Adaptive Runtime Systems meet Needs of Many Task Computing

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

- Some of the MTAGS community is moving towards a context where each task is itself a parallel job
  - These tasks interact in potentially complex work-flow arrangements
  - And they must run on cloud/grid environments
    - Virtualized OSs
    - Latencies
    - Performance Heterogeneity: static and dynamic
    - Resource availability may vary over time
    - Resource needs may vary over time
Outline

• How adaptive runtime systems within jobs can help make parallel jobs fit within grid/cloud environment

• ARTS and their place in HPC
• Charm++ model and successes

• Charm++ Features of relevance:
  – Task parallelism
  – Handling latency, and variation/heterogeneity
  – Multi-cluster jobs
  – Shrink/expand, faucets project, scheduler, bid
  – Interacting with parallel jobs
  – Support for replica’s: loosely communicating tightly-parallel jobs
  – Theme: Please experiment with it
Migratable Objects Execution Model

• Programmer
  – Decomposes computation into a large number of work/data units (WUDUs)
  – Grainsize independent of number of processors

• The runtime system
  – Assigns these units to processors,
  – Changes the assignment at runtime
  – Mediates communication between the units

• Message–driven execution model
  – Since there are multiple units on each PE

• Programmer’s mental model doesn’t have “processor” in it
Object Based Over-decomposition: Charm++

- Multiple “indexed collections” of C++ objects
- Indices can be multi-dimensional and/or sparse
- Programmer expresses communication between objects – with no reference to processors

System implementation

User View
Adaptive Runtime Systems

- Decomposing program into a large number of WUDUs empowers the RTS, which can:
  - Migrate WUDUs at will
  - Schedule DEBS at will
  - Instrument computation and communication at the level of these logical units
    - WUDU x communicates y bytes to WUDU z every iteration
    - SEB A has a high cache miss ratio
  - Maintain historical data to track changes in application behavior
    - Historical => previous iterations
    - E.g., to trigger load balancing
Over-decomposition and message-driven execution

Migratability

Introspective and adaptive runtime system

Scalable Tools

Automatic overlap, prefetch, compositionality

Emulation for Perf Prediction

Fault Tolerance

Dynamic load balancing (topology-aware, scalable)

Temperature/power considerations
Message-driven execution model

- Adaptive overlap of communication and computation
- A strong principle of prediction for data and code use
  - Much stronger than principle of locality
    - Can use to scale memory wall:
      - Prefetching needed data:
        - into scratch pad memories, for example
Impact on communication

• Current use of communication network:
  – Compute–communicate cycles in typical MPI apps
  – So, the network is used for a fraction of time,
  – and is on the critical path
• So, current *communication networks are over-engineered for by necessity*
• With overdecomposition
  – Communication is spread over an iteration
Decomposition Independent of numCores

- Rocket simulation example under traditional MPI

1
Solid
Fluid

2
Solid
Fluid

. . .

P
Solid
Fluid

- With migratable-objects:

Solid_1
Fluid_1

Solid_2
Fluid_2

Solid_3
Fluid_3

. . .

Solid_n
Fluid_m

- Benefit: load balance, communication optimizations, modularity
Enabling CS technology of parallel objects and intelligent runtime systems has led to several CSE collaborative applications.

Well-known Biophysics molecular simulations App Gordon Bell Award, 2002
Object Based Over-decomposition: AMPI

- Each MPI process is implemented as a user-level thread
- Threads are light-weight and migratable!
  - <1 microsecond context switch time, potentially >100k threads per core
- Each thread is embedded in a charm++ object (chare)
A quick Example: Weather Forecasting in BRAMS

- Brams: Brazilian weather code (based on RAMS)
- AMPI version (Eduardo Rodrigues, with Mendes and J. Panetta)
Baseline: 64 objects on 64 processors
Over-decomposition: 1024 objects on 64 processors: Benefits from communication/computation overlap
With Load Balancing:
1024 objects on 64 processors

<table>
<thead>
<tr>
<th>Description</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>No overdecomp (64 threads)</td>
<td>4988 sec</td>
</tr>
<tr>
<td>Overdecomp into 1024 threads</td>
<td>3713 sec</td>
</tr>
<tr>
<td>Load balancing (1024 threads)</td>
<td>3367 sec</td>
</tr>
</tbody>
</table>
Saving Cooling Energy

• Easy: increase A/C setting
  – But: some cores may get too hot
• Reduce frequency if temperature is high
  – Independently for each core or chip
• This creates a load imbalance!
• Migrate objects away from the slowed-down processors
  – Balance load using an existing strategy
  – Strategies take speed of processors into account
• Recently implemented in experimental version
  – SC 2011 paper
• Several new power/energy-related strategies
Fault Tolerance in Charm++/AMPI

• Four Approaches:
  – Disk-based checkpoint/restart
  – In-memory double checkpoint/restart
  – Proactive object migration
  – Message-logging: scalable fault tolerance

• Common Features:
  – Leverages object-migration capabilities
  – Based on dynamic runtime capabilities
In-memory double checkpointing

• Is practical for many apps
  – Relatively small footprint at checkpoint time
  – Also, you can use non-volatile node-local storage (e.g. FLASH)
Checkpoint time is low: 4 milliseconds for MD, essentially, live-data-permutation for any app.

![Checkpoint Time - Intrepid(leanMD)](image)

- **125000 atoms**
- **1 million atoms**
Restart time is low: 150 milliseconds on 64K cores, detection time, and re-execution times not included.
HPC Challenge Competition

• Conducted at Supercomputing 2011
• 2 parts:
  – Class I: machine performance
  – Class II: programming model productivity
    • Has been typically split in two sub-awards
  – We implemented in Charm++
    • LU decomposition
    • RandomAccess
    • LeanMD
    • Barnes–Hut
• Finalists in 2011:
  – Chapel (Cray), CAF (Rice), and Charm++ (UIUC)
Strong Scaling on Hopper for LeanMD

Gemini Interconnect, much less noisy

Performance on Hopper (125,000 atoms)

No LB
Refine LB
CharmLU: productivity and performance

- 1650 lines of source
- 67% of peak on Jaguar
Barnes–Hut

High Density Variation with a *Plummer* distribution of particles

![Barnes-Hut scaling on BG/P](image)
Charm++ interoperates with MPI

(a) Time Sharing

- MPI Control
- Charm++ Control

P(1) P(2) P(N-1) P(N)
Summary of ARTS

• Charm++ is a sophisticated programming “language”,

• It is supported by a rich adaptive runtime system, which supports:
  – Adaptive overlap of communication/computation
  – Parallel composition
  – Dynamic load balancing
  – Fault tolerance

• Is a production-quality system used by many apps in routine use by CSE scientists
So…

• Charm++ is a sophisticated programming “language”,
• It is supported by a rich adaptive runtime system, which supports:
  – Adaptive overlap of communication/computation
  – Parallel composition
  – Dynamic load balancing
  – Fault tolerance
• Is a production-quality system used by many apps in routine use by CSE scientists
• How does it help the MTAGS community?
Support for Task Parallelism
Task Parallelism support

• Dynamic creation of chares, supported by a “seed balancer”, supports
  – Master–slave
  – Divide–and–conquer
  – State–space (combinatorial) search

• One can assign priorities with each task
  – And with each response as well
  – Supported by a prioritized load balancer
Some Examples:

Finding any feasible solution
While controlling mem. usage

With priorities, search tends proceed in this fashion,
Leading to very low memory usage: P +D
(P: processors, D: depth)
Combinatorial Search Examples

• A*, IDA* (memory efficient A*), …
• Branch–and–bound search
• Graph coloring, …
• Game trees
• Parallel logic programming

• All of these have been done well using Charm++
• To the extent Task parallelism is relevant to MTAGS, these capabilities are useful

11/11/2012 Charm and MTAGS
Handling Speed Heterogeneity
Different CPU speeds

• This may happen because
  – Static: a cloud/cluster environment has a mix of nodes with different capabilities
  – Dynamic: physical node may be time–shared (with other VMs, for example)
  – Frequency changes in hot spots

• But is easy to handle:
  – The RTS measures speeds and balances load accordingly
  – Measures idle time, and can adapt to dynamic loads
    • By migrating objects away from time–shared overloaded nodes

• See http://ppl.cs.illinois.edu/research/cloud
Handling Increased or Variable Latencies
Latencies

- Message-Driven execution mitigates the impact of latencies
  - With multiple objects per PE
  - Adaptive and automatic overlap of communication and computation

- Even more dramatic example:
  - Running a single, tightly coupled, application across geographically separated clusters
  - Work from Greg Koenig’s dissertation:
Multi-Cluster Co-Scheduling

- Job co-scheduled to run across two clusters to provide access to large numbers of processors
- But cross-cluster latencies are large
- Virtualization within Charm++ masks high inter-cluster latency by allowing overlap of communication with computation

Cluster A

Cluster B

Intra-cluster latency (microseconds)

Inter-cluster latency (milliseconds)
Five-Point Stencil Results
(2048x2048 mesh, P=16)

Execution Time
(milliseconds/step)

Latency (milliseconds)

Number of Objects = 16
Number of Objects = 64
Number of Objects = 256
Multi-Cluster Co-Scheduling

LeanMD running Hydrophobic Cluster Analysis with 30,652 atoms

Execution Time (seconds/step) vs Latency (milliseconds)

- Processors 2
- Processors 4
- Processors 8
- Processors 16
- Processors 32
- Processors 64

Charm and MTAGS
Live Interaction with Parallel Jobs:
The client-server interface and its uses
Interactive Parallel Jobs

• Need for real-time communication with parallel applications
  – Steering computation
  – Visualizing/Analyzing data
  – Debugging problems

• Long running applications
  – Time consuming to recompile the code (if at all available)
  – Need to wait for application to re-execute

• Communication requirements:
  – Fast (low user waiting time), Scalable
  – Uniform method of connection

• User controlled workflow
Charm++ Client–Server Interface

1) Send request
2) Execute the request
3) Combine results
4) Send back reply later

External Client

Server frontend

Parallel program

Converse Client Server

Client
Large Scale Debugging: Motivations

- Bugs in sequential programs
  - Buffer overflow, memory leaks, pointers, ...
  - More than 50% programming time spent debugging
  - GDB and others

- Bugs in parallel programs
  - Race conditions, non-determinism, ...
  - Much harder to find
    - Effects not only happen later in time, but also on different processors
  - Bugs may appear only on thousands of processors
    - Network latencies delaying messages
    - Data decomposition algorithm
  - TotalView, Allinea DDT
CharmDebug Overview

CharmDebug Java GUI (local machine)  
Firewall

Parallel Application (remote machine)

CCS (Converse Client-Server)

Application

GDB
*** LEAKING ***
Memory type: Message
Set at position 0x10072b8 of size 912 bytes Belonging to share 0. Backtrace:
- function CmMBc (0x440c15) at ??
- function CMAbc (0x4a9510a) at ??
- function CMXMessage_Client:: alloc() (0x45c126f) at jacob2d.cpp def.250
- function CMMessage_Client:: operator new(unsigned long) (0x45c12ea) at jacob2b.cpp def.237
- function Jacobi::beginIteration() (0x46006a) at jacob2d.cpp def.202
- function CMIndex::Jacobii::callBeginIteration(void*, Jacobii*) (0x45c30e) at jacob2d.cpp def.443
- function CMMessage::DeliverMessageRedeately (0x4904a2) at ??
- function CMMessage::DeliverMessage(void*, int, bool) (0x4a9413) at ??
- function CMArrary::beginBroadcast(CMMessage*, ArrayElement*) (0x49d7) at ??
- function CMArray::endBroadcast(CMMessage*, int) (0x4b05c9) at ??
- function CMMessage::DeliverMessageFree (0x4b81e11) at ??
- function _processMessage (0x493c46) at ??
- function CMMessage::handleMessage (0x4a70c3e) at ??
Online, Interactive Access to Parallel Performance Data: Motivations

- Observation of time-varying performance of long-running applications through streaming
  - Re-use of local performance data buffers
- Interactive manipulation of performance data when parameters are difficult to define a priori
  - Perform data-volume reduction before application shutdown
    - k-clustering parameters (like number of seeds to use)
    - Write only one processor per cluster
Projections: Online Streaming of Performance Data

- Parallel Application records performance data on local processor buffers
- Performance data is periodically processed and collected to a root processor
- Charm++ runtime adaptively co-schedules the data collection's computation and messages with the host parallel application's
- Performance data buffers can now be re-used
- Remote tool collects data through CCS
Projections: Online Streaming of Performance Data

- Parallel Application records performance data on local processor buffers
- Performance data is periodically processed and collected to a root processor
- Charm++ runtime adaptively co-schedules
  - The data collection's computation and messages
  - with the host parallel application’s
- Performance data buffers can now be re-used
- Remote tool collects data through CCS
System Overview

1. Broadcast Request for Utilization Profiles Once Per Second

2. Reduction Merges Compressed Utilization Profiles

3. Buffer Utilization Profiles

Periodic Requests
## Impact of Online Performance Data Streaming

### Simple Charm++ Parallel Application

(Iterations of Work + Barriers)

<table>
<thead>
<tr>
<th># Cores</th>
<th>Exec Time in seconds (no Data Collection and Streaming)</th>
<th>Exec Time in seconds (with Data Collection and Streaming*)</th>
</tr>
</thead>
<tbody>
<tr>
<td>4095</td>
<td>21.44s</td>
<td>21.46s</td>
</tr>
<tr>
<td>8191</td>
<td>37.84s</td>
<td>37.71s</td>
</tr>
</tbody>
</table>

* Global Reduction of 8 kilobyte messages from each processor every second.

### NAMD 1-million atom simulation (STMV)

<table>
<thead>
<tr>
<th># Cores</th>
<th>512</th>
<th>1024</th>
<th>2048</th>
<th>4096</th>
<th>8192</th>
</tr>
</thead>
<tbody>
<tr>
<td>Overhead (%) no Data Collection and Streaming to visualization client.</td>
<td>0.69%</td>
<td>0.55%</td>
<td>-3.44%</td>
<td>1.56%</td>
<td>1.29%</td>
</tr>
<tr>
<td>Overhead (%) with Data Collection and Streaming@</td>
<td>0.30%</td>
<td>0.43%</td>
<td>-3.94%</td>
<td>3.47%</td>
<td>6.63%</td>
</tr>
</tbody>
</table>

@ Global Reductions per second of between 3.5 to 11 kilobyte messages from each processor. The visualization client receives 12 kilobytes/second.
Online Visualization of Streamed Performance Data

- Pictures show 10-second snapshots of live NAMD detailed performance profiles from start-up (left) to the first major load-balancing phase (right) on 1024 Cray XT5 processors
- Ssh tunnel between client and compute node through head-node
System Overview

(1) Send Request via TCP using CCS protocol

(4) Update Display

(2) Retrieve a Buffered Utilization Profile

(3) CCS Reply Contains Utilization Profile
Cosmological Data Analysis: Motivations

- Astronomical simulations/observations generate huge amount of data
- This data cannot be loaded into a single machine
- Even if loaded, interaction with user too slow

- Need to parallel analyzer tools capable of
  - Scaling well to large number of processors
  - Provide flexibility to the user
Salsa

Collaboration with Prof. Quinn, (U. Washington) and Prof.
LiveViz

- Every piece is represented by a chare
- Under integration in ChaNGa (simulator)
Faucets Project Experience:
Shrink/Expand jobs, with an adaptive job scheduler
The Faucets Project

• Motivations
  – Increasing trend towards individual organizations owning their own computational resources
  – Computational power is too dispersed and hard to use
  – Workload of most organizations occurs in bursts
  – Rigid job scheduling leads to internal fragmentation of resources

• Objectives
  – Support the metaphor of computing power as a utility
  – Make it easier to use remote compute power
  – Efficient utilization of individual clusters
  – Improve the throughput of jobs in a federation of clusters
Aspects of the Faucets Project

• Theme:
  – Efficient resource allocation via adaptive strategies for
    • Higher throughput/utilization
    • Shorter response times

• Resource Utilization within a cluster
  – Leveraging our adaptive run time system
  – A new cluster scheduler

• Resource Utilization across clusters
  – Meta-scheduling and Market economy

• Supporting a single job on multiple clusters
Inefficient Utilization within a cluster

Current Job Schedulers can lead to low system utilization!
Adaptive Job Scheduler

- Scheduler can take advantage of the adaptivity of AMPI and Charm++ jobs
- Improve system utilization and response time
- Scheduling decisions
  - Shrink existing jobs when a new job arrives
  - Expand jobs to use all processors when a job finishes
- Processor map sent to the job
  - Bit vector specifying which processors a job is allowed to use
    - 00011100 (use 3 4 and 5!)
- Handles regular (non-adaptive) jobs
Two Adaptive Jobs

16 Processor system

- **Job A**
- **Job B**

Max_pe = 10
Min_pe = 1

Allocate B!

Allocate B!

Allocate B!

Allocate B!

Allocate B!

Allocate B!

Allocate B!

Allocate B!

Allocate B!

Allocate B!

Allocate B!

Allocate B!

Allocate B!

Allocate B!

Allocate B!

Allocate B!

Allocate B!

Allocate B!

Allocate B!

Allocate B!

Allocate B!

Allocate B!

Allocate B!

Allocate B!

Allocate B!

Allocate B!

Allocate B!

Allocate B!

Allocate B!

Allocate B!

Allocate B!

Allocate B!

Allocate B!

Allocate B!

Allocate B!

Allocate B!

Allocate B!

Allocate B!

Allocate B!

Allocate B!

Allocate B!

Allocate B!

Allocate B!

Allocate B!

Allocate B!

Allocate B!

Allocate B!

Allocate B!

Allocate B!

Allocate B!

Allocate B!

Allocate B!

Allocate B!

Allocate B!

Allocate B!

Allocate B!

Allocate B!

Allocate B!

Allocate B!

Allocate B!

Allocate B!

Allocate B!

Allocate B!

Allocate B!

Allocate B!

Allocate B!

Allocate B!

Allocate B!

Allocate B!

Allocate B!

Allocate B!

Allocate B!

Allocate B!

Allocate B!

Allocate B!

Allocate B!

Allocate B!

Allocate B!

Allocate B!

Allocate B!

Allocate B!

Allocate B!

Allocate B!

Allocate B!

Allocate B!

Allocate B!

Allocate B!

Allocate B!

Allocate B!

Allocate B!

Allocate B!

Allocate B!

Allocate B!

Allocate B!

Allocate B!

Allocate B!

Allocate B!

Allocate B!

Allocate B!

Allocate B!

Allocate B!

Allocate B!

Allocate B!

Allocate B!

Allocate B!

Allocate B!

Allocate B!

Allocate B!

Allocate B!

Allocate B!

Allocate B!

Allocate B!

Allocate B!

Allocate B!

Allocate B!

Allocate B!

Allocate B!

Allocate B!

Allocate B!

Allocate B!

Allocate B!

Allocate B!

Allocate B!

Allocate B!

Allocate B!

Allocate B!

Allocate B!

Allocate B!

Allocate B!

Allocate B!

Allocate B!

Allocate B!

Allocate B!

Allocate B!

Allocate B!

Allocate B!

Allocate B!

Allocate B!

Allocate B!

Allocate B!

Allocate B!

Allocate B!

Allocate B!

Allocate B!

Allocate B!

Allocate B!

Allocate B!

Allocate B!

Allocate B!

Allocate B!

Allocate B!

Allocate B!

Allocate B!

Allocate B!

Allocate B!

Allocate B!

Allocate B!

Allocate B!

Allocate B!

Allocate B!

Allocate B!

Allocate B!

Allocate B!

Allocate B!

Allocate B!

Allocate B!

Allocate B!

Allocate B!

Allocate B!

Allocate B!

Allocate B!

Allocate B!

Allocate B!

Allocate B!

Allocate B!

Allocate B!

Allocate B!

Allocate B!

Allocate B!

Allocate B!

Allocate B!

Allocate B!

Allocate B!

Allocate B!

Allocate B!

Allocate B!

Allocate B!

Allocate B!

Allocate B!

Allocate B!

Allocate B!

Allocate B!

Allocate B!

Allocate B!

Allocate B!

Allocate B!

Allocate B!

Allocate B!

Allocate B!

Allocate B!

Allocate B!

Allocate B!

Allocate B!

Allocate B!

Allocate B!

Allocate B!

Allocate B!

Allocate B!

Allocate B!

Allocate B!

Allocate B!

Allocate B!
Shrink/Expand

- Problem: Availability of computing platform may change
- Fitting applications on the platform by object migration

Time per step for the million-row CG solver on a 16-node cluster
Additional 16 nodes available at step 600
AQS: Adaptive Queuing System

- Multithreaded
- Reliable and robust
- Deployed on multiple Linux clusters at UIUC
- Supports most features of standard queuing Sys.
- Has the ability to manage adaptive jobs currently implemented in Charm++ and MPI
- Handles regular (non-adaptive) jobs
- For more details: [http://ppl.cs.illinois.edu/research/faucets](http://ppl.cs.illinois.edu/research/faucets)
Experimental Utilization

Simulation Results of System Utilization for Traditional and Adaptive Jobs.
Experimental MRT

[Bar chart showing experimental results of mean response time for traditional and adaptive jobs.]

- Traditional Job
- Adaptive Jobs

System Load (%): 12, 30, 60, 100, 108

Mean Response Time (s): 0, 50, 100, 150, 200, 250, 300, 350
Faucets: Scheduling Across the Grid

• “Central” source of compute power
  – Users
  – Providers of compute resources
  – User account not needed on every resource

• Match users and providers
  – Market economy?
  – QoS requirements, contracts and bidding systems

• GUI or web-based interface
  – Submission
  – monitoring
Parallel systems need to maximize their efficiency!

Faucets

http://ppl.cs.illinois.edu/research/faucets
System Overview

FAUCETS SERVER

GUI CLIENT (or) Web Browser

CLUSTER

CLUSTER DAEMON

ADAPTIVE Q SYSTEM

PE PE PE

CLUSTER
Replica Computations
Replica Methods

• Motivation
  – Scientific studies often require multiple runs
    • with minor changes in initial conditions: results are combined to increase accuracy
    • Forking alternatives …
    • Soft error detection
  – But if working on small problem sizes, strong scaling is not seen – larger systems do not help.

• Solution
  – Run RTS supported “replicas” of simulation
  – Add code for replicas to enable combining of results in situ
Replica in Charm++

- Charm++ RTS divides the allocated processors into *Charm Instances* – users can plugin their partitioning code
- Each instance runs a simulation, and are unaffected by other instances
  - Interact within my instance as before
  - No change in existing code
- Asynchronous, non-blocking communication messages to other instances
  - `RemoteSend(to_partition, rank_within_partition, message)`
- Examples of usage: Thanks to TCBG/Prof. Schulten
First application of parallel tempering is CHARMM Drude-oscillator polarizable force field development by Alex MacKerell (U. Maryland)

Distribution of backbone dihedral angles at different temperatures from 64-replica simulation of Acetyl-(AAQAA)3-amide peptide on Blue Gene/P

Data from Luo & Roux, ANL/UC.
**DBP7: Membrane Transporters** – First BTRC application of replica exchange for **umbrella sampling** on collective variables

Quaternion-based order parameters from collective variables module

Inward-Facing ↔ Outward-Facing transition of GlpT transporter in explicit membrane/water environment (not shown)

**Efficient Reaction Path Sampling**

Free Energy (kcal/mol)

Reaction Path ($\theta_1 + \theta_7$)

12 replicas
Usage and Future Work

• To the command line,
  – Add `+partitions <num_partitions>`
  – This will create block-division based `num_partitions` Charm instances, each with a unique partition number

• Future work
  – Support topology aware partitioning
  – Heterogeneous tasks in partitions
  – Stretch partitions as needed
Conclusion

• Adaptive runtime systems have proved useful in pure HPC settings
• The same adaptivity features, especially migratability and message-driven execution, prove useful in multiple-tasks contexts
• dynamic interactive controllability through scripting, both external and embedded, supports rich variety of job types