CIS 631
Parallel Processing

Lecture 13: Parallel Performance Tools

Allen D. Malony
malony@cs.uoregon.edu

Department of Computer and Information Science
University of Oregon
Acknowledgements

☐ Portions of the lectures slides were adopted from:
Outline

- Parallel performance problem solving
- Complexity and parallel performance tools
- Measurement and analysis techniques
- Parallel performance tools
  - Profiling tools
  - Tracing tools
- Vampir trace visualization system
- TAU parallel performance system
- Automatic performance analysis and Expert
Motivation

- Tools for performance problem solving
  - Empirical-based performance optimization process

```
Performance Tuning
hypotheses

Performance Diagnosis
properties

Performance Experimentation
characterization

Performance Observation

Performance Technology
- Instrumentation
- Measurement
- Analysis
- Visualization
```
Performance Needs ⇔ Performance Technology

- **Performance observability** requirements
  - Multiple levels of software and hardware
  - Different types and detail of performance data
  - Alternative performance problem solving methods
  - Multiple targets of software and system application

- **Performance technology** requirements
  - Broad scope of performance observation
  - Flexible and configurable mechanisms
  - Technology integration and extension
  - Cross-platform portability
  - Open, layered, and modular framework architecture
Parallel Performance Technology

- Performance instrumentation tools
  - Different program code levels
  - Different system levels

- Performance measurement (observation) tools
  - Profiling and tracing of SW/HW performance events
  - Different software (SW) and hardware (HW) levels

- Performance analysis tools
  - Performance data analysis and presentation
  - Online and offline tools

- Performance experimentation

- Performance modeling and prediction tools
Complex Parallel Systems

- Complexity in computing system architecture
  - Diverse parallel system architectures
    - shared / distributed memory, cluster, hybrid, NOW, …
  - Sophisticated processor and memory architectures
  - Advanced network interface and switching architecture

- Complexity in parallel software environment
  - Diverse parallel programming paradigms
    - shared memory multi-threading, message passing, hybrid
  - Hierarchical, multi-level software architectures
  - Optimizing compilers and sophisticated runtime systems
  - Advanced numerical libraries and application frameworks
Complexity Challenges for Performance Tools

- Computing system environment complexity
  - Observation integration and optimization
  - Access, accuracy, and granularity constraints
  - Diverse/specialized observation capabilities/technology
  - Restricted modes limit performance problem solving

- Sophisticated software development environments
  - Programming paradigms and performance models
  - Performance data mapping to software abstractions
  - Uniformity of performance abstraction across platforms
  - Rich observation capabilities and flexible configuration
  - Common performance problem solving methods
How do we create robust and ubiquitous performance technology for the analysis and tuning of parallel and distributed software and systems in the presence of (evolving) complexity challenges?

How do we apply performance technology effectively for the variety and diversity of performance problems that arise in the context of complex parallel and distributed computer systems?
Definitions: Instrumentation

- Instrumentation
  - inserting extra code (hooks) into program
- Source code instrumentation
  - Manual
  - Automatic by compiler or source-to-source translator
- Object code instrumentation
  - “Re-writing” the executable to insert hooks
- Dynamic code instrumentation (like a debugger)
  - Object code instrumentation while program is running
- Pre-instrumented library
  - Typically used for MPI and PVM program analysis
Definitions: Profile Measurement

- Profiling
  - Recording of summary information
    - execution time
    - number of calls
    - number of FLOPS ...
  - Performance data about program entities
    - functions
    - loops
    - basic blocks ...
- Very good for quick, low-cost overview of performance
- Helps to expose potential bottlenecks or hotspots
- Implemented through sampling or instrumentation
Definitions: Trace Measurement

- **Tracing**
  - Recording of information about significant points (events) during program execution
    - Entering or leaving a code region (function, loop, ...)
    - Sending or receiving a message, ...
  - Saving information in *event record* consisting of
    - Timestamp
    - Location identifier (node, CPU, thread)
    - Event type plus event specific information
  - Event *trace* := sequence of event records sorted by time

- Can be used to reconstruct the dynamic behavior
- Typically requires code instrumentation
Event Tracing: Instrumentation, Monitoring, Trace

CPU A:

```c
void master {
    trace(EXIT, 1);
    ...
    trace(SEND, B);
    send(B, tag, buf);
    ...
    trace(EXIT, 1);
}
```

CPU B:

```c
void slave {
    trace(EXIT, 2);
    ...
    recv(A, tag, buf);
    ...
    trace(RECV, A);
    ...
    trace(EXIT, 2);
}
```

```plaintext
MONITOR

<table>
<thead>
<tr>
<th>Event</th>
<th>Time</th>
<th>Source</th>
<th>Event Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>58</td>
<td>A</td>
<td>ENTER</td>
<td>1</td>
</tr>
<tr>
<td>60</td>
<td>B</td>
<td>ENTER</td>
<td>2</td>
</tr>
<tr>
<td>62</td>
<td>A</td>
<td>SEND</td>
<td>B</td>
</tr>
<tr>
<td>64</td>
<td>A</td>
<td>EXIT</td>
<td>1</td>
</tr>
<tr>
<td>68</td>
<td>B</td>
<td>RECV</td>
<td>A</td>
</tr>
<tr>
<td>69</td>
<td>B</td>
<td>EXIT</td>
<td>2</td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```
Event Tracing: “Timeline” Visualization

<table>
<thead>
<tr>
<th></th>
<th>master</th>
<th></th>
<th>slave</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>master</td>
<td></td>
<td>slave</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>slave</td>
<td></td>
<td>slave</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>...</td>
<td></td>
<td>...</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>ENTER</th>
<th></th>
<th>SEND</th>
<th></th>
<th>EXIT</th>
<th></th>
<th>RECV</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>58</td>
<td>A</td>
<td>1</td>
<td>B</td>
<td></td>
<td>A</td>
<td>1</td>
<td>A</td>
<td>2</td>
</tr>
<tr>
<td>60</td>
<td>B</td>
<td>2</td>
<td>B</td>
<td></td>
<td>A</td>
<td>1</td>
<td>A</td>
<td></td>
</tr>
<tr>
<td>62</td>
<td>A</td>
<td></td>
<td>B</td>
<td></td>
<td>A</td>
<td></td>
<td>A</td>
<td></td>
</tr>
<tr>
<td>64</td>
<td>A</td>
<td></td>
<td>B</td>
<td></td>
<td>A</td>
<td></td>
<td>A</td>
<td></td>
</tr>
<tr>
<td>68</td>
<td>B</td>
<td></td>
<td>A</td>
<td></td>
<td>B</td>
<td></td>
<td>A</td>
<td></td>
</tr>
<tr>
<td>69</td>
<td>B</td>
<td></td>
<td>A</td>
<td></td>
<td>B</td>
<td></td>
<td>A</td>
<td></td>
</tr>
</tbody>
</table>

...
Unix Profiling Tools (prof)

- Classical Unix profiling tools: prof and gprof
- prof
  - Sample-based measurement
    - Samples program counter (PC) at timer interrupts or traps
    - Match PC with code sections (routines) using symbol table
  - Keeps time histogram
    - Assumes all time since last sample spent in routine
    - Accumulates time per routine
  - Needs large enough samples to obtain statistical accuracy
  - Requires program to be compiled for profiling
    - Need to produce symbol table
Unix Profiling Tools (gprof)

- Interested in seeing routine calling relationships
  - Callpath profiling
- gprof
  - Sample-based measurement
    - Samples program counter (PC) at timer interrupts or traps
    - Match PC with code sections (routines) using symbol table
    - Looks on stack for calling PC and matches to calling routine
  - Keeps time histogram
    - Assumes all time since last sample spent in routine
    - Accumulates time per routine and caller
  - Needs large enough samples to obtain statistical accuracy
  - Requires program to be compiled for profiling
Performance API (PAPI, UTK)

- Time is not the only thing of interest
- Access to hardware counters on modern microprocessors

<table>
<thead>
<tr>
<th>PAPI Low-Level Interface</th>
<th>PAPI High-Level Interface</th>
</tr>
</thead>
<tbody>
<tr>
<td>PAPI Machine Dependent Substrate</td>
<td>Kernel Extension</td>
</tr>
<tr>
<td></td>
<td>Operating System</td>
</tr>
<tr>
<td></td>
<td>Performance Counter Hardware</td>
</tr>
</tbody>
</table>

- *dynaprof*
  - PAPI-based profiler
- Program Counter Library (PCL, Research Center Juelich)
Perfometer
Parallel Profiling

- All of the Unix profiling tools are sequential profilers
  - Process-oriented
- What does parallel profiling mean?
  - Capture profiles for all “threads of execution”
    - shared-memory threads
    - processes
  - What about interactions?
    - synchronization
    - communication
  - Correctly save profiles for analysis
  - How do you do analysis?
How to capture message communication events?
- MPI standard defined an interface for instrumentation
  - Alternative entry points to each MPI routine
  - “Standard” routine entry linked to instrumented library
  - Instrumented library performs measurement then calls alternative entry point for corresponding routine
    - library interposition
    - wrapper library
- PMPI used for most MPI performance measurement
- PMPI also can be used for debugging
- PERUSE (LLNL) project is a follow-on project
How to address dual performance technology goals?

- Robust capabilities + widely available methods
- Contend with problems of system diversity
- Flexible tool composition/configuration/integration

Approaches

- Restrict computation types / performance problems
  - machines, languages, instrumentation technique, ...
  - limited performance technology coverage and application
- Base technology on abstract computation model
  - general architecture and software execution features
  - map features/methods to existing complex system types
  - develop capabilities that can be adapted and optimized
General Complex System Computation Model

- **Node**: physically distinct shared memory machine
  - Message passing *node interconnection network*
- **Context**: distinct virtual memory space within node
- **Thread**: execution threads (user/system) in context

![Diagram of interconnection network showing physical and model views, nodes, contexts, thread spaces, and message communication.]
Framework for Performance Problem Solving

- Model-based composition
  - Instrumentation / measurement / execution models
    - performance observability constraints
    - performance data types and events
  - Analysis / presentation model
    - performance data processing
    - performance views and model mapping
  - Integration model
    - performance tool component configuration / integration

Can performance problem solving framework be designed based on general complex system model?
TAU Performance System Framework

- Tuning and Analysis Utilities
- Performance system framework for scalable parallel and distributed high-performance computing
- Targets a general complex system computation model
  - nodes / contexts / threads
  - Multi-level: system / software / parallelism
  - Measurement and analysis abstraction
- Integrated toolkit for performance instrumentation, measurement, analysis, and visualization
  - Portable performance profiling/tracing facility
  - Open software approach
- University of Oregon, LANL, FZJ Germany
TAU Performance System Architecture
Definitions – Instrumentation

- Instrumentation
  - Insertion of extra code (hooks) into program
  - Source instrumentation
    - done by compiler, source-to-source translator, or manually
    - portable
    - links back to program code
    - re-compile is necessary for (change in) instrumentation
    - requires source to be available
    - hard to use in standard way for mix-language programs
    - source-to-source translators hard to develop (e.g., C++, F90)
  - Object code instrumentation
    - “re-writing” the executable to insert hooks
Definitions – Instrumentation (continued)

- **Dynamic code instrumentation**
  - a debugger-like instrumentation approach
  - executable code instrumentation on running program
  - *DynInst* and *DPCL* are examples
  +/– opposite compared to source instrumentation

- **Pre-instrumented library**
  - typically used for MPI and PVM program analysis
  - supported by link-time library interposition
  + easy to use since only re-linking is necessary
  – can only record information about library entities
TAU Instrumentation

- Flexible instrumentation mechanisms at multiple levels
  - Source code
    - Manual
    - automatic
      - Program Database Toolkit (*PDT*)
      - OpenMP directive rewriting (*Opari*)
  - Object code
    - pre-instrumented libraries (e.g., MPI using *PMPI*)
    - statically linked and dynamically linked
  - Executable code
    - dynamic instrumentation (pre-execution) (*DynInstAPI*)
    - Java virtual machine instrumentation using (*JVMPI*)
TAU Instrumentation Approach

- Targets common measurement interface
  - TAU API

- Object-based design and implementation
  - Macro-based, using constructor/destructor techniques
  - Program units: function, classes, templates, blocks
  - Uniquely identify functions and templates
    - name and type signature (name registration)
    - static object creates performance entry
    - dynamic object receives static object pointer
    - runtime type identification for template instantiations
  - C and Fortran instrumentation variants

- Instrumentation and measurement optimization
Program Database Toolkit (PDT)

- Program code analysis framework
  - develop source-based tools
- High-level interface to source code information
- Integrated toolkit for source code parsing, database creation, and database query
  - Commercial grade front end parsers
  - Portable IL analyzer, database format, and access API
  - Open software approach for tool development
- Multiple source languages
- Automated performance instrumentation tools
  - TAU instrumentor
PDT Architecture and Tools

Application / Library

C / C++ parser
Fortran 77/90 parser

C / C++ IL analyzer
Fortran 77/90 IL analyzer

IL
IL

Program Database Files

PDBhtml
Program documentation

SILOON
Application component glue

CHASM
C++ / F90 interoperability

TAU_instr
Automatic source instrumentation

DUCTAPE
PDT Components

- **Language front end**
  - Edison Design Group (EDG): C, C++, Java
  - Mutek Solutions Ltd.: F77, F90
  - Creates an intermediate-language (IL) tree

- **IL Analyzer**
  - Processes the intermediate language (IL) tree
  - Creates “program database” (PDB) formatted file

- **DUCTAPE** (Bernd Mohr, FZJ/ZAM, Germany)
  - C++ program Database Utilities and Conversion Tools Application Environment
  - Processes and merges PDB files
  - C++ library to access the PDB for PDT applications
Definitions – Profiling

Profiling

- Recording of summary information during execution
  - execution time, # calls, hardware statistics, …
- Reflects performance behavior of program entities
  - functions, loops, basic blocks
  - user-defined “semantic” entities
- Very good for low-cost performance assessment
- Helps to expose performance bottlenecks and hotspots
- Implemented through
  - sampling: periodic OS interrupts or hardware counter traps
  - instrumentation: direct insertion of measurement code
Definitions – Tracing

☐ Tracing

☑ Recording of information about significant points (events) during program execution
  ➢ entering/exiting code regions (function, loop, block, …)
  ➢ thread/process interactions (e.g., send/receive messages)

☑ Save information in event record
  ➢ timestamp
  ➢ CPU identifier, thread identifier
  ➢ Event type and event-specific information

☐ Event trace is a time-sequenced stream of event records

☑ Can be used to reconstruct dynamic program behavior

☑ Typically requires code instrumentation
TAU Measurement

- Performance information
  - Performance events
  - High-resolution timer library (real-time / virtual clocks)
  - General software counter library (user-defined events)
  - Hardware performance counters
    - PCL (Performance Counter Library) (ZAM, Germany)
    - PAPI (Performance API) (UTK, Ptools Consortium)
    - consistent, portable API

- Organization
  - Node, context, thread levels
  - Profile groups for collective events (runtime selective)
  - Performance data mapping between software levels
**TAU Measurement Options**

- **Parallel profiling**
  - Function-level, block-level, statement-level
  - Supports user-defined events
  - TAU parallel profile database
  - Hardware counts values
  - Multiple counters
  - Callpath profiling

- **Tracing**
  - All profile-level events
  - Inter-process communication events
  - Timestamp synchronization

- **Configurable measurement library (user controlled)**
**TAU Measurement System Configuration**

- **configure [OPTIONS]**
  - `{-c++=<CC>, -cc=<cc>}` Specify C++ and C compilers
  - `{-pthread, -sproc , -smarts}` Use pthread, SGI sproc, smarts threads
  - `-openmp` Use OpenMP threads
  - `-opari=<dir>` Specify location of Opapi OpenMP tool
  - `{-papi ,-pcl=<dir>}` Specify location of PAPI or PCL
  - `-pdt=<dir>` Specify location of PDT
  - `{-mpiinc=<d>, mpilib=<d>}` Specify MPI library instrumentation
  - `-TRACE` Generate TAU event traces
  - `-PROFILE` Generate TAU profiles
  - `-PROFILECALLPATH` Generate Callpath profiles (1-level)
  - `-MULTIPLECOUNTERS` Use more than one hardware counter
  - `-CPUTIME` Use usertime+system time
  - `-PAPIWALLCLOCK` Use PAPI to access wallclock time
  - `-PAPIVIRTUAL` Use PAPI for virtual (user) time …
TAU Measurement API

- Initialization and runtime configuration
  - TAU_PROFILE_INIT(argc, argv);
  - TAU_PROFILE_SET_NODE(myNode);
  - TAU_PROFILE_SET_CONTEXT(myContext);
  - TAU_PROFILE_EXIT(message);

- Function and class methods
  - TAU_PROFILE(name, type, group);

- Template
  - TAU_TYPE_STRING(variable, type);
  - TAU_PROFILE(name, type, group);
  - CT(variable);

- User-defined timing
  - TAU PROFILE TIMER(timer, name, type, group);
  - TAU_PROFILE_START(timer);
  - TAU_PROFILE_STOP(timer);
TAU Measurement API (continued)

- User-defined events
  - TAU_REGISTER_EVENT(variable, event_name);
  - TAU_EVENT(variable, value);
  - TAU_PROFILE_STMT(statement);

- Mapping
  - TAU_MAPPING(statement, key);
  - TAU_MAPPING_OBJECT(funcIdVar);
  - TAU_MAPPING_LINK(funcIdVar, key);
  - TAU_MAPPING_PROFILE(funcIdVar);
  - TAU_MAPPING_PROFILE_TIMER(timer, funcIdVar);
  - TAU_MAPPING_PROFILE_START(timer);
  - TAU_MAPPING_PROFILE_STOP(timer);

- Reporting
  - TAU_REPORT_STATISTICS();
  - TAU_REPORT_THREAD_STATISTICS();
TAU Analysis

- Parallel profile analysis
  - Pprof
    - parallel profiler with text-based display
  - Racy
    - graphical interface to pprof (Tcl/Tk)
  - jRacy
    - Java implementation of Racy

- Trace analysis and visualization
  - Trace merging and clock adjustment (if necessary)
  - Trace format conversion (ALOG, SDDF, Vampir, Paraver)
  - Vampir (Pallas) trace visualization
**Pprof Command**

```plaintext
pprof [-c|-b|-m|-t|-e|-i] [-r] [-s] [-n num] [-f file] [-l] [nodes]

- **-c**  Sort according to number of calls
- **-b**  Sort according to number of subroutines called
- **-m**  Sort according to msecs (exclusive time total)
- **-t**  Sort according to total msecs (inclusive time total)
- **-e**  Sort according to exclusive time per call
- **-i**  Sort according to inclusive time per call
- **-v**  Sort according to standard deviation (exclusive usec)
- **-r**  Reverse sorting order
- **-s**  Print only summary profile information
- **-n num**  Print only first number of functions
- **-f file**  Specify full path and filename without node ids
- **-l nodes**  List all functions and exit (prints only info about all contexts/threads of given node numbers)
```
Pprof Output (NAS Parallel Benchmark – LU)

- Intel Quad PIII Xeon
- F90 + MPICH
- Profile
  - Node
  - Context
  - Thread
- Events
  - code
  - MPI

Lecture 13
jRacy (NAS Parallel Benchmark – LU)

n: node
c: context
t: thread

Global profiles

Routine profile across all nodes

Event legend

Individual profile
TAU + PAPI (NAS Parallel Benchmark – LU)

- Floating point operations
- Replaces execution time
- Only requires re-linking to different TAU library
Callpath Profiling Example (NAS LU v2.3)

% configure -PROFILECALLPATH -SGITIMERS -arch=sgi64
   -mpiinc=/usr/include -mpilib=/usr/lib64 -useropt=-O2

---

<table>
<thead>
<tr>
<th>%time</th>
<th>msec</th>
<th>total msec</th>
<th>#call</th>
<th>#subrs</th>
<th>usec/call</th>
<th>name</th>
</tr>
</thead>
<tbody>
<tr>
<td>30.7</td>
<td>13,037</td>
<td>17,495</td>
<td>301</td>
<td>602</td>
<td>58126</td>
<td>bcast_inputs =&gt; rhs</td>
</tr>
<tr>
<td>30.7</td>
<td>13,037</td>
<td>17,495</td>
<td>301</td>
<td>602</td>
<td>58126</td>
<td>rhs</td>
</tr>
<tr>
<td>17.3</td>
<td>8,195</td>
<td>9,838</td>
<td>9300</td>
<td>18600</td>
<td>1058</td>
<td>bcast_inputs =&gt; buts</td>
</tr>
<tr>
<td>17.3</td>
<td>8,195</td>
<td>9,838</td>
<td>9300</td>
<td>18600</td>
<td>1058</td>
<td>buts</td>
</tr>
<tr>
<td>21.0</td>
<td>7,669</td>
<td>11,998</td>
<td>9300</td>
<td>18600</td>
<td>1290</td>
<td>bcast_inputs =&gt; blts</td>
</tr>
<tr>
<td>21.0</td>
<td>7,669</td>
<td>11,998</td>
<td>9300</td>
<td>18600</td>
<td>1290</td>
<td>blts</td>
</tr>
<tr>
<td>12.8</td>
<td>7,320</td>
<td>7,320</td>
<td>9300</td>
<td>0</td>
<td>787</td>
<td>bcast_inputs =&gt; jacd</td>
</tr>
<tr>
<td>12.8</td>
<td>7,320</td>
<td>7,320</td>
<td>9300</td>
<td>0</td>
<td>787</td>
<td>jacd</td>
</tr>
<tr>
<td>10.6</td>
<td>6,049</td>
<td>6,049</td>
<td>9300</td>
<td>0</td>
<td>651</td>
<td>bcast_inputs =&gt; jacu</td>
</tr>
<tr>
<td>10.6</td>
<td>6,049</td>
<td>6,049</td>
<td>9300</td>
<td>0</td>
<td>651</td>
<td>jacu</td>
</tr>
<tr>
<td>7.7</td>
<td>4,385</td>
<td>4,385</td>
<td>18600</td>
<td>0</td>
<td>236</td>
<td>MPI_Recv()</td>
</tr>
<tr>
<td>7.7</td>
<td>4,385</td>
<td>4,385</td>
<td>18600</td>
<td>0</td>
<td>236</td>
<td>exchange_1 =&gt; MPI_Recv()</td>
</tr>
<tr>
<td>6.5</td>
<td>3,700</td>
<td>3,700</td>
<td>606</td>
<td>0</td>
<td>6106</td>
<td>MPI_Wait()</td>
</tr>
<tr>
<td>6.5</td>
<td>3,700</td>
<td>3,700</td>
<td>604</td>
<td>0</td>
<td>6126</td>
<td>exchange_3 =&gt; MPI_Wait()</td>
</tr>
<tr>
<td>95.8</td>
<td>1,882</td>
<td>54,609</td>
<td>2.25</td>
<td>37517</td>
<td>24270831</td>
<td>bcast_inputs</td>
</tr>
<tr>
<td>95.7</td>
<td>1,882</td>
<td>54,604</td>
<td>1.25</td>
<td>37508</td>
<td>43683863</td>
<td>applu =&gt; bcast_inputs</td>
</tr>
<tr>
<td>1.8</td>
<td>1,012</td>
<td>1,012</td>
<td>1</td>
<td>0</td>
<td>1012219</td>
<td>MPI_Finalize()</td>
</tr>
<tr>
<td>1.8</td>
<td>1,012</td>
<td>1,012</td>
<td>1</td>
<td>0</td>
<td>1012219</td>
<td>applu =&gt; MPI_Finalize()</td>
</tr>
<tr>
<td>10.5</td>
<td>996</td>
<td>5,971</td>
<td>37200</td>
<td>37200</td>
<td>161</td>
<td>exchange_1</td>
</tr>
<tr>
<td>1.5</td>
<td>882</td>
<td>882</td>
<td>19205</td>
<td>0</td>
<td>46</td>
<td>MPI_Send()</td>
</tr>
<tr>
<td>1.7</td>
<td>833</td>
<td>950</td>
<td>1</td>
<td>44686.5</td>
<td>950499</td>
<td>applu =&gt; setiv</td>
</tr>
</tbody>
</table>
Callpath Parallel Profile Display

- 0-level and 1-level callpath grouping

0-Level Callpath

<table>
<thead>
<tr>
<th>Mean n,c,t</th>
<th>0.0,0</th>
<th>1.0,0</th>
<th>2.0,0</th>
<th>3.0,0</th>
</tr>
</thead>
<tbody>
<tr>
<td>n,c,t</td>
<td>0,0</td>
<td>1,0</td>
<td>2,0</td>
<td>3,0</td>
</tr>
</tbody>
</table>

1-Level Callpath

<table>
<thead>
<tr>
<th>Mean n,c,t</th>
<th>0.0,0</th>
<th>1.0,0</th>
<th>2.0,0</th>
<th>3.0,0</th>
</tr>
</thead>
<tbody>
<tr>
<td>n,c,t</td>
<td>0,0</td>
<td>1,0</td>
<td>2,0</td>
<td>3,0</td>
</tr>
</tbody>
</table>

File Options Windows Help

0.2286% rhs
14.37% buts
13.45% blts
12.84% jacld
10.61% jcu
7.69% MPI_Recv()
6.49% MPI_Wait()
3.3% bcast_inputs
1.77% MPI_Finalize()
1.75% exchange_1
1.55% MPI_Send()
1.46% setiv
0.89% exchange_3
0.26% exact
0.25% error
0.18% MPI_Init()
0.11% erhs

File Options Windows Help

22.86% bcast_inputs => rhs
14.37% bcast_inputs => buts
13.45% bcast_inputs => blts
12.84% bcast_inputs => jacld
10.61% bcast_inputs => jcu
7.69% exchange_1 => MPI_Recv()
6.49% exchange_3 => MPI_Wait()
3.3% applu => bcast_inputs
1.77% applu => MPI_Finalize()
1.46% applu => setiv
1.03% exchange_1 => MPI_Send()
0.9% blts => exchange_1
0.89% rhs => exchange_3
0.85% buts => exchange_1
0.51% exchange_3 => MPI_Send()
0.25% applu => error
0.21% setiv => exact
TAU + Vampir (NAS Parallel Benchmark – LU)

Timeline display

Callgraph display

Parallelism display

Communications display

Lecture 13
Next Class

☐ More parallel performance tools