: an experimental language for high productivity programming of scalable systems. In *Proceedings of Workshop on Productivity and Performance in High-End Computing (P-PHEC)*, 2005. In conjunction with *Symposium on High Performance Computer Architecture (HPCA)*.

[9] 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.

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

[11] R. Hoffmann, M. Korch, and T. Rauber. Performance evaluation of task pools based on hardware synchronization. In *SC '04: Proceedings of the 2004 ACM/IEEE conference on Supercomputing*, page 44, Washington, DC, USA, 2004. IEEE Computer Society.

[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 Comput. Surv.*, 31(4):406–471, 1999.

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

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

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

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

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

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