Adaptive Runtimes: Charm++ case study and Lessons for Exascale

Laxmikant (Sanjay) Kale
http://charm.cs.illinois.edu
Parallel Programming Laboratory
Department of Computer Science
University of Illinois at Urbana Champaign
Exascale Challenges

- Main challenge: variability
  - Static/dynamic
  - Heterogeneity: processor types, process variation, ..
  - Power/Temperature/Energy
  - Component failure

- Exacerbated by strong scaling needs from apps
  - Why?

- To deal with these, we must seek
  - Not full automation
  - Not full burden on app-developers
  - But: a good division of labor between the system and app developers
A couple of forks

- MPI + x
- “Task Models”
  - Asynchrony
- Overdecomposition:
  - Most adaptivity
My Mantra

verdecompositio

synchrony

igratability
Overdecomposition

- Decompose the work units & data units into many more pieces than execution units
  - Cores/Nodes/..
- Not so hard: we do decomposition anyway
Migratability

• Allow these work and data units to be migratable at runtime
  – i.e. the programmer or runtime, can move them

• Consequences for the app–developer
  – Communication must now be addressed to logical units with global names, not to physical processors
  – But this is a good thing

• Consequences for RTS
  – Must keep track of where each unit is
  – Naming and location management
Asynchrony: Message-Driven Execution

• Now:
  – You have multiple units on each processor
  – They address each other via logical names

• Need for scheduling:
  – What sequence should the work units execute in?
  – One answer: let the programmer sequence them
    • Seen in current codes, e.g. some AMR frameworks
  – Message-driven execution:
    • Let the work-unit that happens to have data (“message”) available for it execute next
    • Let the RTS select among ready work units
    • Programmer should not specify what executes next, but can influence it via priorities
Message-driven Execution

Processor 1
Scheduler
Message Queue

A[..].foo(…)

Processor 2
Scheduler
Message Queue
Overview of features

• Objects, called chares:
  – Organized into multiple collections, each with its own indexing
  – Asynchronous method invocations

• User-level “run” threads embedded in chares

• Asynchronous (non-blocking) reductions

• “structured dagger”:
  – script-like notation to express dependencies among computations and messages within chares
Empowering the RTS

The Adaptive RTS can:

- Dynamically balance loads
- Optimize communication:
  - Spread over time, async collectives
- Automatic latency tolerance
- Prefetch data with almost perfect predictability
NAMD: Biomolecular Simulations

• Collaboration with K. Schulten
• With over 70,000 registered users
• Scaled to most top US supercomputers
• In production use on supercomputers and clusters and desktops
• Gordon Bell award in 2002

Recent success:
Determination of the structure of HIV capsid by researchers including Prof Schulten
Time Profile of ApoA1 on Power7 PERCS

92,000 atom system, on 500+ nodes (16k cores)

2ms total

A snapshot of optimization in progress. Not the final result

Overlapped steps, as a result of asynchrony
Timeline of ApoA1 on Power7 PERCS

230us
NAMD: Strong Scaling

- HIV Capsid was a 64 million atom simulation, including explicit water atoms
- Most biophysics systems of interests are 10M atoms or less… maybe 100M
- Strong scaling desired, to execute billions of steps

Exascale lesson: strong scaling will be required
Structured AMR miniApp
Structured AMR: State Machine

- Required depth: $d$
- Initial state: $d$
- Decision: $d$
- Received message
- Local error condition
- Termination detection

Graph:
- States: $d - 1$, $d$, $d + 1$
- Transitions:
  - $d - 1$ to $d$ by Coarsen
  - $d$ to $d + 1$ by Refine
  - $d$ to $d - 1$ by Sibling
  - $d$ to $d + 1$ by Coarsen, Stay
  - $d + 1$ to $d + 2$ by Refine

- Dotted lines indicate conditions: $d$, $d - 1$, $d + 1$, $d + 2$
Structured AMR: Performance

Testbed: IBM BG/Q Mira Cray XK/6 Titan

Advection Benchmark
First order method in 3d-space

![Graph showing performance of different load balancing strategies](image.png)
Production–Quality System with Many Apps

• Used on current PF class machines on many applications:
  – NAMD
  – Charisma
  – OpenAtom
  – Episimdemics
  – ...

• Correspondingly, production–quality system
  – Nightly builds with testing on dozens of platforms
  – Covers a variety of OS, hardware, compiler combinations
## MiniApps: [http://charmplusplus.org/benchmarks](http://charmplusplus.org/benchmarks)

<table>
<thead>
<tr>
<th>Mini-App</th>
<th>Features</th>
<th>Machine</th>
<th>Max cores</th>
</tr>
</thead>
<tbody>
<tr>
<td>AMR</td>
<td>Custom array index, Message priorities, Load Balancing, Checkpoint restart</td>
<td>BG/Q</td>
<td>131,072</td>
</tr>
<tr>
<td>LeanMD</td>
<td>Load Balancing, Checkpoint restart, Power awareness</td>
<td>BG/P BG/Q</td>
<td>131,072 32,768</td>
</tr>
<tr>
<td>Barnes–Hut (n-body)</td>
<td>Message priorities, Load Balancing</td>
<td>Blue Waters</td>
<td>16,384</td>
</tr>
<tr>
<td>LULESH 2.02</td>
<td>AMPI, Load Balancing</td>
<td>Hopper</td>
<td>8,000</td>
</tr>
<tr>
<td>PDES</td>
<td>Message priorities, TRAM</td>
<td>Stampede</td>
<td>4,096</td>
</tr>
<tr>
<td>1D FFT</td>
<td>Interoperable with MPI</td>
<td>BG/P BG/Q</td>
<td>65,536 16,384</td>
</tr>
<tr>
<td>Dense LU</td>
<td>SDAG</td>
<td>XT5</td>
<td>8,192</td>
</tr>
<tr>
<td>GTC</td>
<td>SDAG</td>
<td>BG/Q</td>
<td>1,024</td>
</tr>
</tbody>
</table>
Where are Exascale Issues?

- These techniques were needed for dynamic irregular apps even on yesterday’s machines
  - At exascale, they need to be applied to even regular apps
  - Charm++ meets exascale challenges already, almost
    - How we got so lucky: because of these irregular apps

The adaptivity that was created via overdecomposition, migratibility, & asynchrony, for dynamic applications, is also useful for handling machine variability at exascale
Agenda for Exascale RTS from Charm++ point of view
Costs of Overdecomposition?

- We examined the “Pro”s so far
- Cons and remedies:
  - Scheduling overhead?
    - Not much at all
    - In fact get benefits due to blocking
  - Memory in ghost layer increases
    - Fuse local regions with compiler support
    - Fetch one ghost layer at a time
    - Hybridize (pthreads/openMP inside objects/DEBs)
- Less control over scheduling?
  - i.e. too much asynchrony?
  - But can be controlled in various ways by an observant RTS/programmer
- For domain–decomposition based solvers, may increase number of iterations
  - You can lift it to node–level overdecomposition (use openMP inside)
  - Also, other ideas:
Exascale RTS components

**XARTS**

**WUDUs:** Indexed collection, Migratable threads, Scalable sections (sub-communicators), Location services

Data-driven **scheduler,** user-level threads, priority queues

Continuous Introspection Framework

**Fault tolerance protocols**

**Load balancers:** intra-node, inter-node
Power-aware, Thermal-aware, Topo-aware

**Communication Libs**
(Colletives/persistence)

**LRTS:** m/c specific implementations:
(start-up, communication, virtual mem. management)

**Scalable Tools**
Analysis, Debugging
Fault Tolerance in Charm++/AMPI

- Four approaches available:
  - Disk-based checkpoint/restart
  - In-local-storage double checkpoint w auto restart
    • Demoed on 64k cores
  - Proactive object migration
  - Message-logging: scalable fault tolerance
    • Can tolerate frequent faults
    • Parallel restart and potential for handling faults during recovery

- Common Features:
  - Easy checkpoint: migrate-to-disk
  - Based on dynamic runtime capabilities
  - Use of object-migration
  - Can be used in concert with load-balancing schemes
Extensions to fault recovery

• Based on the same over-decomposition ideas
  – Use NVRAM instead of DRAM for checkpoints
    • Non-blocking variants
    • [Cluster 2012] Xiang Ni et al.
  – Replica-based soft-and-hard-error handling
    • As a “gold-standard” to optimize against

• A **vision for future**:
  – But depends on whether faults are frequent
Power/Energy/Temperature/Time

• Usually: power and temperature are constraints
• Whole machine problem
• But needs application RTS to cooperate with global OS
Saving Cooling Energy

• Easy: increase A/C setting
  – But: some cores may get too hot
• So, reduce frequency if temperature is high (DVFS)
  – Independently for each chip
• But, this creates a load imbalance!
• No problem, we can handle that:
  – Migrate objects away from the slowed-down processors
  – Balance load using an existing strategy
  – Strategies take speed of processors into account
• Implemented in experimental version
  – SC 2011 paper, IEEE TC paper
• Several new power/energy–related strategies
  – http://charm.cs.illinois.edu/research/energy
App-facing and whole-machine RTS

• These are two distinct entities.. Have to be.
  – Jobs come and go
• Today, you can think of SLURM etc. as “whole machine RTS”
  – But that’s very limited
• We need a two-way dialogue between them,
  – And not just at job-start time
PARM: Power Aware Resource Manager

- Charm++ RTS facilitates malleable jobs
- PARM can improve throughput under a fixed power budget using:
  - overprovisioning (adding more nodes than conventional data center)
  - RAPL (capping power consumption of nodes)
  - Job malleability and moldability
Controlling Asynchrony

• Tighter control over scheduling is needed
• NAMD example:
  – Messages for the next iteration start arriving even before I have received all messages for the current iteration
  – The problem: It's not the scheduling (which is easier to solve) but interference..
Sub–computations support

• E.g. ability to fire subscale simulations
  – Overdecomposition creates a potential for handling this,
  – But new scheduling/load balancing techniques are needed
Interoperability

• Don’t take over “main”. At least offer a mode
• But there are more challenges to solve when interoperating between a task-model and an MPI-like model
• We need to evolve a consensus on:
  – Control transfer mechanisms
  – Data sharing and data transfer mechanisms
Collectives on sections

• A subset of objects spread over a subset of nodes, under dynamic redistribution.
  – Efficient collective algorithms in this setting
  – Adapting to whatever fraction of nodes is occupied by the section
  – May seem straightforward, but has some twists
Reconfiguring Apps and RTS

• We need a set of knobs in the application
  – Can be created by the compiler, the programming language, or the user
  – Each knob ("control point") specifies its "effects" in a formal registration

• We need a closed loop control system:
  – Periodically, collect introspected data,
  – Analyze it and decide the issue that needs fixing
  – Select one or more knobs that can fix it, and turn

• *PICS* in Charm++ is one attempt