

HPCMP User Training

Jeff Larkin <larkin@cray.com>

Alan Minga <aminga@cray.com>

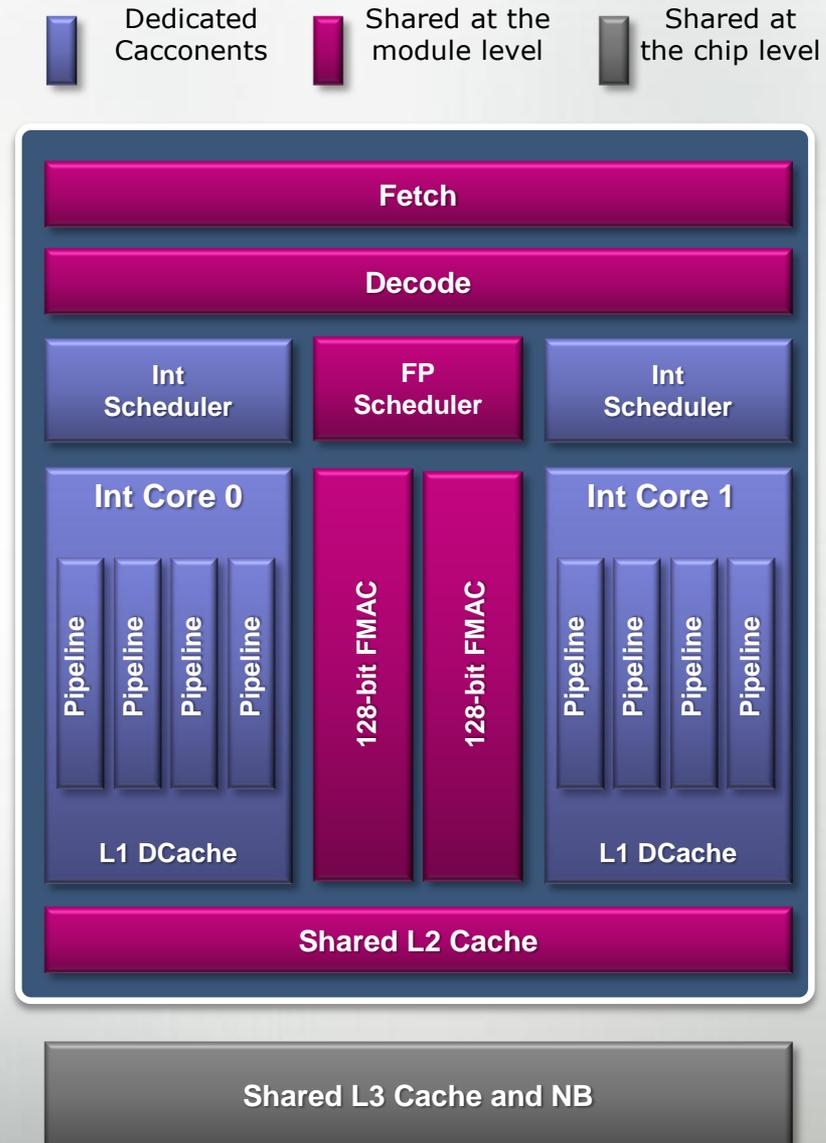
Interlagos

POP Quiz: How many “cores” is an Iterlagos?

- If you ask AMD or your OS: 16
 - The chip has 16 integer schedulers and can run 16 threads of execution.
- If you ask a Cray benchmarker: “it depends”
 - The chip has 8 floating point schedulers and 8 memory controllers, but can run 16 threads of execution.
- Modern Intel processors have a similar ambiguity, 8 physical cores, but hyperthreading reports 16 cores to the OS.
- Lesson: Don’t talk about your requirements in terms of “cores” and be specific what you actually need.

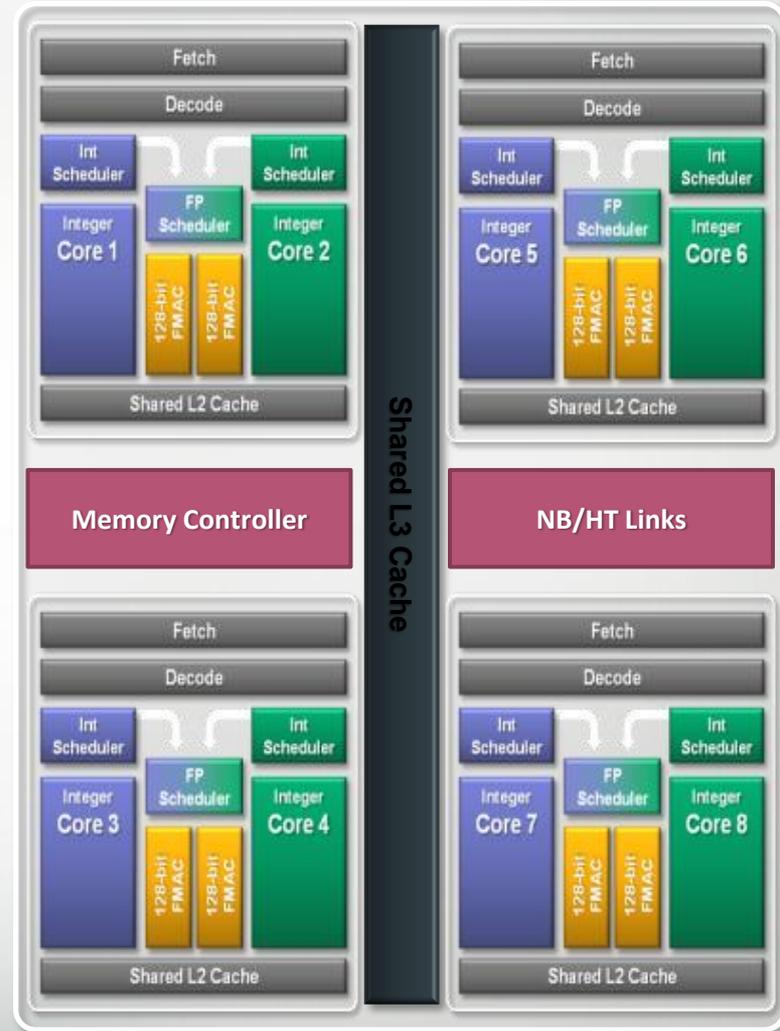
Interlagos Processor Architecture

- Interlagos is composed of a number of “Bulldozer modules” or “Compute Unit”
- A compute unit has shared and dedicated components
 - There are two independent integer units; shared L2 cache, instruction fetch, lcache; and a *shared*, 256-bit Floating Point resource
- A single Integer unit can make use of the entire Floating Point resource with 256-bit AVX instructions
 - Vector Length
 - 32 bit operands, VL = 8
 - 64 bit operands, VL = 4

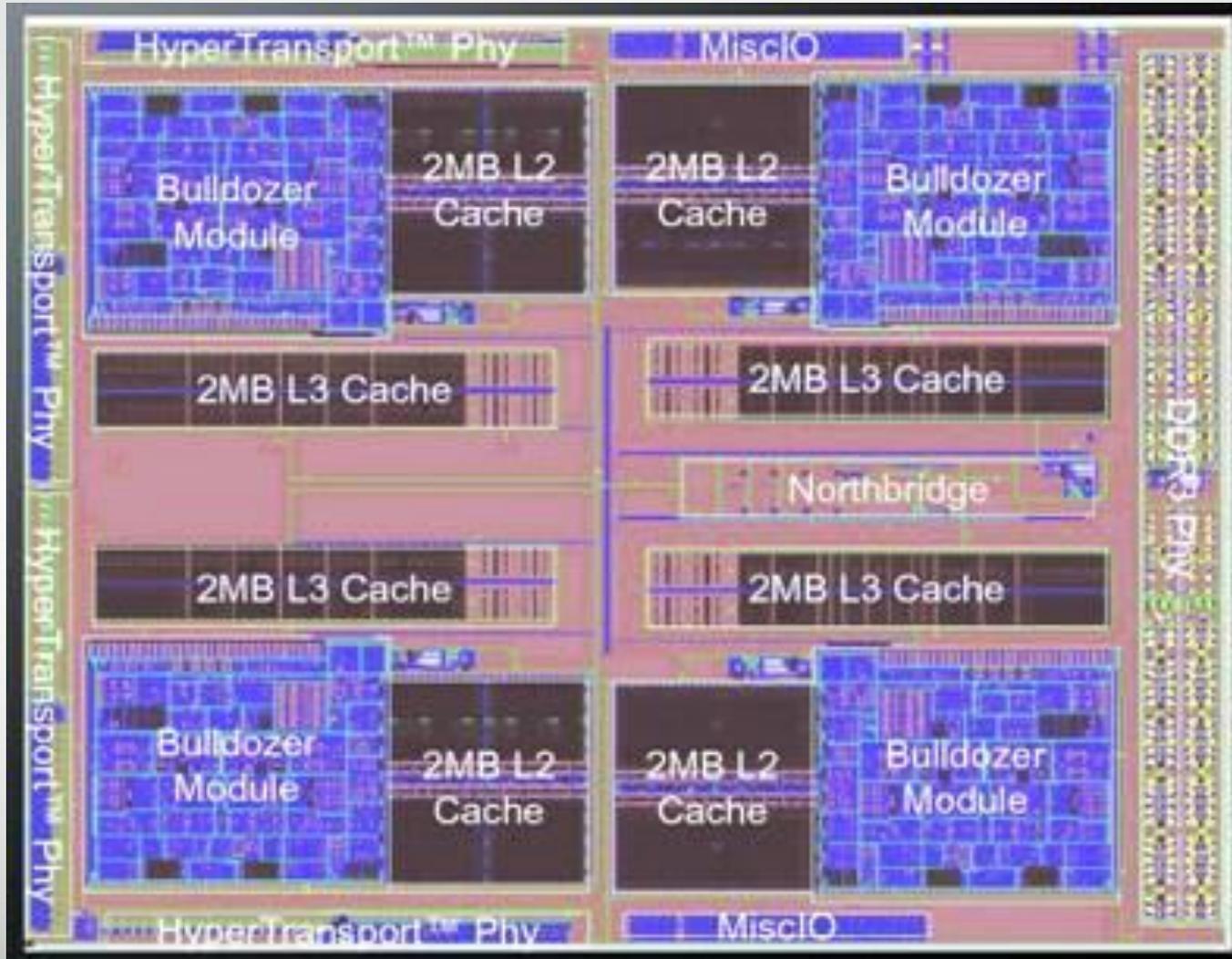


Building an Interlagos Processor

- Each processor die is composed of 4 compute units
- The 4 compute units share a memory controller and 8MB L3 data cache
- Each processor die is configured with two DDR3 memory channels and multiple HT3 links



Interlagos Die Floorplan

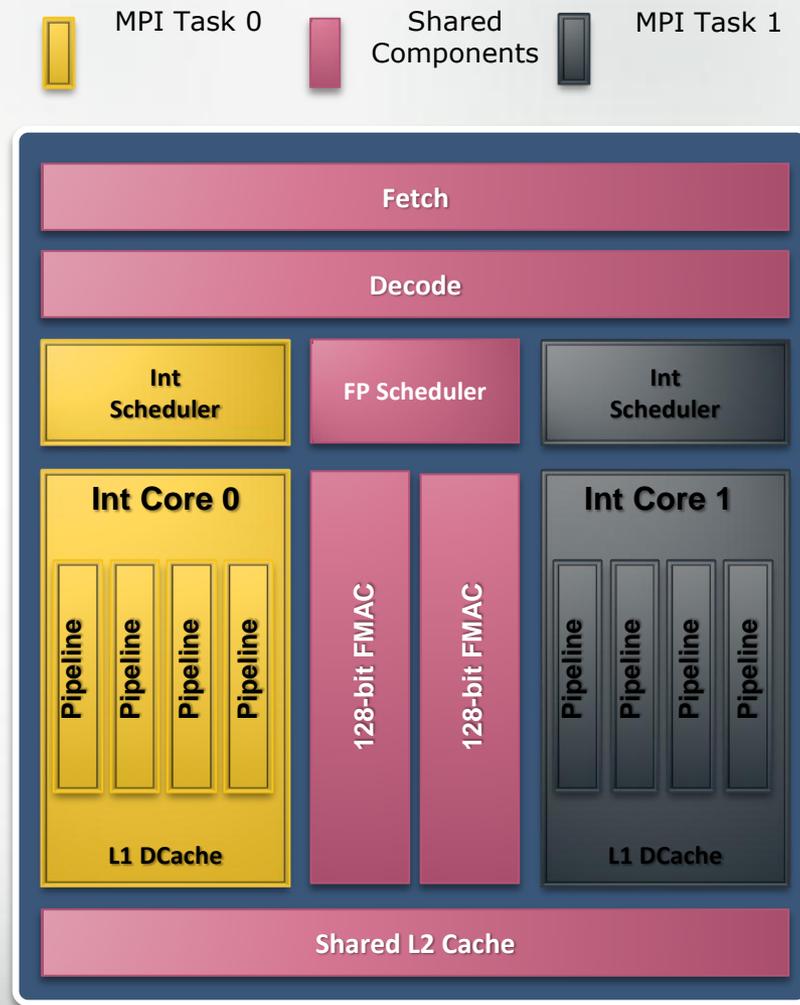


Interlagos Caches and Memory

- L1 Cache
 - 16 KB, 4-way predicted, parity protected
 - Write-through and inclusive with respect to L2
 - 4 cycle load to use latency
- L2 Cache
 - 2MB, Shared within core-module
 - 18-20 cycle load to use latency
- L3 Cache
 - 8 MB, non-inclusive victim cache (mostly exclusive)
 - Entries used by multiple core-modules will remain in cache
 - 1 to 2 MB used by probe filter (snoop bus)
 - 4 sub-caches, one close to each compute module
 - Minimum Load to latency of 55-60 cycles
- Minimum latency to memory is 90-100 cycles

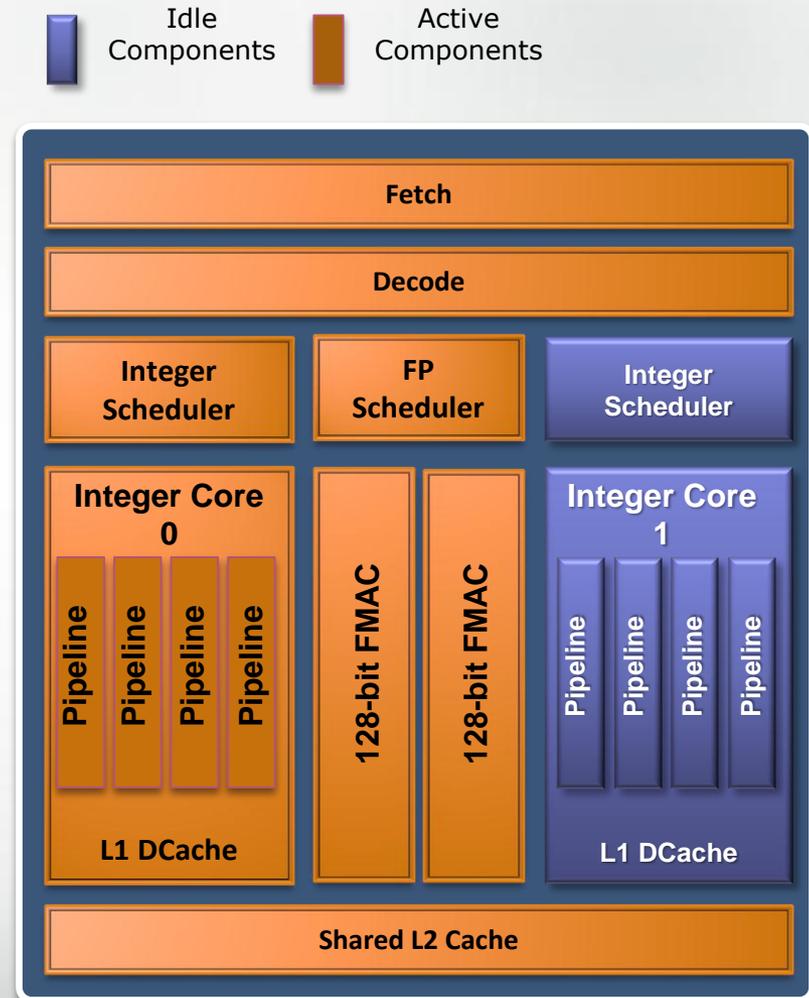
Two MPI Tasks on a Compute Unit ("Dual-Stream Mode")

- An MPI task is pinned to each integer unit
 - Each integer unit has exclusive access to an integer scheduler, integer pipelines and L1 Dcache
 - The 256-bit FP unit, instruction fetch, and the L2 Cache are shared between the two integer units
 - 256-bit AVX instructions are dynamically executed as two 128-bit instructions if the 2nd FP unit is busy
- When to use
 - Code is highly scalable to a large number of MPI ranks
 - Code can run with a 2GB per task memory footprint
 - Code is not well vectorized



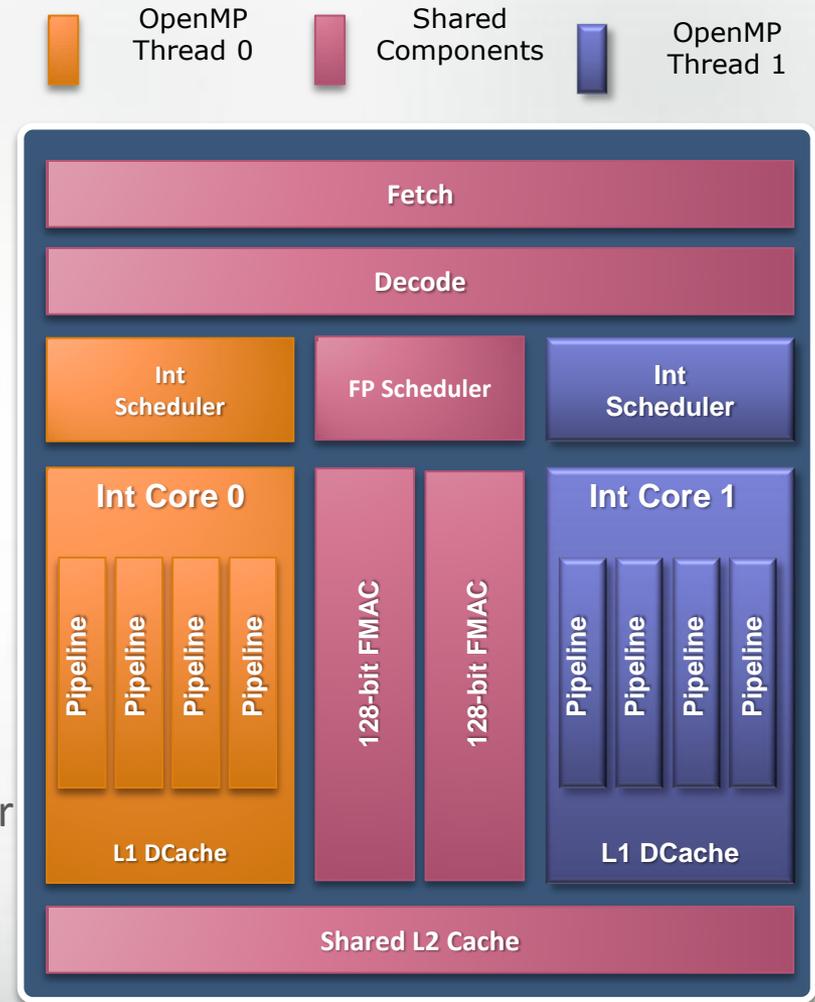
One MPI Task on a Compute Unit ("Single Stream Mode")

- Only one integer unit is used per compute unit
 - This unit has exclusive access to the 256-bit FP unit and is capable of 8 FP results per clock cycle
 - The unit has twice the memory capacity and memory bandwidth in this mode
 - The L2 cache is effectively twice as large
 - The peak of the chip is not reduced
- When to use
 - Code is highly vectorized and makes use of AVX instructions
 - Code benefits from higher per task memory size and bandwidth



One MPI Task per compute unit with Two OpenMP Threads ("Dual-Stream Mode")

- An MPI task is pinned to a compute unit
- OpenMP is used to run a thread on each integer unit
 - Each OpenMP thread has exclusive access to an integer scheduler, integer pipelines and L1 Dcache
 - The 256-bit FP unit and the L2 Cache is shared between the two threads
 - 256-bit AVX instructions are dynamically executed as two 128-bit instructions if the 2nd FP unit is busy
- When to use
 - Code needs a large amount of memory per MPI rank
 - Code has OpenMP parallelism at each MPI rank



AVX (Advanced Vector Extensions)

- Max Vector length doubled to 256 bit
- Much cleaner instruction set
 - Result register is unique from the source registers
 - Old SSE instruction set always destroyed a source register
- Floating point multiple-accumulate
 - $A(1:4) = B(1:4) * C(1:4) + D(1:4)$! Now one instruction
- Next gen of both AMD and Intel will have AVX
- Vectors are becoming more important, not less

Running in Dual-Stream mode

- Dual-Stream mode is the current default mode on the Cray XE6 systems. General use does not require any options. CPU affinity is set automatically by ALPS.
- Use the `aprun -d` option to set the number of OpenMP threads per process. If OpenMP is not used, no `-d` option is required. The `aprun -N` option is used to specify the number of MPI processes to assign per compute node. This is generally needed if OpenMP threads are used and more than one node is used.

Running in Single-Stream mode

- Single-Stream mode is simple to specify on the Cray XE6 systems if no OpenMP threads are used. The `aprun -d` option is set to a value of 2, and CPU affinity is set automatically by ALPS. (Make sure `$OMP_NUM_THREADS` is not set, or is set to a value of 1.)
- When OpenMP threads are used, careful setting of the `aprun -cc cpu_list` option is required to get the desired CPU affinity. A `cpu_list` is map of CPUs to threads. Each process is assigned a list of CPUs, with one CPU per thread. See the `aprun(1)` man page for more details. The `aprun -N` option is used to specify the number of MPI processes to assign per compute node. This is generally needed if more than one node is used in Single-Stream mode. Also, the environment variable `$OMP_NUM_THREADS` needs to be set to the correct number of threads per process.

aprun Examples

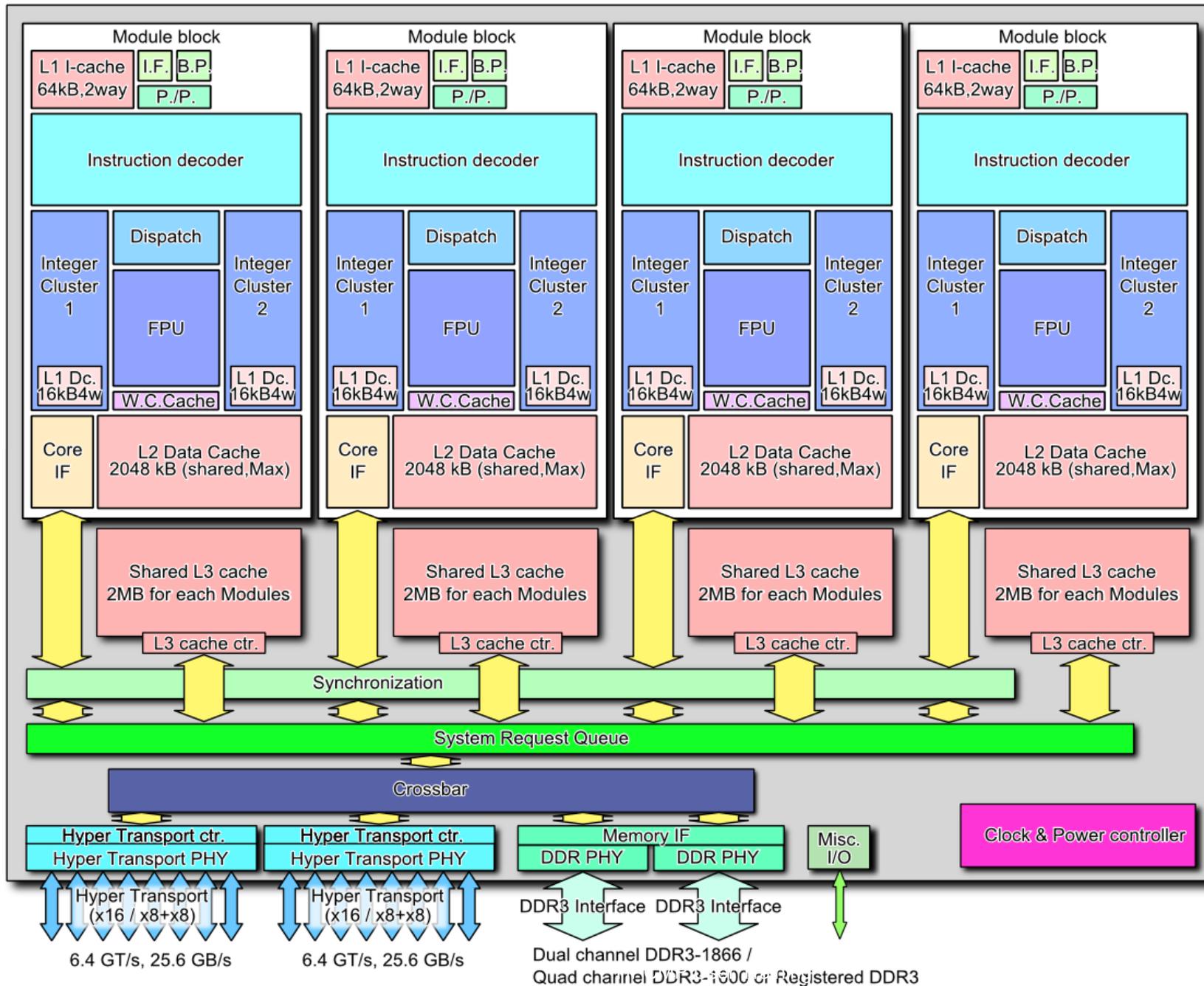
- No OpenMP or 1 OpenMP thread per process, 16 processes per compute node
`-d 2`
- 2 OpenMP threads per MPI process, 8 processes per compute node
`-N 8 -cc`
`0, 2:4, 6:8, 10:12, 14:16, 18:20, 22:24, 26:28, 30`
- 4 OpenMP threads per MPI process, 4 processes per compute node
`-N 4 -cc`
`0, 2, 4, 6:8, 10, 12, 14:16, 18, 20, 22:24, 26, 28, 30`
- 8 OpenMP threads per MPI process, 2 processes per compute node
`-N 2 -cc`
`0, 2, 4, 6, 8, 10, 12, 14:16, 18, 20, 22, 24, 26, 28, 30`
- 16 OpenMP threads per MPI process, 1 process per compute node
`-N 1 -cc`
`0, 2, 4, 6, 8, 10, 12, 14, 16, 18, 20, 22, 24, 26, 28, 30`

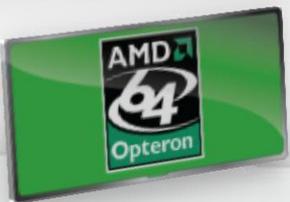
NUMA Considerations

- An XE6 compute node with 2 Interlagos processors has 4 NUMA memory domains, each with 4 Bulldozer Modules. Access to memory located in a remote NUMA domain is slower than access to local memory. Bandwidth is lower, and latency is higher.
- OpenMP performance is usually better when all threads in a process execute in the same NUMA domain. For the Dual-Stream case, 8 CPUs share a NUMA domain, while in Single-Stream mode 4 CPUs share a NUMA domain. Using a larger number of OpenMP threads per MPI process than these values may result in lower performance due to cross-domain memory access.

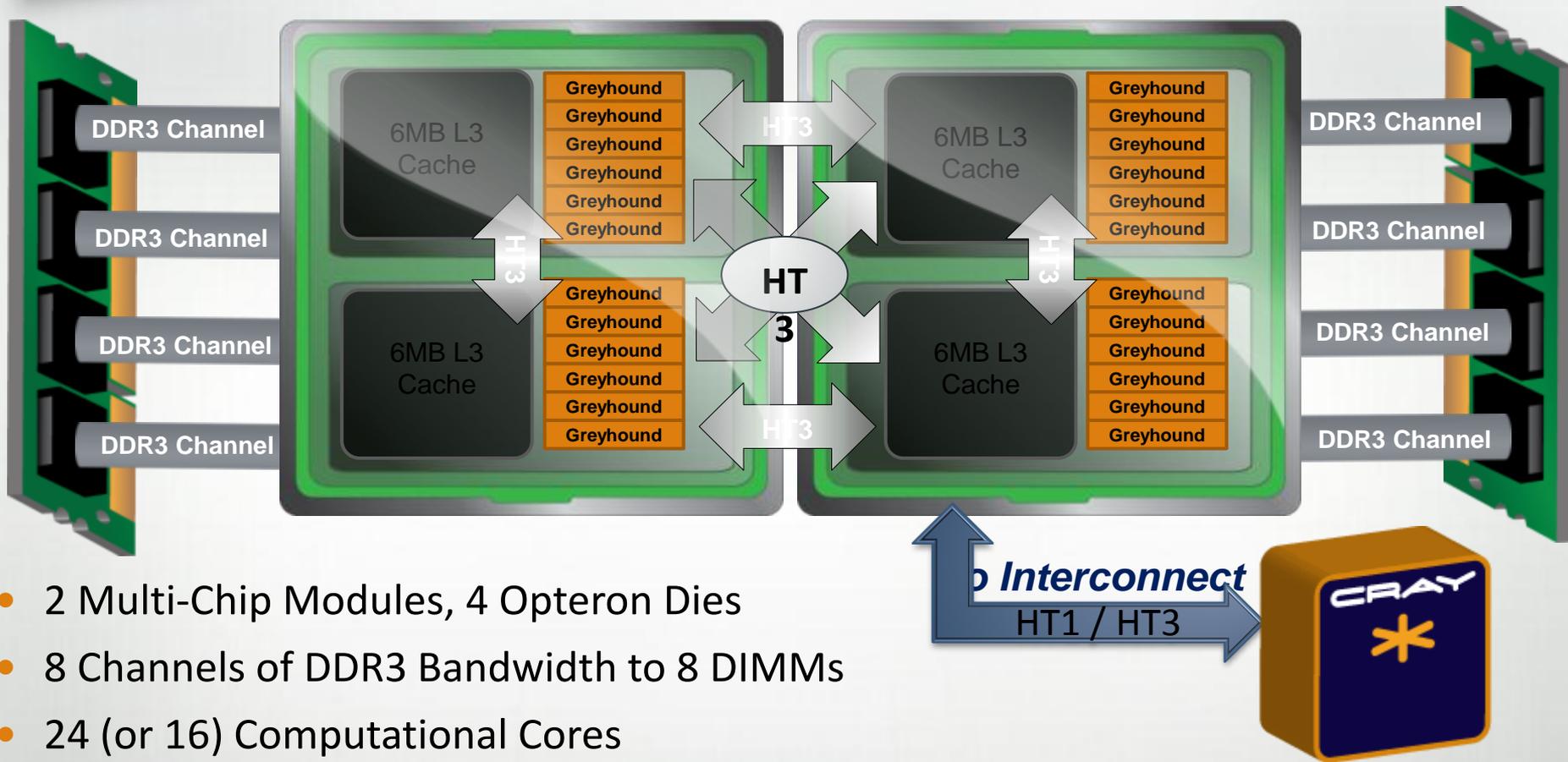
aprun Options Summary

Run Type	Dual-Stream	Single-Stream
No OpenMP	No option needed	-d 2 (note: \$OMP_NUM_THREADS not set)
2 OpenMP threads	-N 16 -d 2	-N 8 -cc 0,2:4,6:8,10:12,14:16,18:20,22:24,26:28,30
4 OpenMP threads	-N 8 -d 4	-N 4 -cc 0,2,4,6:8,10,12,14:16,18,20,22:24,26,28,30
8 OpenMP threads	-N 4 -d 8	-N 2 -cc 0,2,4,6,8,10,12,14:16,18,20,22,24,26,28,30
16 OpenMP threads	-N 2 -d 16	-N 1 -cc 0,2,4,6,8,10,12,14,16,18,20,22,24,26,28,30
32 OpenMP threads	-N 1 -d 32	Not Applicable



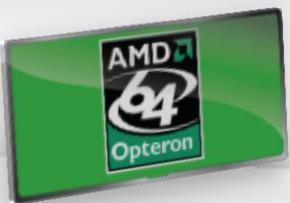


XE6 Node Details: 24-core Magny Cours



- 2 Multi-Chip Modules, 4 Opteron Dies
- 8 Channels of DDR3 Bandwidth to 8 DIMMs
- 24 (or 16) Computational Cores
 - 64 KB L1 and 512 KB L2 caches for each core
 - 6 MB of shared L3 cache on each die
- Dies are fully connected with HT3
- Snoop Filter Feature Allows 4 Die SMP to scale well

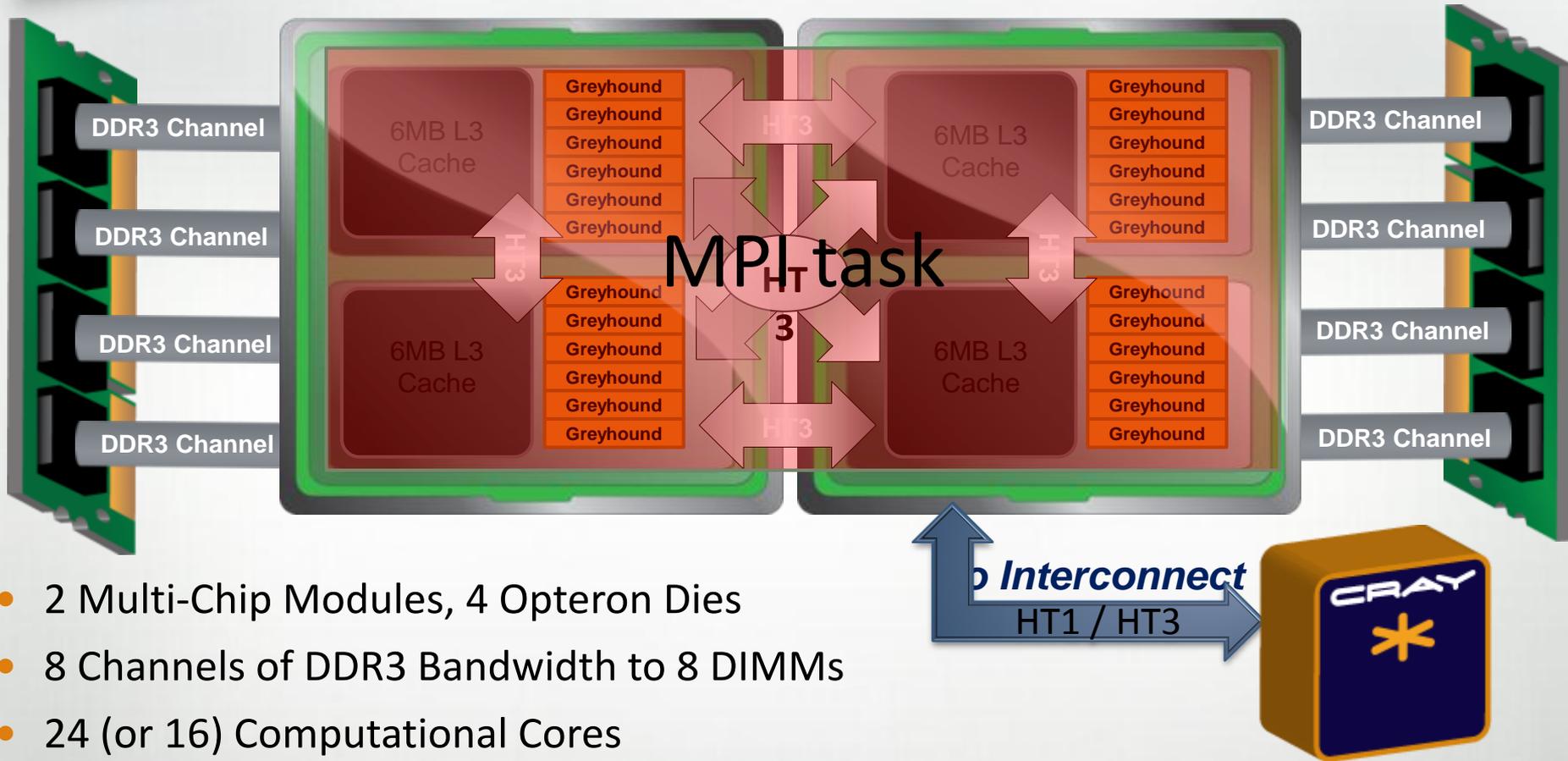




XE6 Node Details: 24-core Magny Cours

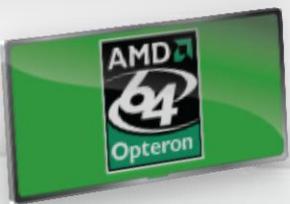


Run using 1 MPI task on the node



- 2 Multi-Chip Modules, 4 Opteron Dies
- 8 Channels of DDR3 Bandwidth to 8 DIMMs
- 24 (or 16) Computational Cores
 - 64 KB L1 and 512 KB L2 caches for each core
 - 6 MB of shared L3 cache on each die
- Dies are fully connected with HT3
- Snoop Filter Feature Allows 4 Die SMP

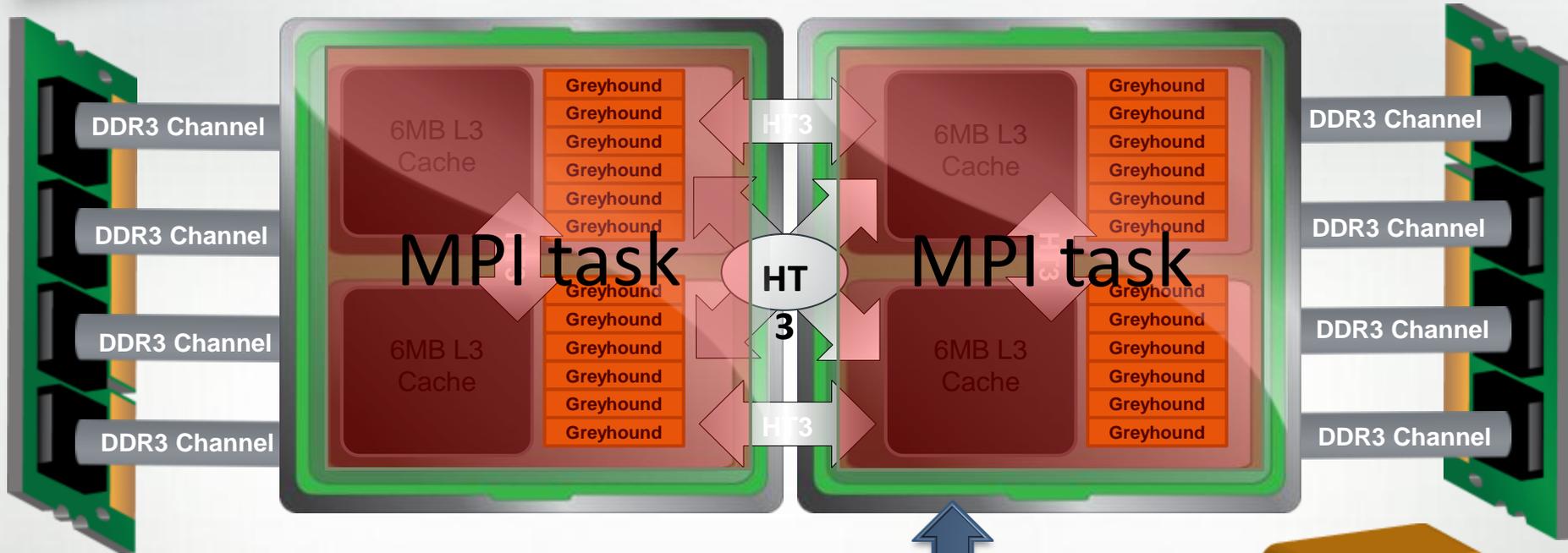
Use OpenMP across all 24 cores



XE6 Node Details: 24-core Magny Cours



Run using 2 MPI tasks on the node
One on Each Die



- 2 Multi-Chip Modules, 4 Opteron Dies
- 8 Channels of DDR3 Bandwidth to 8 DIMMs
- 24 (or 16) Computational Cores
 - 64 KB L1 and 512 KB L2 caches for each core
 - 6 MB of shared L3 cache on each die
- Dies are fully connected with HT3
- Snoop Filter Feature Allows 4 Die SMP to scale well

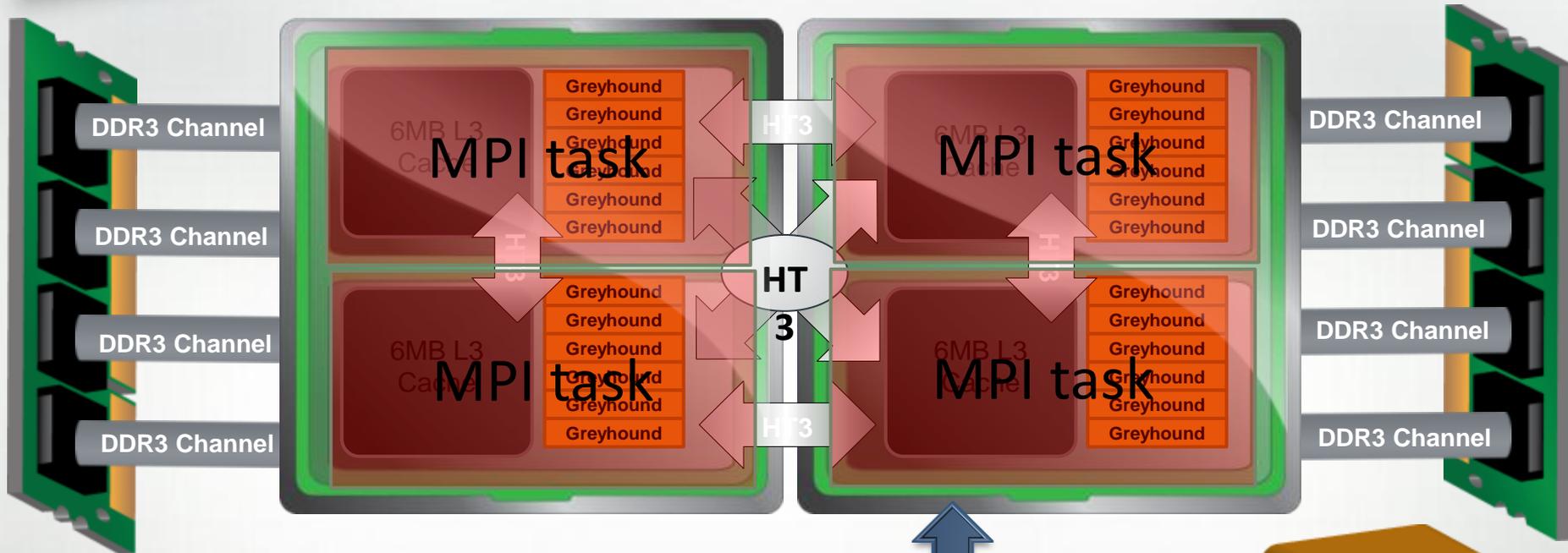
Use OpenMP across all 12 cores
in the Die



XE6 Node Details: 24-core Magny Cours



Run using 4 MPI tasks on the node
One on Each Socket



- 2 Multi-Chip Modules, 4 Opteron Dies
- 8 Channels of DDR3 Bandwidth to 8 DIMMs
- 24 (or 16) Computational Cores
 - 64 KB L1 and 512 KB L2 caches for each core
 - 6 MB of shared L3 cache on each die
- Dies are fully connected with HT3
- Snoop Filter Feature Allows 4 Die SMP to scale well



Use OpenMP across all 6 cores
in the Socket

Core Specialization

- Cray has put significant effort into reducing OS services and interruptions in the compute node OS
 - Improved Scalability
 - Increased time doing running user code
- Some services are still necessary in order to function, so still some interruptions remain, which can reduce scalability of codes with frequent synchronization
- Core specialization provides a way to dedicate a number of cores to handling the interrupts that must happen so the application will be interrupted less often.

Core Specialization (cont.)

Two types of interrupts:

- Directed at a particular core, such as from the network for data directed at a particular core.
 - Always handled by the specified core
- Handled by any available core, such as system/OS services
 - When run without core specialization, these interrupts will go to whatever core is most idle at that time (unpredictable)
 - Core specialization dedicates a number of cores to handling these interrupts in a predictable way.

Core Specialization

- Core Specialization is a feature that moves most systems services (noise sources) to a dedicated set of cores on each node
 - MPI ranks run on the remaining cores
- Noise-sensitive applications run faster on the same number of nodes with Core Specialization
 - Even though fewer cores are used overall to run MPI ranks
 - 20%-30% speedup measured for POP
- Core Specialization becomes more favorable moving forward
 - Cores are under-served with memory bandwidth
 - A single core represents a shrinking fraction of the compute capacity

Using Core Specialization

- Core specialization is an aprun option:
 - `aprun -j 1 ...` # dedicate 1 core to services (Interlagos single-stream mode)
 - `aprun -j 2 ...` # dedicate 2 cores to services (Interlagos dual-stream mode)

- Dual-stream mode is more likely to see performance improvements than single-stream mode

POP Results

Cores	ET	TOTAL	STEP	BAROCLINIC	BAROTROPIC
Core Spec: 7168	410	387	387	198	94
Standard: 8192	528	499	499	264	128
ratio	.77	.77	.77	.75	.73

`aprun -n 8192 POP`
`aprun -n 7168 -j 1 POP`

Indications of Noise Amplification

- Frequent synchronization
- Large amount of time spent in collectives
 - Especially if it increases faster than expected with increasing core count
- Unexplained scaling problems

Cray Debugging Support Tools or How to Debug Peta-scale Applications

Stack Trace Analysis Tool (STAT)
-- My application hangs! --

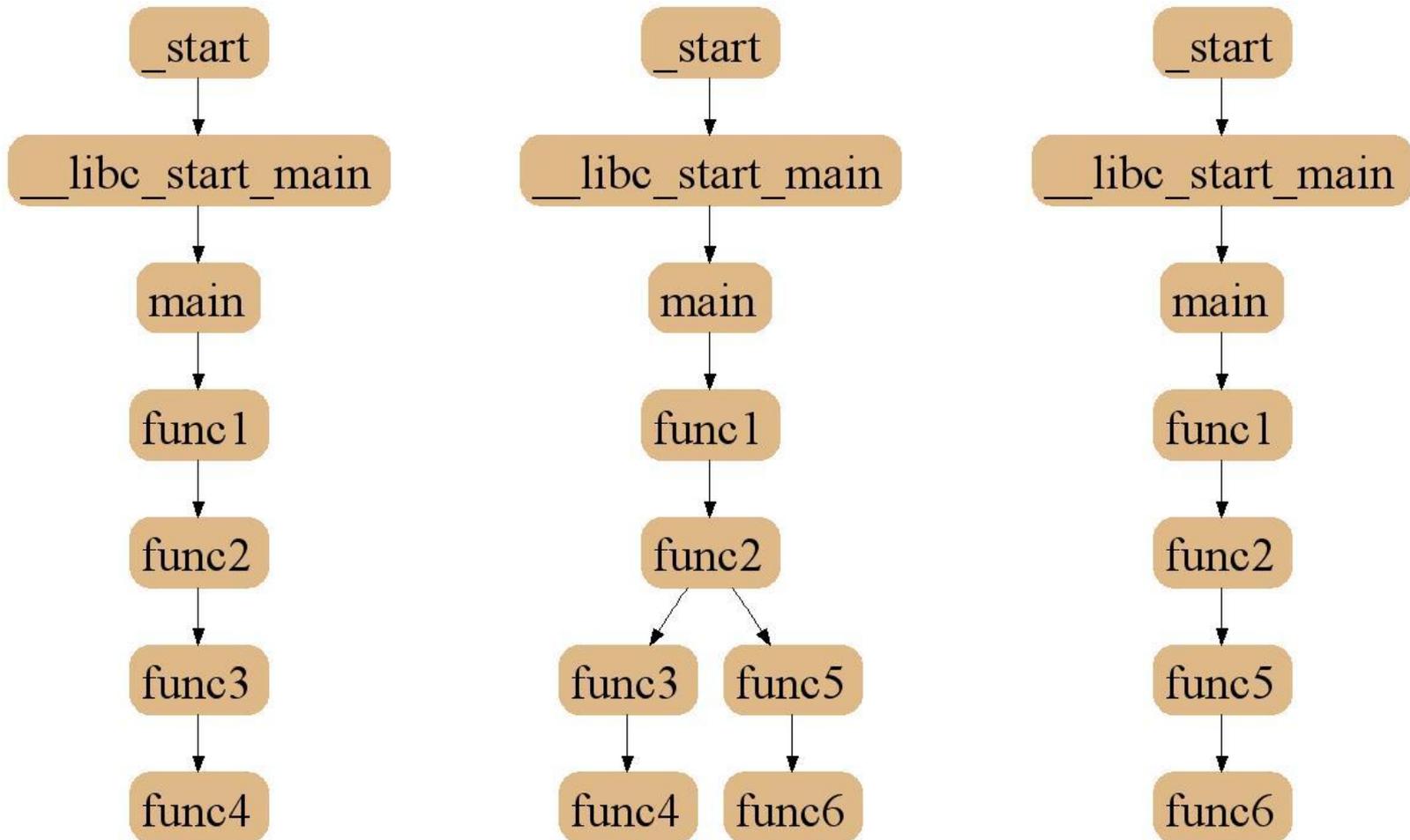
What is STAT?

- Stack trace sampling and analysis for large scale applications from Lawrence Livermore Labs and the University of Wisconsin
 - Creates a merged stack trace tree
 - Groups ranks with common behaviors
 - Fast: Collects traces for 100s of 1000s of cores in under a second
 - Compact: Stack trace tree only a few mega bytes
- Extreme scale
 - Jaguar: 200K cores
 - Hopper: 125K cores

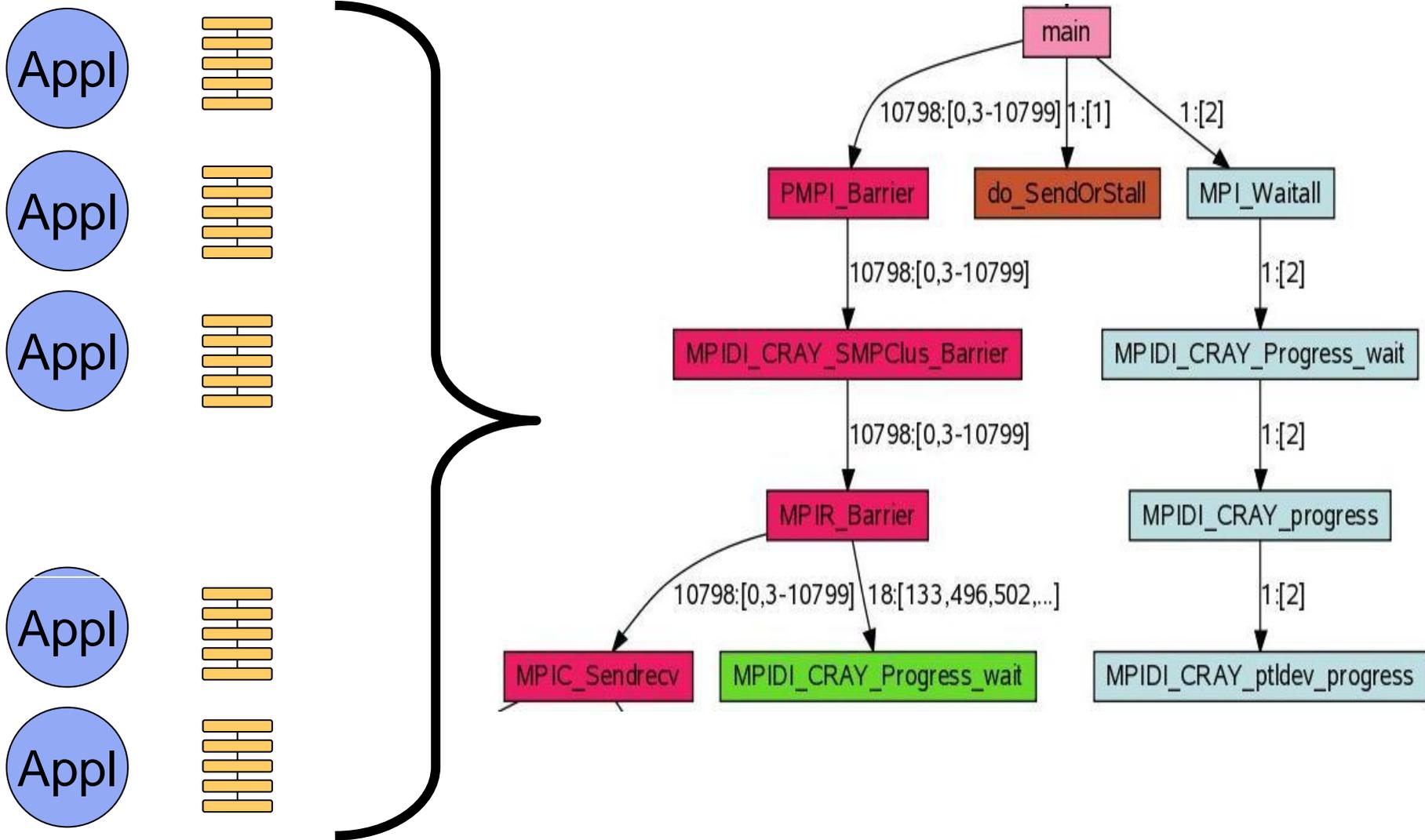
Merged stack trace trees

- Sampling across ranks
- Sampling across time
- Scalable visualization
 - Shows the big picture
 - Pin points subset for heavy weight debuggers

Stack Trace Merge Example



2D-Trace/Space Analysis



NERSC Plasma Physics Application

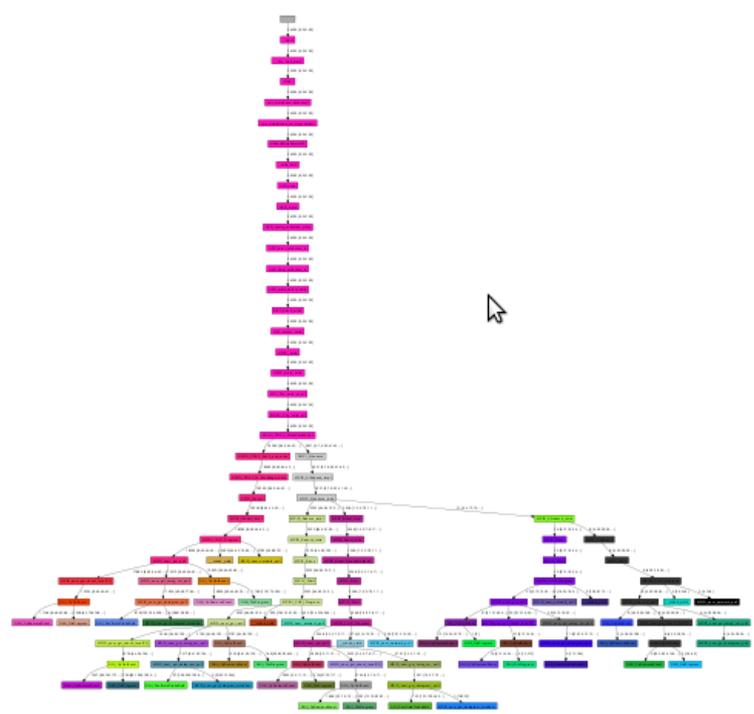
- Production, plasma physics PIC (Particle in Cell) code, run with 120K cores on hopper, and using HDF5 for parallel I/O
- Mixed MPI/OpenMP
- STAT helped them to see the big picture, as well as eliminate code possibilities since they were not in the tree

STATview (on kaibab)

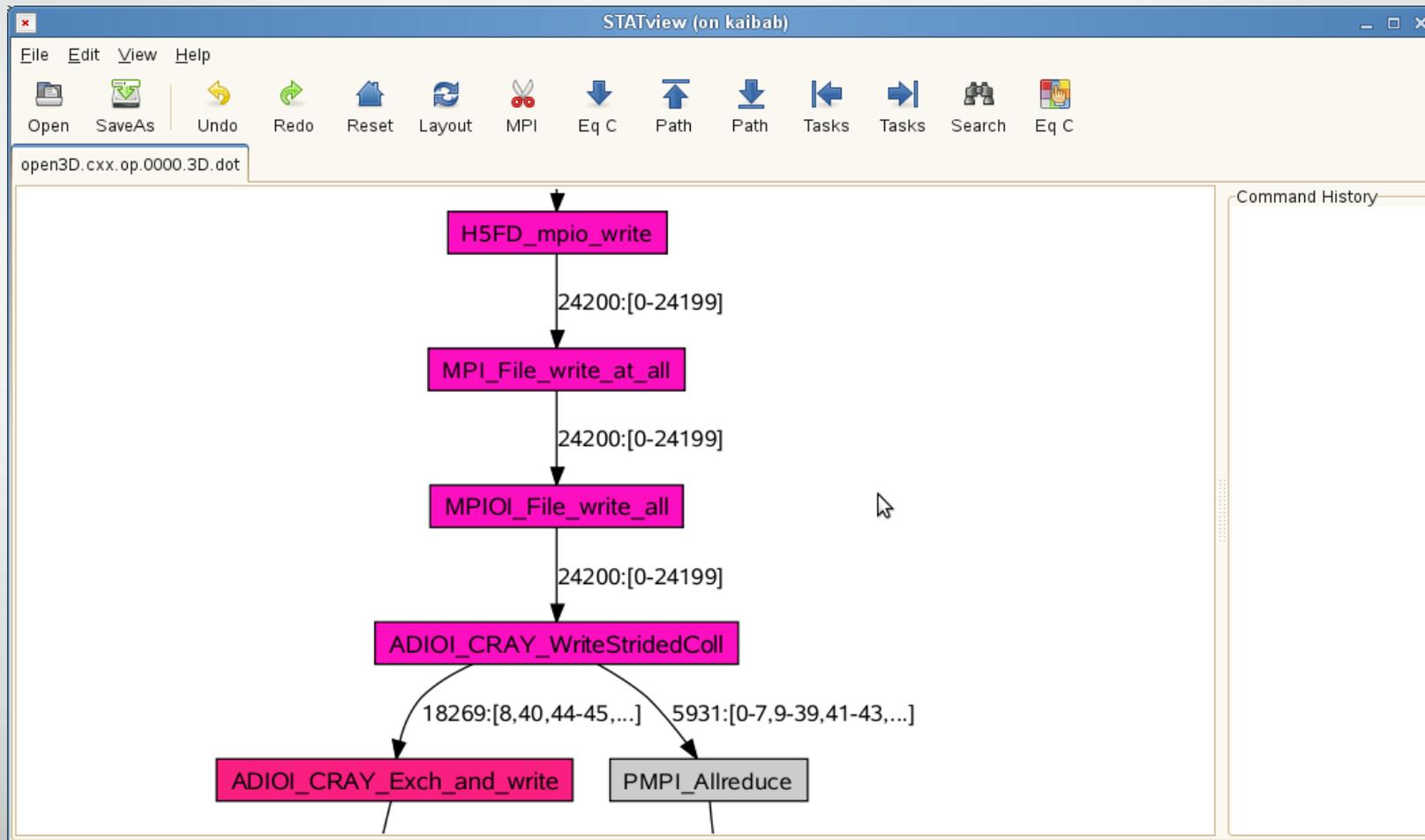
File Edit View Help

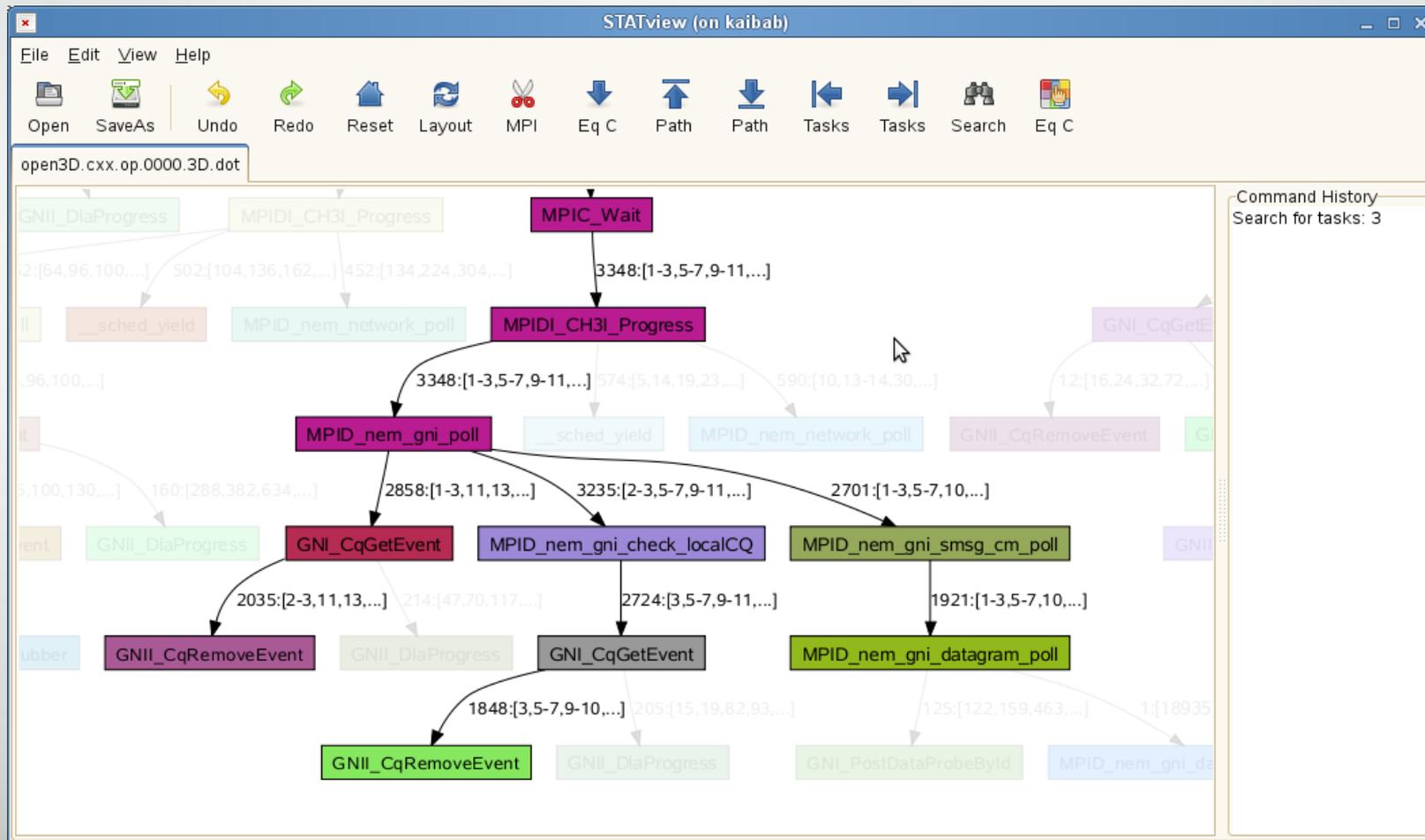
Open SaveAs Undo Redo Reset Layout MPI Eq C Path Path Tasks Tasks Search Eq C

open3D.cxx.op.0000.3D.dot



Command History





STAT: Since We Were Here Last Year

- Last year, STAT was a great tool that wasn't quite ported
- Now (since November 2011), STAT is a great tool that is fully ported.
- And, a new addition: **STATGUI**
 - Work bench for repeated requests
 - Change granularity
 - Change sampling
 - Continue then resample
 - Launches or attaches

STAT 1.2.1.1

- module load stat
- man STAT
- STAT <pid_of_aprun>
 - Creates
 STAT_results/<app_name>/<merged_st_file>
- statview <merged_st_file>
- STATGUI
- Scaling **no longer** limited by number file descriptors

Abnormal Termination Processing (ATP)

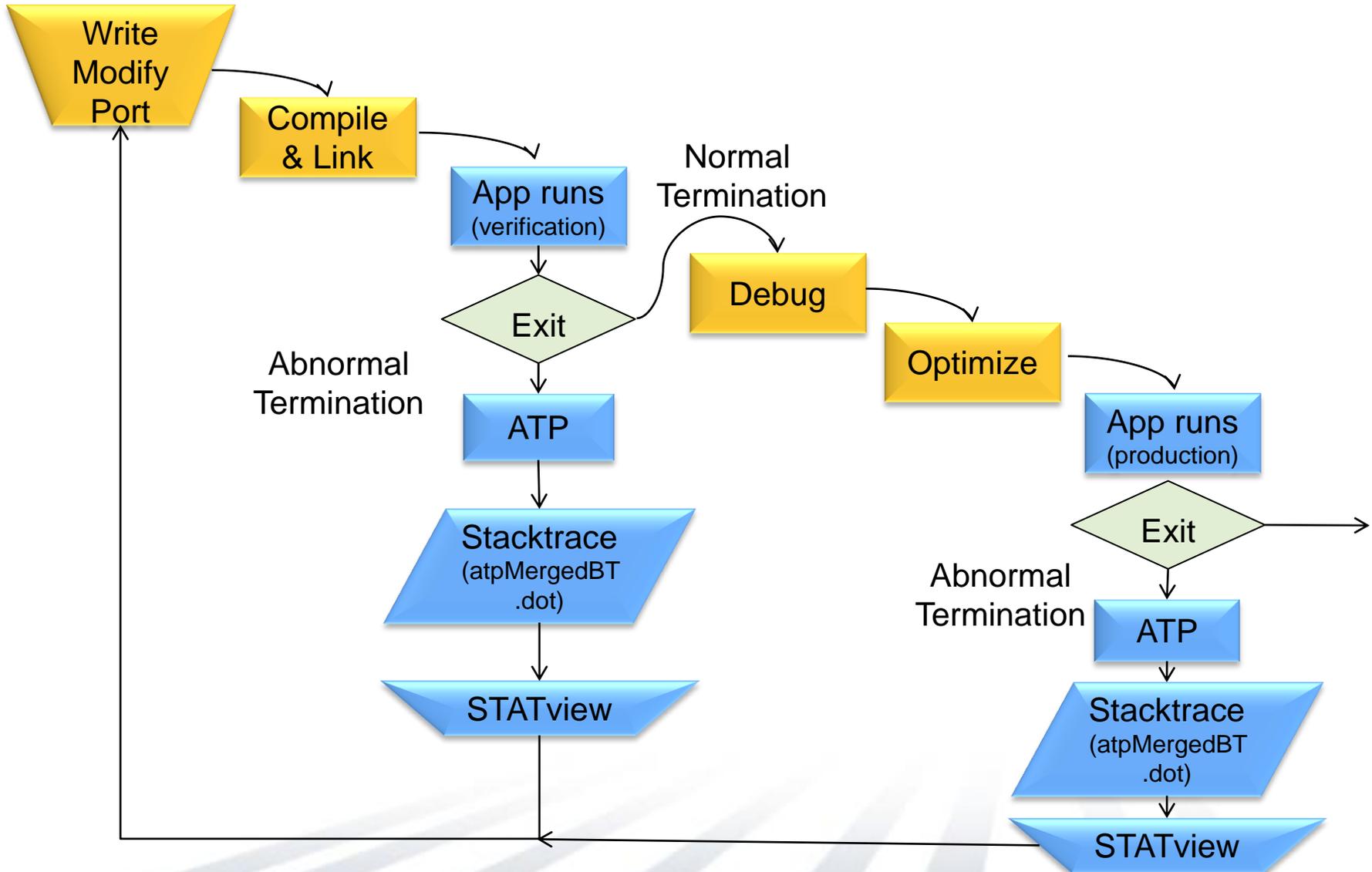
-- My application crashes! --

- Applications on Cray systems use hundreds of thousands of processes
- On a crash one, many, or all of them might trap
- No one wants that many core files
- No one wants that many stack backtraces
- They are too slow and too big.
- They are too much to comprehend

ATP Description

- System of light weight back-end monitor processes on compute nodes
- Coupled together as a tree with MRNet
- Automatically launched by aprun
- Leap into action on any application process trapping
- Stderr **backtrace** of first process to trap
- STAT like analysis provides **merged stack backtrace** tree
- Leaf nodes of tree define a modest set of processes to **core dump**
- Or, a set of processes to attach to with a debugger

ATP – Abnormal Termination Processing

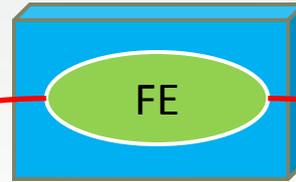


ATP Components

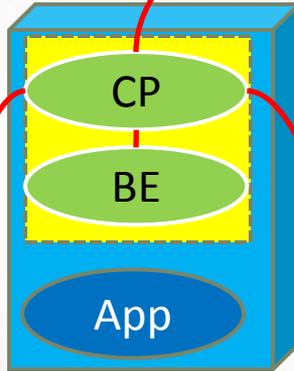
- Application process signal handler (**atpAppSigHandler**)
 - triggers analysis
- Back-end monitor (**atpBackend**)
 - collects backtraces via StackwalkerAPI
 - forces core dumps as directed using core_pattern
- Front-end controller (**atpFrontend**)
 - coordinates analysis via MRNet
 - selects process set that is to dump core
- Once initial set up complete, all components comatose

ATP Communications Tree

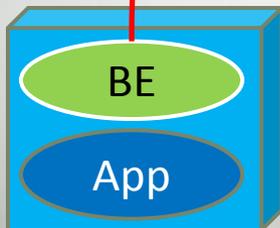
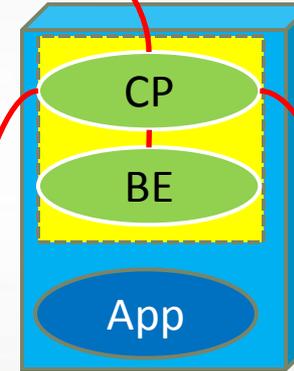
Front-end



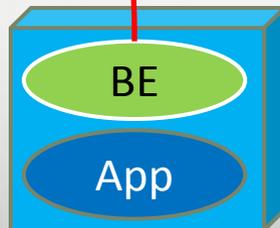
Back-end



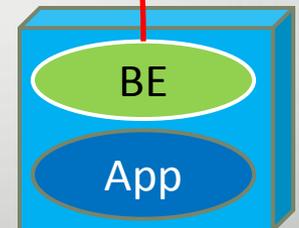
...



...



...



ATP Since We Were Here Last Year

- Added support for:
 - Dynamic Applications
 - Threaded Applications
 - Medium memory model compiles
 - Analysis on queuing system wall clock time out
- Eliminated use of LD_LIBRARY_PATH
- Numerous bug fixes.

Current Release: ATP 1.4.3

- Automatic
 - ATP module loaded by default
 - Signal handler added to application and registered
 - Aprun launches ATP in parallel with application launch
 - Run time enabled/disabled via `ATP_ENABLED` environment variable (can be set by site)
- Provides:
 - backtrace of first crash to stderr
 - merged backtrace trees
 - dumps core file set (if limit/ulimit allows)
- Tested at 15K PEs

What's Next for ATP?

- Support for Checkpoint/Restart
- Support higher scale
- Improved output file naming system
- E-mail on crash, if user requesting HOLD

Cray Performance Tools

- **Assist** the user with application performance analysis and optimization
 - Help user identify important and meaningful information from potentially massive data sets
 - Help user identify problem areas instead of just reporting data
 - Bring optimization knowledge to a wider set of users

- Focus on **ease** of use and **intuitive** user interfaces
 - Automatic program instrumentation
 - Automatic analysis

- Target **scalability** issues in all areas of tool development
 - Data management
 - Storage, movement, presentation

Provide a complete solution from instrumentation to measurement to analysis to visualization of data

- Performance measurement and analysis **on large systems**
 - Automatic Profiling Analysis
 - Load Imbalance
 - HW counter derived metrics
 - Predefined trace groups provide performance statistics for libraries called by program (blas, lapack, pgas runtime, netcdf, hdf5, etc.)
 - Observations of inefficient performance
 - Data collection and presentation filtering
 - Data correlates to user source (line number info, etc.)
 - Support MPI, SHMEM, OpenMP, UPC, CAF
 - Access to network counters
 - Minimal program perturbation

- Usability on large systems
 - Client / server
 - Scalable data format
 - Intuitive visualization of performance data

- Supports “recipe” for porting MPI programs to many-core or hybrid systems

- Integrates with other Cray PE software for more tightly coupled development environment

- Supports traditional post-mortem performance analysis
 - Automatic identification of performance problems
 - Indication of causes of problems
 - Suggestions of modifications for performance improvement
 - `pat_build`: provides automatic instrumentation
 - `CrayPat` run-time library collects measurements (transparent to the user)
 - `pat_report` performs analysis and generates text reports
 - `pat_help`: online help utility
 - `Cray Apprentice2`: graphical visualization tool

- CrayPat
 - Instrumentation of optimized code
 - No source code modification required
 - Data collection transparent to the user
 - Text-based performance reports
 - Derived metrics
 - Performance analysis

- Cray Apprentice2
 - Performance data visualization tool
 - Call tree view
 - Source code mappings

Steps to Using the Tools

- pat_build is a stand-alone utility that automatically instruments the application for performance collection

- Requires **no** source code or makefile **modification**
 - Automatic instrumentation at group (function) level
 - Groups: mpi, io, heap, math SW, ...

- Performs link-time instrumentation
 - **Requires object files**
 - Instruments optimized code
 - Generates stand-alone instrumented program
 - Preserves original binary

- Supports two categories of experiments
 - asynchronous experiments (**sampling**) which capture values from the call stack or the program counter at specified intervals or when a specified counter overflows
 - Event-based experiments (**tracing**) which count some events such as the number of times a specific system call is executed
- While tracing provides most useful information, it can be very heavy if the application runs on a large number of cores for a long period of time
- Sampling can be useful as a starting point, to provide a first overview of the work distribution

- Large programs
 - Scaling issues more dominant
 - Use automatic profiling analysis to quickly identify top time consuming routines
 - Use loop statistics to quickly identify top time consuming loops

- Small (test) or short running programs
 - Scaling issues not significant
 - Can skip first sampling experiment and directly generate profile
 - For example: `% pat_build -u -g mpi my_program`

Where to Run Instrumented Application

- **MUST run on Lustre** (/mnt/snx3/... , /lus/..., /scratch/...,etc.)
- Number of files used to store raw data
 - 1 file created for program with 1 – 256 processes
 - \sqrt{n} files created for program with 257 – n processes
 - Ability to customize with `PAT_RT_EXPFIL_MAX`

- Runtime controlled through PAT_RT_XXX environment variables
- See [intro_craypat\(1\)](#) man page
- Examples of control
 - Enable full trace
 - Change number of data files created
 - Enable collection of HW counters
 - Enable collection of network counters
 - Enable tracing filters to control trace file size (max threads, max call stack depth, etc.)

- Optional timeline view of program available
 - export `PAT_RT_SUMMARY=0`
 - View trace file with Cray Apprentice²

- Number of files used to store raw data:
 - 1 file created for program with 1 – 256 processes
 - \sqrt{n} files created for program with 257 – n processes
 - Ability to customize with `PAT_RT_EXPFIL_MAX`

- Request hardware performance counter information:
 - export `PAT_RT_HWPC=<HWPC Group>`
 - Can specify events or predefined groups

- Performs data conversion
 - Combines information from binary with raw performance data

- Performs analysis on data

- Generates text report of performance results

- Formats data for input into Cray Apprentice²

Why Should I generate an “.ap2” file?

- The “.ap2” file is a self contained compressed performance file
- Normally it is about 5 times smaller than the “.xf” file
- Contains the information needed from the application binary
 - Can be reused, even if the application binary is no longer available or if it was rebuilt
- It is the only input format accepted by Cray Apprentice²

Files Generated and the Naming Convention

File Suffix	Description
a.out+pat	Program instrumented for data collection
a.out...s.xf	Raw data for sampling experiment, available after application execution
a.out...t.xf	Raw data for trace (summarized or full) experiment, available after application execution
a.out...st.ap2	Processed data, generated by pat_report, contains application symbol information
a.out...s.apa	Automatic profiling analysis template, generated by pat_report (based on pat_build -O apa experiment)
a.out+apa	Program instrumented using .apa file
MPICH_RANK_ORDER.Custom	Rank reorder file generated by pat_report from automatic grid detection and reorder suggestions

- Automatic profiling analysis (APA)
 - Provides simple procedure to instrument and collect performance data for novice users
 - Identifies top time consuming routines
 - Automatically creates instrumentation template customized to application for future in-depth measurement and analysis

Steps to Collecting Performance Data

- Access performance tools software

```
% module load perftools
```

- Build application keeping .o files (CCE: -h keepfiles)

```
% make clean  
% make
```

- Instrument application for automatic profiling analysis

- You should get an instrumented program a.out+pat

```
% pat_build -O apa a.out
```

- Run application to get top time consuming routines

- You should get a performance file (“<sdatafile>.xf”) or multiple files in a directory <sdatadir>

```
% aprun ... a.out+pat (or qsub <pat script>)
```

- Generate report and .apa instrumentation file

```
% pat_report -o my_sampling_report [<sdatafile>.xf |  
  <sdatadir>]
```

- Inspect .apa file and sampling report
- Verify if additional instrumentation is needed

APA File Example

```
# You can edit this file, if desired, and use it
# to reinstrument the program for tracing like this:
#
# pat_build -O standard.cray-xt.PE-2.1.56HD.pgi-8.0.amd64.pat-5.0.0.2-
# Oapa.512.quad.cores.seal.090405.1154.mpi.pat_rt_exp=default.pat_rt_hwpc=non
# e.14999.xf.xf.apa
#
# These suggested trace options are based on data from:
#
# /home/users/malice/pat/Runs/Runs.seal.pat5001.2009Apr04/./pat.quad/homme/st
# andard.cray-xt.PE-2.1.56HD.pgi-8.0.amd64.pat-5.0.0.2-
# Oapa.512.quad.cores.seal.090405.1154.mpi.pat_rt_exp=default.pat_rt_hwpc=non
# e.14999.xf.xf.cdb
# -----
# HWPC group to collect by default.
#
# -Drtenv=PAT_RT_HWPC=1 # Summary with TLB metrics.
# -----
# Libraries to trace.
#
# -g mpi
# -----
# User-defined functions to trace, sorted by % of samples.
#
# The way these functions are filtered can be controlled with
# pat_report options (values used for this file are shown):
#
# -s apa_max_count=200 No more than 200 functions are listed.
# -s apa_min_size=800 Commented out if text size < 800 bytes.
# -s apa_min_pct=1 Commented out if it had < 1% of samples.
# -s apa_max_cum_pct=90 Commented out after cumulative 90%.
#
# Local functions are listed for completeness, but cannot be traced.
#
# -w # Enable tracing of user-defined functions.
# Note: -u should NOT be specified as an additional option.
```

```
# 31.29% 38517 bytes
# -T prim_advance_mod_preq_advance_exp_
#
# 15.07% 14158 bytes
# -T prim_si_mod_prim_diffusion_
#
# 9.76% 5474 bytes
# -T derivative_mod_gradient_str_nonstag_
#
# ...
# 2.95% 3067 bytes
# -T forcing_mod_apply_forcing_
#
# 2.93% 118585 bytes
# -T column_model_mod_applycolumnmodel_
#
# Functions below this point account for less than 10% of samples.
#
# 0.66% 4575 bytes
# -T bndry_mod_bndry_exchangev_thsave_time_
#
# 0.10% 46797 bytes
# -T baroclinic_inst_mod_binst_init_state_
#
# 0.04% 62214 bytes
# -T prim_state_mod_prim_printstate_
#
# ...
# 0.00% 118 bytes
# -T time_mod_timelevel_update_
#
# -----
# -o preqx.cray-xt.PE-2.1.56HD.pgi-8.0.amd64.pat-5.0.0.2.x+apa
# # New instrumented program.
#
# /AUTO/cray/css.pe_tools/malice/craypat/build/pat/2009Apr03/2.1.56HD/amd64/ho
# mme/pgi/pat-5.0.0.2/homme/2005Dec08/build.Linux/preqx.cray-xt.PE-
# 2.1.56HD.pgi-8.0.amd64.pat-5.0.0.2.x # Original program.
```

- Instrument application for further analysis (*a.out+apa*)

```
% pat_build -O <apafilename>.apa
```

- Run application

```
% aprun ... a.out+apa (or qsub <apa script>)
```

- Generate text report and visualization file (.ap2)

```
% pat_report -o my_text_report.txt [<datafile>.xf |  
  <datadir>]
```

- View report in text and/or with Cray Apprentice²

```
% app2 <datafile>.ap2
```

HW Performance Counters

- AMD Family 10H Opteron Hardware Performance Counters
 - Each core has 4 48-bit performance counters
 - Each counter can monitor a single event
 - Count specific processor events
 - » the processor increments the counter when it detects an occurrence of the event
 - » (e.g., cache misses)
 - Duration of events
 - » the processor counts the number of processor clocks it takes to complete an event
 - » (e.g., the number of clocks it takes to return data from memory after a cache miss)
 - Time Stamp Counters (TSC)
 - Cycles (user time)

- AMD Family 15H Opteron Hardware Performance Counters
 - Each node has **4** 48-bit NorthBridge counters
 - Each core has **6** 48-bit performance counters
 - Not all events can be counted on all counters
 - Supports multi-events
 - events have a maximum count per clock that exceeds one event per clock

- Common set of events deemed relevant and useful for application performance tuning
 - Accesses to the memory hierarchy, cycle and instruction counts, functional units, pipeline status, etc.
 - The “papi_avail” utility shows which predefined events are available on the system – execute on compute node

- PAPI also provides access to native events
 - The “papi_native_avail” utility lists all AMD native events available on the system – execute on compute node

- PAPI uses perf_events Linux subsystem

- Information on PAPI and AMD native events
 - pat_help counters
 - man intro_papi (points to PAPI documentation: <http://icl.cs.utk.edu/papi/>)
 - <http://lists.eecs.utk.edu/pipermail/perfapi-devel/2011-January/004078.html>

- HW counter collection enabled with PAT_RT_HWPC environment variable

- PAT_RT_HWPC <set number> | <event list>
 - A set number can be used to select a group of predefined hardware counters events (recommended)
 - CrayPat provides 23 groups on the Cray XT/XE systems
 - See [pat_help\(1\)](#) or the [hwpc\(5\)](#) man page for a list of groups

 - Alternatively a list of hardware performance counter event names can be used

 - Hardware counter events are not collected by default

- Raw data
- Derived metrics
- Desirable thresholds

Predefined Interlagos HW Counter Groups

See `pat_help -> counters -> amd_fam15h -> groups`

0: Summary with instructions metrics

1: Summary with TLB metrics

2: L1 and L2 Metrics

3: Bandwidth information

4: <Unused>

5: Floating operations dispatched

6: Cycles stalled, resources idle

7: Cycles stalled, resources full

8: Instructions and branches

9: Instruction cache

10: Cache Hierarchy (unsupported for IL)

11: Floating point operations dispatched

12: Dual pipe floating point operations dispatched

13: Floating point operations SP

14: Floating point operations DP

L3 (socket and core level) (unsupported)

19: Prefetchs

20: FP, D1, TLB, MIPS <<-new for Interlagos

21: FP, D1, TLB, Stalls

22: D1, TLB, MemBW

- Group 20: FP, D1, TLB, MIPS

 - PAPI_FP_OPS

 - PAPI_L1_DCA

 - PAPI_L1_DCM

 - PAPI_TLB_DM

 - DATA_CACHE_REFILLS_FROM_NORTHBRIDGE

 - PAPI_TOT_INS

- Group 21: FP, D1, TLB, Stalls

 - PAPI_FP_OPS

 - PAPI_L1_DCA

 - PAPI_L1_DCM

 - PAPI_TLB_DM

 - DATA_CACHE_REFILLS_FROM_NORTHBRIDGE

 - PAPI_RES_STL

Check spelling via `papi_native_avail`

PAPI_DP_OPS

■ AMD Family 10H:

- `RETIRED_SSE_OPERATIONS:DOUBLE_ADD_SUB_OPS:DOUBLE_MUL_OPS:DOUBLE_DIV_OPS`

■ AMD Family 15H:

- `RETIRED_SSE_OPS:DOUBLE_ADD_SUB_OPS:DOUBLE_MUL_OPS:DOUBLE_DIV_OPS:DOUBLE_MUL_ADD_OPS`

Example: HW counter data and Derived Metrics

```
PAPI_TLB_DM  Data translation lookaside buffer misses
PAPI_L1_DCA  Level 1 data cache accesses
PAPI_FP_OPS  Floating point operations
DC_MISS      Data Cache Miss
User_Cycles  Virtual Cycles
```

=====

USER

```
Time%                98.3%
Time                 4.434402 secs
Imb.Time             -- secs
Imb.Time%            --
Calls                0.001M/sec    4500.0 calls
PAPI_L1_DCM          14.820M/sec    65712197 misses
PAPI_TLB_DM          0.902M/sec    3998928 misses
PAPI_L1_DCA          333.331M/sec  1477996162 refs
PAPI_FP_OPS          445.571M/sec  1975672594 ops
User time (approx)   4.434 secs    11971868993 cycles  100.0%Time
Average Time per Call
CrayPat Overhead : Time      0.1%
HW FP Ops / User time    445.571M/sec    1975672594 ops    4.1%peak (DP)
HW FP Ops / WCT          445.533M/sec
Computational intensity    0.17 ops/cycle    1.34 ops/ref
MFLOPS (aggregate)        1782.28M/sec
TLB utilization           369.60 refs/miss    0.722 avg uses
D1 cache hit,miss ratios   95.6% hits          4.4% misses
D1 cache utilization (misses) 22.49 refs/miss    2.811 avg hits
```

PAT_RT_HWPC=1
Flat profile data
Raw counts
Derived metrics

Profile Visualization with `pat_report` and Cray Apprentice2

Examples of Recent Scaling Efforts

New .ap2 Format + Client/Server Model

- Reduced pat_report processing and report generation times
- Reduced app2 data load times
- Graphical presentation handled locally (not passed through ssh connection)
- Better tool responsiveness
- Minimizes data loaded into memory at any given time
- Reduced server footprint on Cray XT/XE service node
- Larger data files handled (1.5TB .xf -> 800GB .ap2)

■ CPMD

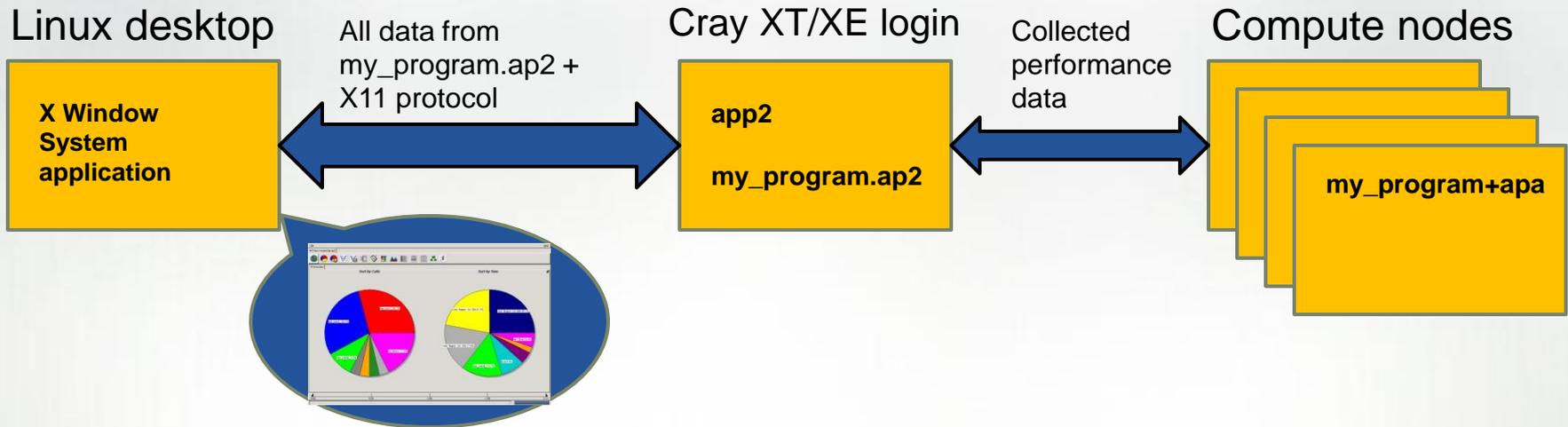
- MPI, instrumented with `pat_build -u`, HWPC=1
- 960 cores

	Perftools 5.1.3	Perftools 5.2.0
<code>.xf -> .ap2</code>	88.5 seconds	22.9 seconds
<code>ap2 -> report</code>	1512.27 seconds	49.6 seconds

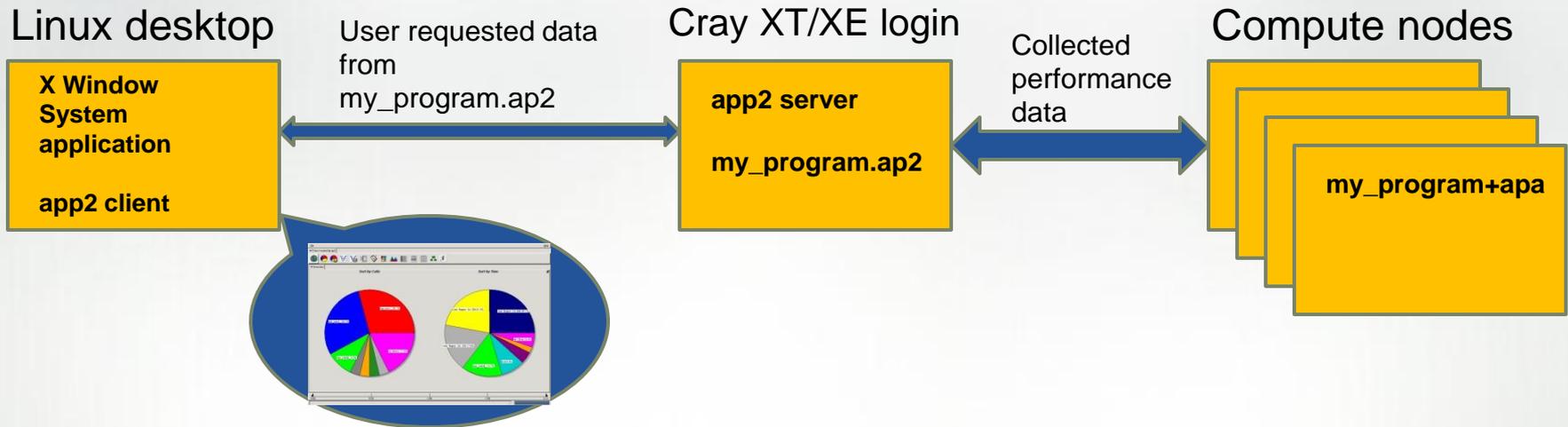
■ VASP

- MPI, instrumented with `pat_build -gmpi -u`, HWPC=3
- 768 cores

	Perftools 5.1.3	Perftools 5.2.0
<code>.xf -> .ap2</code>	45.2 seconds	15.9 seconds
<code>ap2 -> report</code>	796.9 seconds	28.0 seconds



- Log into Cray XT/XE login node
% `ssh -Y kaibab`
- Launch Cray Apprentice2 on Cray XT/XE login node
% `app2 /lus/scratch/mydir/my_program.ap2`
 - User interface displayed on desktop via ssh X11 forwarding
 - Entire .ap2 file loaded into memory on login node (can be Gbytes of data)



- Launch Cray Apprentice2 on desktop, point to data
 - % `app2 kaibab:/lus/scratch/mydir/my_program.ap2`
 - User interface displayed on desktop via X Windows-based software
 - Minimal subset of data from.ap2 file loaded into memory on login node at any given time
 - Only data requested sent from server to client

pat_report: Job Execution Information

CrayPat/X: Version 5.2.3.8078 Revision 8078 (xf 8063) 08/25/11 ...

Number of PEs (MPI ranks): 16

Numbers of PEs per Node: 16

Numbers of Threads per PE: 1

Number of Cores per Socket: 12

Execution start time: Thu Aug 25 14:16:51 2011

System type and speed: x86_64 2000 MHz

Current path to data file:

`/lus/scratch/heidi/ted_swim/mpi-openmp/run/swim+pat+27472-34t.ap2`

Notes for table 1:

...

Notes for table 1:

Table option:

-O profile

Options implied by table option:

-d ti%@0.95,ti,imb_ti,imb_ti%,tr -b gr,fu,pe=HIDE

Other options:

-T

Options for related tables:

-O profile_pe.th

-O profile_th_pe

-O profile+src

-O load_balance

-O callers

-O callers+src

-O calltree

-O calltree+src

The Total value for Time, Calls is the sum for the Group values.

The Group value for Time, Calls is the sum for the Function values.

The Function value for Time, Calls is the avg for the PE values.

(To specify different aggregations, see: pat_help report options s1)

This table shows only lines with Time% > 0.

Percentages at each level are of the Total for the program.

(For percentages relative to next level up, specify:

-s percent=r[relative])

```
...  
Instrumented with:  
  pat_build -gmpi -u himenoBMTxpr.x
```

```
Program invocation:  
  ../bin/himenoBMTxpr+pat.x
```

```
Exit Status:  0 for 256 PEs
```

```
CPU  Family: 15h  Model: 01h  Stepping: 2
```

```
Core Performance Boost:  Configured for  0 PEs  
                        Capable      for 256 PEs
```

```
Memory pagesize:  4096
```

```
Accelerator Model: Nvidia X2090 Memory: 6.00 GB Frequency: 1.15 GHz
```

```
Programming environment:  CRAY
```

```
Runtime environment variables:  
  OMP_NUM_THREADS=1
```

```
...
```

Sampling Output (Table 1)

Notes for table 1:

...

Table 1: Profile by Function

Samp %	Samp	Imb. Samp	Imb. Samp %	Group Function PE='HIDE'
100.0%	775	--	--	Total
94.2%	730	--	--	USER
43.4%	336	8.75	2.6%	mlwxyz
16.1%	125	6.28	4.9%	half
8.0%	62	6.25	9.5%	full
6.8%	53	1.88	3.5%	artv
4.9%	38	1.34	3.6%	bnd
3.6%	28	2.00	6.9%	currentf
2.2%	17	1.50	8.6%	bndsf
1.7%	13	1.97	13.5%	model
1.4%	11	1.53	12.2%	cfl
1.3%	10	0.75	7.0%	currenth
1.0%	8	5.28	41.9%	bndbo
1.0%	8	8.28	53.4%	bndto
5.4%	42	--	--	MPI
1.9%	15	4.62	23.9%	mpi_sendrecv
1.8%	14	16.53	55.0%	mpi_bcast
1.7%	13	5.66	30.7%	mpi_barrier

Table 1: Profile by Function Group and Function

Time %	Time	Imb. Time	Imb. Time %	Calls	Group	Function
						PE='HIDE'
100.0%	104.593634	--	--	22649	Total	

71.0%	74.230520	--	--	10473	MPI	

69.7%	72.905208	0.508369	0.7%	125	mpi_allreduce_	
1.0%	1.050931	0.030042	2.8%	94	mpi_alltoall_	
=====						
25.3%	26.514029	--	--	73	USER	

16.7%	17.461110	0.329532	1.9%	23	selfgravity_	
7.7%	8.078474	0.114913	1.4%	48	ffte4_	
=====						
2.5%	2.659429	--	--	435	MPI_SYNC	

2.1%	2.207467	0.768347	26.2%	172	mpi_barrier_(sync)	
=====						
1.1%	1.188998	--	--	11608	HEAP	

1.1%	1.166707	0.142473	11.1%	5235	free	
=====						

pat_report: Message Stats by Caller

Table 4: MPI Message Stats by Caller

MPI Msg Bytes	MPI Msg Count	MsgSz <16B Count	4KB<= MsgSz <64KB Count	Function Caller PE [mmm]
15138076.0	4099.4	411.6	3687.8	Total

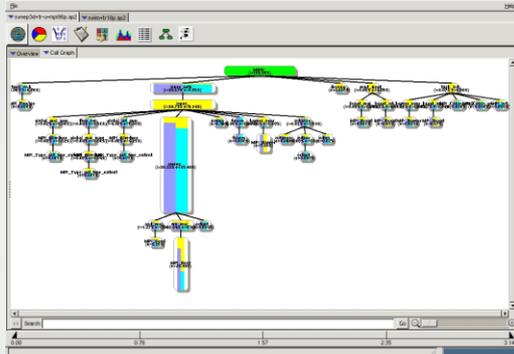
15138028.0	4093.4	405.6	3687.8	MPI_ISEND

8080500.0	2062.5	93.8	1968.8	calc2_ MAIN_

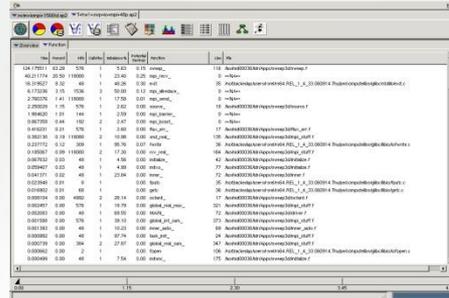
8216000.0	3000.0	1000.0	2000.0	pe.0
8208000.0	2000.0	--	2000.0	pe.9
6160000.0	2000.0	500.0	1500.0	pe.15
=====				
6285250.0	1656.2	125.0	1531.2	calc1_ MAIN_

8216000.0	3000.0	1000.0	2000.0	pe.0
6156000.0	1500.0	--	1500.0	pe.3
6156000.0	1500.0	--	1500.0	pe.5
=====				
. . .				

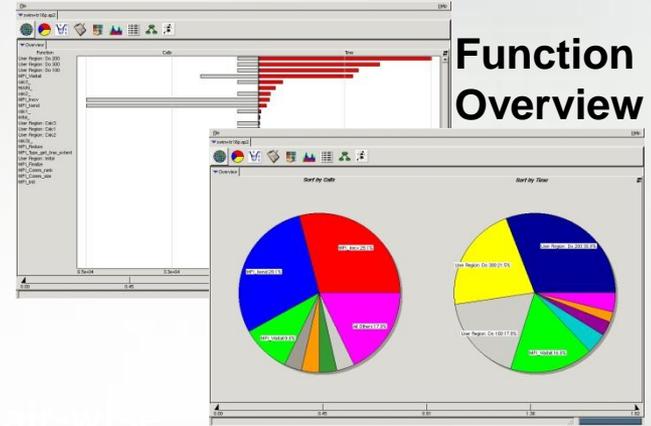
Call Graph Profile



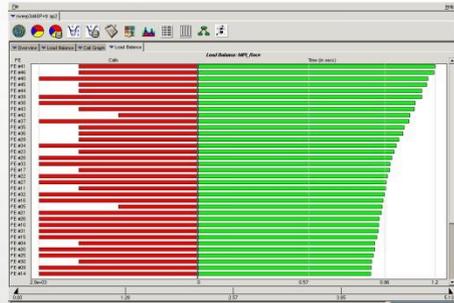
Function Profile



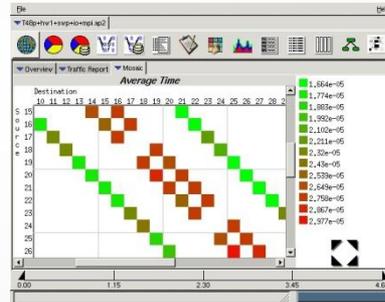
Function Overview



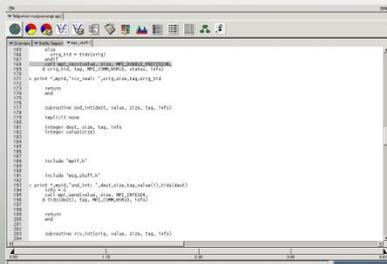
Load based views



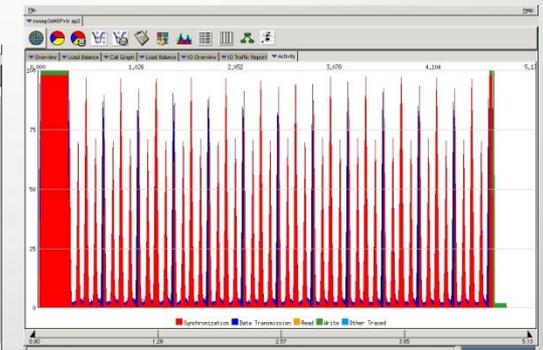
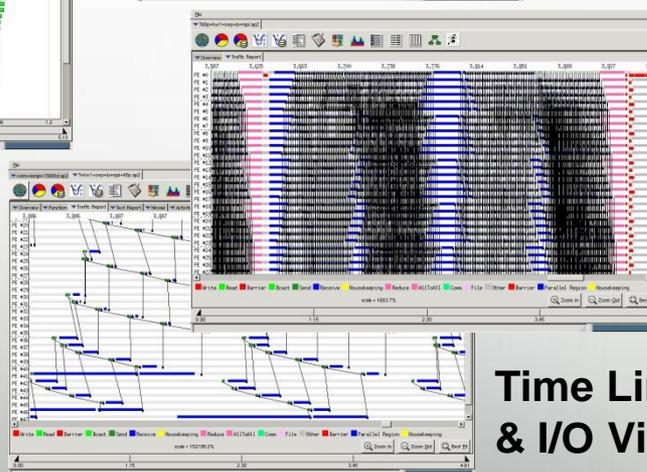
Communication View



Source code mapping

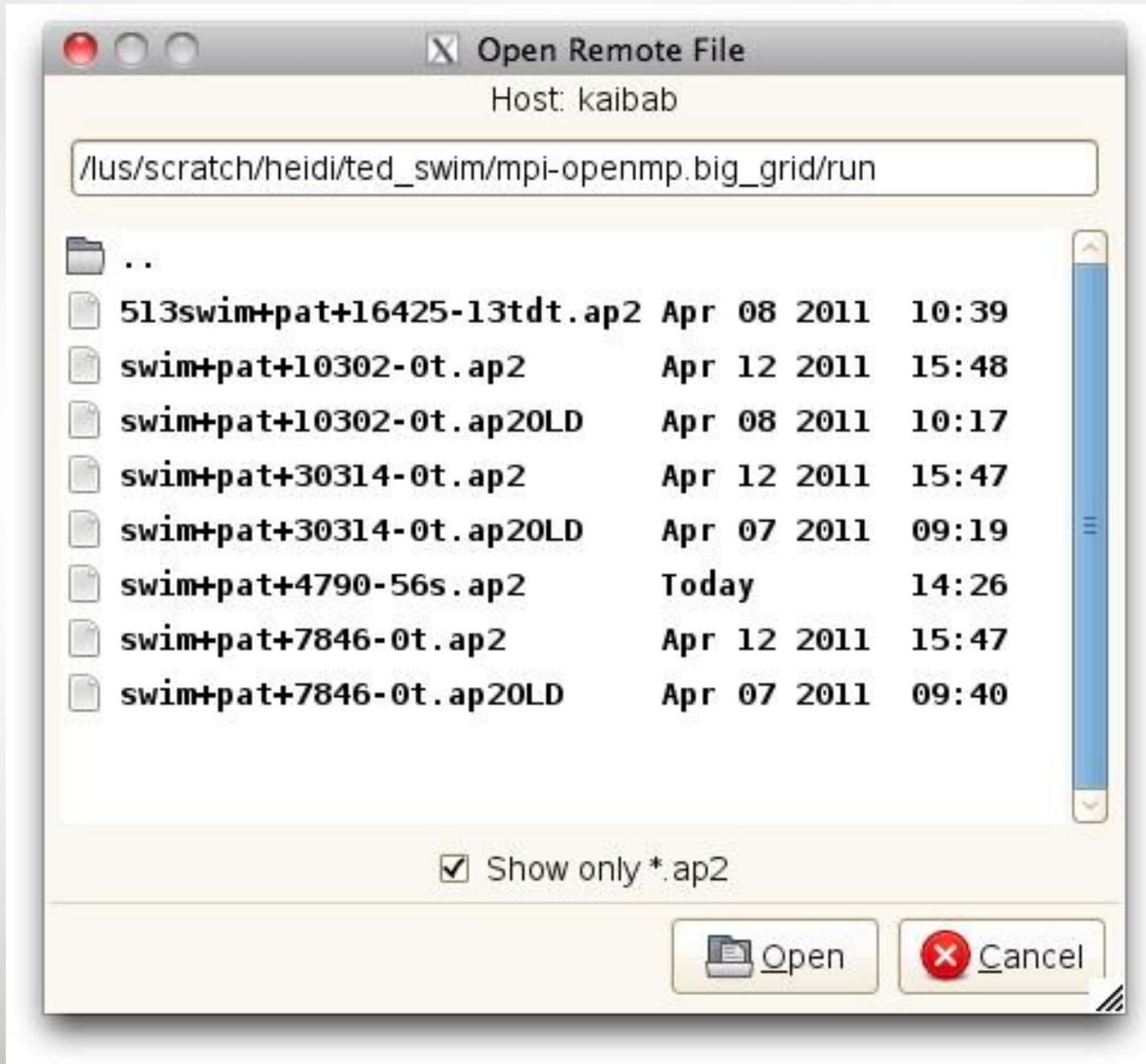


Time Line & I/O Views

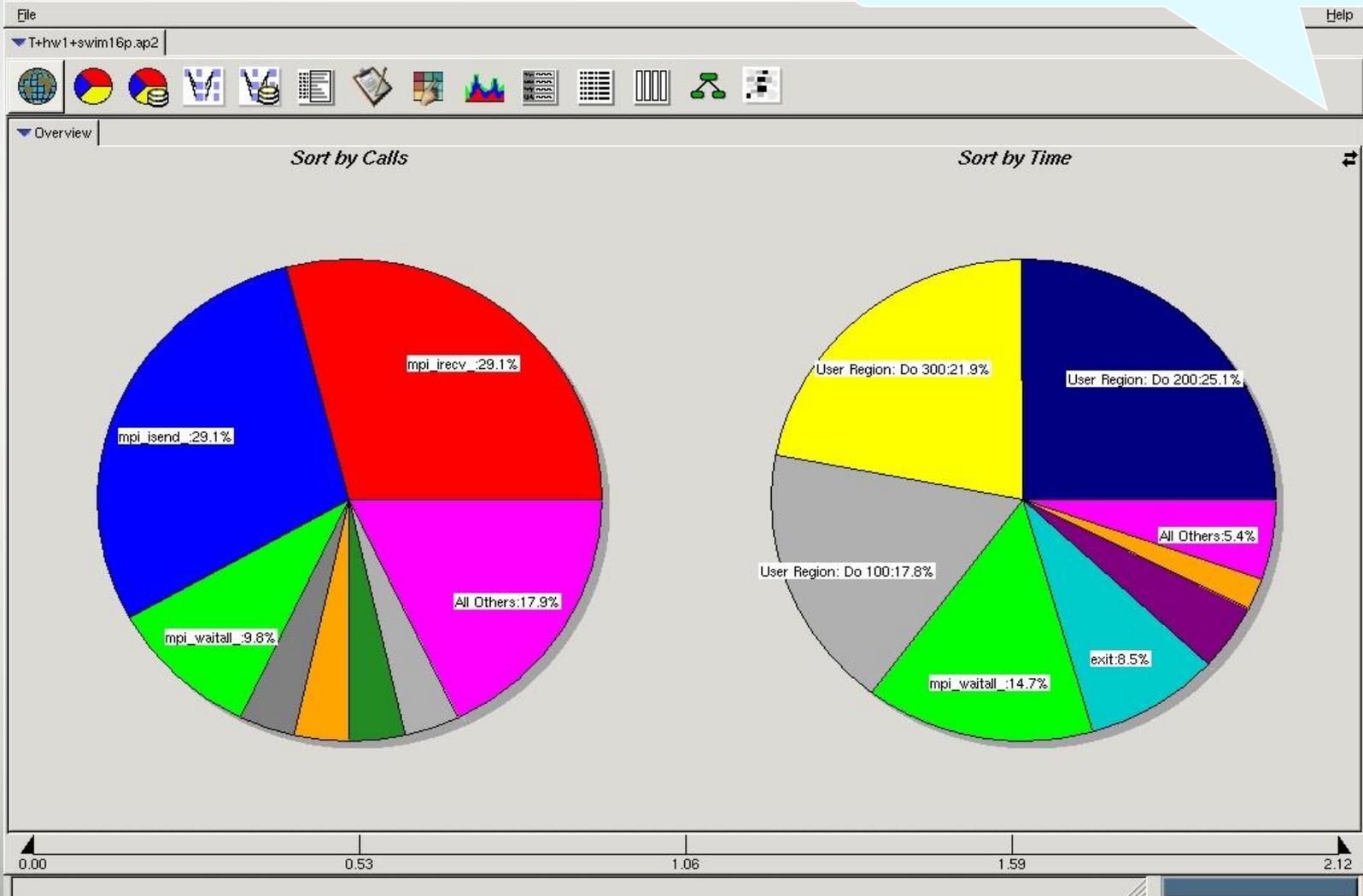


- Call graph profile
 - Communication statistics
 - Time-line view
 - Communication
 - I/O
 - Activity view
 - Pair-wise communication statistics
 - Text reports
 - Source code mapping
- Cray Apprentice² helps identify:
 - Load imbalance
 - Excessive communication
 - Network contention
 - Excessive serialization
 - I/O Problems

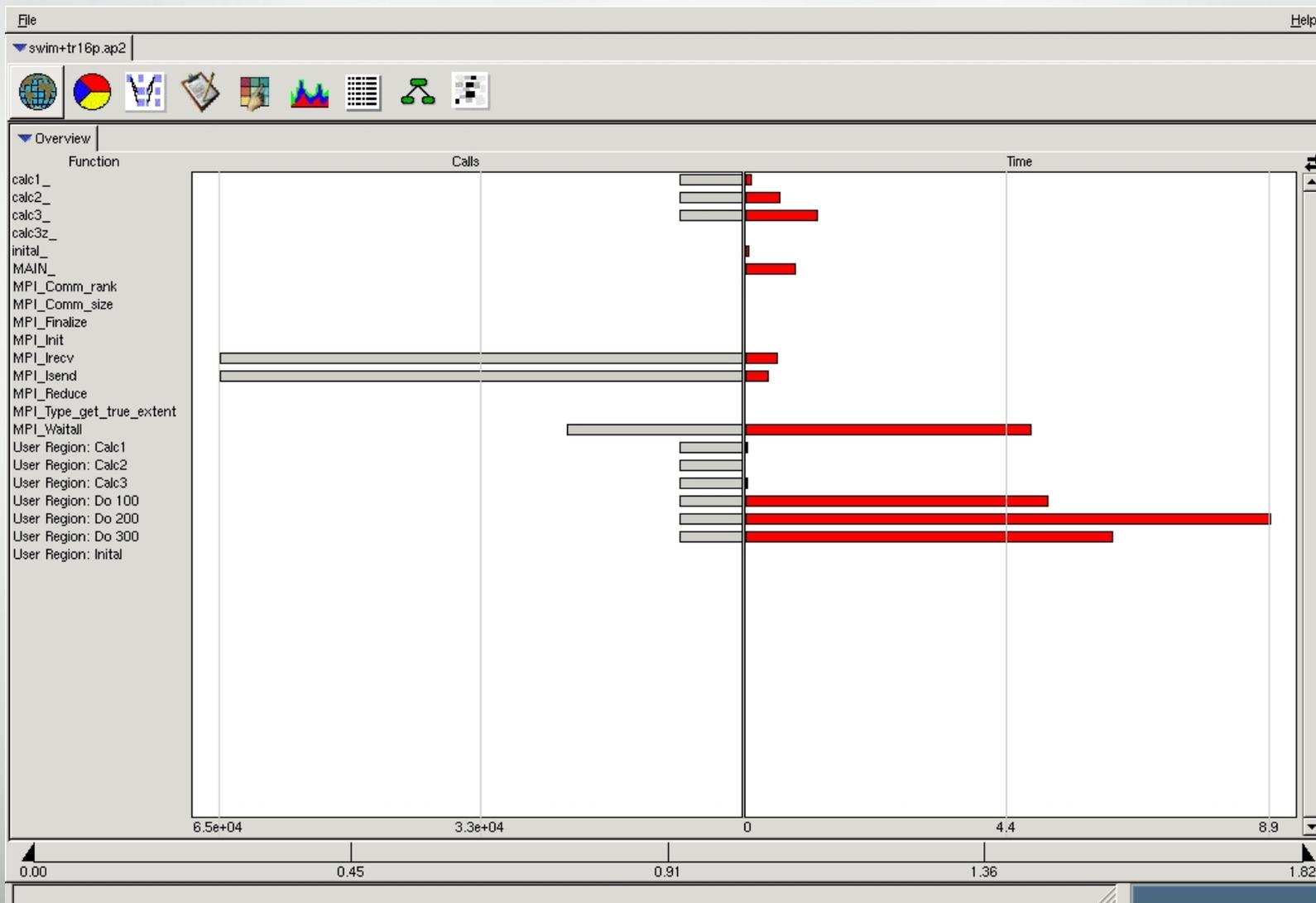
Accessing Remote File from app2 Client



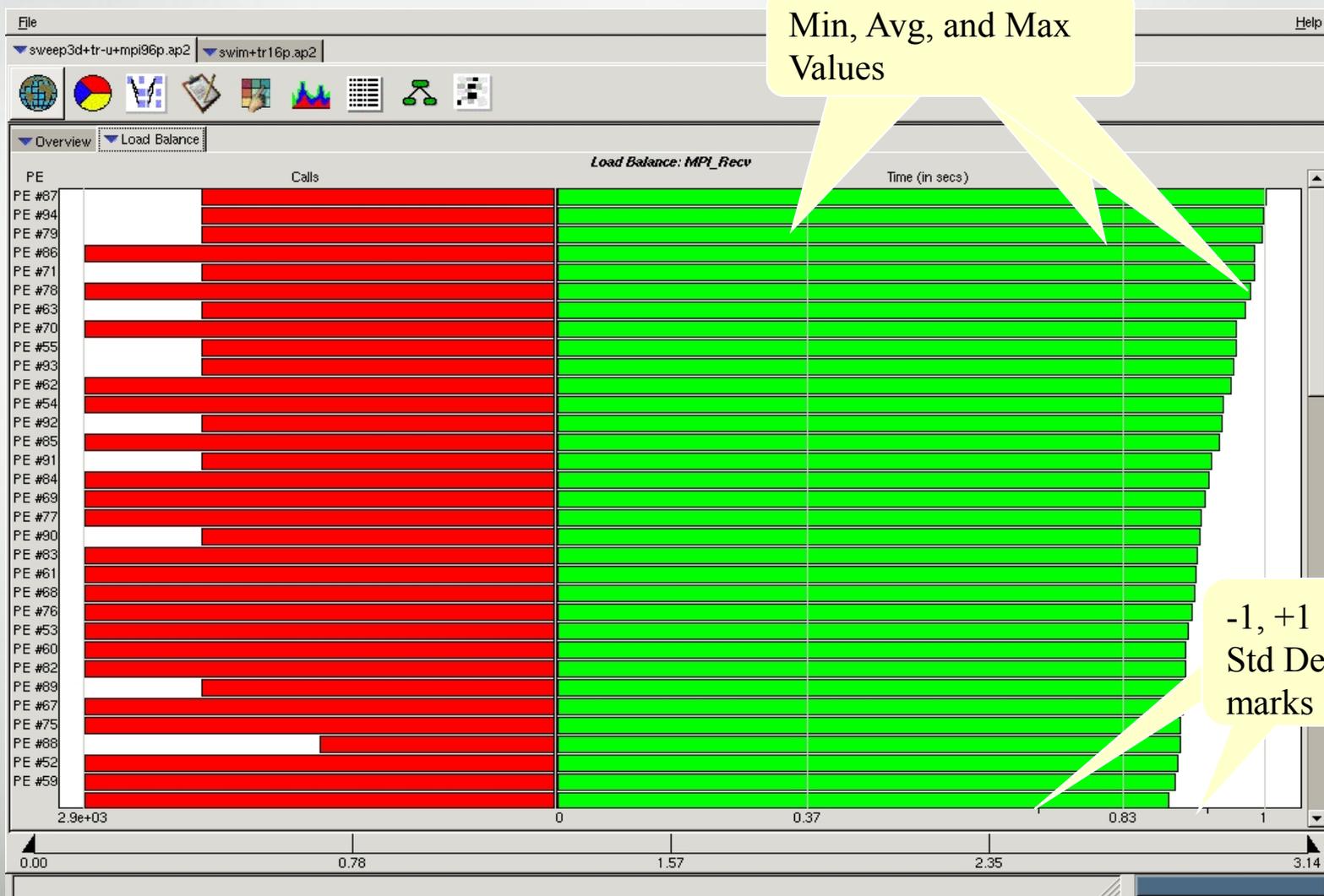
Switch Overview display



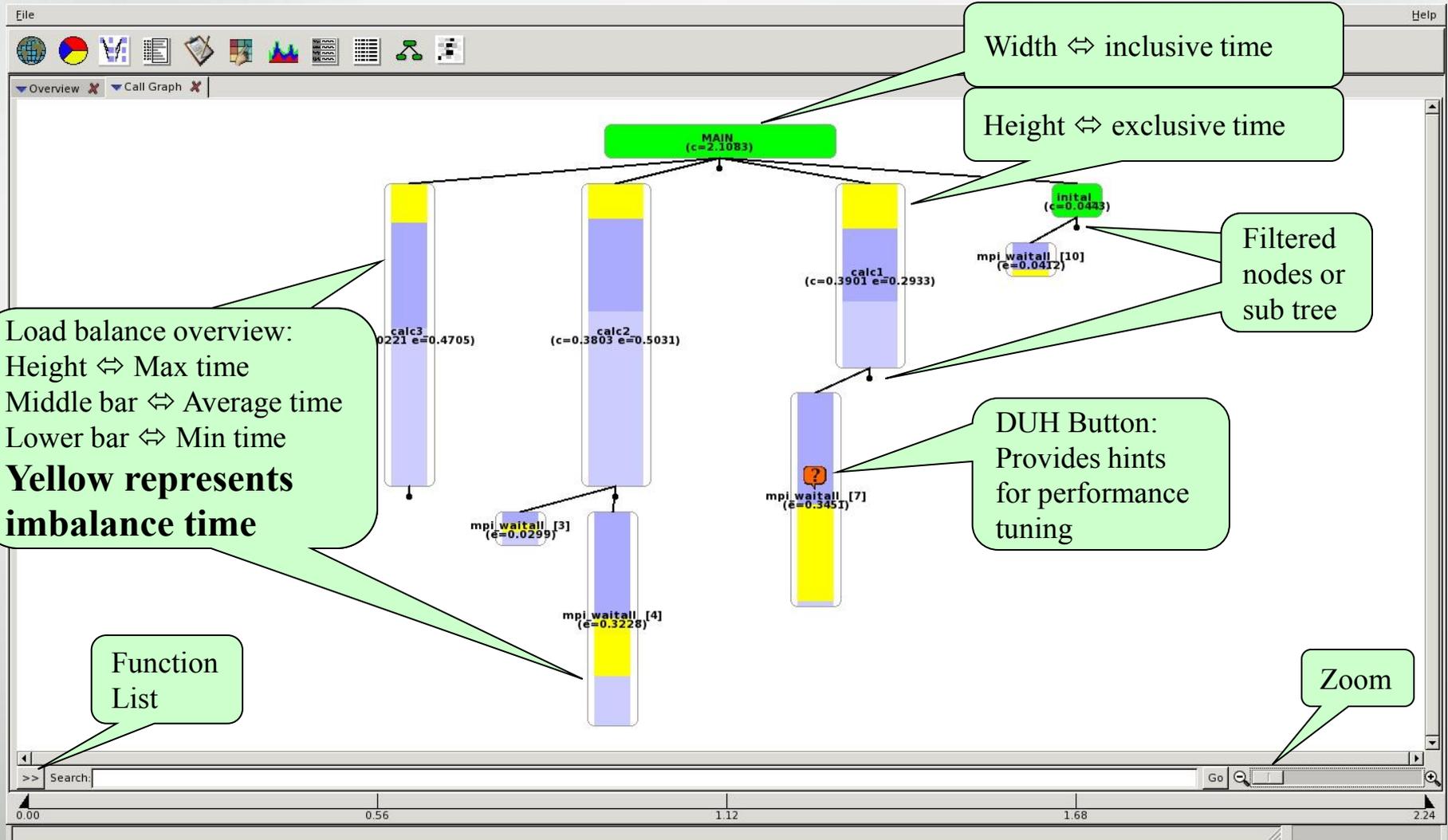
Function Profile



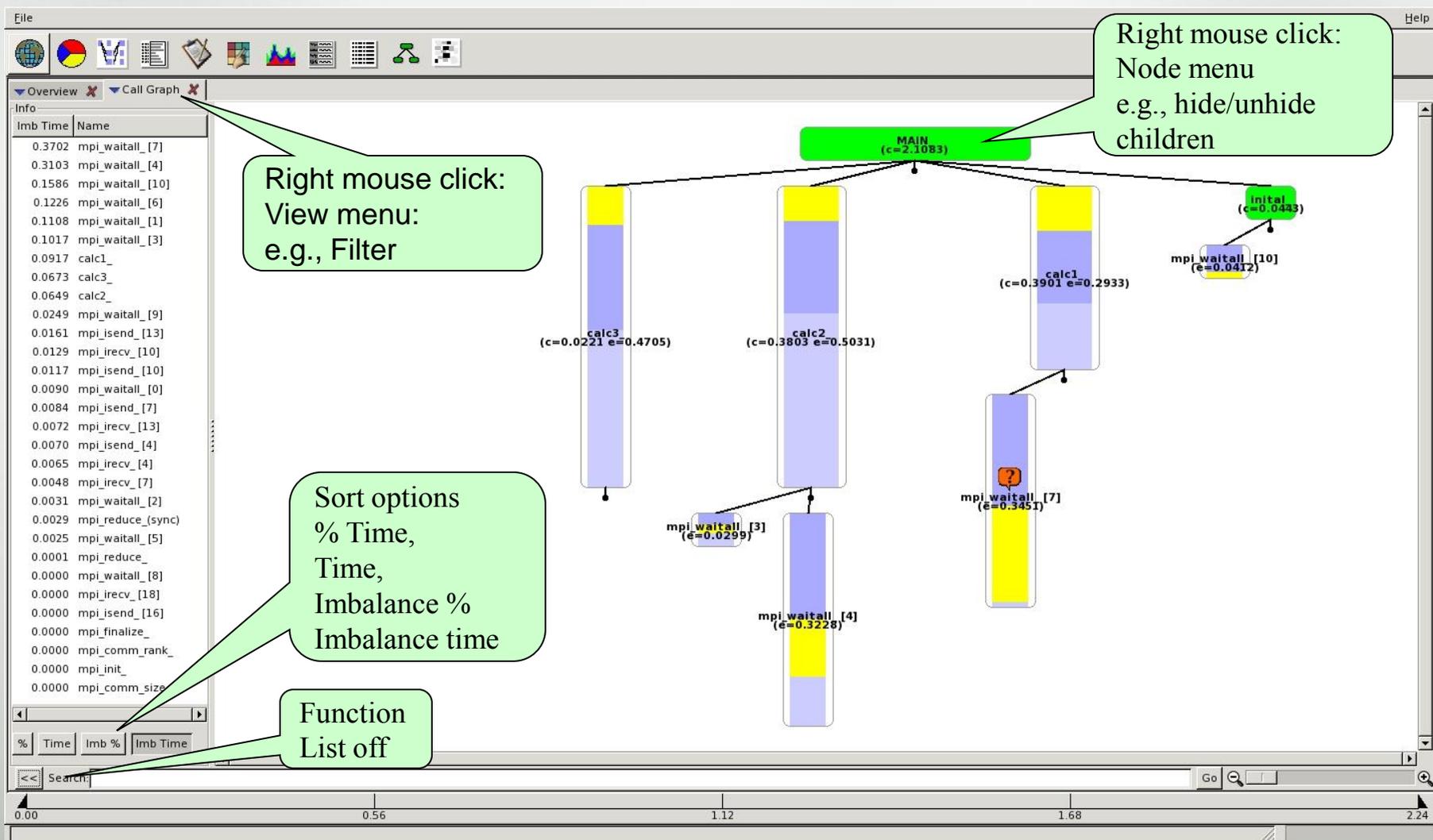
Load Balance View (Aggregated from Overview)



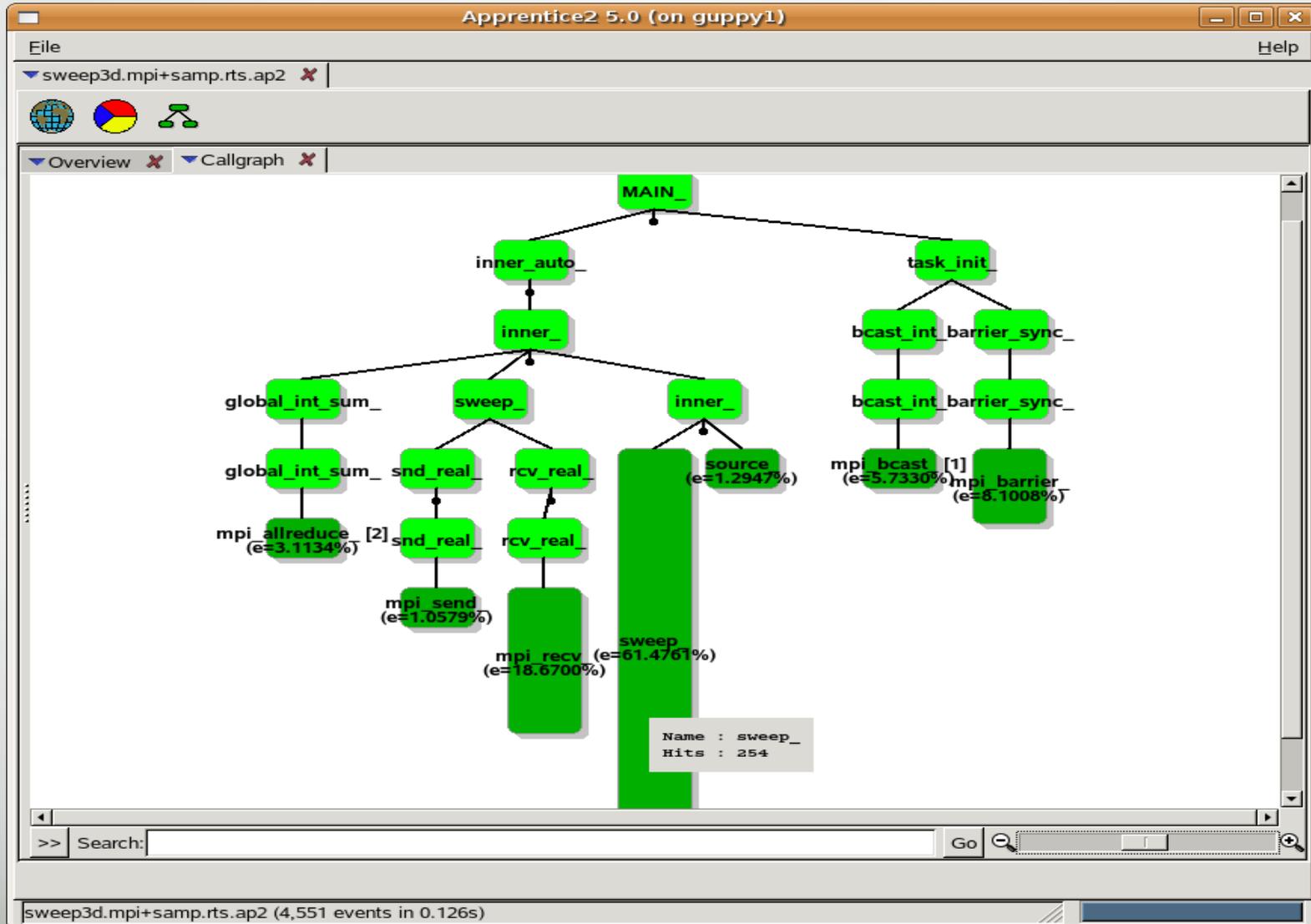
Call Tree View



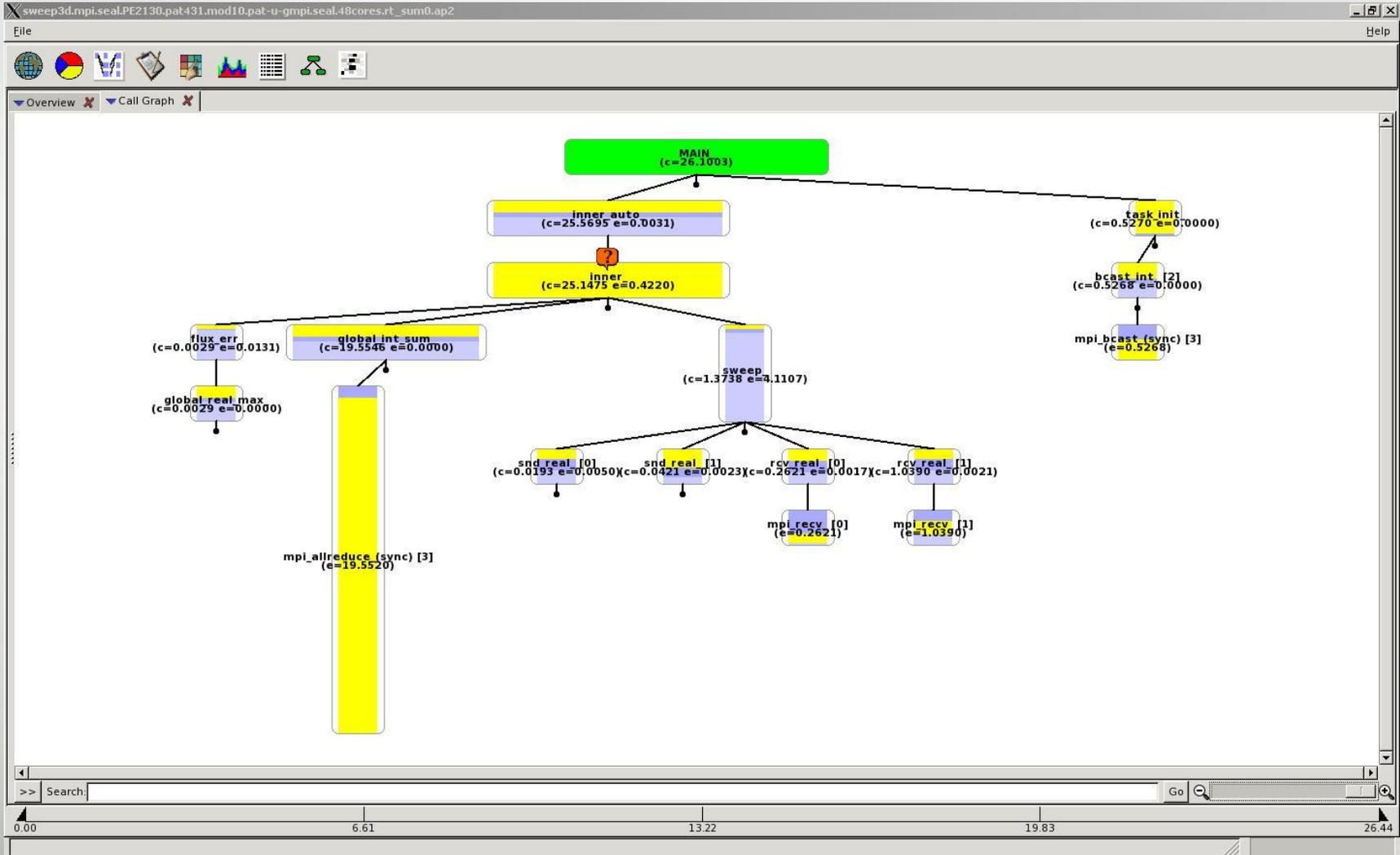
Call Tree View – Function List



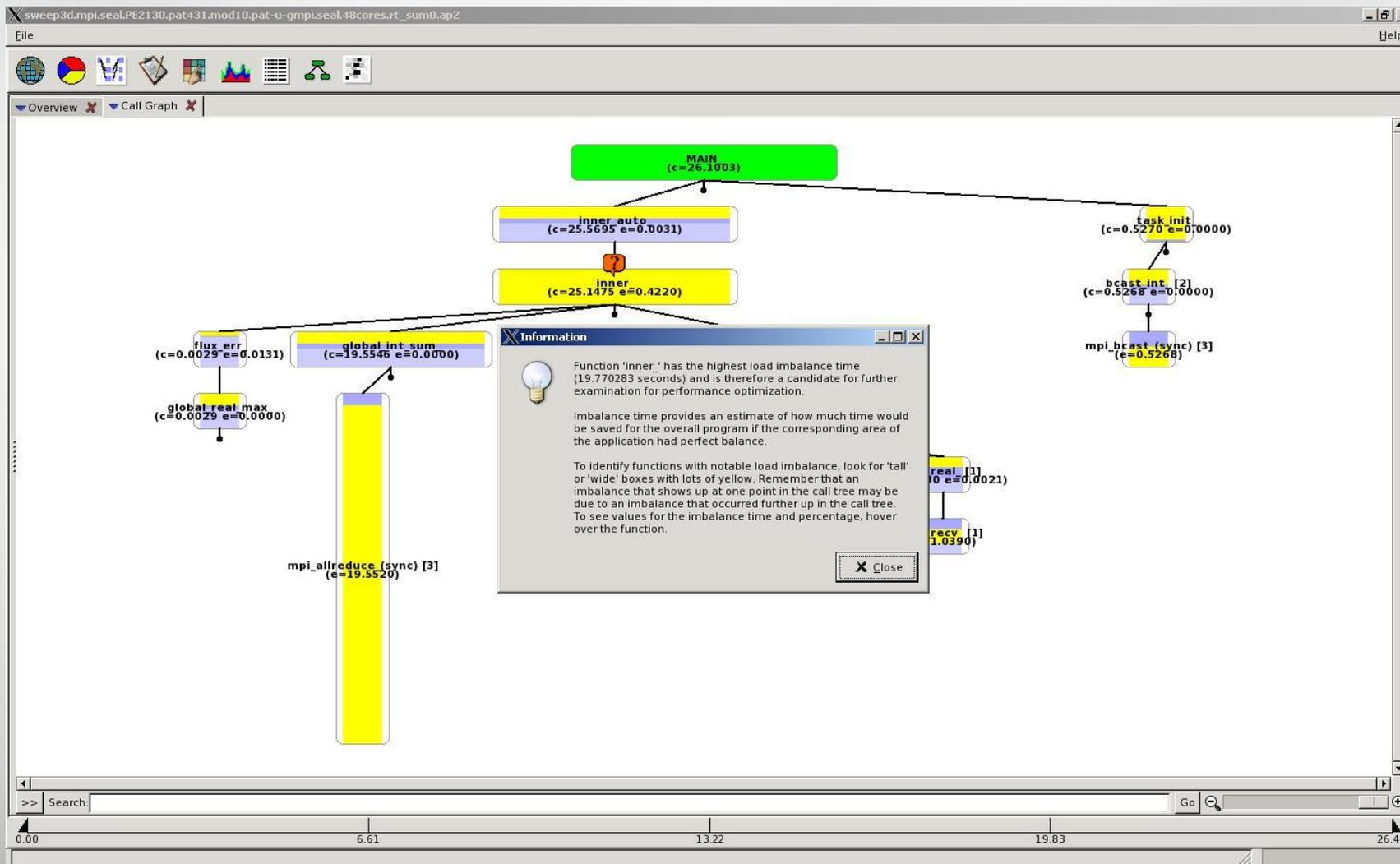
Apprentice² Call Tree View of Sampled Data



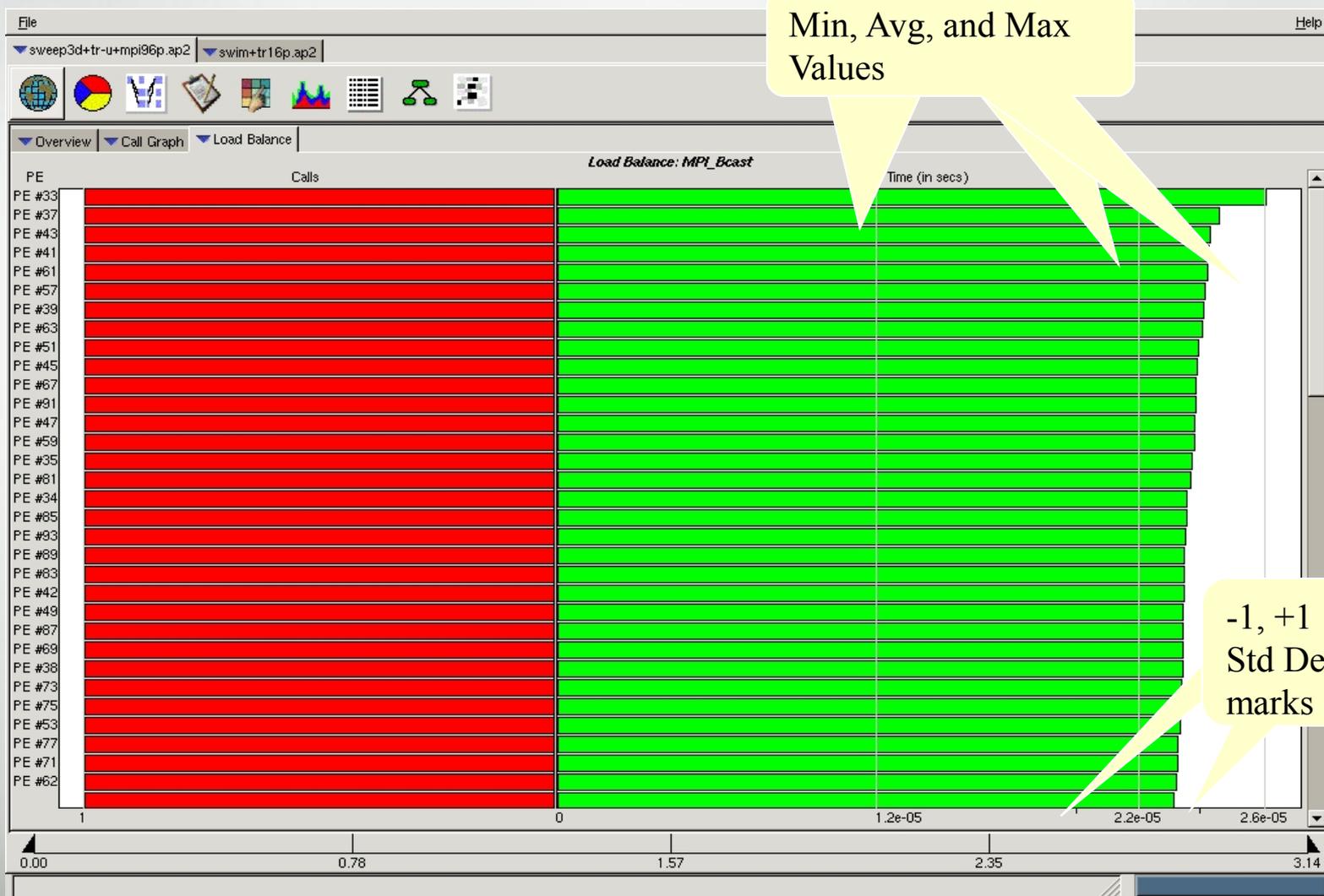
Call Tree Visualization



Discrete Unit of Help (DUH Button)



Load Balance View (from Call Tree)



Source Mapping from Call Tree

```
165
166 c angle pipelining loop (batches of rmi angles)
167 c
168     DO mo = 1, rmo
169     mio = (mo-1)*rmi
170
171 c K-inflows (k=k0 boundary)
172 c
173     if (k2.lt.0 .or. kbc.eq.0) then
174     do mi = 1, rmi
175     do j = 1, jt
176     do i = 1, it
177     phikb(i,j,mi) = 0.0d+0
178     end do
179     end do
180     end do
181     else
182     if (do_dsa) then
183     leak = 0.0
184     k = k0 - k2
185     do mi = 1, rmi
186     m = mi + mio
187     do j = 1, jt
188     do i = 1, it
189     phikb(i,j,mi) = phikbc(i,j,m)
190     leak = leak
191     & + wtsi(m)*phikb(i,j,mi)*di(i)*dj(j)
192     & face(i,j,k+k3,3) = face(i,j,k+k3,3)
193     & + wtsi(m)*phikb(i,j,mi)
194     end do
195     end do
196     end do
197     leakage(5) = leakage(5) + leak
199     else
```

- Complimentary performance data available in one place
- Drop down menu provides quick access to most common reports
- Ability to easily generate different views of performance data
- Provides mechanism for more in depth explanation of data presented

Example of pat_report Tables in Cray Apprentice2

The screenshot shows the Cray Apprentice2 interface with a 'Text' window open. The window displays the following text:

```
CrayPat/X: Version 5.2 Revision 7190 (xf 705... 04/06/11 02:52:12)
Number of PEs (MPI ranks): 16
Numbers of PEs per Node: 16
Numbers of Threads per PE: 1
Number of Cores per Socket: 12
Execution start time: Thu Apr 7 09:50:13 2011
System type and speed: x86_64 2000 MHz
Current path to data file: swim+pat+10302-0t.ap2

Notes for table 1:

Table option:
-O profile
Options implied by table option:
-d ti%@0.95,ti,imb_ti,imb_ti%,tr -b gr,fu,pe=HIDE,th=HIDE

The Total value for Time, Calls is the sum for the Group values.
The Group value for Time, Calls is the sum for the Function values.
The Function value for Time, Calls is the avg for the PE values.
The PE value for Time, Calls is the max for the Thread values.
(To specify different aggregations, see: pat_help report options s1)

This table shows call times with Time = 0.05
```

At the bottom of the window, a progress bar shows values: 0.00, 49.30, 98.59, 147.89, 197.18. The task name 'swim+pat+10302-0t.ap2' and its duration '(1.373s)' are also visible.

New text table icon

Right click for table generation options

Generating New pat_report Tables

- Profile
- Custom...

- Source
- Calltree
- Callers

- Show Notes
- Show All PE's
- Show HWPC
- Use Thresholds

Select All

Select None

Panel Actions >

Panel Help

Load Imbalance Analysis

■ Load imbalance

- Identifies computational code regions and synchronization calls that could benefit most from load balance optimization (some processes have less work than others, some are waiting longer on barriers, etc)
- Estimates savings if corresponding section of code were balanced
- MPI sync time (determines late arrivers to barriers)
- MPI rank placement suggestions (maximize on-node communication)
- Imbalance metrics (user functions, MPI functions, OpenMP threads)

- Increasing system software and architecture complexity
 - Current trend in high end computing is to have systems with tens of thousands of processors
 - This is being accentuated with multi-core processors

- Applications have to be very well balanced In order to perform at scale on these MPP systems
 - Efficient application scaling includes a balanced use of requested computing resources

- Desire to minimize computing resource “waste”
 - Identify slower paths through code
 - Identify inefficient “stalls” within an application

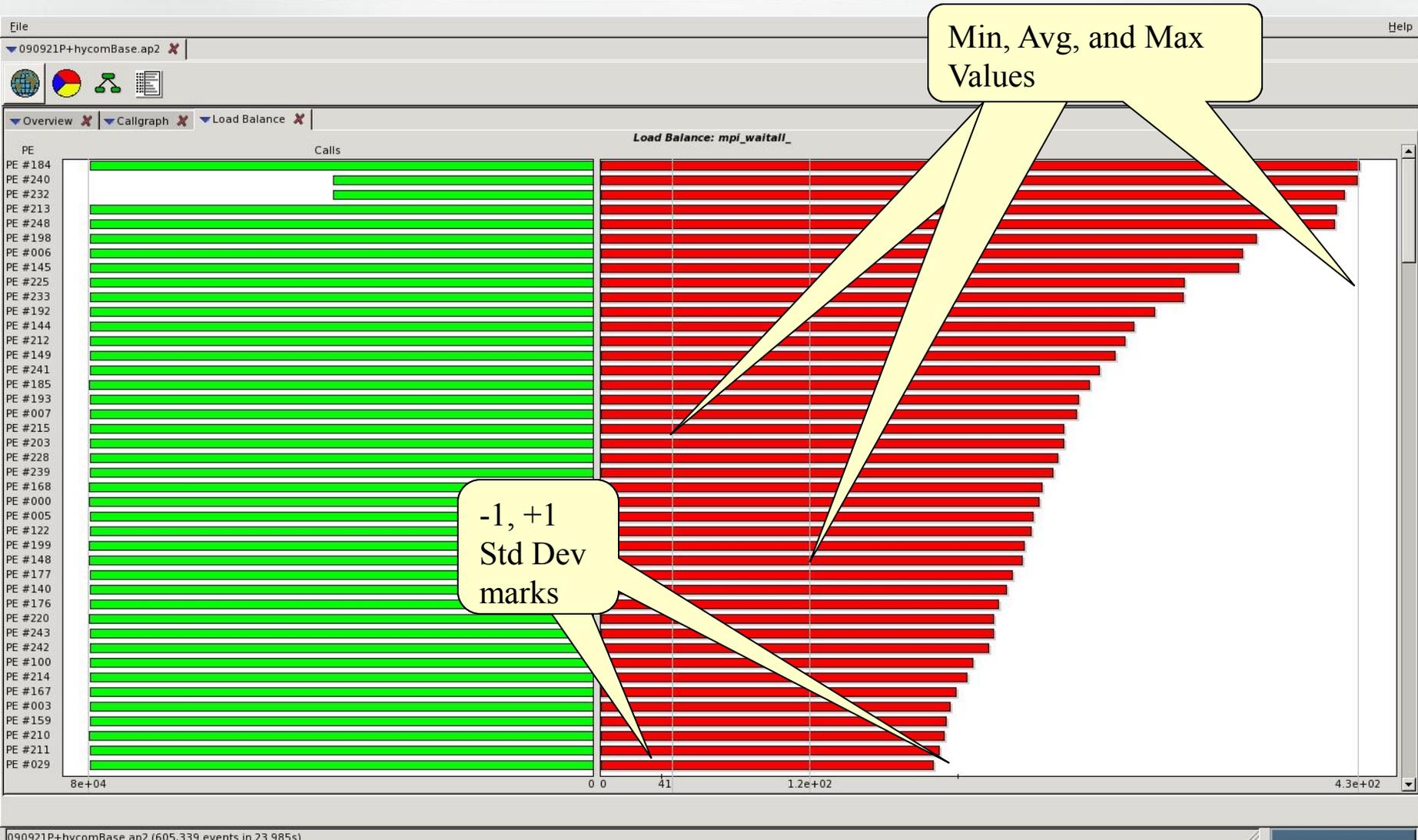
- Measure load imbalance in programs instrumented to trace MPI functions to determine if MPI ranks arrive at collectives together
- Separates potential load imbalance from data transfer
- Sync times reported by default if MPI functions traced
- If desired, `PAT_RT_MPI_SYNC=0` deactivates this feature

- Metric based on execution time
- It is dependent on the type of activity:
 - User functions
 - Imbalance time = Maximum time – Average time**
 - Synchronization (Collective communication and barriers)
 - Imbalance time = Average time – Minimum time**
- Identifies computational code regions and synchronization calls that could benefit most from load balance optimization
- Estimates how much overall program time could be saved if corresponding section of code had a perfect balance
 - Represents upper bound on “potential savings”
 - Assumes other processes are waiting, not doing useful work while slowest member finishes

$$\text{Imbalance\%} = 100 \times \frac{\text{Imbalance time}}{\text{Max Time}} \times \frac{N}{N - 1}$$

- Represents % of resources available for parallelism that is “wasted”
- Corresponds to % of time that rest of team is not engaged in useful work on the given function
- Perfectly balanced code segment has imbalance of 0%
- Serial code segment has imbalance of 100%

Load Distribution



MPI Rank Placement Suggestions

- Analyze runtime performance data to identify grids in a program to maximize on-node communication
 - Example: nearest neighbor exchange in 2 dimensions
 - Sweep3d uses a 2-D grid for communication
- Determine whether or not a custom MPI rank order will produce a significant performance benefit
- Grid detection is helpful for programs with significant point-to-point communication
- Doesn't interfere with MPI collective communication optimizations

- Tools produce a custom rank order if it's beneficial based on grid size, grid order and cost metric
- Summarized findings in report
- Available if MPI functions traced (-g mpi)
- Describe how to re-run with custom rank order

Example: Observations and Suggestions

MPI Grid Detection: There appears to be point-to-point MPI communication in a 22 X 18 grid pattern. The 48.6% of the total execution time spent in MPI functions might be reduced with a rank order that maximizes communication between ranks on the same node. The effect of several rank orders is estimated below.

A file named `MPICH_RANK_ORDER.Custom` was generated along with this report and contains the Custom rank order from the following table. This file also contains usage instructions and a table of alternative rank orders.

Rank Order	On-Node Bytes/PE	On-Node Bytes/PE% of Total Bytes/PE	MPICH_RANK_REORDER_METHOD
Custom	7.80e+06	78.37%	3
SMP	5.59e+06	56.21%	1
Fold	2.59e+05	2.60%	2
RoundRobin	0.00e+00	0.00%	0

MPICH_RANK_ORDER File Example

```
# The 'Custom' rank order in this file targets nodes with multi-core
# processors, based on Sent Msg Total Bytes collected for:
#
# Program:    /lus/nid00030/heidi/sweep3d/mod/sweep3d.mpi
# Ap2 File:   sweep3d.mpi+pat+27054-89t.ap2
# Number PEs: 48
# Max PEs/Node: 4
#
# To use this file, make a copy named MPICH_RANK_ORDER, and set the
# environment variable MPICH_RANK_REORDER_METHOD to 3 prior to
# executing the program.
#
# The following table lists rank order alternatives and the grid_order
# command-line options that can be used to generate a new order.
...
```

Example 2 - Hycom

===== Observations and suggestions =====

MPI grid detection:

There appears to be point-to-point MPI communication in a 33 X 41 grid pattern. The 26.1% of the total execution time spent in MPI functions might be reduced with a rank order that maximizes communication between ranks on the same node. The effect of several rank orders is estimated below.

A file named MPICH_RANK_ORDER.Custom was generated along with this report and contains the Custom rank order from the following table. This file also contains usage instructions and a table of alternative rank orders.

Rank Order	On-Node Bytes/PE	On-Node Bytes/PE% of Total Bytes/PE	MPICH_RANK_REORDER_METHOD
Custom	1.20e+09	32.21%	3
SMP	8.70e+08	23.27%	1
Fold	3.55e+07	0.95%	2
RoundRobin	1.99e+05	0.01%	0

===== End Observations =====

- Run on 1353 MPI ranks, 24 ranks per node

- Overall program wallclock:
 - Default MPI rank order: 1450s
 - Custom MPI rank order: 1315s
 - ~10% improvement in execution time!

- Time spent in MPI routines:
 - Default rank order: 377s
 - Custom rank order: 303s

Loop Work Estimates

- Helps identify loops to optimize (parallelize serial loops):
 - Loop timings approximate how much work exists within a loop
 - Trip counts can be used to help carve up loop on GPU
- Enabled with CCE `-h profile_generate` option
 - Should be done as separate experiment – **compiler optimizations are restricted with this feature**
- Loop statistics reported by default in `pat_report` table
- Next enhancement: integrate loop information in profile
 - Get exclusive times and loops attributed to functions

- Load PrgEnv-cray software
- Load perftools software

- Compile **AND** link with `-h profile_generate`

- Instrument binary for tracing
 - `pat_build -u my_program` or
 - `pat_build -w my_program`

- Run application
- Create report with loop statistics
 - `pat_report my_program.xf > loops_report`

Example Report – Loop Work Estimates

Table 1: Profile by Function Group and Function

Time%	Time	Imb.	Imb.	Calls	Group
		Time	Time%		Function
					PE=HIDE
					Thread=HIDE
100.0%	176.687480	--	--	17108.0	Total

85.3%	150.789559	--	--	8.0	USER

85.0%	150.215785	24.876709	14.4%	2.0	jacobi_.LOOPS
=====					
12.2%	21.600616	--	--	16071.0	MPI

11.9%	21.104488	41.016738	67.1%	3009.0	mpi_waitall
=====					
2.4%	4.297301	--	--	1007.0	MPI_SYNC

2.4%	4.166092	4.135016	99.3%	1004.0	mpi_allreduce_(sync)
=====					

Example Report – Loop Work Estimates (2)



Table 3: Inclusive Loop Time from -hprofile_generate

Loop	Incl	Time	Loop	Hit	Loop	Trips	Loop	Trips	Function=/.LOOP[.]
		Total				Min		Max	

...									
	175.676881		2		0		1003		jacobi_.LOOP.07.li.267
	0.917107		1003		0		260		jacobi_.LOOP.08.li.276
	0.907515		129888		0		260		jacobi_.LOOP.09.li.277
	0.446784		1003		0		260		jacobi_.LOOP.10.li.288
	0.425763		129888		0		516		jacobi_.LOOP.11.li.289
	0.395003		1003		0		260		jacobi_.LOOP.12.li.300
	0.374206		129888		0		516		jacobi_.LOOP.13.li.301
	126.250610		1003		0		256		jacobi_.LOOP.14.li.312
	126.223035		127882		0		256		jacobi_.LOOP.15.li.313
	124.298650		16305019		0		512		jacobi_.LOOP.16.li.314
	20.875086		1003		0		256		jacobi_.LOOP.17.li.336
	20.862715		127882		0		256		jacobi_.LOOP.18.li.337
	19.428085		16305019		0		512		jacobi_.LOOP.19.li.338
=====									

Other Interesting Performance Data

- Sampling is useful to determine where the program spends most of its time (functions and lines)
- The environment variable **PAT_RT_EXPERIMENT** allows the specification of the type of experiment prior to execution
 - **samp_pc_time** (default)
 - Samples the PC at intervals of 10,000 microseconds
 - Measures user CPU and system CPU time
 - Returns total program time and absolute and relative times each program counter was recorded
 - Optionally record the values of hardware counters specified with PAT_RT_HWPC
 - **samp_pc_ovfl**
 - Samples the PC at a given overflow of a HW counter
 - Does not allow collection of hardware counters
 - **samp_cs_time**
 - Sample the call stack at a given time interval

-g tracegroup (subset)

- blas Basic Linear Algebra subprograms
- CAF Co-Array Fortran (Cray CCE compiler only)
- HDF5 manages extremely large and complex data collections
- heap dynamic heap
- io includes stdio and sysio groups
- lapack Linear Algebra Package
- math ANSI math
- mpi MPI
- omp OpenMP API
- omp-rtl OpenMP runtime library (not supported on Catamount)
- pthreads POSIX threads (not supported on Catamount)
- shmem SHMEM
- sysio I/O system calls
- system system calls
- upc Unified Parallel C (Cray CCE compiler only)

For a full list, please see `man pat_build`

Specific Tables in pat_report

```
heidi@kaibab:/lus/scratch/heidi> pat_report -O -h
```

```
pat_report: Help for -O option:
```

```
Available option values are in left column, a prefix can be specified:
```

ct	-O calltree
defaults	<Tables that would appear by default.>
heap	-O heap_program,heap_hiwater,heap_leaks
io	-O read_stats,write_stats
lb	-O load_balance
load_balance	-O lb_program,lb_group,lb_function
mpi	-O mpi_callers

D1_D2_observation	Observation about Functions with low
D1+D2 cache hit ratio	
D1_D2_util	Functions with low D1+D2 cache hit ratio
D1_observation	Observation about Functions with low D1
cache hit ratio	
D1_util	Functions with low D1 cache hit ratio
TLB_observation	Observation about Functions with low TLB
refs/miss	
TLB_util	Functions with low TLB refs/miss

- -g heap
 - calloc, cfree, malloc, free, malloc_trim, malloc_usable_size, mallopt, memalign, posix_memalign, pvalloc, realloc, valloc

- -g heap
- -g sheap
- -g shmem
 - shfree, shfree_nb, shmalloc, shmalloc_nb, shrealloc

- -g upc (automatic with `-O apa`)
 - upc_alloc, upc_all_alloc, upc_all_free, uc_all_lock_alloc, upc_all_lock_free, upc_free, upc_global_alloc, upc_global_lock_alloc, upc_lock_free

Notes for table 5:

Table option:

-O heap_hiwater

Options implied by table option:

-d am@,ub,ta,ua,tf,nf,ac,ab -b pe=[mmm]

This table shows only lines with Tracked Heap HiWater MBytes >

Table 5: Heap Stats during Main Program

Tracked Heap HiWater MBytes	Total Allocs	Total Frees	Tracked Objects Not Freed	Tracked MBytes Not Freed	PE [mmm]
9.794	915	910	4	1.011	Total
9.943	1170	1103	68	1.046	pe.0
9.909	715	712	3	1.010	pe.22
9.446	1278	1275	3	1.010	pe.43

- Fortran

```
include "pat_apif.h"
```

```
...
```

```
call PAT_region_begin(id, "label", ierr)
```

```
do i = 1,n
```

```
...
```

```
enddo
```

```
call PAT_region_end(id, ierr)
```

- C & C++

```
include <pat_api.h>
```

```
...
```

```
ierr = PAT_region_begin(id, "label");
```

```
< code segment >
```

```
ierr = PAT_region_end(id);
```

OpenMP Support

- Measure overhead incurred entering and leaving
 - Parallel regions
 - Work-sharing constructs within parallel regions

- Show per-thread timings and other data

- Trace entry points automatically inserted by Cray and PGI compilers
 - Provides per-thread information

- Can use sampling to get performance data without API (per process view... no per-thread counters)
 - Run with `OMP_NUM_THREADS=1` during sampling
 - Watch for calls to `omp_set_num_threads()`

- Load imbalance calculated across all threads in all ranks for mixed MPI/OpenMP programs
 - Can choose to see imbalance to each programming model separately

- Data displayed by default in `pat_report` (no options needed)
 - Focus on where program is spending its time
 - Assumes all requested resources should be used

- profile_pe.th (default view)
 - Imbalance based on the set of all threads in the program

- profile_pe_th
 - Highlights imbalance across MPI ranks
 - Uses max for thread aggregation to avoid showing under-performers
 - Aggregated thread data merged into MPI rank data

- profile_th_pe
 - For each thread, show imbalance over MPI ranks
 - Example: Load imbalance shown where thread 4 in each MPI rank didn't get much work

Profile by Function Group and Function (with -T)

Table 1: Profile by Function Group and Function

Time %	Time	Imb. Time	Imb. Time %	Calls	Group	Function	PE.Thread='HIDE'
100.0%	12.548996	--	--	7944.7	Total		

97.8%	12.277316	--	--	3371.8	USER		

35.6%	4.473536	0.072259	1.6%	498.0	calc3_.LOOP@li.96		
29.1%	3.653288	0.070551	1.9%	500.0	calc2_.LOOP@li.74		
28.3%	3.545677	0.056303	1.6%	500.0	calc1_.LOOP@li.69		
.....							
=====							
1.2%	0.155028	--	--	1000.5	MPI_SYNC		

1.2%	0.154899	0.674518	82.0%	999.0	mpi_barrier_(sync)		
0.0%	0.000129	0.000489	79.8%	1.5	mpi_reduce_(sync)		
=====							
0.7%	0.082943	--	--	3197.2	MPI		

0.4%	0.047471	0.158820	77.6%	999.0	mpi_barrier_		
0.1%	0.015157	0.295055	95.9%	297.1	mpi_waitall_		
.....							
=====							
0.3%	0.033683	--	--	374.5	OMP		

0.1%	0.013098	0.078620	86.4%	125.0	calc2_.REGION@li.74 (ovhd)		
0.1%	0.010298	0.052760	84.3%	124.5	calc3_.REGION@li.96 (ovhd)		
0.1%	0.010287	0.068428	87.6%	125.0	calc1_.REGION@li.69 (ovhd)		
=====							
0.0%	0.000027	0.000128	83.0%	0.8	PTHREAD		
					pthread_create		
=====							

OpenMP Parallel DOs
 <function>.<region>@<line>
 automatically instrumented

OpenMP overhead is normally small and is filtered out on the default report (< 0.5%). When using "-T" the filter is deactivated

Hardware Counters Information at Loop Level

```
=====
USER / calc3_.LOOP@li.96
-----
```

```
Time%                37.3%
Time                 6.826587 secs
Imb.Time             0.039858 secs
Imb.Time%            0.6%
Calls                72.9 /sec          498.0 calls
DATA_CACHE_REFILLS:
  L2_MODIFIED:L2_OWNED:
  L2_EXCLUSIVE:L2_SHARED      64.364M/sec      439531950 fills
DATA_CACHE_REFILLS_FROM_SYSTEM:
  ALL                        10.760M/sec      73477950 fills
PAPI_L1_DCM              64.973M/sec      443686857 misses
PAPI_L1_DCA              135.699M/sec      926662773 refs
User time (approx)       6.829 secs      15706256693 cycles  100.0%Time
Average Time per Call    0.013708 sec
CrayPat Overhead : Time  0.0%
D1 cache hit,miss ratios 52.1% hits          47.9% misses
D1 cache utilization (misses) 2.09 refs/miss      0.261 avg hits
D1 cache utilization (refills) 1.81 refs/refill    0.226 avg uses
D2 cache hit,miss ratio   85.7% hits          14.3% misses
D1+D2 cache hit,miss ratio 93.1% hits          6.9% misses
D1+D2 cache utilization  14.58 refs/miss      1.823 avg hits
System to D1 refill      10.760M/sec          73477950 lines
System to D1 bandwidth   656.738MB/sec        4702588826 bytes
D2 to D1 bandwidth       3928.490MB/sec       28130044826 bytes
=====
```

- No support for nested parallel regions
 - To work around this until addressed disable nested regions by setting `OMP_NESTED=0`
 - Watch for calls to `omp_set_nested()`

- If compiler merges 2 or more parallel regions, OpenMP trace points are not merged correctly
 - To work around this until addressed, use `-h thread1`

- We need to add tracing support for barriers (both implicit and explicit)
 - Need support from compilers

- When code is network bound
 - Look at collective time, excluding sync time: this goes up as network becomes a problem
 - Look at point-to-point wait times: if these go up, network may be a problem

- When MPI starts leveling off
 - Too much memory used, even if on-node shared communication is available
 - As the number of MPI ranks increases, more off-node communication can result, creating a network injection issue

- Adding OpenMP to memory bound codes may aggravate memory bandwidth issues, but you have more control when optimizing for cache



Follow-up Slides from Questions

Questions

- How do I increase the traced functions limit for pat_report:
 - -D trace-max=N (N is 1024 by default)
- How do I know which “core” my ranks and threads are pinned to?
 - Set the MPICH_CPUMASK_DISPLAY environment variable at runtime.

```
[PE_0]: cpumask set to 1 cpu on nid00036, cpumask = 00000001
[PE_1]: cpumask set to 1 cpu on nid00036, cpumask = 00000010
[PE_2]: cpumask set to 1 cpu on nid00036, cpumask = 00000100
[PE_3]: cpumask set to 1 cpu on nid00036, cpumask = 00001000
[PE_4]: cpumask set to 1 cpu on nid00036, cpumask = 00010000
[PE_5]: cpumask set to 1 cpu on nid00036, cpumask = 00100000
[PE_6]: cpumask set to 1 cpu on nid00036, cpumask = 01000000
[PE_7]: cpumask set to 1 cpu on nid00036, cpumask = 10000000
```

Memory Allocation: First Touch

- Linux has a “first touch policy” for memory allocation
 - *alloc functions don’t actually allocate your memory
 - Memory gets allocated when “touched”
- Problem: A code can allocate more memory than available
 - Linux assumes “swap space,” we don’t have any
 - Applications won’t fail from over-allocation until the memory is finally touched
- Problem: Memory will be put on the core of the “touching” thread
 - Only a problem if thread 0 allocates all memory for a node
- Solution: Always initialize your memory in a threaded region immediately after allocating it
 - If you over-allocate, it will fail immediately, rather than a strange place in your code
 - If every thread touches its own memory, it will be allocated on the proper socket / die.