Comparison and analysis of parallel tasking performance for an irregular application

Patrick Atkinson, University of Bristol (p.atkinson@bristol.ac.uk)

Simon McIntosh-Smith, University of Bristol
Motivation

- Exploring task parallelism through a new mini-app (https://github.com/UoB-HPC/minifmm)
- Discovering limitations in OpenMP tasking model
- Optimising OpenMP implementation of algorithm through alternatives to task constructs
- Comparing performance of tasking in OpenMP runtime implementations and to other parallel frameworks
- Determining whether using tasks can perform as well as data-parallel implementations whilst reducing code-size
Fast Multipole Method overview

- Used for solving N-body problems
- Reduces time complexity from $O(n^2)$ to $O(n)$
- Compute bound method
- Good fit for tasking due to complex control flow – dependant on particle data
- Applications include: astrophysics, electrostatics, fluid dynamics, electromagnetics
FMM domain decomposition
FMM domain decomposition
FMM domain decomposition
FMM domain decomposition
Method

- Each node in the tree will perform interactions with many other nodes.
- Interaction type determined by distance between nodes and user-defined parameter.
- Recurse until either:
  - If two nodes are well-separated the interaction is approximated (node to node interaction).
  - The leaf level is reached and the particle interaction is calculated directly (particle to particle interaction).
Method

- Each node in the tree will perform interactions with many other nodes.
- Interaction type determined by distance between nodes and user-defined parameter.
- Recurse until either:
  - If two nodes are well-separated the interaction is approximated (node to node).
  - The leaf level is reached and the particle interaction is calculated directly (particle to particle).
Method

- Each node in the tree will perform interactions with many other nodes.
- Interaction type determined by distance between nodes and user-defined parameter.
- Recurse until either:
  - If two nodes are well-separated the interaction is approximated (node to node).
  - The leaf level is reached and the particle interaction is calculated directly (particle to particle).
Using tasks for FMM

• We have many interactions to perform between groups of particles

• Interaction type dependant on distance between tree nodes – not known until runtime

• Tree could be highly imbalanced
Using tasks for FMM

• We have many interactions to perform between groups of particles

• Interaction type dependant on distance between tree nodes – not known until runtime

• Tree could be highly imbalanced

Solution? Use tasks

• Create task for each interaction

• Letting some thread complete the required work at any time

• Need a way to enforce two threads don’t update same values…
Intuitive implementation with task dependencies

```c
function DTT(target, source)
    // calculate distance between target and source
    ...
    if source and target well separated then
        #pragma omp task depend(inout: target)
        ApproximateForce(target, source)
    else if target and source are leaves then
        #pragma omp task depend(inout: target)
        DirectForce(target, source)
    else
        if target.radius > source.radius then
            for each child in target do
                DTT(child, source)
        else
            for each child in source do
                DTT(target, child)
```

- Generate task for each interaction type
- Nodes/cells typically contain $O(100)$ particles - enough work to for single task
- Allows for fine-grained synchronisation with other stages of algorithm using task dependencies
- The order tasks are generated in determines order of execution
Effect of enforcing unnecessary ordering

- Plotting execution of each of the calculation functions
- Whitespace = thread idle time
- Unnecessary ordering of dependencies causes large amounts of idle time
Performance gain from removing dependencies

function DTT(target, source)
...
if source and target well separated then
    #pragma omp task depend(inout: target)
    ApproximateForce(target, source)
else if target and source are leaves then
    #pragma omp task depend(inout: target)
    DirectForce(target, source)
else
    // recurse
    ...

• Investigation – what happens if we remove dependencies?

• Incorrect behaviour due to multiple threads updating same nodes

• However, much better thread utilisation…
Effect of a single thread generating tasks – 24 core Ivybridge

Significantly less idle time than before, however…
Effect of a single thread generating tasks – 24 core Ivybridge

Threads lacking tasks to execute

Thread generating tasks

Intel® HPC Developer Conference 2017
Effect of a single thread generating tasks – 256 threads, KNL

Problem even worse for KNL

Thread generating tasks

Threads 0 – 204 not shown
So two issues…

• Need an efficient way to handle race condition
  -> Ensure mutual exclusion through locks or atomics

• Can’t generate all tasks from single thread
  -> Need to perform tree traversal in parallel
Locking nodes of tree

- Lock target node while updating values
- \texttt{taskyield} – allows programmer to specify task can be suspended
- Combine \texttt{taskyield} with locks so thread encountering task can switch to another task
- \texttt{untied task} – task can be resumed by any thread
- Can combine both \texttt{taskyield} and \texttt{untied} with locks

```c
void force_calculation(node target, node source) {
    omp_set_lock(&target->lock);
    // calculate or approximate force
    ...
    omp_unset_lock(&target->lock);
}
```

```c
void force_calculation(node target, node source) {
    int locked = 0;
    while (!locked)
    {
        locked = omp_test_lock(&target->lock);
        if (!locked)
        {
            //pragma omp taskyield
        }
        // calculate or approximate force
        ...
        omp_unset_lock(&target->lock);
    }
}
```
Atomically updating values

- Alternatively can atomically update values instead of locking entire node
- Four atomics per node update (task)
- Which is better locks or atomics? It depends
- On KNL atomics performed worse, on Xeon CPU depends if we can keep lock contention low
- Can lower lock contention with less work per node
Using different lock implementations

- Can specify in OpenMP which lock implementation to use
- First supported in Intel OpenMP - still not present in GCC (7.2)
- Can use locks that are better for high contention and/or speculative locks
- Default lock implementation worked best in miniFMM, all other combinations resulted in poorer performance

```c
omp_lock_t lock;
omp_init_lock_with_hint(&lock, omp_lock_hint_<type>);
omp_lock_init_none
omp_lock_init_contended
omp_lock_init_uncontended
omp_lock_init_speculative
omp_lock_init_nonspeculative
```
Commutative dependencies

- Commutative dependency type specifies tasks can run in any order regardless of when they were generated.
- Feature in OmpSs.
- Would mean entire method could be implemented using task dependencies – allows for fine-grained synchronisation between stages.
- But we would still suffer from starvation problem with one thread generating tasks.

```c
#pragma omp task depend(inout: target)
approximate_force(target, source)
```

```c
#pragma omp task depend(commute: target)
approximate_force(target, source)
```
Performance comparison overview

- OpenMP implementations: Intel (17.2), GCC (6.3), Cray (8.5.8), BOLT

- Programming models: OpenMP, OmpSs, CILK, TBB

- Also compared to data-parallel implementation where list of interactions are collected and then performed in a loop over the target nodes

- Typical problem size $\sim O(10^6)$ particles with maximum 500 particles per node
<table>
<thead>
<tr>
<th>Hardware</th>
<th>Broadwell</th>
<th>KNL</th>
<th>Skylake</th>
</tr>
</thead>
<tbody>
<tr>
<td>2x Intel Xeon E5-2699 v4 2.20 GHz</td>
<td><strong>Intel Xeon Phi</strong> 7210 1.30 GHz</td>
<td>2x Intel Xeon Gold 6152 2.10 GHz</td>
<td></td>
</tr>
<tr>
<td>2 Sockets</td>
<td>64 cores</td>
<td>2 Sockets</td>
<td></td>
</tr>
<tr>
<td>22 cores per socket</td>
<td>Up to 4 threads per core</td>
<td>22 cores per socket</td>
<td></td>
</tr>
<tr>
<td>Up to 2 threads per core</td>
<td>512-bit width vectors</td>
<td>Up to 2 threads per core</td>
<td></td>
</tr>
<tr>
<td>256-bit width vectors</td>
<td></td>
<td>512-bit width vectors</td>
<td></td>
</tr>
</tbody>
</table>
Results – Dual socket 22-core Broadwell

- Most OpenMP implementations, CILK, TBB, and OmpSs scale well and are close to data-parallel algorithm

- Intel runtimes (OpenMP, CILK, TBB) and OmpSs perform best whilst Cray and GCC lag behind

- Can be explained by measuring time outside of computational work, at 44 cores:
  - Intel 2.01%
  - GNU 8.31%
  - Cray 9.13%
Results – 64 core KNL

1 thread per core

- Data-parallel code slightly outperforms task-parallel implementations

- Good OmpSs performance required changing scheduler to use queue per thread

- Performance **degrades** >~120 threads using GCC

4 threads per core

---

Intel® HPC Developer Conference 2017

University of BRISTOL
Results – Dual socket 22-core **Skylake**

![Graph showing speedup vs. cores for OMP-Intel and OMP-GNU](image-url)

- Blue dots represent OMP-Intel.
- Red squares represent OMP-GNU.
Summary

- Tasks can significantly reduce lines of code whilst achieving good performance.

- Difficult to express parallelism in irregular methods like FMM using current OpenMP task constructs – future changes in OpenMP could remedy this.

- In the meantime, alternatives to task dependencies exist.

- Most programming models and implementations achieve good scaling/performance until scaling to high thread counts.
Publications

Pragmatic Kernels, and Mini-apps including TeaLeaf, CloverLeaf, miniFMM, and SNAP
https://github.com/UK-MAC/
https://github.com/UoB-HPC/

On the performance of parallel tasking runtimes for an irregular fast multipole method application
Atkinson, Patrick and McIntosh-Smith, Simon

Assessing the performance portability of modern parallel programming models using TeaLeaf
Martineau, Matt, McIntosh-Smith, Simon, and Gaudin, Wayne

Many-core Acceleration of a Discrete Ordinates Transport Mini-app at Extreme Scale
Deakin, Tom, McIntosh-Smith, Simon N, and Gaudin, Wayne

The Productivity, Portability and Performance of OpenMP 4.5 for Scientific Applications Targeting Intel CPUs, IBM CPUs, and NVIDIA GPUs
Martineau, Matt and McIntosh-Smith, Simon
Extra slides
1 threads per core Broadwell

2 threads per core KNL