Scalable Interaction with Parallel Applications

Filippo Gioachin
Chee Wai Lee
Laxmikant V. Kalé

Department of Computer Science
University of Illinois at Urbana-Champaign
Outline

• Overview
  - Charm++ RTS
  - Converse Client Server (CCS)
• Case Studies
  - CharmDebug (parallel debugger)
  - Projections (performance analysis tool)
  - Salsa (particle analysis tool)
• Conclusions
Overview

• 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**
Charm++ Overview

- Middleware written in C++
  - Message passing paradigm (asynchronous comm.)
- User decomposes work among objects (chars)
  - The objects can be virtual MPI processors
- System maps chars to processors
  - automatic load balancing
  - communication optimizations
Adaptive overlap and modules

- Allow easy integration of different modules
- Automatic overlap of communication and computation

Diagram showing overlap and modules with arrows connecting A, B, and C, and timelines indicating busy and idle states.
Develop Abstractions in Context of Full-Scale Applications

- Quantum Chemistry
  - LeanCP

- Computational Cosmology

- NAMD: Molecular Dynamics
  - STMV virus simulation

- Parallel Objects, Adaptive Runtime System
  - Libraries and Tools

- Protein Folding
  - Space-time meshes

- Rocket Simulation

- Dendritic Growth

- Crack Propagation
**Charm++ RTS**

Processors 0, 1, and N-1 are illustrated in a block diagram. Each processor contains a Charm++ RTS Module and a Python Module within the Machine Layer. The Charm++ RTS Module is responsible for executing requests, while the Python Module is used for external client interactions.

### Flow of Operations

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

The diagram also shows the interconnect between the processors and the external client, indicating the flow of data and operations within the Charm++ RTS framework.

**Server frontend**

**Client**

**Parallel program**
Case study: Parallel Debugging
Large Scale Debugging: Motivations

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

• Bugs on 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**

![Diagram showing CharmDebug Java GUI (local machine), Firewall, Parallel Application (remote machine), CCS (Converse Client-Server), Application, and GDB.]

- CharmDebug Java GUI (local machine)
- Firewall
- Parallel Application (remote machine)
- CCS (Converse Client-Server)
- Application
- GDB
Main Program View

- **entry methods**
- **processor subsets**
- **output**
- **messages queued**
- **message details**
CharmDebug at scale

• Current parallel debuggers don't work
  – Direct connection to every processor
  – STAT (MRNet) not a full debugger

• Kraken: Cray XT4 at NICS
  – Parallel operation collecting total allocated memory
  – Time at the client 16~20 ms
  – Up to 4K processors
  – Other tests to come

• Attaching to the running program took also very little (few seconds)
CharmDebug: Introspection

```python
def method(self):
    height = charm.getStatic(block_height)
    width = charm.getStatic(block_width)
    temp = charm.getValue(self, Jacobi, temperature)
    for i in range(height+2):
        row = charm.getArray(temp, double*, i)
        for j in range(width+2):
            value = charm.getArray(row, double, j)
            if (value < 0.0 or value > 255.0):
                return [i,j]
```

- **B: ghostsFromLeft(int width, const char* arg)**
- **B: ghostsFromRight(int width, const char* arg)**
- **ghostsFromTop(int width, const char* arg)**
- **A: ghostsFromBottom(int width, const char* arg)**
- **compute(void)**
- **requestNextFrame(liveVizRequest)**
Severe leak: ghost layer messages leaked every iteration
Case study: Performance Analysis
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
# Impact of Online Performance Data Streaming

## Simple Charm++ Parallel Application

(Iterations of Work + Barriers)

<table>
<thead>
<tr>
<th># Cores</th>
<th></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></td>
<td>21.44s</td>
<td>21.46s</td>
</tr>
<tr>
<td>8191</td>
<td></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
Case study: Cosmological Data Analysis
Comsological 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
**Write your own piece of Python script**

```
numParticles = charm.countParticles()
cek.print("number of particles: "+repr(numParticles))
```

number of particles: 1235400

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

- Every piece is represented by a chare

- Under integration in ChaNGa (simulator)
How well are we doing?

- JPEG is CPU bound
  - Inefficient on high bandwidth networks
- Bitmap is network bound
  - Bad on slow networks
- The bottleneck is on the client (network or processor)
  - Parallel application: use enough processors

<table>
<thead>
<tr>
<th>Window size</th>
<th>Gigabit network</th>
<th>2MB/s wireless</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Bitmap</td>
<td>JPEG</td>
</tr>
<tr>
<td>256x256</td>
<td>333</td>
<td>25</td>
</tr>
<tr>
<td>512x512</td>
<td>166</td>
<td>24</td>
</tr>
<tr>
<td>1024x1024</td>
<td>50</td>
<td>15</td>
</tr>
<tr>
<td>2048x2048</td>
<td>13</td>
<td>4</td>
</tr>
</tbody>
</table>

Courtesy of Prof. Lawlor, U.Alaska
Impostors: Basic Idea

Camera

Impostor

Geometry

Courtesy of Prof. Lawlor, U.Alaska
Particle Set to Volume Impostors

Courtesy of Prof. Lawlor, U.Alaska
Summary

• Generic framework (CCS) to connect to a running parallel application, and interact with it
• Demonstration in different scenarios:
  – Parallel debugging
    • Low response time
  – Performance analysis
    • Low runtime overhead
  – Application (cosmological) data analysis
    • High frame rate
• All code is open source and available on our website
Questions?

Thank you

http://charm.cs.uiuc.edu/