Contents

1 Introduction

2 Parallel Force Calculation

3 Interaction List Compression
Introduction

- Short ranged particle simulation (Lennard Jones Potential)
- Cutoff-radius to specify interaction partners of a particle

for $i = 1$ to $N$ do 
  foreach $j \in interaction\_list(i)$ do 
    interact((i, j))

procedure interact((i, j))
  $r =$ distance vector between $P_i$ and $P_j$
  ...
  if $|r| < r_{cut}$ then
    $t_{force} =$ compute\_force($P_i$, $P_j$, $r$)
    force update $P_i$
    force update $P_j$

- Interaction list stores particle interactions
- Interaction list is computed and used for several time steps (not shown here)
- To handle particles moving into Cutoff-Radius, it is slightly increased
Motivation

- Interaction lists are part of lot of professional simulation codes
- Interaction lists may require a large amount of memory, sometimes more than particle data like positions and forces
- Caching interaction list not possible (larger than cache, each element read only once)
- Loading the interaction list requires a large amount of memory bandwidth possibly shared by several cores
- Size of simulation limited by particle data and interaction list data

→ Compressing interaction list will reduce memory consumption and increase simulation speed
Interaction Matrix $\Sigma$

- Divide simulation volume into $N_1 \times N_2 \times N_3$ subvolumes $B_u$, $1 \leq u \leq N_1 \cdot N_2 \cdot N_3$
- Partitioning of interaction list according to subvolumes where particles reside creates interaction matrix $\Sigma$
- Entry $S_{uv}$ from $\Sigma$ is the set of interactions between particles residing in subvolume $B_u$ and $B_v$, that were originally located in the interaction list
- $\Sigma$ has only nonempty entries above the main diagonal
- Number of interactions in $S_{uv}$ varies; main diagonal elements have highest number of interactions
- In the parallel algorithm all interactions in $S_{uv}$ are computed by one thread; computing $S_{uv}$ is a task
Recursive Mapping of Tasks to Cores

- Set of available cores $T_0$ and $\Sigma$ are both split recursively
  - $T_0$ is split into $T_1$ and $T_2$
  - $\Sigma$ is split into $S_1$, $S_2$, $S_3$, and $S_4$
  - $S_1$ ($S_4$) is computed recursively by the cores in $T_1$ ($T_2$)
  - If $S_3$ is below main diagonal it is not computed and $S_2$ is computed by all threads in $T_0$
  - Else ( $S_3$ is above main diagonal) $S_2$ ($S_3$) is computed by cores in $T_2$ ($T_1$)

- Example on right shows a mapping of tasks (sets of interactions) to 16 cores
Recursive Mapping of Tasks to Cores

- Set of available cores $T_0$ and $\Sigma$ are both split recursively
  - $T_0$ is split into $T_1$ and $T_2$
  - $\Sigma$ is split into $S_1$, $S_2$, $S_3$, and $S_4$
  - $S_1$ ($S_4$) is computed recursively by the cores in $T_1$ ($T_2$)
  - If $S_3$ is below main diagonal it is not computed and $S_2$ is computed by all threads in $T_0$
  - Else ( $S_3$ is above main diagonal) $S_2$ ($S_3$) is computed by cores in $T_2$ ($T_1$)
- Example on right shows a mapping of tasks (sets of interactions) to 16 cores
Recursive Mapping of Tasks to Cores

- Set of available cores $T_0$ and $\Sigma$ are both split recursively
  - $T_0$ is split into $T_1$ and $T_2$
  - $\Sigma$ is split into $S_1$, $S_2$, $S_3$, and $S_4$
  - $S_1$ ($S_4$) is computed recursively by the cores in $T_1$ ($T_2$)
  - If $S_3$ is below main diagonal it is not computed and $S_2$ is computed by all threads in $T_0$
  - Else ( $S_3$ is above main diagonal) $S_2$ ($S_3$) is computed by cores in $T_2$ ($T_1$)
- Example on right shows a mapping of tasks (sets of interactions) to 16 cores
Recursive Mapping of Tasks to Cores

- Set of available cores $T_0$ and $\Sigma$ are both split recursively
  - $T_0$ is split into $T_1$ and $T_2$
  - $\Sigma$ is split into $S_1$, $S_2$, $S_3$, and $S_4$
  - $S_1$ ($S_4$) is computed recursively by the cores in $T_1$ ($T_2$)
  - If $S_3$ is below main diagonal it is not computed and $S_2$ is computed by all threads in $T_0$
  - Else ($S_3$ is above main diagonal) $S_2$ ($S_3$) is computed by cores in $T_2$ ($T_1$)

- Example on right shows a mapping of tasks (sets of interactions) to 16 cores
Recursive Mapping of Tasks to Cores

- Set of available cores $T_0$ and $\Sigma$ are both split recursively
  - $T_0$ is split into $T_1$ and $T_2$
  - $\Sigma$ is split into $S_1$, $S_2$, $S_3$, and $S_4$
  - $S_1$ ($S_4$) is computed recursively by the cores
    in $T_1$ ($T_2$)
  - If $S_3$ is below main diagonal it is not computed
    and $S_2$ is computed by all threads in $T_0$
  - Else ($S_3$ is above main diagonal) $S_2$ ($S_3$) is computed
    by cores in $T_2$ ($T_1$)

- Example on right shows a mapping of tasks
  (sets of interactions) to 16 cores
Recursive Mapping of Tasks to Cores

- Set of available cores $T_0$ and $\Sigma$ are both split recursively
  - $T_0$ is split into $T_1$ and $T_2$
  - $\Sigma$ is split into $S_1$, $S_2$, $S_3$, and $S_4$
  - $S_1$ ($S_4$) is computed recursively by the cores in $T_1$ ($T_2$)
  - If $S_3$ is below main diagonal it is not computed and $S_2$ is computed by all threads in $T_0$
  - Else ( $S_3$ is above main diagonal) $S_2$ ($S_3$) is computed by cores in $T_2$ ($T_1$)

- Example on right shows a mapping of tasks (sets of interactions) to 16 cores
\( N = 100000 \)

- Highest speedup 11.1 (Egypt), 12.6 (Barcelona) and 7.6 (Clovertown)
Overview over Interaction List Compression Techniques

Four considered techniques:

1. **Redundant Computations**
   - Ignores interaction lists when computing a set $S_{uv}$

2. **Hybrid Technique**
   - Uses interaction lists for some tasks $S_{uv}$
   - The first technique is used on the others

3. **Refinement Technique**
   - Orders particle data within a subvolume along a space filling curve
   - Uses particle position along the space filling curve to filter out interactions from the interaction list
   - Restores the interactions, when computing the task $S_{uv}$

4. **Variable Coding of interactions**
   - Compresses the interaction partners of each particle in a subvolume with a variable byte code
Redundant Computations Technique

- Parallel algorithm remains unmodified only computation of interactions of a set $S_{uv}$ is modified:

Original computation:

```
foreach \((i, j) \in S_{uv}\) interact\(((i, j))\)
```

Redundant Computation Technique:

```
foreach \(i \in B_u\)
    foreach \(j \in B_v\) interact\(((i, j))\)
```

- Advantage:
  - No interaction list is required $\rightarrow$ reduced storage
  - The information which particle is in which box, which has to be loaded $\rightarrow$ can be cached

- Disadvantage:
  - More interactions considered $\rightarrow$ requires filtering (cutoff radius)
     $\rightarrow$ redundant computations
Hybrid Technique

- Uses Interaction List Technique for sparse tasks and Redundant Computations Technique for dense tasks.
- Uses ratio $\epsilon$ of number of interactions in $S_{uv}$ divided by number of maximum possible interactions $N_{uv}$ between sets $B_u$ and $B_v$:

$$\epsilon = \frac{|S_{uv}|}{N_{uv}}$$

$$N_{uv} = \begin{cases} 
\frac{|B_u| \cdot (|B_u| - 1)}{2}, & \text{if } u = v \\
|B_u| \cdot |B_v|, & \text{otherwise}
\end{cases}$$

- Choose $\epsilon_0$, $0 \leq \epsilon_0 \leq 1$ constant for computation of $\Sigma$.
- For each task $S_{uv}$ calculate $\epsilon$ and decide:
  - If $\epsilon > \epsilon_0$ sparse task $\rightarrow$ use Interaction List
  - Else dense task $\rightarrow$ use Redundant Computations Technique
Decomposition into $16 \times 16 \times 16$ subvolumes

$N = 100000$ particles

- $\epsilon_0 = 1 \rightarrow$ all tasks $S_{uv}$ use interaction list $\rightarrow$ runtime of original implementation
- $\epsilon_0 = 0 \rightarrow$ all tasks $S_{uv}$ do not use interaction list
- At $\epsilon_0 = 0$ runtime increase through redundant computations
Refinement Technique

- Sets $S_{uv}$, with $u = v$, have highest density
  $\rightarrow$ compression has highest benefit

- Approach:
  - Refine each subvolume $B_u$ into $\tilde{N}_1 \times \tilde{N}_2 \times \tilde{N}_3$ subvolumes
  - Sort particles within $B_u$ according to a space filling curve (Z-Morton)
    $\rightarrow$ near distant particles are stored in consecutive memory positions
  - In computation the interaction list: Do not store the interaction between particle $P_i$ and $P_j$, if the distance of the memory position indices between $P_i$ and $P_j$ is smaller than a constant $l_z$
  - Replace code for computing task $S_{uv}$:

```cpp
if u = v then
  for $g_1 = 0$ to $|B_u| - 1$ do
    for $g_2 = g_1 + 1$ to $\min(g_1 + l_z, |B_u| - 1)$ do
      interact(first($B_u$) + $g_1$, first($B_u$) + $g_2$)

Computed remaining interactions with interaction list
```
Variable Coding of Interactions

After constructing the interaction list:
For each particle $P_i$ from $B_{uv}$ and for each set $S_{uv}$
- Sort interaction partners of $P_i$
- Calculate difference sequence of the sorted sequence
- Encode the sequence with a variable byte code of 1 to 5 Bytes

In the parallel calculation:
- Decode the variable byte code sequence and immediately use the elements for force calculation
## Runtime Comparision of the Techniques

<table>
<thead>
<tr>
<th>Description</th>
<th>Egypt</th>
<th>Barcelona</th>
<th>Clover-town</th>
</tr>
</thead>
<tbody>
<tr>
<td>Orig. Implementation</td>
<td>0.096</td>
<td>0.084</td>
<td>0.127</td>
</tr>
<tr>
<td>Redundant Computations</td>
<td>0.477</td>
<td>0.443</td>
<td>0.810</td>
</tr>
<tr>
<td>Hybrid</td>
<td>0.078</td>
<td>0.074</td>
<td>0.127</td>
</tr>
<tr>
<td>Subrefinement</td>
<td>0.081</td>
<td>0.079</td>
<td>0.130</td>
</tr>
<tr>
<td>Variable Coding</td>
<td>0.096</td>
<td>0.086</td>
<td>0.134</td>
</tr>
</tbody>
</table>

- 19% smaller runtime on Barcelona and 12% smaller runtime on system Egypt for Hybrid Technique compared to original implementation.
- 16% smaller runtime on Barcelona and 6% smaller runtime on Egypt for Subrefinement Technique compared to original implementation.
Compression versus Runtime

Compression vs. Runtime Clovertown

- Redundant
- Hybrid
- Refinement
- Variable Coding

Runtime [s] vs. Compression in Percent
Conclusion

- Compared four different techniques for reducing the size of the interaction list in a parallel algorithm.
- Interaction list compression techniques can reduce the storage requirements for storing the interaction lists.
- Parallel algorithm with compression techniques can reach the same runtime or even decrease the runtime of the parallel computation.
Thanks for your attention!

Questions?
## Multicore Systems Used for the Experiments

<table>
<thead>
<tr>
<th>Code Name</th>
<th><strong>Egypt</strong></th>
<th><strong>Barcelona</strong></th>
<th><strong>Clovertown</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>Processor</td>
<td>AMD Opteron 870</td>
<td>AMD Opteron 8347</td>
<td>Intel XEON E5345</td>
</tr>
<tr>
<td>Frequency</td>
<td>2.0 GHz</td>
<td>1.9 GHz</td>
<td>2.33 GHz</td>
</tr>
<tr>
<td>GFlop/s per Core</td>
<td>4</td>
<td>7.6</td>
<td>9.32</td>
</tr>
<tr>
<td>Total GFlop/s</td>
<td>64</td>
<td>121.6</td>
<td>74.56</td>
</tr>
<tr>
<td>Number of CPUs</td>
<td>8</td>
<td>4</td>
<td>2</td>
</tr>
<tr>
<td>Cores per CPU</td>
<td>2</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>Total Cores</td>
<td>16</td>
<td>16</td>
<td>8</td>
</tr>
<tr>
<td>Memory Architecture</td>
<td>Numa</td>
<td>Numa</td>
<td>Bus</td>
</tr>
<tr>
<td>L1-Cache</td>
<td>64KByte</td>
<td>64KByte</td>
<td>32KByte</td>
</tr>
<tr>
<td>L2-Cache</td>
<td>1MByte</td>
<td>512KByte</td>
<td>4MByte per 2 Cores</td>
</tr>
<tr>
<td>L3-Cache</td>
<td></td>
<td>2MByte per 4 Cores</td>
<td></td>
</tr>
</tbody>
</table>

16.2.2012 • Michael Schwind
Runtime Parallel Algorithm

![Graph showing runtime vs. number of cores for different platforms: Egypt, Barcelona, and Clovertown.](image)
L1-Misses of the Hybrid Technique depending on $\epsilon_0$

![Graph showing L1-Misses vs $\epsilon_0$ for Egypt, Barcelona, and Clovertown.](image)
L2-Misses of the Hybrid Technique depending on $\epsilon_0$

![Graph showing L2-Misses vs epsilon0 for different locations: Egypt, Barcelona, Clovertown. The graph illustrates the increase in L2-Misses as epsilon0 increases.]
TLB-Misses of the Hybrid Technique depending on $\epsilon_0$

![Graph showing TLB-Misses for different $\epsilon_0$ values for Egypt, Barcelona, and Clovertown.](image)

- **TLB-Misses vs. $\epsilon_0$**
- **Axes:** TLB-Misses on the y-axis and $\epsilon_0$ on the x-axis.
- **Lines:**
  - Red solid line for Egypt.
  - Green dashed line for Barcelona.
  - Blue dotted line for Clovertown.

**Legend:**
- Egypt (Red) *
- Barcelona (Green) X
- Clovertown (Blue) *

**Graph Notes:**
- The graph shows an increasing trend in TLB-Misses as $\epsilon_0$ increases for all systems.

**References:**
- Michael Schwind (16.2.2012)