Adaptive MPI: Overview & Recent Work

Sam White
PPL, UIUC
Motivation

• Main challenge for applications: variability
  – Hardware variation
    • Static/dynamic, heterogeneity, failures, power, etc.
  – Dynamic program behavior
    • AMR, particle movements, subscale simulations, ...

• To deal with this:
  – Rewrite applications in new languages ...
  – Or, implement existing APIs on different runtime systems
Adaptive MPI

- MPI-2.2 implementation on top of Charm++
  - MPI ranks are lightweight, migratable user-level threads associated with Charm++ objects
Adaptive MPI

• Q: What can Charm++ and its runtime system offer MPI programmers?

• A: Application-independent features for MPI codes:
  – Process virtualization
  – Automatic overlap of comm. & comp.
  – Static and dynamic mapping
  – Automatic fault tolerance
  – OpenMP runtime integration
Overdecomposition

• MPI programmers already decompose to MPI ranks:
  – One rank per node/core/…

• AMPI virtualizes MPI ranks, allowing multiple ranks to execute per node/core/…
  – Benefits: cache usage, comm. overlap, etc.
  – Issue: multiple ranks in same OS process now share all their global/static variables
Overdecomposition

• MPI programmers already decompose to MPI ranks:
  – One rank per node/core/…

• AMPI virtualizes MPI ranks, allowing multiple ranks to execute per node/core/…
  – Benefits: cache usage, comm. overlap, etc.
  – Issue: multiple ranks in same OS process now share all their global/static variables
    • AMPI programs are MPI programs without mutable global/static variables
    • Compiler support for automating this privatization
Asynchrony

• With multiple MPI ranks per core, how do we schedule them?

• Message-driven execution:
  – Let the work-unit that happens to have data (a matching message) available for it execute next
  – Let the RTS select among ready work units
Message-driven Execution

MPI_Send()
Migratability

- AMPI ranks are migratable at runtime
  - Thread stack + heap
Migratibility

- AMPI ranks are migratable at runtime
  - Thread stack + heap

- Isomalloc makes migration automatic
  - No application Pack–UnPack (PUP) code needed
  - Productive, easy to experiment with

- PUP routines are only an optimization
  - Portability: no need for 64-bit VM
  - Performance: only migrate the data that will be needed after migration
Dynamic Load Balancing

• AMPI ranks can be dynamically load balanced between nodes/cores
  – Based on measured idle time, or user-level information
  – Suite of built-in Charm++ strategies available
  – Application developers can write their own strategies too

• User code needs to call AMPI_Migrate() and choose balancer at runtime:
  – srun -n 100 ./pgm +vp 1000 +balancer RefineLB
Fault Tolerance

• Basic ideas:
  – Checkpoints are just migrations to storage
  – Underlying storage can be various things
  – Can be used in concert with load balancing

• Four approaches available:
  – Disk-based checkpoint/restart
  – In-memory double checkpoint w/ auto restart
  – Proactive object migration
  – Message-logging
PlasComCM

• The Center for Exascale Simulation of Plasma–Coupled Combustion (XPACC)
  – PSAAPII center at UIUC
  – Collaboration of experimentalists, computational scientists, and computer scientists

• Main simulation code: PlasComCM
  – 150K lines of Fortran90/MPI: runs on AMPI
    • Benefits from overdecomposition
    • Fault tolerance demonstrated
    • Dynamic load imbalance coming in future
**PlasComCM Strong Scaling**

- Virtualization benefits ($V =$ ranks/core)
Fault Tolerance

PlasComCM: iteration = 96, dt = 0.870094D-02, time = 0.835290D+00, cfl = 0.500000D+00, maxT = 0.2980000D+03
PlasComCM: iteration = 97, dt = 0.870094D-02, time = 0.843991D+00, cfl = 0.500000D+00, maxT = 0.2980000D+03
PlasComCM: iteration = 98, dt = 0.870094D-02, time = 0.852692D+00, cfl = 0.500000D+00, maxT = 0.2980000D+03
PlasComCM: iteration = 99, dt = 0.870094D-02, time = 0.861393D+00, cfl = 0.500000D+00, maxT = 0.2980000D+03
PlasComCM: iteration = 100, dt = 0.870094D-02, time = 0.870094D+00, cfl = 0.500000D+00, maxT = 0.2980000D+03
[0] Checkpoint started
[0] Checkpoint finished in 0.455819 seconds
PlasComCM: iteration = 101, dt = 0.870094D-02, time = 0.878795D+00, cfl = 0.500000D+00, maxT = 0.2980000D+03
PlasComCM: iteration = 102, dt = 0.870094D-02, time = 0.887496D+00, cfl = 0.500000D+00, maxT = 0.2980000D+03
PlasComCM: iteration = 103, dt = 0.870094D-02, time = 0.896197D+00, cfl = 0.500000D+00, maxT = 0.2980000D+03
Fault Tolerance

1. Checkpoint

2. Failure
Fault Tolerance

1. Checkpoint

2. Failure

3. Recover

PlasComCM: iteration = 96, dt = 0.870094D-02, time = 0.835290D+00, cfl = 0.500000D+00, maxT = 0.298000D+03
PlasComCM: iteration = 97, dt = 0.870094D-02, time = 0.843991D+00, cfl = 0.500000D+00, maxT = 0.298000D+03
PlasComCM: iteration = 98, dt = 0.870094D-02, time = 0.852692D+00, cfl = 0.500000D+00, maxT = 0.298000D+03
PlasComCM: iteration = 99, dt = 0.870094D-02, time = 0.861393D+00, cfl = 0.500000D+00, maxT = 0.298000D+03
PlasComCM: iteration = 100, dt = 0.870094D-02, time = 0.870094D+00, cfl = 0.500000D+00, maxT = 0.298000D+03

[0] Checkpoint started
[0] Checkpoint finished in 0.455819 seconds
PlasComCM: iteration = 101, dt = 0.870094D-02, time = 0.878795D+00, cfl = 0.500000D+00, maxT = 0.298000D+03
PlasComCM: iteration = 102, dt = 0.870094D-02, time = 0.887496D+00, cfl = 0.500000D+00, maxT = 0.298000D+03
PlasComCM: iteration = 103, dt = 0.870094D-02, time = 0.896197D+00, cfl = 0.500000D+00, maxT = 0.298000D+03

Socket closed before recv.
Socket 4 failed

Charmrun finished launching new process in 1.153346 seconds
Charmrun says Processor 1 failed on Node 1
1. Restarting after crash
1. Restart finished in 0.458689 seconds at 0.463579.
Fault Tolerance

1. Checkpoint
2. Failure
3. Recover
4. Resume execution

PlasComCM: iteration = 96, dt = 0.870094D-02, time = 0.835290D+00, cfl = 0.500000D+00, maxT = 0.298000D+03
PlasComCM: iteration = 97, dt = 0.870094D-02, time = 0.843991D+00, cfl = 0.500000D+00, maxT = 0.298000D+03
PlasComCM: iteration = 98, dt = 0.870094D-02, time = 0.852692D+00, cfl = 0.500000D+00, maxT = 0.298000D+03
PlasComCM: iteration = 99, dt = 0.870094D-02, time = 0.861393D+00, cfl = 0.500000D+00, maxT = 0.298000D+03
PlasComCM: iteration = 100, dt = 0.870094D-02, time = 0.870094D+00, cfl = 0.500000D+00, maxT = 0.298000D+03
[0] Checkpoint started
[0] Checkpoint finished in 0.455819 seconds
PlasComCM: iteration = 101, dt = 0.870094D-02, time = 0.878795D+00, cfl = 0.500000D+00, maxT = 0.298000D+03
PlasComCM: iteration = 102, dt = 0.870094D-02, time = 0.887496D+00, cfl = 0.500000D+00, maxT = 0.298000D+03
PlasComCM: iteration = 103, dt = 0.870094D-02, time = 0.896197D+00, cfl = 0.500000D+00, maxT = 0.298000D+03

Socket closed before recv.
Socket 4 failed

Charmrun finished launching new process in 1.153346 seconds
Charmrun says Processor 1 failed on Node 1
[1] Restarting after crash
[1] Restart finished in 0.458689 seconds at 0.463579.
PlasComCM: iteration = 101, dt = 0.870094D-02, time = 0.878795D+00, cfl = 0.500000D+00, maxT = 0.298000D+03
Fault Tolerance

1. Checkpoint

2. Failure

3. Recover

4. Resume execution

5. Load balance
Fault Tolerance

- Double in-memory checkpoint is scalable

- Minimal changes needed to PlasComCM
Kripke

- LLNL ASC proxy app for deterministic particle transport codes
  - Solves the Boltzmann transport equation using parallel sweeps over a 3D domain space

- Given:
  - 3D domain of known materials
  - Initial flow of particles through domain
  - Particle-generating sources inside the domain
  - Boundary conditions

- Solution:
  - Particle flux at every point inside the domain at a later time
Kripke

- Key communication pattern: parallel sweep
Kripke

- Key communication pattern: parallel sweep
Kripke

- Key communication pattern: parallel sweep
Mapping

• Blocked mapping of subdomains to ranks is efficient within-node
Mapping

- Scattered mapping increases concurrency
  - 5–10% improvement at scale
OpenMP Integration

- Charm++ version of GNU OpenMP 4.0 works with AMPI
  - (A)MPI+OpenMP configurations on P cores/node:

<table>
<thead>
<tr>
<th>Notation</th>
<th>Ranks/Node</th>
<th>Threads/Rank</th>
<th>MPI(+OpenMP)</th>
<th>AMPI(+OpenMP)</th>
</tr>
</thead>
<tbody>
<tr>
<td>P:1</td>
<td>P</td>
<td>1</td>
<td>✔️</td>
<td>✔️</td>
</tr>
<tr>
<td>1:P</td>
<td>1</td>
<td>P</td>
<td>✔️</td>
<td>✔️</td>
</tr>
<tr>
<td>P:P</td>
<td>P</td>
<td>P</td>
<td></td>
<td>✔️</td>
</tr>
</tbody>
</table>

- AMPI+OpenMP can do P:P without oversubscription of system resources
Kripke Weak Scaling

- (A)MPI-only suffers from transient load imbalance during the sweep
OpenMP Interoperation

- (A)MPI+OpenMP (1:P) loses out on the sweep’s pipeline parallelism
OpenMP Integration

- Kripke benefits from AMPI+OpenMP (P:P)
  - Pipeline parallelism + within-node load balancing
Recent Progress

• Charm++ 6.7.1 is a feature release for AMPI
  – AMPI extensions now prefixed with ‘AMPI_’
  – MPI-2.2 compliance
  – MPI-3.1 nonblocking & neighborhood collectives
  – Improved performance for test, wait routines
  – ampicc is more compatible with autoconf/cmake

• Ongoing work:
  – Conformance to MPI-3.1
  – True RDMA for MPI’s RMA routines
  – Optimization of AMPI+OpenMP integration
Summary

• Adaptive MPI provides Charm++’s high-level features to pre-existing MPI applications
  – Overdecomposition
  – Overlap of communication and computation
  – Configurable static mapping
  – Dynamic load balancing
  – Automatic fault tolerance
  – OpenMP runtime integration
Thank you