Executing Task Graphs Using Work-Stealing

Jim Sukha
MIT CSAIL
IPDPS, 4/21/2010

Kunal Agrawal (Washington University in St. Louis)
Charles E. Leiserson (MIT)
Task Graphs

A *task graph* is a directed acyclic graph (dag) where
- Every node $A$ is a task requiring computation,
- Every edge $(A, B)$ means that the computation of $B$ depends on the result of $A$’s computation.
Example: Counting Paths

Counting the number of paths through a dag can be expressed as a task graph.

Source starts with value 1.

The value of a node \( X \) is the sum of the values of \( X \)'s immediate predecessors.
Some task graphs can be executed in parallel using a fork-join language such as Cilk++.

```c
void f1() {
    A();
    cilk_spawn B();
    C();
    cilk_sync;
    D();
}

void f2() {
    E(); F(); G();
}

int main() {
    cilk_spawn f1();
    f2();
    cilk_sync;
}
```
Graphs with Arbitrary Dependencies

Unfortunately, one can not directly express task graphs with arbitrary dependencies using only spawn and sync.

Counting paths in a non-series-parallel dag.

Question: Can we efficiently execute arbitrary task graphs in parallel in a fork-join language such as Cilk++?
Our Contributions: Nabbit

We are developing Nabbit, a Cilk++ library for executing task graphs with arbitrary dependencies.

- Nabbit is built on top of Cilk++. It utilizes Cilk++’s provably-efficient work-stealing scheduler without any modification to the Cilk++ runtime.
- Using Nabbit, the computation of an individual task graph node can itself be parallel.
Provable Bounds for Nabbit

We are developing Nabbit, a Cilk++ library for executing task graphs with arbitrary dependencies.

- Nabbit offers provable bounds on the time required for parallel execution of (static and dynamic) task graphs.
- The time bounds for Nabbit are asymptotically optimal for task graphs whose nodes have constant in-degree and out-degree.
Outline

- Static Task Graphs using Nabbit
- Nabbit Implementation
- Completion Time Bound
- Dynamic Task Graphs and Other Extensions
Dynamic-Programming Example

Generic dynamic programs can often be expressed as a task graph.

\[ M(i, j) = \max \begin{cases} g_1(M(i-1, j-1)) \\ E(i, j) \\ F(i, j) \end{cases} \]

\[ E(i, j) = g_2( M(1, j), M(2, j), \ldots, M(i-1, j) ) \]

\[ F(i, j) = g_3( M(i, 1), M(i, 2), \ldots, M(i, j-1) ) \]
Static Task Graphs

For this example, we can use a static task graph, i.e., a task graph where the structure of the dag is known before the execution begins.

\[
M(i, j) = \max \begin{cases} 
  g_1(M(i-1, j-1)) \\
  E(i, j) \\
  F(i, j) 
\end{cases}
\]

\[
E(i, j) = g_2( M(1, j), M(2, j),..., M(i-1, j) )
\]

\[
F(i, j) = g_3( M(i, 1), M(i, 2),..., M(i, j-1) )
\]

Create a node for every cell \(M(i, j)\).
Static Task Graphs

For this example, we can use a static task graph, i.e., a task graph where the structure of the dag is known before the execution begins.

\[
M(i, j) = \max \begin{cases} 
g_1(M(i-1, j-1)) \\
E(i, j) \\
F(i, j) 
\end{cases}
\]

\[
E(i, j) = g_2( M(1, j), M(2, j),..., M(i-1, j) )
\]

\[
F(i, j) = g_3( M(i, 1), M(i, 2),..., M(i, j-1) )
\]

Create a node for every cell \(M(i, j)\). Then add dependency edges.

* In Nabbit, static task graphs still require dynamic scheduling. We assume the compute time for each node may be unknown.
Interface for Static Nabbit

For static task graphs, each task graph node is derived from Nabbit’s `DAGNode` class, and overrides the node’s `Compute()` method.

```cpp
class Mnode: public DAGNode {  
  Mnode(int key, MDag* dag);  
  void Compute();  
};
```

Typically, each node needs to know its identity, and global parameters for task graph.

```cpp
class MDag {  
  int N; int *M;  
  Mnode* g;  
  MDag(int N_, int* M_);  
};
```

Programmer builds their own task graph.
Constructing a Static Task Graph

Programmers use Nabbit’s **AddDep** method to specify dependencies between task graph nodes.

```cpp
class MDag {
    int N; int* s; Mnode* g;
    MDag(int n_, int* M_) : N(N_), M(M_) {
        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].key = k; g[k].dag = this;
                if (i > 0) g[k].AddDep(&Mnode[k-N]);
                if (j > 0) g[k].AddDep(&Mnode[k-1]);
            }
        }
    }
};
```

*M(i, j)* has edges from *M(i-1, j)* and *M(i, j-1).*
Implementing Task Nodes

Task graph nodes inherit from a `DAGNode` class, and override the node’s `Compute()` method.

```cpp
class Mnode: public DAGNode {
  int i, j;

  void Compute() {
    int z = INFINITY;
    int Eij = calcE(dag->M, i, j);
    int Fij = calcF(dag->M, i, j);
    if ((i > 0) && (j > 0))
      z = g1(M, i, j);
    dag->M[key] = min(z, Eij, Fij);
  }
};
```

One can call other Cilk functions inside the `Compute()` method, including methods that `spawn` and `sync`.
Outline

- Static Task Graphs using Nabbit
- Nabbit Implementation
- Completion Time Bound
- Dynamic Task Graphs and Other Extensions
Nabbit uses a simple algorithm to execute static task graphs in parallel. Each node
1. Maintains a count of the # of its immediate predecessors that are still incomplete. (Each node keeps a join counter.)
2. Notifies its immediate successors in parallel after it is computed.
3. Recursively computes any successors which become ready.

```cpp
void ComputeAndNotify() {
    this->Compute();
    cilk_for (int q = 0; q < successors().size(); q++) {
        DAGNode* Y = successors[q];
        int val = AtomicDecAndFetch(Y.join);
        if (val == 0) Y.ComputeAndNotify();
    }
}
```

Start execution by calling ComputeAndNotify from the source (root) node.
Work-Stealing in Nabbit

Nabbit is able to rely on Cilk++’s work-stealing scheduler to load-balance the computation.

- When a processor runs out of work, it tries to steal work from other processors.
- Nabbit spawns task nodes in a way that makes the Cilk++ runtime likely to steal nodes along the critical path of the task graph.

Sample Dynamic-Program Execution with $P=4$
As a benchmark, we consider a dynamic program modeling the Smith-Waterman algorithm with a generic penalty gap:

\[ M(i, j) = \max \begin{cases} M(i-1, j-1) + s(i, j) \\ E(i, j) \\ F(i, j) \end{cases} \]

\[ E(i, j) = \max_{k \in \{0, 1, \ldots, i-1\}} M(k, j) + \gamma(i-k) \]

\[ F(i, j) = \max_{k \in \{0, 1, \ldots, j-1\}} M(i, k) + \gamma(j-k) \]

In this example, \( s(i, j) \) and \( \gamma(k) \) are constant arrays.
Comparison with Divide-and-Conquer

For the dynamic program, we compare the task graph evaluation using Nabbit with alternative algorithms.

K-way divide-and-conquer

Wavefront (Synchronous)

For all algorithms, base case is blocks of size $B$ by $B$. Grid is arranged in a cache-oblivious layout.
Comparing implementations of the dynamic program

Opteron Processor 8354: 2.2 Ghz

Serial Running Time = 4.4 s
$c = 4.4e-9$ if $T_S \sim cN^3$

Serial Running Time = 664 s
$c = 5.3e-9$ if $T_S \sim cN^3$
Comparison, $N=3000$, $P=16$

Nabbit

Wavefront

$K=5$

$K=2$
Outline

- Static Task Graphs using Nabbit
- Nabbit Implementation
- Completion Time Bound
- Dynamic Task Graphs and Other Extensions
Definitions

Let $D=(V, E)$ be a task graph to execute. Consider the execution dag associated with the `Compute()` method of a task node $A \in V$.

- $W(A)$: the work of $A$ (# of nodes in execution dag)
- $S(A)$: the span of $A$ (length of longest path in dag)

$M$: # of task nodes on longest path through task graph $D$.

$\Delta$: maximum degree of any task node

$W(A) = 11$
$S(A) = 6$

$M = 5$
$\Delta = 2$
Work and Span of a Task Graph

We can define a “total” work and span for the task graph execution. Define $T_1$ and $T_\infty$ as:

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

$$T_\infty = \max_{\text{all paths } p \text{ through } D} \left\{ \sum_{A \in p} (S(A) + O(1)) \right\}$$

Any execution of the task graph on $P$ processors requires time at least:

$$\max \{ T_1/P, T_\infty \}.$$
**THEOREM 1**: Nabbit executes a static task graph $D = (V, E)$ on $P$ processors in expected time

$$O\left(\frac{T_1}{P} + T_\infty + M \log \Delta + C(D)\right)$$

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

$T_1/P + T_\infty$: Bound for ordinary Cilk-like work-stealing

$M \log \Delta$: span of notifying task node successors

$C(D)$: worst-case contention for atomic decrements.

($\min\{\Delta, P\}$: time a decrement can wait)

Theorem is asymptotically optimal when $\Delta = \Theta(1)$. 

$M$: # of nodes on longest path through task graph $D$.

$\Delta$: maximum degree of any task node
Outline

- Static Task Graphs using Nabbit
- Nabbit Implementation
- Completion Time Bound
- Dynamic Task Graphs and Other Extensions
Nabbit also supports *dynamic task graphs*. Roughly, a dynamic task graph can be thought of as performing a parallel traversal of a two-phase dag, where the first `Init()` phase creates new nodes.
Dynamic task graphs are more complicated because \texttt{Init()} and \texttt{Compute()} happen concurrently.
Theorem 2: Nabbit executes a dynamic task graph \( D = (V, E) \) on \( P \) processors in expected time \( \mathcal{O} \left( \frac{T_1}{P} + T_\infty + M\Delta + C(D) \right) \) where \( C(D) = \mathcal{O} \left( \left( \frac{E}{P} + M \right) \min\{\Delta, P\} \right) \). \( T_1 \) and \( T_\infty \) are modified to account for \texttt{Init()} for each node. \( M\Delta \) : weaker bound because all edges in the graph are not known ahead of time.
Topics for Future Investigation

We are interested in possibly extending Nabbit in several directions:

- **Strongly Dynamic Task Graphs**
  - `Compute()` of a task node can generate a new task.
- **Reusing Nodes and Garbage Collection**
- **Hierarchical Task Graphs**
- **Runtime/Compiler Support for Nabbit**
Applications for Nabbit?

We are interested in possibly extending Nabbit in several directions:

- **Strongly Dynamic Task Graphs**
  - `Compute()` of a task node can generate a new task.
- **Reusing Nodes and Garbage Collection**
- **Hierarchical Task Graphs**
- **Runtime/Compiler Support for Nabbit**
- **Applications!**

The value of these possible extensions to Nabbit depends on programs that use static or dynamic task graphs. We value any feedback regarding potential applications!
Questions?