Dynamic Processor Allocation for Adaptively Parallel Jobs
What is the problem?

[kunal@ygg ~]$ ./strassen --nproc 4

[sidsen@ygg ~]$ ./nfib --nproc 32

[bradley@ygg ~]$ ./nfib --nproc 16

Allocate the processors fairly and efficiently
Why so Dynamic Scheduling?

- Considers all the jobs in the system.
- Programmer doesn’t have to specify the number of processors.
- Parallelism can change during execution.

```
[kunal@ygg ~]$ ./strassen --nproc 4
```

![Diagram showing time vs. parallelism]
Allocation vs. Scheduling

Job 1

Job 2

... (ellipsis)

Job n

Operating System

P1 P2 P3 P4 P5 P6 ...... Pk
Terminology

- The parallelism of a job is dynamic
  - adaptively parallel jobs—jobs for which the number of processors that can be used without waste varies during execution.

- At any given time, each job $j$ has a
  - desire—the maximum number of efficiently usable processors, or the parallelism of the job ($d_j$).
  - allocation—the number of processors allotted to the job ($a_j$).
Terminology

- We want to allocate processors to jobs in a way that is
  - \textit{fair}—whenever a job receives fewer processors than it desires, all other jobs receive at most one more processor than this job received.
    - \( a_j < d_j \Rightarrow (a_j + 1) \) is a max
  - \textit{efficient}—no job receives more processors than it desires, and we use as many processors as possible.
    - \( \forall j \ a_j \leq d_j \)
    - \( \exists j \ a_j < d_j \Rightarrow \) there are no free processors
Overall Goal

Design and implement a *fair* and *efficient* dynamic processor *allocation* system for *adaptively parallel jobs*. 
Example: Fair and Efficient Allocation

Job 1

Job 2

Job 3

Job 4

Job 5

Job 6
Assumptions

- All jobs are Cilk jobs.
- Jobs can enter and leave the system at will.
- All jobs are mutually trusting, in that they will
  - stay within the bounds of their allocations.
  - communicate their desires honestly.
- Each job has at least one processor.
- Jobs have some amount of time to reach their allocations.
High-Level Sequence of Events

1. Estimate desire
2. Report current desire
3. Recalculate allocations
4. Get allocation
5. Adjust allocation (add/remove processors)
Main Algorithms

- **(1, 2)** Dynamically estimate the current desire of a job.
  - Steal rate (Bin Song)
  - Number of threads in ready deque
- **(3)** Dynamically determine the allotment for each job such that the resulting allocation is fair and efficient.
  - SRLBA algorithm (Bin Song)
  - Global allocation algorithm
- **(4, 5)** Converge to the granted allocation by increasing/decreasing number of processors in use.
  - While work-stealing?
  - Periodically by a background thread?
Desire Estimation

- (1) Estimate processor desire $d_j$: add up the number of threads in the ready deques of each processor and divide by a constant.

\[
\begin{align*}
&\begin{array}{c}
H \\
T
\end{array} + 
\begin{array}{c}
H \\
T
\end{array} + 
\begin{array}{c}
H \\
T
\end{array} + 
\begin{array}{c}
H \\
T
\end{array} \\
&k > 3
\end{align*}
\]

- (2) Report the desire to the processor allocation system.
Adjusting the Allocation

- **(4)** Get the allocation $a_{\text{new}}$.

- **(5)** Adjust the allocation.
  - If $a_{\text{new}} < a_{\text{old}}$, remove $(a_{\text{old}} - a_{\text{new}})$ processors
  - If $a_{\text{new}} > a_{\text{old}}$, add $(a_{\text{new}} - a_{\text{old}})$ processors
Implementation Details

- Adding up the number of threads in the ready deques
  - While work-stealing
  - Periodically by a background thread
  Too late!

- Removing processors
  - While work-stealing
  - Periodically by a background thread
  Complicated

- Adding processors
  - While work-stealing
  - Periodically by a background thread
  Bad idea
Processor Allocation

Start-up

- Job 1: Desire=4, Alloc=0
- Job 2: Desire=6, Alloc=0
- Job 3: Desire=5, Alloc=5
- Job 4: Desire=5, Alloc=1

Free Processors: 16
Processor Allocation

- Job 2 decreases desire.

<table>
<thead>
<tr>
<th>Job 1</th>
<th>Job 2</th>
<th>Job 3</th>
<th>Job 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>Desire=4</td>
<td>Desire=6→4</td>
<td>Desire=5</td>
<td>Desire=5</td>
</tr>
<tr>
<td>Alloc=4</td>
<td>Alloc=4</td>
<td>Alloc=4</td>
<td>Alloc=4</td>
</tr>
</tbody>
</table>

Free Processors: 0

No Reallocation!!
Processor Allocation

- Job 1 decreases desire.

```
<table>
<thead>
<tr>
<th>Job 1</th>
<th>Job 2</th>
<th>Job 3</th>
<th>Job 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>Desire=4 → 2</td>
<td>Desire=6</td>
<td>Desire=5</td>
<td>Desire=5</td>
</tr>
<tr>
<td>Alloc=4 → 2</td>
<td>Alloc=4 → 5</td>
<td>Alloc=4 → 5</td>
<td>Alloc=4</td>
</tr>
</tbody>
</table>
```

Free Processors

Reallocation!!
Processor Allocation

- Job 2 Increases desire.

<table>
<thead>
<tr>
<th>Job</th>
<th>Desire</th>
<th>Alloc</th>
</tr>
</thead>
<tbody>
<tr>
<td>Job 1</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>Job 2</td>
<td>6 → 8</td>
<td>5</td>
</tr>
<tr>
<td>Job 3</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>Job 4</td>
<td>5</td>
<td>4</td>
</tr>
</tbody>
</table>

Free Processors: 0

No Reallocation!!
Processor Allocation

Job 1 *Increases* desire.

Job 1

Desire: 2 → 5
Alloc: 2 → 3

Job 2

Desire: 8
Alloc: 5 → 4

Job 3

Desire: 5
Alloc: 5 → 4

Job 4

Desire: 5
Alloc: 4

Free Processors: 0

Reallocate! !!
Implementation Details

<table>
<thead>
<tr>
<th>min_depr_alloc:4</th>
<th>max_alloc:5</th>
</tr>
</thead>
<tbody>
<tr>
<td>Job Id:1</td>
<td>Job Id:2</td>
</tr>
<tr>
<td>Desire:6</td>
<td>Desire:2</td>
</tr>
<tr>
<td>Alloc:4</td>
<td>Alloc:2</td>
</tr>
</tbody>
</table>
Processor Allocation

- Job 1 decreases desire.

```
Job 1: Desire=4 → 2, Alloc=4 → 2
Job 2: Desire=6, Alloc=4 → 5
Job 3: Desire=5, Alloc=4 → 5
Job 4: Desire=5

Free Processors
```

mda=4
ma=5
Implementation

<table>
<thead>
<tr>
<th>min_depr_alloc:4</th>
</tr>
</thead>
<tbody>
<tr>
<td>max_alloc:5</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Job Id:1</th>
<th>Job Id:2</th>
<th>Job Id:3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Design:6</td>
<td>Desire:2</td>
<td>Design:7</td>
</tr>
<tr>
<td>Alloc:4</td>
<td>Alloc:2</td>
<td>Alloc:5</td>
</tr>
</tbody>
</table>

- When desire of job j decreases: if \((\text{new\_desire}<\text{alloc})\)
  - take processors from \(j\) and give to jobs having \(\text{min\_depr\_alloc}\).
- When desire of job j increases: if \((\text{alloc}<\text{mda})\)
  - take processors from jobs having \(\text{max\_alloc}\) and give them to \(j\) until \(j\) reaches \(\text{min\_depr\_alloc}\) or \(\text{new\_desire}\).
Processor Allocation

- **Job 1** Increases desire.

```
Job 1
Desire=2 → 5
Alloc=2 → 3

Job 2
Desire=8
Alloc=5 → 4

Job 3
Desire=5
Alloc=5 → 4

Job 4
Desire=5
Alloc=4
```

Free Processors

- mda=4
- ma=5
Experiments

- Correctness: Does it work?

- Effectiveness: Are there cases where it is better than the static allocation?

- Responsiveness: How long does it take the jobs to reach their allocation?
Conclusions

- The desire estimation and processor allocation algorithms are simple and easy to implement.
- We’ll see how well they do in practice once we’ve performed the experiments.
- There are many ways of improving the algorithms and in many cases it is not clear what we should do.
Job Tasks (Extensions)

- Incorporate heuristics on steal-rate (Bin Song’s idea).
- Remove processors in the background thread, not while work stealing.
  - Need a mechanism for putting processors with pending work to sleep
  - When adding processors, wake up processors with pending work first

Processor Allocation System

1. ...
2. ...
3. ...
4. ...
5. ...
Processor Allocation System (Extensions)

- Use a sorted data structure for job entries.
  - Sort by desires
  - Sort by allocations
  - Group jobs:
    - Desires satisfied ($a_j = d_j$)
    - Minimum deprived allocation ($a_j = \text{min}_\text{depr}_\text{alloc}$)
    - Maximum allocation ($a_j = \text{max}_\text{alloc}$)

- Need fast inserts/deletes and fast sequential walk.

[Diagram of Processor Allocation System with Job j]
Processor Allocation System (Extensions)

- Rethink definitions of fairness and efficiency.
  - Incorporate histories of processor usage for each job
  - Implement a mechanism for assigning different priorities to users or jobs
- Move the processor allocation system into the kernel.
  - Jobs still report desires since they know best
  - How to group the jobs?
    - Make classes of jobs (Cilk, Emacs, etc.)
    - Group by user (sidsen, kunal, etc.)
Questions?