Work-stealing for mixed-mode parallelism by deterministic team-building

Martin Wimmer
Jesper Larsson Träff
University of Vienna
PEPPHER context

- PEPPHER focusses on performance portability and programmability aspects
- Component-based model
  - algorithmic kernels as components
- DAG-structured model of computation with component-tasks
- Scheduler sees component-task as blackbox
  - It may be scheduled to different types of processors
  - Explicit resource requirements
  - It may be a parallel kernel
    - e.g. OpenMP kernel
Support for parallel component-tasks requires extensions to classical DAG-scheduling
- Co-scheduling on multiple processors
- Support for blocking synchronization between threads of a task
- Subsequent numbering of threads executing task
  - Many algorithms require numbering of threads
  - Required for OpenMP kernels
- Memory locality issues
Programmability aspect

- Some parallel algorithms are easier/more efficient to implement in task-based models
  - e.g. divide-and-conquer algorithms
- Others require SPMD-style programming with blocking synchronization
  - Difficult to map to task-based models

- Ability to compose both types of kernels in single applications may be beneficial
- Term: mixed-mode parallel applications
- Model: Task can spawn other tasks with fixed thread-requirement $\geq 1$
Some solutions for mixed-mode parallelism

- Use continuations instead of blocking synchronization
  - Difficult to implement
  - Sometimes small granularity of tasks
- Language extension + compiler support
  - Phasers in Habanero Java
  - Clocks in X10
- Centralized scheduling approaches
  - e.g. Communicating M-tasks
  - Many others
Motivating example: Quicksort

- The classical, well-known task-parallel quicksort:
  - Start off with single task
  - Partition data
  - Spawn one task for each generated subsequence
  - Switch to sequential sorting algorithm for smaller subsequences

```java
if n ≤ CUTOFF then
    return sequential_sort(data, n)
else
    pivot ← partition(data, n)
    async qsort(data, pivot)
    async qsort(data + pivot + 1, pivot − n − 1)
sync
end if
```
Quicksort scalability problems

- At start, no parallelism
- Partitioning is sequential, $O(n)$
- Partitioning must be done at least once before first fork
- At least $\log p$ steps, before all processors have work

- **Sequential bottleneck at least** $O(n)$
Data-parallel partitioning

- Attacking sequential bottleneck
- Proposed by P. Tsigas and Y. Zhang in 2003
- Block-wise decomposition of data
- Threads acquire blocks at each side - try to neutralize
  (all data in neutralized blocks are larger or smaller than pivot)
- Remaining blocks sequentially neutralized at end

```
P2  P3  P1       P3  P1  P2
```

- Unfinished block
- Neutralized block
Quicksort with parallel tasks

- Start off with parallel tasks that do parallel partitioning
- For each newly spawned task determine best number of threads
- For 1-processor tasks use sequential partitioning

```python
if np = 1 then
    return fork_join_qsort(data, n)
else
    pivot ← parallel_partition(data, n)
    if localId = 0 then
        async (getBestNp(pivot))
        par_qsort(data, pivot)
        async (getBestNp(n-pivot-1))
        par_qsort(data+pivot+1, n-pivot-1)
    sync
end if
end if
```
Decentralized scheduling for mixed-mode parallelism

Our solution: work-stealing with deterministic team-building

- Follows the work-stealing philosophy
  - Local work queues
  - Threads act autonomously
  - Only communicate if out of work
  - Depth-first scheduling
- Low overhead
Modifications to standard work-stealing

- Impose a hierarchy on processors in system
  - Should take memory hierarchy into account
- At level 0 each processor is in a group of its own
- At higher levels, processors are grouped together

Teams will be built out of processor groups
We assume a binary tree for the hierarchy
  - allows to calculate partner thread ids on the fly
Modified stealing procedure

- Deterministic stealing pattern
  - Visit log p partners (one for each level in hierarchy) until we find some work

- Partner for level l is selected by XOR of thread-id with x in the range: $2^{l-1} \leq x < 2^l$
  - Depending on policy, x may be fixed (for completely deterministic schemes) or random
For partner visited at level $l$:

- Check whether it is building a team requiring at least $2^l$ threads
  - If so, join team and exit
- Try to steal task requiring at most $2^{l-1}$ threads
  - On success, exit and coordinate stolen task
- Move on to next level
Team building (coordination) procedure

- Required to build team to execute parallel task
- Executed by all threads already in the team

- If team is built, start task execution
- Otherwise go through hierarchy as in stealing
  - Only visit partners required for task execution
  - On successful steal exit coordination
  - Deterministic tie-breaking if conflicting teams are built
Team building (coordination) procedure

- Required to build team to execute parallel task
- Executed by all threads already in the team

- If team is built, start task execution
- Otherwise go through hierarchy as in stealing
  - Only visit partners required for task execution
  - On successful steal exit coordination
  - Deterministic tie-breaking if conflicting teams are built
Team building (coordination) procedure

- Required to build team to execute parallel task
- Executed by all threads already in the team

- If team is built, start task execution
- Otherwise go through hierarchy as in stealing
  - Only visit partners required for task execution
  - On successful steal exit coordination
  - Deterministic tie-breaking if conflicting teams are built
Team building (coordination) procedure

- Required to build team to execute parallel task
- Executed by all threads already in the team

- If team is built, start task execution
- Otherwise go through hierarchy as in stealing
  - Only visit partners required for task execution
  - On successful steal exit coordination
  - Deterministic tie-breaking if conflicting teams are built
Implementation

- Implemented in C++ with pthreads
- Interface comparable to tasks in Intel TBB
- Lock-free implementation
  - Uses compare-and-swap (CAS) and fetch-and-add
  - Registration and deregistration for a team requires a single CAS per thread
  - One word per thread stores team-building information
- Standard lock-free queue implementation for task queues
- Completely deterministic, configurable stealing policy
### Experimental results

- Measured on a 32 core Intel Nehalem EX system.
- Average time over 10 runs in seconds

<table>
<thead>
<tr>
<th>Type</th>
<th>Size</th>
<th>Seq/STL</th>
<th>SeqQS</th>
<th>Fork</th>
<th>SU</th>
<th>Cilk++</th>
<th>SU</th>
<th>MMPar</th>
<th>SU</th>
</tr>
</thead>
<tbody>
<tr>
<td>Random</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>100000000</td>
<td>4.963</td>
<td>5.265</td>
<td>0.294</td>
<td>3.5</td>
<td>0.193</td>
<td>5.3</td>
<td>0.186</td>
<td>5.5</td>
<td></td>
</tr>
<tr>
<td>134217727</td>
<td>15.888</td>
<td>16.617</td>
<td>3.103</td>
<td>5.1</td>
<td>2.525</td>
<td>6.3</td>
<td>1.835</td>
<td>8.7</td>
<td></td>
</tr>
</tbody>
</table>

Future Work

- Investigate further mixed-mode parallel applications
  - PEPPHER benchmarks
- Integration into the PEPPHER framework
  - StarPU scheduler plugin
  - Standalone scheduler
- Support for malleable/moldable tasks within certain limits
  - Automatic selection of thread requirements on spawn
    - depending on processor utilization and task performance
  - Vary thread requirements after stealing
