Overview

Work-stealing scheduler
- $O(pS_1)$ worst case space
- small overhead

Narlikar scheduler\(^1\)
- $O(S_1+pKT_\infty)$ worst case space
- large overhead

Hybrid scheduler
- Idea: combine space saving ideas from Narlikar with the work-stealing scheduler

---

What We Did

- Implemented Narlikar Scheduler for Cilk
  - Replaced WS scheduling code
  - Modified cilk2c
- Designed WS-Narlikar Hybrid Scheduler
- Implemented Hybrid Scheduler
  - Modified WS scheduling code
  - Modified cilk2c
- Performed empirical tests for space and time comparisons
## Results

Data from running the modified fib program on 16 processors

<table>
<thead>
<tr>
<th></th>
<th>Space (Kb)</th>
<th>Ratio (scheduler/Cilk WS)</th>
<th>Time (sec)</th>
<th>Ratio (scheduler/Cilk WS)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cilk WS</td>
<td>491520</td>
<td>1.00</td>
<td>1.8</td>
<td>1.0</td>
</tr>
<tr>
<td>Narlikar</td>
<td>204800</td>
<td>0.41</td>
<td>837.0</td>
<td>465.0</td>
</tr>
<tr>
<td>Hybrid</td>
<td>368640</td>
<td>0.75</td>
<td>2.3</td>
<td>1.3</td>
</tr>
</tbody>
</table>

- Hybrid retains some of the space saving benefits of Narlikar with a much smaller overhead.
Outline

I. Example

II. Narlikar Algorithm
   a. Description
   b. Overheads/Bottlenecks

III. Hybrid Algorithm
   a. Motivation
   b. Description

IV. Empirical Results

V. Future Work

VI. Conclusions
main() {
    for(i = 1 to n)
        spawn F(i, n);
}

F(int i, int n) {
    Temp B[n];
    for(j = 1 to n)
        spawn G(i, j, n);
}
Schedule outer parallelism first

Memory used (heap): $\theta(n^2)$

Similar to work-stealing scheduler ($\theta(pn)$ space)

Green nodes are executed before white nodes
Schedule 2

Schedule inner parallelism first

Memory used (heap): $\theta(n)$

Similar to Narlikar scheduler

$(\theta(n + pK\infty) = \theta(n) \text{ space})$

Green nodes are executed before white nodes
Narlikar Algorithm - Idea

- Perform a $p$-leftmost execution of the DAG

$p$-depth first execution for $p = 2$
Q<sub>in</sub>, Q<sub>out</sub> are FIFO queues that support parallel accesses.

R is a priority queue that maintains the depth first order of all threads in the system.
A processor executes a thread until:
- spawn
- memory allocation
- return

- Processor puts thread in $Q_{in}$, gets new thread from $Q_{out}$
- Scheduler thread moves threads from $Q_{in}$ to $R$, performs spawns, moves the leftmost $p$ to $Q_{out}$
“Voodoo” parameter $K$

If a thread wants to allocate more than $K$ bytes, preempt it

To allocate $M$, where $M > K$, put thread to sleep for $M/K$ scheduling rounds.
Problems with Narlikar

- Large scheduling overhead (can be more than 400 times slower than the WS scheduler)
  - Bad locality: must preempt on every spawn
  - Contention on global data structures
  - Bookkeeping performed by scheduling thread
  - Wasted processor time (bad scalability)

- As of yet, haven’t performed empirical tests to determine a breakdown of overhead
Hybrid Scheduler Idea

- Keeping track of left-to-right ordering is expensive
- What about just delaying the threads that wish to perform large memory allocations?
- Can we achieve some space efficiency with a greedy scheduler biased toward non-memory intensive threads?
Hybrid Algorithm

- Start with randomized Work-stealing scheduler
- Preempt threads that perform large memory allocations and put them to sleep
- Reactivate sleeping threads when work-stealing
## Hybrid Algorithm

<table>
<thead>
<tr>
<th>Deque</th>
<th>Processor</th>
<th>Sleep Queue</th>
</tr>
</thead>
<tbody>
<tr>
<td>wake_time: -</td>
<td></td>
<td>wake_time: 1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>wake_time: 2</td>
</tr>
<tr>
<td></td>
<td></td>
<td>wake_time: 5</td>
</tr>
<tr>
<td></td>
<td></td>
<td>wake_time: 5</td>
</tr>
<tr>
<td></td>
<td></td>
<td>wake_time: 8</td>
</tr>
</tbody>
</table>

**current_time:** 0
Hybrid Algorithm

current_time: 0

Deque

take_time: -

Processor

Sleep Queue

wake_time: 1
wake_time: 2
wake_time: 5
wake_time: 5
wake_time: 8

Get thread from bottom of deque
Hybrid Algorithm

<table>
<thead>
<tr>
<th>Deque</th>
<th>Processor</th>
<th>Sleep Queue</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>wake_time: -</td>
<td>wake_time: 1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>wake_time: 2</td>
</tr>
<tr>
<td></td>
<td></td>
<td>wake_time: 5</td>
</tr>
<tr>
<td></td>
<td></td>
<td>wake_time: 5</td>
</tr>
<tr>
<td></td>
<td></td>
<td>wake_time: 8</td>
</tr>
</tbody>
</table>

Get thread from bottom of deque

Before `malloc(size),`

`sleep_rounds = f(size+current_allocation)`
Hybrid Algorithm

Get thread from bottom of deque

Before `malloc(size)`,
  
  `sleep_rounds = f(size + current_allocation)`

If `sleep_rounds > 0`,
  
  `wake_time = sleep_rounds + current_time`
Hybrid Algorithm

Get thread from bottom of deque

Before `malloc(size),`

`sleep_rounds = f(size+current_allocation)`

If `sleep_rounds > 0,`

`wake_time = sleep_rounds + current_time`

and insert thread into sleep queue
Hybrid Algorithm

Get thread from bottom of deque

**Before** `malloc(size),`

- `sleep_rounds = f(size+current_allocated)`

**If** `sleep_rounds > 0`,

- `wake_time = sleep_rounds + current_time`

and insert thread into sleep queue
Hybrid Algorithm

If no threads on deque, 
increment current_time
Hybrid Algorithm

If no threads on deque,
increment current_time

current_time: 1
wake_time: 1
wake_time: 2
wake_time: 4
wake_time: 5
wake_time: 5
wake_time: 8
Hybrid Algorithm

If no threads on deque, 
**increment** \textit{current\_time} 
if first thread in Sleep Queue is ready, 
get thread from Sleep Queue

<table>
<thead>
<tr>
<th>Deque</th>
<th>Processor</th>
<th>Sleep Queue</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>wake_time: 1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>wake_time: 2</td>
</tr>
<tr>
<td></td>
<td></td>
<td>wake_time: 4</td>
</tr>
<tr>
<td></td>
<td></td>
<td>wake_time: 5</td>
</tr>
<tr>
<td></td>
<td></td>
<td>wake_time: 5</td>
</tr>
<tr>
<td></td>
<td></td>
<td>wake_time: 8</td>
</tr>
</tbody>
</table>
Hybrid Algorithm

If no threads on deque,
increment current_time
if first thread in Sleep Queue is ready,
get thread from Sleep Queue
If no threads on deque,
  increment current_time
if first thread in Sleep Queue is ready,
  get thread from Sleep Queue
  reset wake_time and current_allocated
execute it
Hybrid Algorithm

current_time: 1

Deque

Processor

Sleep Queue

If no threads on deque,
increment current_time
if first thread in Sleep Queue is ready,
get thread from Sleep Queue
reset wake_time and current_allocated
execute it
otherwise, work-steal
How long to Sleep?

- Want sleep time to be proportional to the size of the memory allocation
- Increment time on every work-steal attempt
- Scale with number of processors
- Place for future improvement?

Current function

\[
sleep\_rounds = \text{floor}\left(\frac{\text{size}}{\alpha + \beta \times p}\right)
\]

\(\alpha\) and \(\beta\) are “voodoo” parameters
## Empirical Results

### Peak Memory Usage

<table>
<thead>
<tr>
<th>Number of Processors</th>
<th>Cilk</th>
<th>Hybrid</th>
<th>Narlikar</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>163840</td>
<td>163840</td>
<td>163840</td>
</tr>
<tr>
<td>8</td>
<td>327680</td>
<td>327680</td>
<td>163840</td>
</tr>
<tr>
<td>12</td>
<td>409600</td>
<td>327680</td>
<td>204800</td>
</tr>
<tr>
<td>16</td>
<td>491520</td>
<td>368640</td>
<td>204800</td>
</tr>
<tr>
<td>32</td>
<td>614400</td>
<td>368640</td>
<td>204800</td>
</tr>
</tbody>
</table>

![Graph showing peak memory usage for different numbers of processors and algorithms: Cilk, Hybrid, Narlikar.](image)
## Empirical Results

### Running Time

<table>
<thead>
<tr>
<th>Number of Processors</th>
<th>Cilk</th>
<th>Hybrid</th>
<th>Narlikar</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>6.3</td>
<td>7.8</td>
<td>937</td>
</tr>
<tr>
<td>8</td>
<td>3.2</td>
<td>4.5</td>
<td>878</td>
</tr>
<tr>
<td>12</td>
<td>2.3</td>
<td>2.6</td>
<td>855</td>
</tr>
<tr>
<td>16</td>
<td>1.8</td>
<td>2.3</td>
<td>837</td>
</tr>
<tr>
<td>32</td>
<td>1.8</td>
<td>2.2</td>
<td></td>
</tr>
</tbody>
</table>

### Diagram

A line graph showing running time in seconds for different numbers of processors. The graph compares Cilk, Hybrid, and Narlikar, with Narlikar showing the highest running time at 937 seconds for 4 processors, and a range of 837 seconds to 878 seconds for 8 processors.
Future Work on Hybrid Scheduler

- Find the best sleep function and values for “voodoo” parameters
- Optimize the implementation to reduce scheduling overhead
- Determine theoretical space bound
- More detailed empirical analysis
Conclusions

- Narlikar scheduler provides a provably good space bound but incurs a large scheduling overhead.
- It appears that it is possible to achieve space usage that scales well with the number of processors while retaining much of the efficiency of work-stealing.