A MASSIVELY PARALLEL DISTRIBUTED N-BODY APPLICATION IMPLEMENTED WITH HPX

Zahra Khatami
Hartmut Kaiser
Patricia Grubel
Adrian Serio
J. Ramanujam
CONTENTS

- Introduction
- HPX
- N-Body
- Experimental Results
- Conclusion
Computer scientists and programmers face the difficulty of improving the scalability of their applications while using conventional programming techniques only.

- N-Body:
  - communication intensive,
  - large message latencies,
  - large overheads

degrade parallel performance
WHAT DO WE NEED?

reducing:

a. poor utilization of resources caused by lack of available work (Starvation),
b. the time-distance delay of accessing remote resources (Latencies),
c. the cost for managing parallel actions (Overhead)
d. the cost imposed by oversubscription of shared resources (Waiting)
HPX

- HPX is a parallel C++ runtime system that facilitates distributed operations and enables fine-grained task parallelism. Using fine-grained tasks results in better load balancing, lower communication overheads, and better system utilization:
  - future concept
  - dataflow object
FUTURE

- A future is a computational result that is initially unknown but becomes available at a later time. The goal of using future is to let every computation proceed as far as possible.
- Other threads do not stop their progress even if the thread, which waits for the value to compute, is suspended. Threads access the future value by performing a `future::get()` . When the result becomes available, the future resumes all HPX suspended threads waiting for the value.
DATAFLOW

\[ \text{fnc1} \]
\[ \text{fnc2} \]
\[ \ldots \]
\[ \text{fncn} \]

\[ \text{in}_1 \rightarrow F(\text{in}_1, \text{in}_2 \ldots, \text{in}_n) \rightarrow \text{out} \]

\[ \text{in}_1 \rightarrow F(\text{in}_1, \text{in}_2 \ldots, \text{in}_n) \rightarrow \text{out} \]

\[ \text{in}_1 \rightarrow F(\text{in}_1, \text{in}_2 \ldots, \text{in}_n) \rightarrow \text{out} \]
ADVANTAGES

1. removing unnecessary barrier synchronization
2. interleaving different loops
3. automatically creating the dependency tree
4. dynamically determining the parameters during running time
1. Octree construction
2. Interaction list creation
3. Force computation
Each cube is subdivided into eight equally sized subcubes if it has more than \( N \) particles and in each step, each particle is reassigned to one of the newly created subcubes.

The gravitational potential of a distant group of particles is approximated as the potential of a single particle located in the center of mass of all particles in the cube.

The interaction list of the particle holds the list of the particles that are near objects to that particle and the cubes that are modeled as faraway objects to that particle.

Improved based on FMM
ALGORITHM IMPLEMENTED WITH HPX

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>hpx::id_type</td>
<td>A global address (GID)</td>
</tr>
<tr>
<td>hpx::find_here</td>
<td>Retrieving this’ locality’s id</td>
</tr>
<tr>
<td>hpx::find_all_localities</td>
<td>Finding all localities</td>
</tr>
<tr>
<td>hpx::register_with_basename</td>
<td>Registering a unique GID</td>
</tr>
</tbody>
</table>
ALGORITHM IMPLEMENTED WITH HPX

- future-based request buffer is used between different nodes and along each spatial direction to send/receive data to/from the remote nodes.
- data distribution across the nodes is designed based on assigning a unique id provided by HPX.
- the algorithm minimizes idle nodes and maintains high utilization.

Initialization Steps:
1) Constructing an Octree
2) Assigning id to each cube
3) Finding each locality’s id with hpx::find-locality()
4) Finding remote localities’ id with hpx::find-all-localities()
5) Locating each cube and all its descendants with Eq.5
6) Discovering the neighbors of each locality
7) Creating shared buffers between two neighbor localities

For each time step Do:
8) Computing the force on each particle with Eq.2
9) Computing the new position of each particle with Eq.4
10) Storing $p_i$ needed to be sent, in the:
    - left buffer, if $x_{p_i} < x_{parent}$
    - right buffer, if $x_{p_i} > x_{parent}$
    - above buffer, if $y_{p_i} > y_{parent}$
    - down buffer, if $y_{p_i} < y_{parent}$
    - back buffer, if $z_{p_i} > z_{parent}$
    - front buffer, if $z_{p_i} < z_{parent}$
11) Receiving $p_i$ that doesn’t belong to $l_i$, from left, then storing it in the right buffer right, then storing it in the left buffer above, then storing it in the down buffer down, then storing it in the above buffer back, then storing it in the front buffer front, then storing it in the back buffer
12) Sending data in buffer $i$ in the direction $i$,
    $i \in \{\text{left, right, up, down, back, front}\}$

Loop
EXPERIMENTAL RESULTS

- Graph 1: Speedup vs. Number of Nodes for different data sets.
  - 1,000,000,000
  - 100,000,000
  - 10,000,000
  - 1,000,000
  - 100,000

- Graph 2: Speedup vs. Threads for different parallel strategies.
  - hpx::async
  - hpx::parallel::for_each
  - ideal
  - OpenMP
EXPERIMENTAL RESULTS

![Graph 1: Efficiency vs. Number of Nodes](image1.png)

![Graph 2: Time vs. Processors](image2.png)
EXPERIMENTAL RESULTS
HPX is able to control a grain size at runtime by using for_each and as a result the resource starvation is reduced as well.

HPX helps keeping the better load balancing and lower overheads compared to MPI and OpenMP.

The better performance by HPX for the larger number of the nodes, which is due to using a future -based request buffer between the remote nodes that allows the continuation of the process without waiting for the previous step to be completed.

128x speedup on 128 distributed nodes for 1 billion particles using HPX that indicates it has the potential to continue to scale on more even cores.
Thank you for your attention