A Promising Approach to Dynamic Load Balancing of Weather Forecast Models

Jairo Panetta
Eduardo Rocha Rodrigues
Philippe O. A. Navaux
Celso L. Mendes
Laxmikant V. Kale
CAS, Annecy, Sep 2011
BRAMS: Full Microphysics
(8 categories of water)

(A) Chuva (mm, 24h)
BRAMS 1 km

<table>
<thead>
<tr>
<th>City</th>
<th>Measured</th>
<th>Forecasted</th>
</tr>
</thead>
<tbody>
<tr>
<td>Nova Friburgo (NF)</td>
<td>162</td>
<td>158</td>
</tr>
<tr>
<td>Teresopolis (TE)</td>
<td>78</td>
<td>88</td>
</tr>
<tr>
<td>Petropolis (PE)</td>
<td>7</td>
<td>4</td>
</tr>
</tbody>
</table>
CATT-BRAMS

Air Pollution due to biomass burning and urban areas
The Problem

Dynamic Load Imbalance Limits BRAMS Scalability
Desired Solution

Automatic load balance with minimum (zero?) code intrusion
Research Strategy

- Over-decompose the domain
  - More MPI ranks than real processors
  - Processor virtualization

- Move MPI ranks across real processors to balance the load
  - Use AMPI, an MPI library build on top of Charm++ (charm.cs.uiuc.edu)

- Explore:
  - Virtualization costs and benefits
  - Load Balancing Algorithms
  - Triggering factor to balance the load
Virtualization Costs and Benefits
## Processor Virtualization

<table>
<thead>
<tr>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

- **4 MPI ranks**
- **4 real processors**

<table>
<thead>
<tr>
<th>12</th>
<th>13</th>
<th>14</th>
<th>15</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>9</td>
<td>10</td>
<td>11</td>
</tr>
<tr>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
</tbody>
</table>

- **16 MPI ranks**
- **4 real processors**

[www.cptec.inpe.br](http://www.cptec.inpe.br)
Why AMPI?

- AMPI implements MPI ranks as user-level threads
  - Fast context switch

- AMPI build-in scheduler keeps a thread executing until it is blocked (e.g., waiting for communication)
  - Overlaps communication and computation

- AMPI ranks are “migratable” and AMPI has a built-in set of thread migration algorithms
  - Ready to be tested

- But not all codes can use AMPI:
  - Source code cannot have static or global variables
  - Fortran: no MODULE public variables, no COMMONS, no SAVEEd variables, etc...

www.cptec.inpe.br
Source Code Changes

- Only one line of BRAMS source code was modified (introduced), to invoke the load balancer:

  \[
  \text{if \ (<\text{triggering factor}>\text{>) call MPI\_Migrate()}\]

- But, to use AMPI, BRAMS original static and global variables were privatized
  - By changing gfortran and runtime libraries to generate (and run) code that supports Thread Local Storage at user-level threads
Results: Virtualization

- BRAMS at 10M Grid Cells, 1.6 km resolution, 4 hours of simulated time, 6 seconds time-step
- 64 real processors @ Kraken

<table>
<thead>
<tr>
<th>Virtual Processors</th>
<th>Execution Time (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1x64</td>
<td>4970</td>
</tr>
<tr>
<td>4x64</td>
<td>3857</td>
</tr>
<tr>
<td>16x64</td>
<td>3713</td>
</tr>
<tr>
<td>32x64</td>
<td>4437</td>
</tr>
</tbody>
</table>

www.cptec.inpe.br
Why?

- Benefits of virtualization through user-level threads:
  - Improves cpu utilization by overlapping computation with communication
    - keeps executing a rank until it waits for a message and then switches to another ready-to-run, non-waiting rank
    - On non-virtualized runs, the single rank keeps waiting on communication
  - Reduces cache misses by using smaller domain per rank and keeping the rank longer in control of a CPU
Load Balancing Algorithm
Load Balancing Algorithm

- Fix a triggering policy to invoke the load balancer and experiment with a set of load balancing algorithms
- Tentative triggering policy: once every forecast hour
- Pre-existing load balancing algorithms at AMPI:
  - Greedy (load only)
  - RefineCommunication (load and communication)
  - RecursiveBissection (load and communication)
  - Metis (load and communication)
## Results: LB Algorithm

- Same input data and forecast period
- 64 real processors, 16x64 virtual processors

<table>
<thead>
<tr>
<th>LB Algorithm</th>
<th>Execution Time (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>No over-decomposition</td>
<td>4987</td>
</tr>
<tr>
<td><strong>OvrDec, no LB</strong></td>
<td><strong>3713</strong></td>
</tr>
<tr>
<td>OvrDec + Greedy LB</td>
<td>3768</td>
</tr>
<tr>
<td>OvrDec + RefineComm LB</td>
<td>3714</td>
</tr>
<tr>
<td>OvrDec + RecBissection LB</td>
<td>4527</td>
</tr>
<tr>
<td><strong>OvrDec + Metis LB</strong></td>
<td><strong>3393</strong></td>
</tr>
</tbody>
</table>
An LB algorithm should achieve:
- Good Load Balance
- Low Communication Cost
- Low Execution Cost

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Balance</th>
<th>Comm.</th>
<th>Execution</th>
</tr>
</thead>
<tbody>
<tr>
<td>Greedy</td>
<td>Good</td>
<td>High</td>
<td>High</td>
</tr>
<tr>
<td>RefineComm</td>
<td>Poor</td>
<td>Low</td>
<td>Low</td>
</tr>
<tr>
<td>RefineBissec</td>
<td>Good</td>
<td>High</td>
<td>High</td>
</tr>
<tr>
<td>Metis</td>
<td>Good</td>
<td>Low</td>
<td>High</td>
</tr>
</tbody>
</table>
A New Load Balancing Algorithm Based Upon Hilbert Space Filling Curve

- Take a 1D enumeration of a 2D finite and discrete space
  - A line passing through each 2D space point just once
  - e.g., MPI ranks over the horizontal simulation domain

- Cut the line in segments that are similar in some metric
  - e.g., load in real processors

- Hilbert Space-filling Curve keeps together 2D nearest neighbours in the 1D enumeration
  - As so, it may reduce communication cost
  - Faster algorithm than Metis
Hilbert Curve
Expanding on Low Load
Shrinking on High Load
### Results: LB Algorithm

<table>
<thead>
<tr>
<th>LB Algorithm</th>
<th>Execution Time (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>No OvrDec</td>
<td>4987</td>
</tr>
<tr>
<td>OvrDec, no LB</td>
<td>3713</td>
</tr>
<tr>
<td>OvrDec + Greedy</td>
<td>3768</td>
</tr>
<tr>
<td>OvrDec + RefineComm.</td>
<td>3714</td>
</tr>
<tr>
<td>OvrDec + RecBissect</td>
<td>4527</td>
</tr>
<tr>
<td>OvrDec + Metis</td>
<td>3393</td>
</tr>
<tr>
<td>OvrDec + Hilbert</td>
<td>3366</td>
</tr>
</tbody>
</table>
Initial Mapping of MPI ranks to 64 processors (8x8)
Mapping after rebalance at one hour
Load imbalance on the first hour prior to rebalance
Load imbalance on the first hour after rebalance
Load imbalance on second hour prior to rebalance
Load imbalance on second hour after rebalance
Load Balancing Can Be Beneficial

But Past Load Is Not a Precise Estimate of Future Load
Triggering Policy
Triggering Policy

- So far, rebalance the load at every simulated hour (600 timesteps)
  - Regardless of the current load imbalance

- But:
  - Measuring load imbalance is cheap
  - Migrating tasks is expensive

- Triggering policy:
  - Measure load imbalance frequently
  - Rebalance only when load imbalance is higher than a fixed threshold
Results: Triggering Policy

- Same forecast
- 64 real processors, 16x64 virtual processors

<table>
<thead>
<tr>
<th>LB Interval (Timesteps)</th>
<th>Imbalance Threshold</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>50%</td>
</tr>
<tr>
<td>600</td>
<td>-</td>
</tr>
<tr>
<td>100</td>
<td>3639.54</td>
</tr>
<tr>
<td>10</td>
<td>3554.07</td>
</tr>
<tr>
<td>1</td>
<td>-</td>
</tr>
</tbody>
</table>

Gain by changing triggering policy

- Automatic selection of the imbalance threshold reduced execution time even further
Conclusions

- There is evidence that processor virtualization and dynamic load balancing are beneficial.
  - On our test case, execution time reduction from 4987s to 3713s (by virtualization) and to 3128s (by load balancing).

- Negligible source code changes.
  - Provided no static or global variables.

- Hopefully, persistent over new parameterizations.

- Further test cases are required...
Bibliography

THANK YOU