Scheduling Task Parallelism on Multi-Socket Multicore Systems

Stephen Olivier, UNC Chapel Hill
Allan Porterfield, RENCI
Kyle Wheeler, Sandia National Labs
Jan Prins, UNC Chapel Hill
Outline

Introduction and Motivation

Scheduling Strategies

Evaluation

Closing Remarks
Outline

Introduction and Motivation

Scheduling Strategies

Evaluation

Closing Remarks
Task Parallel Programming in a Nutshell

• A task consists of executable code and associated data context, with some bookkeeping metadata for scheduling and synchronization.

• Tasks are significantly more lightweight than threads.
  • Dynamically generated and terminated at run time
  • Scheduled onto threads for execution

• Used in Cilk, TBB, X10, Chapel, and other languages
  • Our work is on the recent tasking constructs in OpenMP 3.0.
Simple Task Parallel OpenMP Program: Fibonacci

```c
int fib(int n) {
    int x, y;
    if (n < 2) return n;

    #pragma omp task
    x = fib(n - 1);
    #pragma omp task
    y = fib(n - 2);
    #pragma omp taskwait

    return x + y;
}
```

Useful Applications

- Recursive algorithms
  - E.g. Mergesort
- List and tree traversal
- Irregular computations
  - E.g., Adaptive Fast Multipole
- Parallelization of while loops
- Situations where programmers might otherwise write a difficult-to-debug low-level task pool implementation in pthreads
Goals for Task Parallelism Support

- **Programmability**
  - Expressiveness for applications
  - Ease of use

- **Performance & Scalability**
  - Lack thereof is a serious barrier to adoption
  - Must improve software run time systems
Issues in Task Scheduling

- Load Imbalance
  - Uneven distribution of tasks among threads

- Overhead costs
  - Time spent creating, scheduling, synchronizing, and load balancing tasks, rather than doing the actual computational work

- Locality
  - Task execution time depends on the time required to access data used by the task
The Current Hardware Environment

• Shared Memory is not a free lunch.
  • Data can be accessed without explicitly programmed messages as in MPI, but not always at equal cost.

• However, OpenMP has traditionally been agnostic toward affinity of data and computation.
  • Most vendors have (often non-portable) extensions for thread layout and binding.
  • First-touch traditionally used to distribute data across memories on many systems.
Example UMA System

- Incarnations include Intel server configurations prior to Nehalem and the Sun Niagara systems
- Shared bus to memory
Example Target NUMA System

- Incarnations include Intel Nehalem/Westmere processors using QPI and AMD Opterons using HyperTransport.
- Remote memory accesses are typically higher latency than local accesses, and contention may exacerbate this.
Outline

Introduction and Motivation

Scheduling Strategies

Evaluation

Closing Remarks
Work Stealing

- Studied and implemented in Cilk by Blumofe et al. at MIT
- Now used in many task-parallel run time implementations
- Allows dynamic load balancing with low critical path overheads since idle threads steal work from busy threads
- Tasks are enqueued and dequeued LIFO and stolen FIFO for exploitation of local caches

Challenges

- Not well suited to shared caches now common in multicore chips
- Expensive off-chip steals in NUMA systems
PDFS (Parallel Depth-First Schedule)

- Studied by Blelloch et al. at CMU
- Basic idea: Schedule tasks in an order close to serial order
- If sequential execution has good cache locality, PDFS should as well.
- Implemented most easily as a shared LIFO queue
- Shown to make good use of shared caches

Challenges

- Contention for the shared queue
- Long queue access times across chips in NUMA systems
Our Hierarchical Scheduler

- Basic idea: Combine benefits of work stealing and PDFS for multi-socket multicore NUMA systems
- Intra-chip shared LIFO queue to exploit shared L3 cache and provide natural load balancing among local cores
- FIFO work stealing between chips for further low overhead load balancing while maintaining L3 cache locality
  - Only one thief thread per chip performs work stealing when the on-chip queue is empty
  - Thief thread steals enough tasks, if available, for all cores sharing the on-chip queue
Implementation

• We implemented our scheduler, as well as other schedulers (e.g., work stealing, centralized queue), in extensions to Sandia’s Qthreads multithreading library.

• We use the ROSE source-to-source compiler to accept OpenMP programs and generate transformed code with XOMP outlined functions for OpenMP directives and run time calls.

• Our Qthreads extensions implement the XOMP functions.

• ROSE-transformed application programs are compiled and executed with the Qthreads library.
Outline

Introduction and Motivation

Scheduling Strategies

Evaluation

Closing Remarks
Evaluation Setup

• Hardware: Shared memory NUMA system
  • Four 8-core Intel x7550 chips fully connected by QPI
• Compiler and Run time systems: ICC, GCC, Qthreads
  • Five Qthreads implementations
    • Q: Per-core FIFO queues with round robin task placement
    • L: Per-core LIFO queues with round robin task placement
    • CQ: Centralized queue
    • WS: Per-core LIFO queues with FIFO work stealing
    • MTS: Per-chip LIFO queues with FIFO work stealing
Evaluation Programs

• From the Barcelona OpenMP Tasks Suite (BOTS)
  • Described in ICPP ‘09 paper by Duran et al.
  • Available for download online

• Several of the programs have cut-off thresholds
  • No further tasks created beyond a certain depth in the computation tree
Health Simulation Performance

![Graph showing performance improvement with different numbers of threads](image)

The University of North Carolina at Chapel Hill
Health Simulation Performance

Stock Qthreads scheduler (per-core FIFO queues)
Health Simulation Performance

Per-core LIFO queues
Health Simulation Performance

Per-core LIFO queues with FIFO work stealing
Per-chip LIFO queues with FIFO work stealing
Health Simulation Performance

![Bar chart showing performance comparison across different thread counts and libraries]
Sort Benchmark

![Sort Benchmark Graph](image)

The University of North Carolina at Chapel Hill
NQueens Problem
Fibonacci
Strassen Multiply

![Graph showing speedup of Strassen Multiply with different numbers of threads and various algorithms]

- MTS
- WS
- CQ
- L
- Q
- ICC
- GCC

The University of North Carolina at Chapel Hill
Protein Alignment

Single task startup

For loop startup
Sparse LU Decomposition

Single task startup

For loop startup
Per-Core Work Stealing vs. Hierarchical Scheduling

- Per-core work stealing exhibits lower variability in performance on most benchmarks
- Both per-core work stealing and hierarchical scheduling Qthreads implementations had smaller standard deviations than ICC on almost all benchmarks

<table>
<thead>
<tr>
<th>Configuration</th>
<th>Alignment (single)</th>
<th>Alignment (for)</th>
<th>Fib</th>
<th>Health</th>
<th>NQueens</th>
<th>Sort</th>
<th>SparseLU (single)</th>
<th>SparseLU (for)</th>
<th>Strassen</th>
</tr>
</thead>
<tbody>
<tr>
<td>ICC 32 threads</td>
<td>4.4</td>
<td>2.0</td>
<td>3.7</td>
<td>2.0</td>
<td>3.2</td>
<td>4.0</td>
<td>1.1</td>
<td>3.9</td>
<td>1.8</td>
</tr>
<tr>
<td>GCC 32 threads</td>
<td>0.11</td>
<td>0.34</td>
<td>2.8</td>
<td>0.35</td>
<td>0.77</td>
<td>1.8</td>
<td>0.49</td>
<td>N/A</td>
<td>1.4</td>
</tr>
<tr>
<td>Qthreads MTS 32 workers</td>
<td>0.28</td>
<td>1.5</td>
<td>3.3</td>
<td>1.3</td>
<td>0.78</td>
<td>1.9</td>
<td>0.15</td>
<td>0.16</td>
<td>1.9</td>
</tr>
<tr>
<td>Qthreads WS 32 shepherds</td>
<td>0.035</td>
<td>1.8</td>
<td>2.0</td>
<td>0.29</td>
<td>0.60</td>
<td>0.90</td>
<td>0.060</td>
<td>0.24</td>
<td>3.0</td>
</tr>
</tbody>
</table>

Standard deviation as a percent of the fastest time
Per-Core Work Stealing vs. Hierarchical Scheduling

- Hierarchical scheduling benefits
- Significantly fewer remote steals observed on almost all programs

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>MTS</th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Steals</td>
<td>Failed</td>
<td>Steals</td>
<td>Failed</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Alignment (single)</td>
<td>1016</td>
<td>88</td>
<td>3695</td>
<td>255</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Alignment (for)</td>
<td>109</td>
<td>122</td>
<td>1431</td>
<td>286</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Fib</td>
<td>633</td>
<td>331</td>
<td>467</td>
<td>984</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Health</td>
<td>28948</td>
<td>10323</td>
<td>295637</td>
<td>47538</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>NQueens</td>
<td>102</td>
<td>141</td>
<td>1428</td>
<td>389</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Sort</td>
<td>1134</td>
<td>404</td>
<td>19330</td>
<td>3283</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SparseLU (single)</td>
<td>18045</td>
<td>8133</td>
<td>68927</td>
<td>24506</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SparseLU (for)</td>
<td>13486</td>
<td>11889</td>
<td>68099</td>
<td>32205</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Strassen</td>
<td>227</td>
<td>157</td>
<td>14042</td>
<td>823</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Per-Core Work Stealing vs. Hierarchical Scheduling

• Hierarchical scheduling benefits

• Lower L3 misses, QPI traffic, and fewer memory accesses as measured by HW performance counters on health, sort

<table>
<thead>
<tr>
<th>Metric</th>
<th>MTS</th>
<th>WS</th>
<th>%Diff</th>
</tr>
</thead>
<tbody>
<tr>
<td>Health</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>L3 Misses</td>
<td>1.16e+06</td>
<td>2.58e+06</td>
<td>38</td>
</tr>
<tr>
<td>Bytes from Memory</td>
<td>8.23e+09</td>
<td>9.21e+09</td>
<td>5.6</td>
</tr>
<tr>
<td>Bytes on QPI</td>
<td>2.63e+10</td>
<td>2.98e+10</td>
<td>6.2</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Metric</th>
<th>MTS</th>
<th>WS</th>
<th>%Diff</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sort</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>L3 Misses</td>
<td>1.03e+07</td>
<td>3.42e+07</td>
<td>54</td>
</tr>
<tr>
<td>Bytes from Memory</td>
<td>2.27e+10</td>
<td>2.53e+10</td>
<td>5.5</td>
</tr>
<tr>
<td>Bytes on QPI</td>
<td>4.35e+10</td>
<td>4.87e+10</td>
<td>5.6</td>
</tr>
</tbody>
</table>
Stealing Multiple Tasks

![Bar Chart](chart.jpg)

- **Performance relative to chunk size 1**
- **Chunk size (number of tasks stolen per steal operation)**

- **Legend:**
  - Bar height indicates the performance relative to chunk size 1 for different chunk sizes.
  - The chart shows a trend where performance increases with larger chunk sizes, peaking around chunk size 8 and then decreasing slightly for larger chunk sizes.

---

35
Outline

- Introduction and Motivation
- Scheduling Strategies
- Evaluation
- Closing Remarks
Looking Ahead

- Our prototype Qthreads run time is competitive with and on some applications outperforms ICC and GCC.
  - Implementing non-blocking task queues could further improve performance.
- Hierarchical scheduling shows potential for scheduling on hierarchical shared memory architectures.
  - System complexity is likely to increase rather than decrease with hardware generations.
Thanks.

- Stephen Olivier, UNC Chapel Hill
- Allan Porterfield, RENCI
- Kyle Wheeler, Sandia National Labs
- Jan Prins, UNC Chapel Hill