Projections: Scalable Performance Analysis and Visualization

Jonathan Lifflander, Laxmikant V. Kale

{jliffl2, 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

→ 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
- Runtime system schedules which method to execute next (based on messages that have arrived)

Charm++

→ Collections of Objects

- Often communication patterns can be represented nicely by interactions between a collection of elements

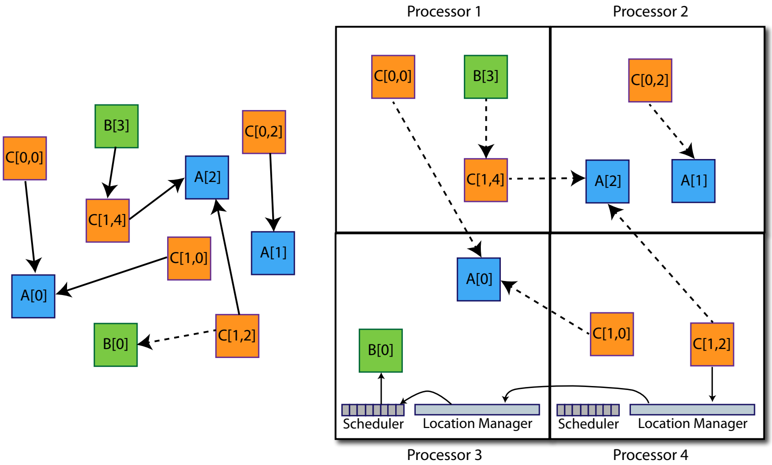
Charm++

→ Collections of Objects

- 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

Charm++

- 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

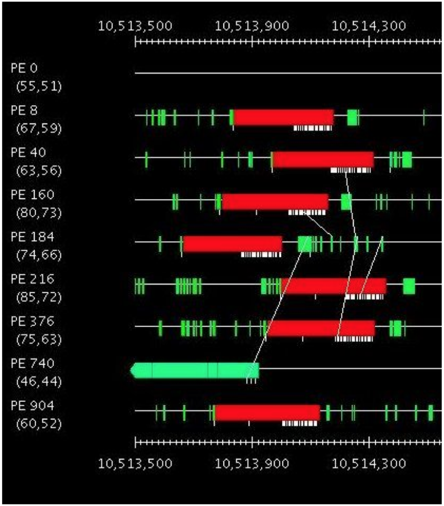
Charm++: Runtime Data Collection

- 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

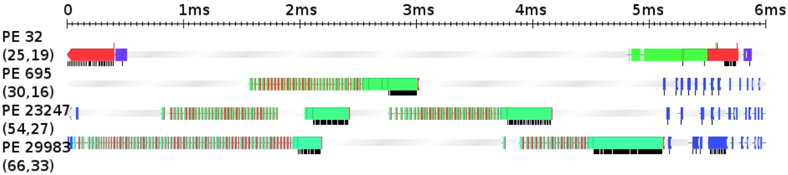
Timeline



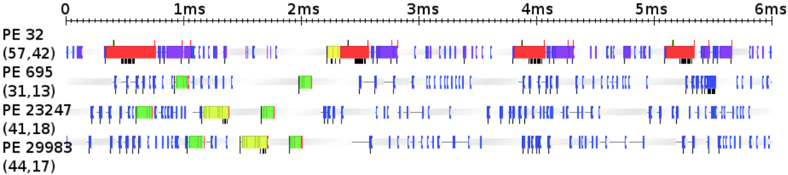
10,513,500
10,513,900
10,514,300
PE 0
(55,51)
PE 8
(67,59)
PE 40
(63,56)
PE 160
(80,73)
PE 184
(74,66)
PE 216
(85,72)
PE 376
(75,63)
PE 740
(46,44)
PE 904
(60,52)
10,513,500
10,513,900
10,514,300

Timeline

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



(a) Standard PME implementation

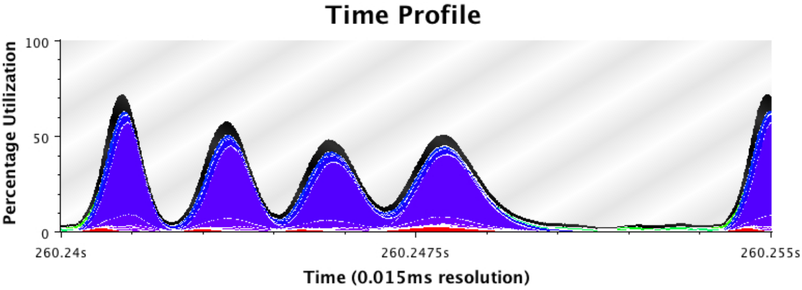


(b) Optimized PME with many-to-many

Integration | PME work | Non-bonded | Communication | Idle time | Overhead

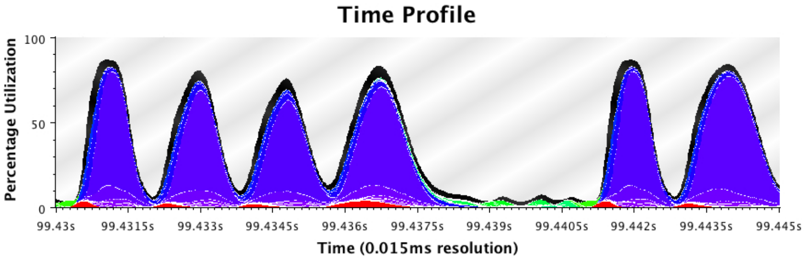
Time Profile

→ **NAMD: Apoa1 system, 92k atoms, no communication thread**



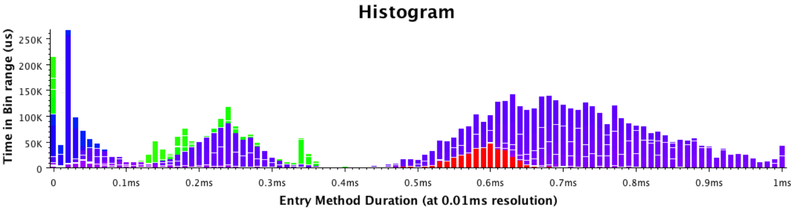
Time Profile

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



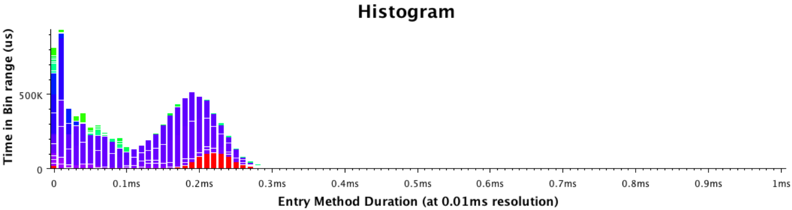
Histogram

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



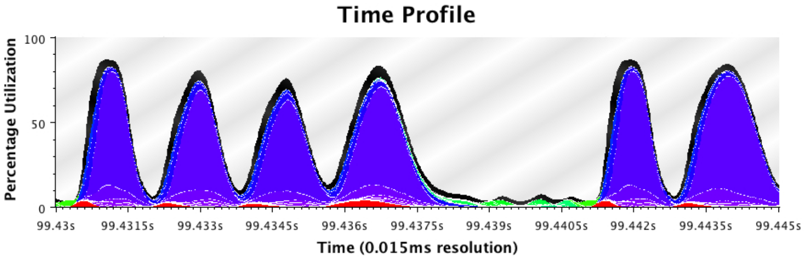
Histogram

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

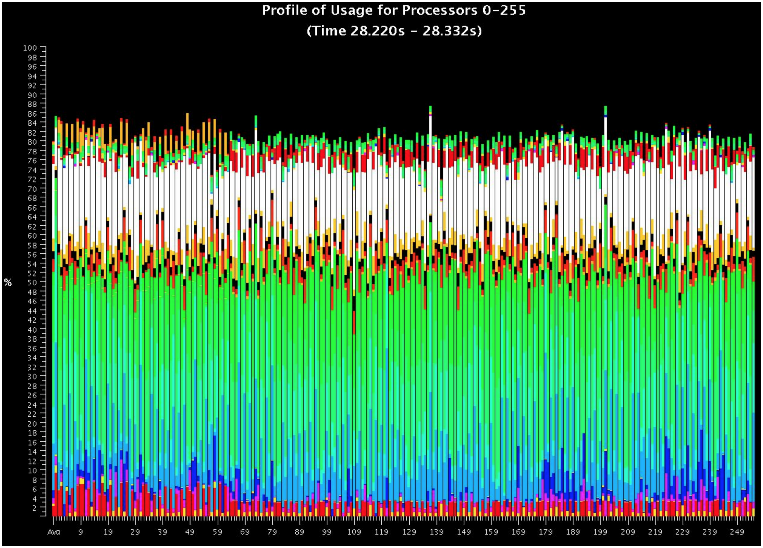


Time Profile

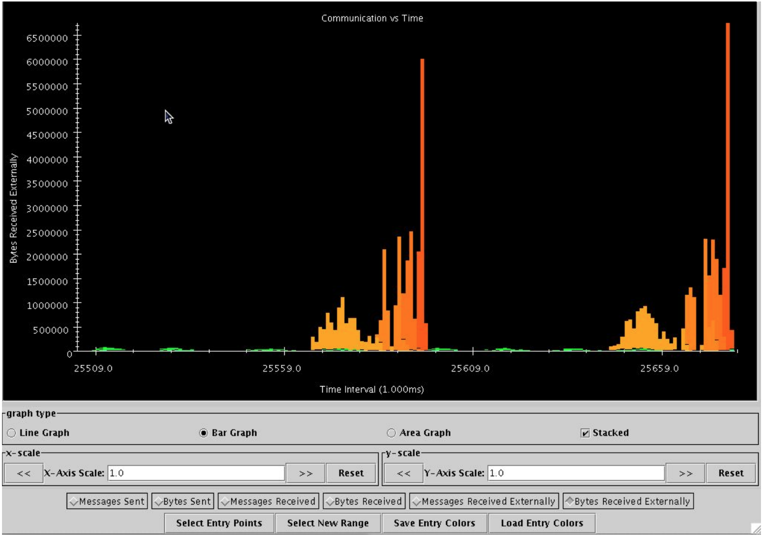
→ **NAMD: Apoa1 system, 92k atoms, with communication thread**



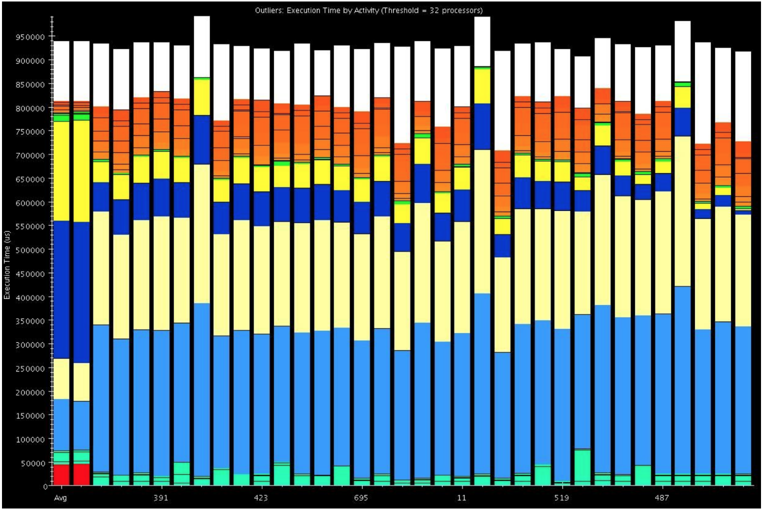
Usage Profile



Communication Over Time

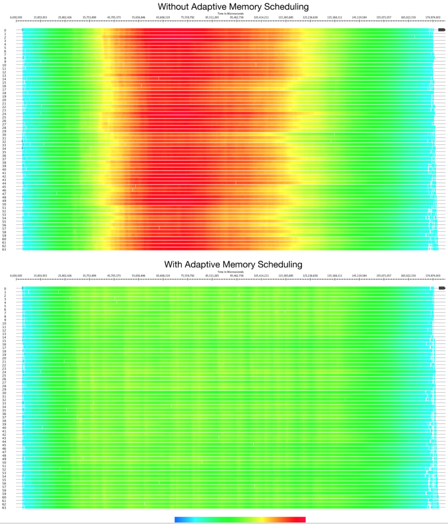


Outlier/Extrema View

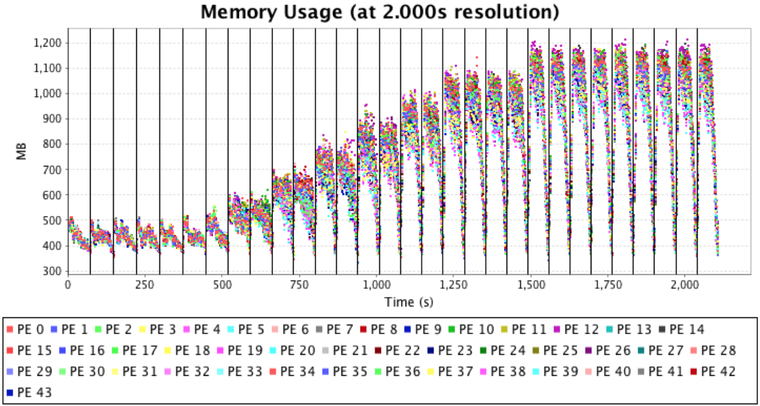


Timeline

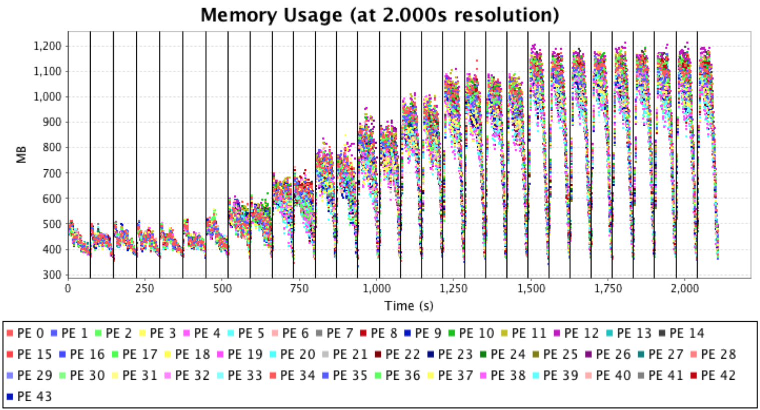
→ Colored by memory for LU



Profile Memory Scatter



Profile Memory Scatter



Demo

Live Analysis

- 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