Projections: Scalable Performance Analysis and Visualization

Jonathan Lifflander, Laxmikant V. Kale
{jliff12, kale}@illinois.edu

University of Illinois Urbana-Champaign

October 14, 2013
Programming Model

→ Charm++

- Work is decomposed into objects that interact.
Programming Model

→ Charm++

- Work is decomposed into objects that interact
- Objects are logical, location-oblivious entities
Programming Model

→ Charm++

- Work is decomposed into objects that interact
- Objects are logical, location-oblivious entities
- Runtime maps them to a processor
  - May migrate them during execution due to dynamic load imbalance
Programming Model

→ Charm++

- Work is decomposed into objects that interact
- Objects are logical, location-oblivious entities
- Runtime maps them to a processor
  - May migrate them during execution due to dynamic load imbalance
- Method invocation between objects causes communication if the objects are not in the same memory domain
Programming Model

→ Charm++

- Work is decomposed into objects that interact
- Objects are logical, location-oblivious entities
- Runtime maps them to a processor
  - May migrate them during execution due to dynamic load imbalance
- Method invocation between objects causes communication if the objects are not in the same memory domain
- Communication is asynchronous and *drives* the computation
Programming Model

- Work is decomposed into objects that interact
- Objects are logical, location-oblivious entities
- Runtime maps them to a processor
  - May migrate them during execution due to dynamic load imbalance
- Method invocation between objects causes communication if the objects are not in the same memory domain
- Communication is asynchronous and *drives* the computation
- Runtime system schedules which method to execute next (based on messages that have arrived)
Often communication patterns can be represented nicely by interactions between a collection of elements.
Often communication patterns can be represented nicely by interactions between a collection of elements.

Objects can be organized into typed, indexed collections:
- Dense
- Sparse
- Multi-dimensional (1d-6d)
- Elements can be dynamically inserted into or deleted
Charm++
→ Collections of Objects
Challenges

- Many more objects than processors
  - Anywhere from tens to hundreds per processor
- Fine-grained resolution of events
  - May be as small as tens of microseconds per event
- Logical entities (objects) are distinct from physical (processors)
  - Mapping may change over time
Most of the code is written in C++
Parallel objects have a corresponding parallel interface in a .ci file
The .ci file is translated to C++ code
  ▶ We have some compiler level support we can leverage
Methodology

→ Event Tracing

- **Trace-based instrumentation of events**
  - Certain methods in the system are marked as *entry* methods
    - Meaning they can be invoked remotely
    - These remote methods are automatically traced by the system
  - Messages sent and received
  - System events
    - Certain scheduler-level events or system states are recorded: processor idleness, communication overhead, message serialization, etc.
User Intervention  
→ Event Tracing

- Language gives flexibility to the user
  - Methods can be annotated by the `notrace` attribute, which causes the code generation to eliminate tracing overhead altogether
  - Non-entry methods (not traced by default), can be annotated as `local` to automatically add tracing

- API provides further control to the programmer
  - Turn tracing on or off
    - On a subset of the processors or objects
    - During some times
  - Register user-defined functions for tracing
  - Trace point events or bracketed events (register name and then call API when it occurs)
  - Save memory usage at a point in the program execution
Charm++ has several strategies built-in that have varying data/memory overheads

- Full tracing
  - An event is composed of the time, sending/receiving processor, entry method, object, etc.
  - Each event is logged per processor in memory and then is incrementally written to disk

- Summary
  - Each processor is allotted a fixed number of equally sized time bins that hold averages over the time range
Projections

- Research on this began in 1992
- Java-based visualization tool that reads traces (summary or full)
- Supports many different ways of visualizing the data
- Scaling
  - Tested with over 100k cores
  - It is multi-threaded and has been optimized for memory usage
- How to use it
  - Download the .jar, works out of the box with Charm++
  - Link with the flag `--tracemode projections`
  - `git://charm.cs.uiuc.edu/projections.git`
- Support beyond Charm++
  - We are actively improving the prototyped MPI tracing layer
  - Support for Global Arrays exists in alpha form
Timeline

→ **NAMD: Apoa1 system, 92k atoms, 32k cores, about 3 atoms per core!**
Time Profile

NAMD: Apoa1 system, 92k atoms, no communication thread
Time Profile

→ NAMD: Apoa1 system, 92k atoms, with communication thread
Histogram

→ NAMD: Apoa1 system, 92k atoms, 1-away decomposition
NAMD: Apoa1 system, 92k atoms, 2-away decomposition

Histogram
Time Profile

→ NAMD: Apoa1 system, 92k atoms, with communication thread
Usage Profile

Profile of Usage for Processors 0–255
(Time 28.220s – 28.332s)
Communication Over Time

![Communication vs Time Graph]

- Graph Type: Stacked
- X-Axis Scale: 1.0
- Y-Axis Scale: 1.0

Legend: Messages Sent, Bytes Sent, Messages Received, Bytes Received, Messages Received Externally, Bytes Received Externally

Select Entry Points, Select New Range, Save Entry Colors, Load Entry Colors
Outlier/Extrema View
Timeline

Colored by memory for LU
Profile Memory Scatter

Memory Usage (at 2,000s resolution)

Time (s)

MB

0 250 500 750 1,000 1,250 1,500 1,750 2,000

300 400 500 600 700 800 900 1,000 1,100 1,200

PE 0 PE 1 PE 2 PE 3 PE 4 PE 5 PE 6 PE 7 PE 8 PE 9 PE 10 PE 11 PE 12 PE 13 PE 14
PE 15 PE 16 PE 17 PE 18 PE 19 PE 20 PE 21 PE 22 PE 23 PE 24 PE 25 PE 26 PE 27 PE 28
PE 29 PE 30 PE 31 PE 32 PE 33 PE 34 PE 35 PE 36 PE 37 PE 38 PE 39 PE 40 PE 41 PE 42
PE 43
Profile Memory Scatter
Can we monitor performance as the application is actually running?

- Uses the Converse client/Server interface
  - We can interact with the runtime as the program runs using python
  - Allows us to stream performance data to Projections
- Demo: utilization
End-of-run Analysis

- When we scale over 100k cores the data becomes very large and unmanageable
- Deathbed analysis
  - Use the full parallel machine at the end of the execution for some analysis
  - e.g. k-means clustering to pick out exemplar processors
- We are currently developing algorithms for this
Conclusion

- Projections
  - We are constantly improving it
  - A mature tool that grew over the years out of necessity
- We are not experts in graphics or visualization
  - As the number of cores increases along with data volume, we need better techniques and help from the broader community