Cray Programming Environment for XE/XK7 Systems

Heidi Poxon
Technical Lead & Manager, Performance Tools
Cray Inc.

Agenda

- **Overview of the Cray programming environment**
  - Compiler
  - Message Passing Toolkit
  - Performance Tools
  - Scientific Libraries
  - Debugging Tools

- **The Cray X86/GPU programming environment**

- **Using Cray and 3rd Party Compilers**

Cray Programming Environment
Focus on Performance and Productivity –

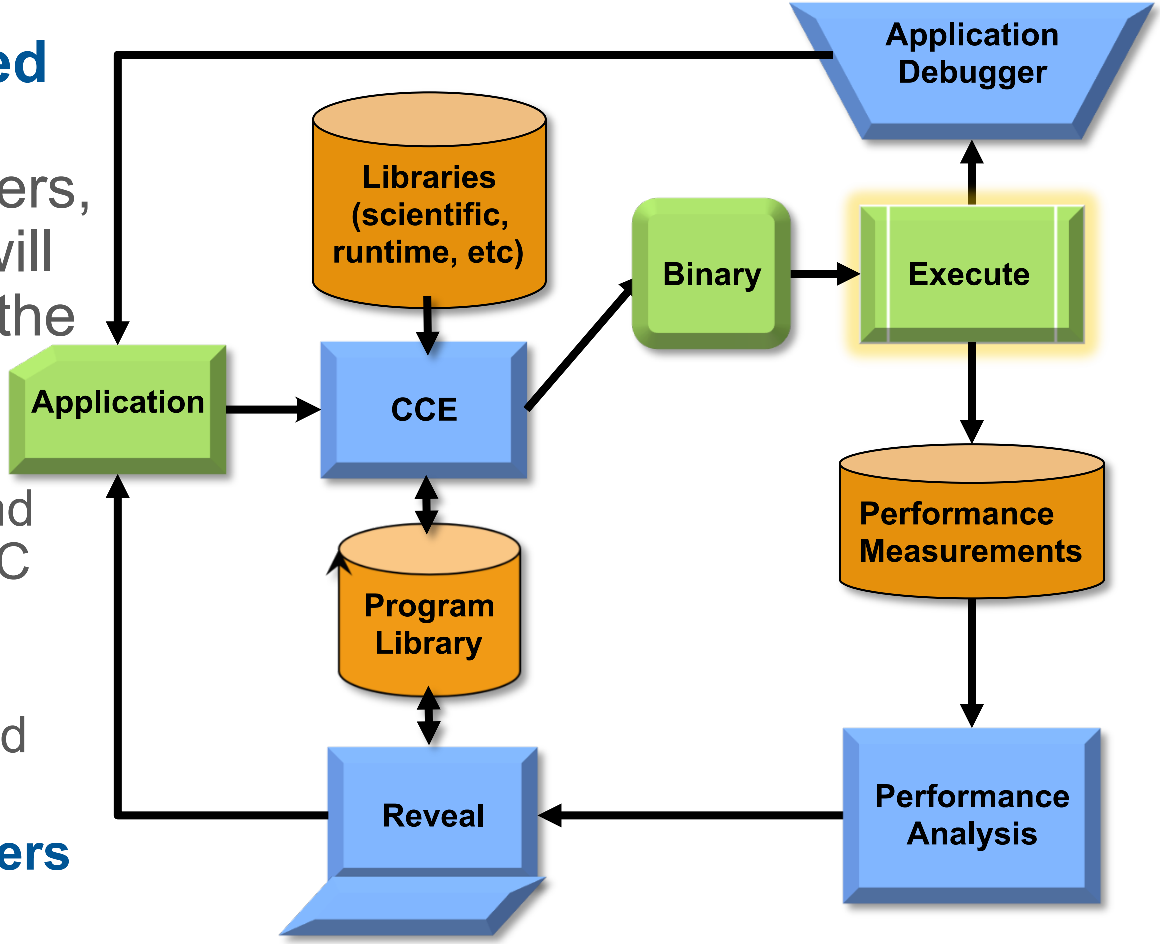
Programming Languages	Programming models	Compilers	Tools	Optimized Scientific Libraries	I/O Libraries
Fortran	Distributed Memory (Cray MPT) • MPI • SHMEM	Cray Compiling Environment (CCE)	**Environment setup**: Modules	LAPACK	NetCDF
C	Shared Memory • OpenMP 3.0 • OpenACC	GNU	**Debuggers**: TotalView Technologies; DDT; lgdb	ScaLAPACK	HDF5
C++	PGAS & Global View • UPC (CCE) • CAF (CCE) • Chapel	3rd Party Compilers	**Debugging Support Tools**: • Abnormal Termination Processing; STAT	BLAS (libgoto)	
Python			**Performance Analysis**: • CrayPat • Cray Apprentice²	Iterative Refinement Toolkit	
			Scoping Analysis: Reveal	Cray Adaptive FFTs (CRAFFT)	
				FFTW	
				Cray PETSc (with CASK)	
				Cray Trilinos (with CASK)	

Cray developed
Licensed ISV SW
3rd party packaging
Cray added value to 3rd party

Cray Programming Environment Vision

- It is the role of the Programming Environment to **close the gap** between observed performance and achievable performance

- Provide a **tightly coupled** high level programming environment with compilers, libraries, and tools that will **hide the complexity** of the system
  - Address issues of scale and complexity of high end HPC systems
  - Target **ease of use** with extended **functionality** and increased **automation**
  - Close **interaction with users**
    - For feedback targeting functionality enhancements



Cray Programming Environment Roadmap

The Cray Compiling Environment

- **Cray technology focused on scientific applications**
  - Takes advantage of **automatic vectorization**
  - Takes advantage of **automatic shared memory parallelization**

- **Standard conforming languages and programming models**
  - **Fortran 2008 standard compliant**
    - Fortran 2008 compliance planned for CCE 8.1 (3Q12)
  - C++98/2003 compliant
  - **OpenMP 3.0 compliant**, working on OpenMP 3.1 and OpenMP 4.0

- **OpenMP and automatic multithreading fully integrated**
  - Share the same runtime and resource pool
  - Aggressive loop restructuring and scalar optimization done in the presence of OpenMP
  - Consistent interface for managing OpenMP and automatic multithreading

- **PGAS languages (UPC & Fortran Coarrays) fully optimized and integrated into the compiler**
  - UPC 1.2 and Fortran 2008 coarray support
  - No preprocessor involved
  - Target the network appropriately
  - Full debugger support with Allinea's DDT

Cray MPI & Cray SHMEM

- **MPI**
  - Implementation based on MPICH2 from ANL
  - Optimized Remote Memory Access (one-sided) fully supported including passive RMA
  - Full MPI-2 support with the exception of
    - Dynamic process management (MPI_Comm_spawn)
  - MPI3 Forum active participant
- **Cray SHMEM**
  - Fully optimized Cray SHMEM library supported
    - Cray XT/XE implementation close to the T3E model
    - Cray XE Implementation on top of the Distributed Memory Applications API (DMAPP)
  - Recent enhancements include:
    - Leveraging local memory access through Cross Process Memory Mapping (XPMEM)
      - Provides the ability for one process to map arbitrary portions of another local process
    - Distributed locking
    - Collectives optimization

Cray Performance Tools

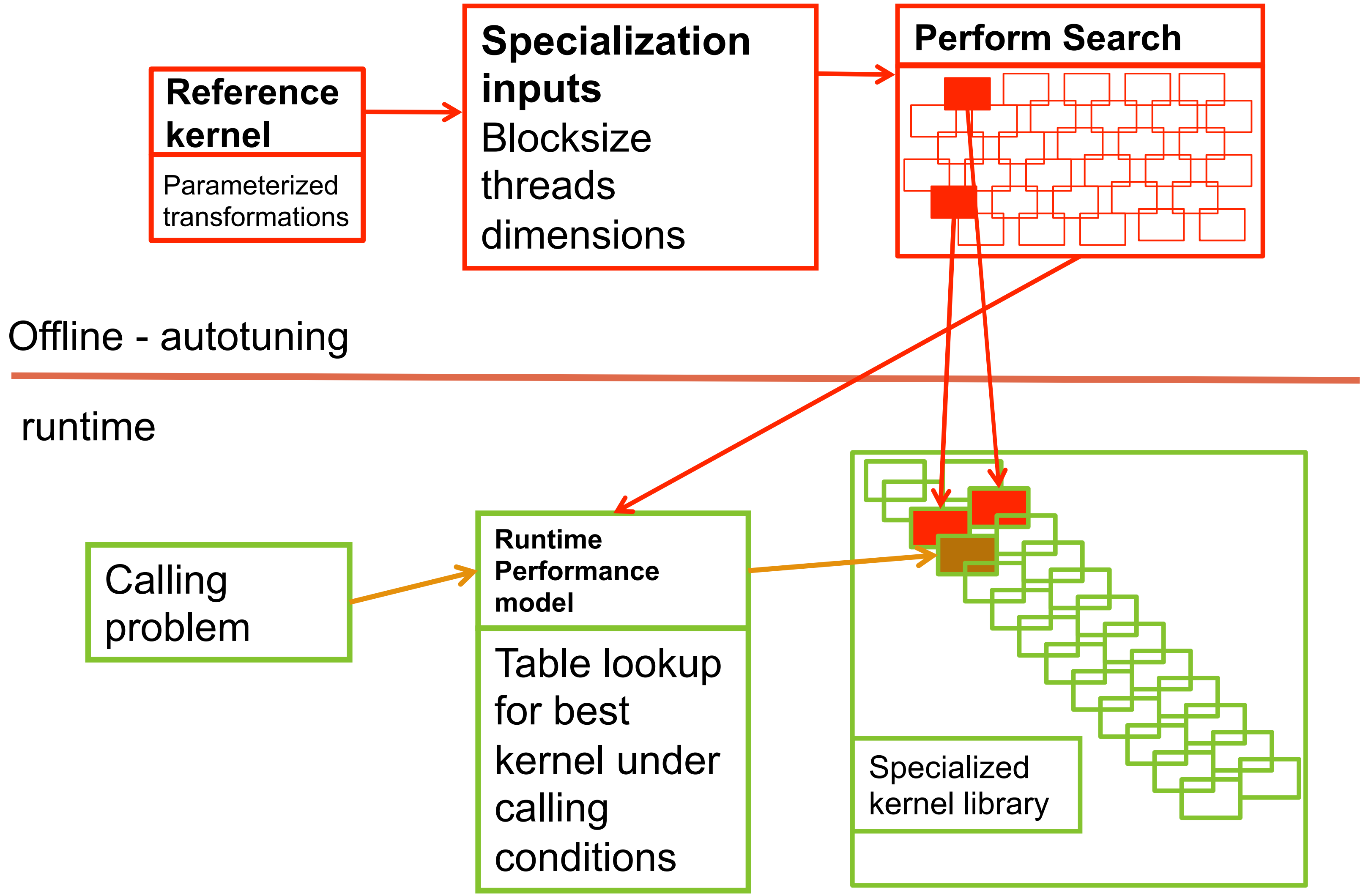
- From performance measurement to performance analysis
- Extend performance measurement tools to assist with optimization (observations, CCE compiler optimization information)
- Focus on **automation** (simplify tool usage, provide feedback based on analysis)
- Enhance support for multiple programming models within a program (MPI, PGAS, OpenMP, OpenACC, SHMEM)
- Improve **scaling** (larger jobs, more data, better tool response)
- Support new processors and interconnects

Adaptive Scientific Libraries

- **Scientific Libraries today have three concentrations to increase productivity with enhanced performance**
  - **Standardization**
  - **Autotuning**
  - **Adaptive Libraries**

- **Cray adaptive model**
  - Runtime analysis allows **best** library/kernel to be **used dynamically**
  - Extensive offline testing allows **library to make decisions** or remove the need for those decisions
  - Decision depends on the system, on previous performance info, obtained previously, and characteristics of calling problem

Adaptation, Auto-tuning and Specialization



This is all invisible to the user :: all you will see is good performance

The Next Generation of Debuggers on Cray Systems

- **Systems with hundreds of thousands of threads of execution need a new debugging paradigm**
  - Innovative techniques for productivity and scalability
    - Scalable Solutions based on MRNet from University of Wisconsin

      STAT - Stack Trace Analysis Tool
      - Scalable generation of a single, merged, stack backtrace tree
        - running at 216K back-end processes

      ATP - Abnormal Termination Processing
        - Scalable analysis of a sick application, delivering a STAT tree and a minimal, comprehensive, core file set.
    - Fast Track Debugging
      - Debugging optimized applications
      - Added to Allinea's DDT 2.6 (June 2010)
  - Support for traditional debugging mechanism
    - TotalView, DDT, and gdb

Unified X86/GPU Programming Environment

The New Generation of Supercomputers

- **Hybrid multicore has arrived and is here to stay**
  - Fat nodes are getting fatter
  - Accelerators have leapt into the Top500
- **Programming accelerators efficiently is hard**
  - Three levels of parallelism required
    - MPI between nodes or sockets
    - Shared memory programming on the node
    - Vectorization for low level looping structures
  - Need a hybrid programming model to support these new systems
  - Need a high level programming environment
    - Compilers, tools, & libraries

Cray Vision for Accelerated Computing

- **Most important hurdle for widespread adoption of accelerated computing is programming difficulty**
  - Need a single programming model that is **portable across machine types**, and also **forward scalable** in time
    - Portable expression of heterogeneity and multi-level parallelism
    - Programming model and optimization should not be significantly difference for “accelerated” nodes and multi-core x86 processors
    - **Allow users to maintain a single code base**
- **Cray's approach to Accelerator Programming is to provide an** ease of use **tightly coupled high level programming environment with compilers, libraries, and tools that will hide the complexity of the system**
- **Ease of use is possible with**
  - Compiler makes it feasible for users to write applications in Fortran, C, C++
  - Tools to help users port and optimize for heterogeneous systems
  - Auto-tuned scientific libraries

Programming for a Node with Accelerator

- **Fortran, C, and C++ compilers**
  - **OpenACC directives to drive compiler optimization**
  - Compiler does the “heavy lifting” to split off the work destined for the accelerator and perform the necessary data transfers
  - Compiler optimizations to take advantage of accelerator and multi-core X86 hardware appropriately
  - Advanced users can mix CUDA functions with compiler-generated accelerator code
  - Debugger support

- **Cray Reveal, built upon an internal compiler representation of the application (the CCE Program Library)**
  - Source code browsing tool that provides interface between the user, the compiler, and the performance tool
    - **Scoping tool** to help users port and optimize applications
    - **Performance measurement and analysis** information for porting and optimization

- **Scientific Libraries support**
  - Auto-tuned libraries (using Cray Auto-Tuning Framework)

OpenACC Accelerator Programming Model

- **Why a new model? There are already many ways to program:**
  - CUDA and OpenCL
    - All are quite low-level and closely coupled to the GPU
  - PGI CUDA Fortran
    - Still CUDA just in a better base language
  - PGI accelerator directives, CAPS HMPP
    - First steps in the right direction – Needed standardization

- **User needs to write specialized kernels:**
  - **Hard** to write and debug
  - **Hard** to optimize for specific GPU
  - **Hard** to update (porting/functionality)

- **OpenACC Directives provide high-level approach**
  - **Simple programming model for heterogeneous systems**
  - **Easier to maintain/port/extend code**
    - The same source code can be compiled for multicore CPU
  - Based on the work in the OpenMP Accelerator Subcommittee
    - Proposed to the OpenMP Language Committee
      - Subcommittee of OpenMP ARB, aiming for OpenMP 4.0
  - Possible performance sacrifice
    - A small performance gap is acceptable (do you still hand-code in assembler?)
    - Goal is to provide at least 90% of the performance obtained with hand coded CUDA
      - Already seeing this in many cases, more tuning ongoing

- http://www.openacc.org/


The OpenACC™ API
QUICK REFERENCE GUIDE

The OpenACC Application Program Interface describes a collection of compiler directives to specify loops and regions of code in standard C, C++ and Fortran to be offloaded from a host CPU to an attached accelerator, providing portability across operating systems, host CPUs and accelerators.

Most OpenACC directives apply to the immediately following structured block or loop; a structured block is a single statement or a compound statement (C or C++) or a sequence of statements (Fortran) with a single entry point at the top and a single exit at the bottom.

CAPS
CRAY THE SUPERCOMPUTER COMPANY
NVIDIA
PGI

Version 1.0, November 2011

© 2011 OpenACC-standard.org all rights reserved.

Using Cray and 3rd Party Compilers

Modules

- **Access to software is managed using the GNU module command**
  - To see which modules are currently loaded, type: "**module list**"
  - To see which modules are available, type: "**module avail**"
    - You can wildcard the end of the names, e.g.: "**module avail PrgEnv***"
    - For more complicated grepping, you need to redirect stderr to stdout, e.g.
      - **module avail 2>&1 | grep "Env"**
  - You load a new module by typing: "**module load <module name>**"
  - Some modules (e.g. different compiler versions) conflict, so you should first "**module unload**" the old version (or use "**module swap**")

Modules (2)

- **To access the different compilers:**
  - You select these by loading a Programming Environment (PE) module
    - PrgEnv-cray for CCE (the default)
    - PrgEnv-pgi for PGI
    - PrgEnv-gnu for GNU
  - Once one of these is loaded, you can then select a compiler suite
    - CCE: **module avail cce**
    - PGI: **module avail pgi**
  - For GPU programming (CUDA, OpenACC...)
    - Make sure you target the GPU when building:
      - **Example: module load craype-accel-nvidia35**

Compiler Choices – Relative Strengths

- **CCE – Outstanding Fortran, very good C, and improving C++**
  - Very good vectorization
  - Very good Fortran language support; only real choice for Coarrays
  - C support is quite good, with UPC support
  - Very good scalar optimization and automatic parallelization
  - Clean implementation of OpenMP 3.0, with tasks
  - Sole delivery focus is on Linux-based Cray hardware systems
  - Best bug turnaround time (if it isn't, let us know!)
  - Cleanest integration with other Cray tools (performance tools, debuggers, upcoming productivity tools)
  - No inline assembly support
  - OpenACC support for accelerators

- **GNU pretty-good Fortran, outstanding C and C++ (if you ignore vectorization)**
  - Very good scalar optimizer
  - Vectorization capabilities focus mostly on inline assembly
  - De-facto C++ compiler (for better or worse)

Compiler Choices – Relative Strengths (2)

- **PGI – Very good Fortran and C, pretty good C++**
  - Good vectorization
  - Good functional correctness with optimization enabled
  - Good manual and automatic prefetch capabilities
  - Very interested in the Linux HPC market, although that is not their only focus
  - Excellent working relationship with Cray, good bug responsiveness
  - OpenACC support for accelerators

- **Intel – Good Fortran, excellent C and C++ (if you ignore vectorization)**
  - Automatic vectorization capabilities are modest, compared to PGI and CCE
  - Use of inline assembly is encouraged
  - Focus is more on best speed for scalar, non-scaling apps
  - Tuned for Intel architectures, but actually works well for some applications on AMD
  - Does not support the Interlagos FMA instruction, so achievable floating point performance is cut in half

Using the Compilers

- **Cray Systems come with compiler wrappers to simplify building parallel applications (similar the `mpicc`/`mpif90`)**
  - Fortran Compiler: `ftn`
  - C Compiler: `cc`
  - C++ Compiler: `CC`
- **Using these wrappers ensures that your code is built for the compute nodes and linked against important libraries**
  - Cray MPT (MPI, Shmem, etc.)
  - Cray LibSci (BLAS, LAPACK, etc.)
  - …
- **Do not call the PGI, Cray, etc. compilers directly**
- **Cray Compiler wrappers try to hide the complexities of using the proper header files and libraries**
  - So does autoconf (./configure) and CMake, so unfortunately, sometimes these tools need massaging to work with compiler wrappers, especially in a cross-compiling environment, like titan

Using the Cray Compiler

- **To access the Cray compiler**
  - module load PrgEnv-cray
  - For titan: module swap PrgEnv-pgi PrgEnv-cray
- **To target the various chip**
  - module load craype-interlagos (loaded by default)
- **To enable OpenACC**
  - module load craype-accel-nvidia35
- **Once you have loaded the module “cc” and “ftn” are the Cray compilers**
  - Recommend just using default options
- **man crayftn**

Some Cray Compilation Environment Basics

- **CCE-specific features:**
  - Optimization: **-O2** is the default and you should usually use this
  - OpenMP is supported by default (no flag needed to enable)
    - if you don't want it, use either **-hnoomp** or **-xomp** compiler flags
  - OpenACC is supported by default if GPU targeting module (craype-accel-nvidia*) is loaded
  - CCE only gives minimal information to stderr when compiling
    - To see more information, you should request a compiler listing file
      - flags **-ra** for ftn or **-hlist=a** for cc
      - writes a file with extension .lst
      - contains annotated source listing, followed by explanatory messages
    - Each message is tagged with an identifier, e.g.: **ftn-6430**
      - to get more information on this, type: **explain <identifier>**
    - Cray Reveal can display all this information (and more)

Compiler Feedback

- **Compiler feedback is extremely important**
  - Did the compiler recognise the accelerator directives?
    - A good sanity check
  - How will the compiler move data?
    - Only use data clauses if the compiler is over-cautious on the copy*
    - Or you want to declare an array to be scratch (create clause)

    - The first main code optimisation is removing unnecessary data movements
  - How will the compiler schedule loop iterations across GPU threads?
    - Did it parallelise the loopnests?
    - Did it schedule the loops sensibly?

    - The other main optimisation is correcting obviously-poor loop scheduling

- **Compiler teams work very hard to make feedback useful**
  - **Advice: use it, it's free!** (i.e. no impact on performance to generate it)
    - CCE: ftn -ra ; cc -hlist=a Produces commentary files <stem>.lst
    - PGI: ftn -Minfo ; cc -Minfo Feedback to STDERR

Example: Cray Loopmark Messages

ftn –rm …   or   cc –hlist=m …

```
 29.  b-------<        do i3=2,n3-1
30.  b b-----<           do i2=2,n2-1
31.  b b Vr--<              do i1=1,n1
32.  b b Vr                    u1(i1) = u(i1,i2-1,i3) + u(i1,i2+1,i3)
33.  b b Vr          >                + u(i1,i2,i3-1) + u(i1,i2,i3+1)
34.  b b Vr                    u2(i1) = u(i1,i2-1,i3-1) + u(i1,i2+1,i3-1)
35.  b b Vr          >                + u(i1,i2-1,i3+1) + u(i1,i2+1,i3+1)
36.  b b Vr-->              enddo
37.  b b Vr--<              do i1=2,n1-1
38.  b b Vr                    r(i1,i2,i3) = v(i1,i2,i3)
39.  b b Vr          >                     - a(0) * u(i1,i2,i3)
40.  b b Vr          >                     - a(2) * ( u2(i1) + u1(i1-1) + u1(i1+1) )
41.  b b Vr          >                     - a(3) * ( u2(i1-1) + u2(i1+1) )
42.  b b Vr-->              enddo
43.  b b----->           enddo
44.  b------->        enddo
```

Example: Cray Loopmark Messages (cont)

```
ftn-6289 ftn: VECTOR File = resid.f, Line = 29
  A loop starting at line 29 was not vectorized because a recurrence was
   found on "U1" between lines 32 and 38.
ftn-6049 ftn: SCALAR File = resid.f, Line = 29
  A loop starting at line 29 was blocked with block size 4.
ftn-6289 ftn: VECTOR File = resid.f, Line = 30
  A loop starting at line 30 was not vectorized because a recurrence was
   found on "U1" between lines 32 and 38.
ftn-6049 ftn: SCALAR File = resid.f, Line = 30
  A loop starting at line 30 was blocked with block size 4.
ftn-6005 ftn: SCALAR File = resid.f, Line = 31
  A loop starting at line 31 was unrolled 4 times.
ftn-6204 ftn: VECTOR File = resid.f, Line = 31
  A loop starting at line 31 was vectorized.
ftn-6005 ftn: SCALAR File = resid.f, Line = 37
  A loop starting at line 37 was unrolled 4 times.
ftn-6204 ftn: VECTOR File = resid.f, Line = 37
  A loop starting at line 37 was vectorized.
```

Interoperability

- **OpenACC is a complete programming model**
  - But there are still situations where it is useful to interface OpenACC code with other GPU programming models
- **Why might this be useful?**
  - You want to call accelerated scientific libraries from your code
    - without having to transfer data back and forth between the host
  - You want to call CUDA kernels from your code
    - also without unnecessary data transfers
  - You want to exploit Nvidia GPUdirect (or similar) to streamline communication of data between accelerators.
- **Interfacing requires access to the lower-level information**
  - Typically the GPU memory locations of OpenACC-created data arrays
  - The compiler normally hides this information from the user.

host_data Directive

- **OpenACC runtime manages GPU memory implicitly**
  - user does not need to worry about memory allocation/free-ing
- **Sometimes it can be useful to know where data is held in device memory, e.g.:**
  - so a hand-optimised CUDA kernel can be used to process data already held on the device
  - so a third-party GPU library can be used to process data already held on the device (Cray libsci_acc, cuBLAS, cuFFT etc.)
  - so optimised communication libraries can be used to streamline data transfer from one GPU to another
- **host_data directive provides mechanism for this**
  - nested inside OpenACC data region
  - subprogram calls within host_data region then pass pointer in device memory rather than in host memory

Interoperability with CUDA

- **Why would you want to do this?**

- **Two situations:**
  - You have already ported an application to OpenACC
    - A few key kernels get improved performance using hand-tuned CUDA
      - (performance at the cost of reduced portability)
    - These CUDA kernels should process data that was already placed in GPU memory using OpenACC

  - Or, you have ported a few key kernels to the GPU using CUDA
    - but data movement costs outweigh the performance gain
    - OpenACC provides an efficient way of porting the remainder of the application

CUDA Interoperability

```fortran
PROGRAM main
  INTEGER :: a(N)
  <stuff>
!$acc data copy(a)
! <Populate a(:) on device
!  as before>
!$acc host_data use_device(a)
  CALL dbl_cuda(a)
!$acc end host_data
!$acc end data
  <stuff>
END PROGRAM main
```

```c
__global__ void dbl_knl(int *c) {
  int i = \
        blockIdx.x*blockDim.x+threadIdx.x;
  if (i < N) c[i] *= 2;
}

extern "C" void dbl_cuda_(int *b_d) {
  cudaThreadSynchronize();
  dbl_knl<<<NBLOCKS,BSIZE>>>(b_d);
  cudaThreadSynchronize();
}
```

- **host_data region exposes accelerator memory address on host**
  - Nested inside data region
- **Call CUDA-C wrapper (compiled with nvcc; linked with CCE)**
  - Must include cudaThreadSynchronize()
    - Before: so asynchronous accelerator kernels definitely finished
    - After: so CUDA kernel definitely finished
  - CUDA kernel written as usual
  - Or use same mechanism to call existing CUDA library

Using the NVIDIA Compiler for CUDA

- **Target build for the NVIDIA GPU and access NVIDIA compiler**
  - **module load craype-accel-nvidia35**

- **Compile CPU code with PrgEnv "cc" wrapper**
  - Either **PrgEnv-gnu** for gcc; or **PrgEnv-cray** for craycc

- **Compile GPU CUDA-C kernels with nvcc**
  - **nvcc -O3 -arch=sm_20**

- **Link program with PrgEnv "cc" wrapper**
  - Only GPU flag needed: `-lcudart`
    - e.g. no CUDA `-L` flags needed (added in cc wrapper)

Interoperability with Libraries

- **Why would you want to do this?**
  - You should always use libraries if they are available
    - A lot of effort goes into optimizing them
    - They are likely to use a lot more tricks that you have time/inclination to try
- **Examples of libraries:**
  - Cray libsci_acc
  - cuBLAS
  - cuFFT
  - ...
- **To use these with OpenACC code**
  - Place calls to the library inside host_data regions

Unified X86/GPU Programming Environment

- **The Cray XK7 includes the first-generation of the Cray Unified X86/GPU Programming Environment**

- **The Cray XK7 PE supports three classes of users:**
  1. “Hardcore" GPU programmers with existing CUDA ports
  2. Users with parallel codes, ideally with some OpenMP experience, but less GPU knowledge
  3. Users with serial codes looking for portable parallel performance with and without GPUs

Cray Debugging Support Tools

What is Included?

- STAT (Stack Trace Analysis Tool)
- ATP (Abnormal Termination Processing)
- MRNet (Multicast/Reduction Network)
- FTD (Fast Track Debugging)
  - Supported in lgdb and DDT
- *Coming*: ccdb (Cray Comparative Debugger)

Stack Trace Analysis Tool (STAT)

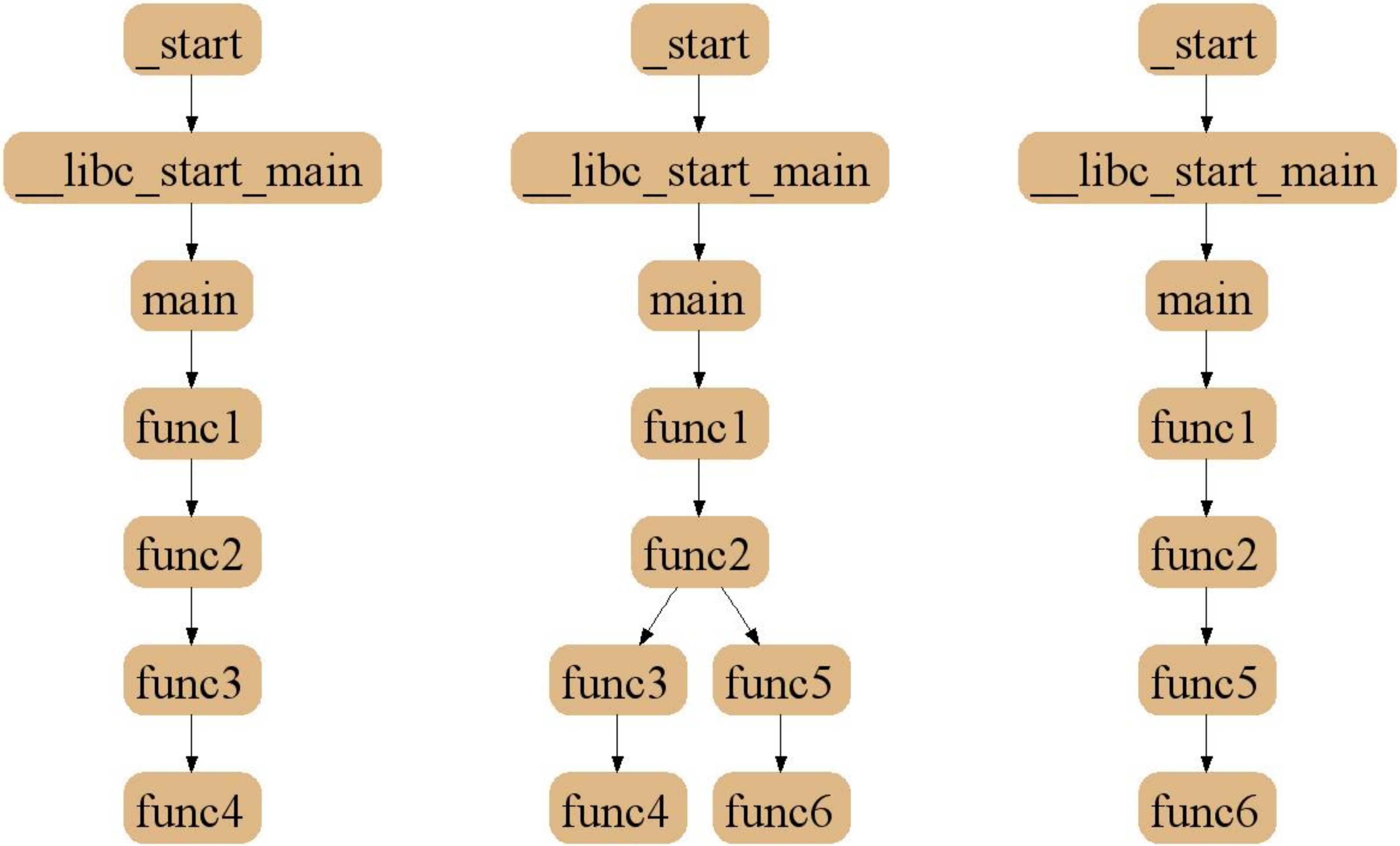
My application hangs!

Stack Trace Analysis Tool (STAT)

- **Stack trace sampling and analysis for large scale applications**
  - Sample application stack traces
  - Scalable generation of a single, merged, stack backtrace tree
    - A comprehensible view of the entire application
    - Discover equivalent process behavior
      - Group similar processes
      - Reduce number of tasks to debug
    - 128K processes analyzed in 2.7 seconds, using MRNet

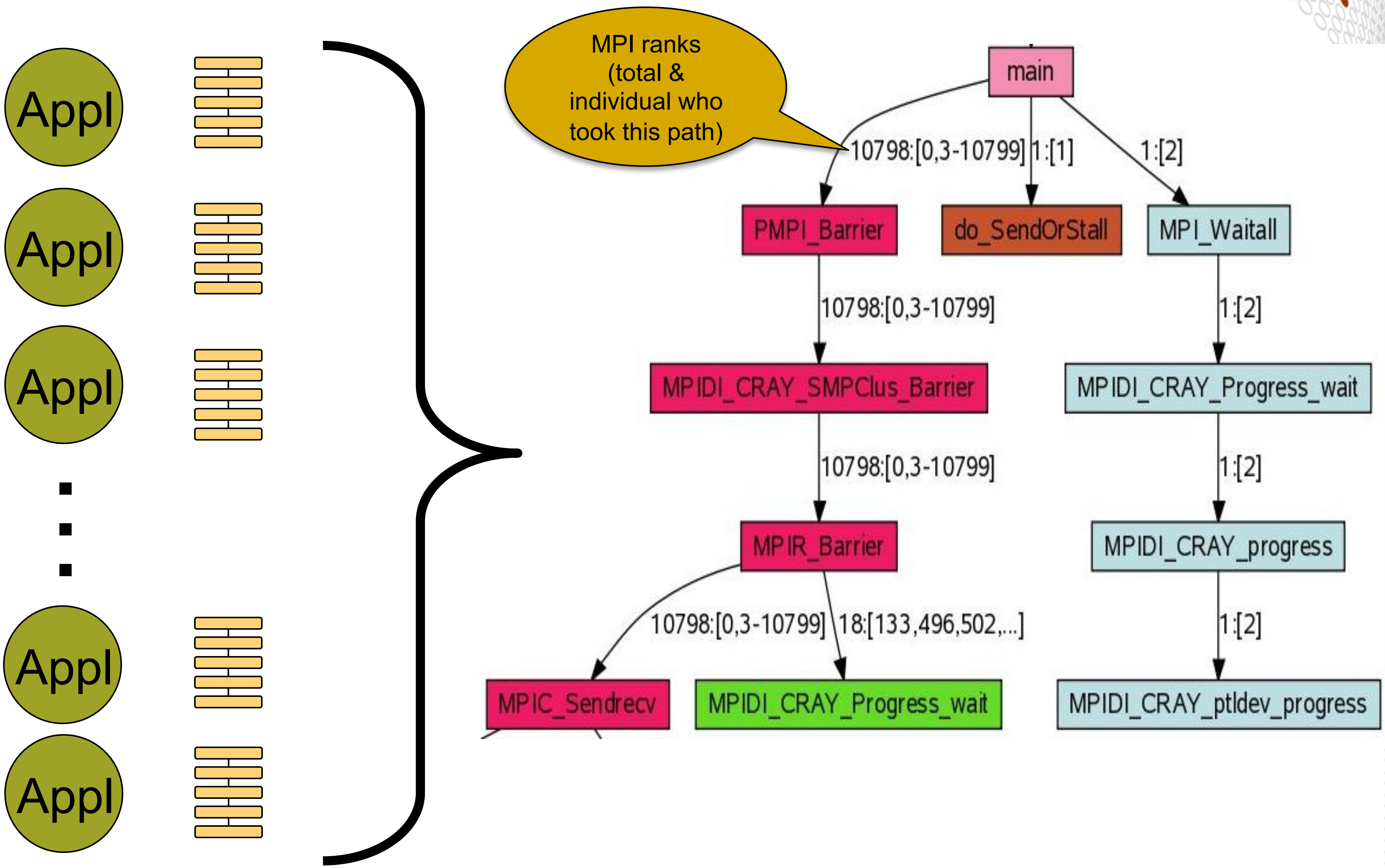
- **Merge/analyze traces:**
  - Facilitate scalable analysis/data presentation
  - Multiple traces over space or time
  - Create call graph prefix tree
    - Compressed representation
    - Scalable visualization
    - Scalable analysis

Stack Trace Merge Example



Courtesy of University of Wisconsin

2D-Trace/Space Analysis

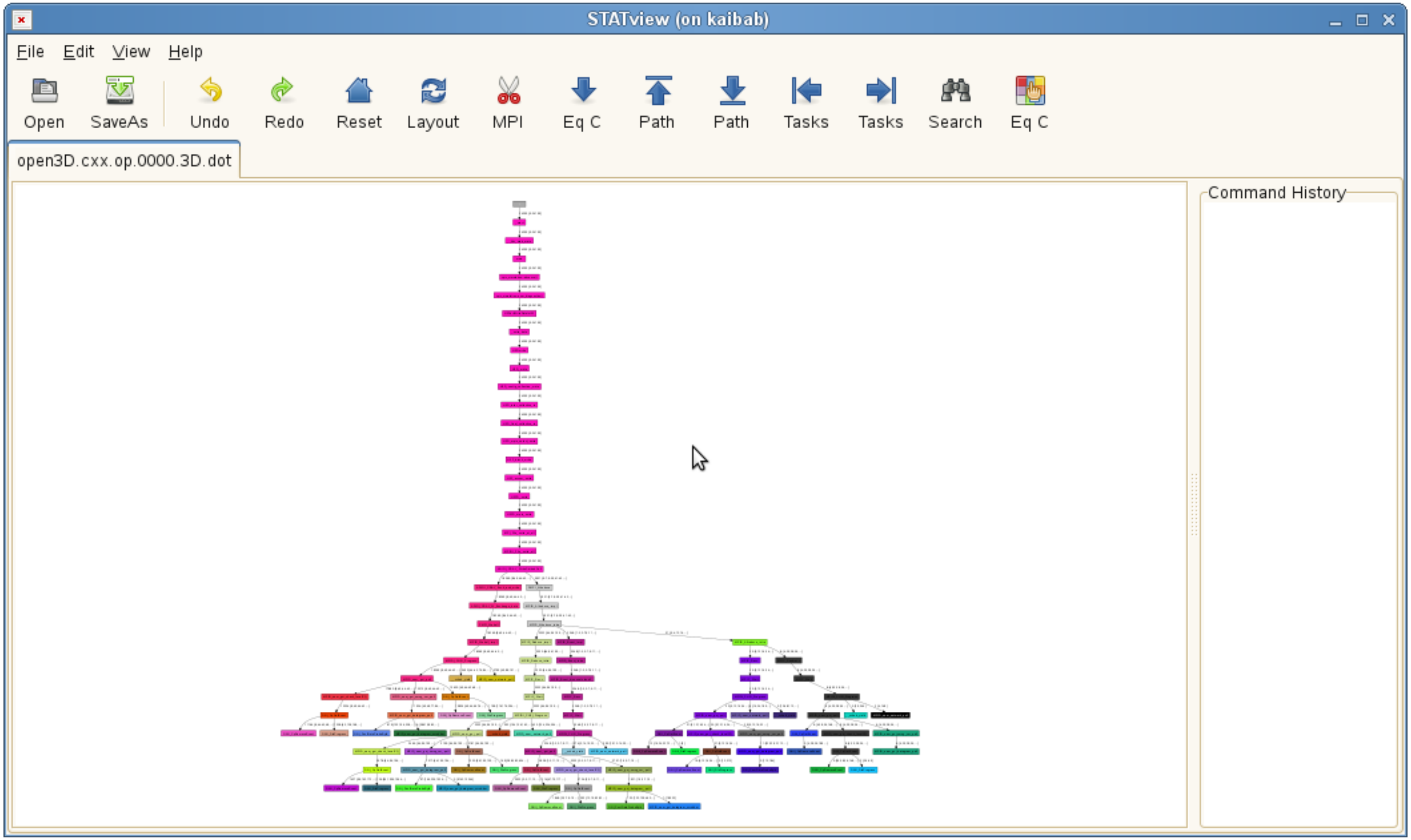


Courtesy of University of Wisconsin

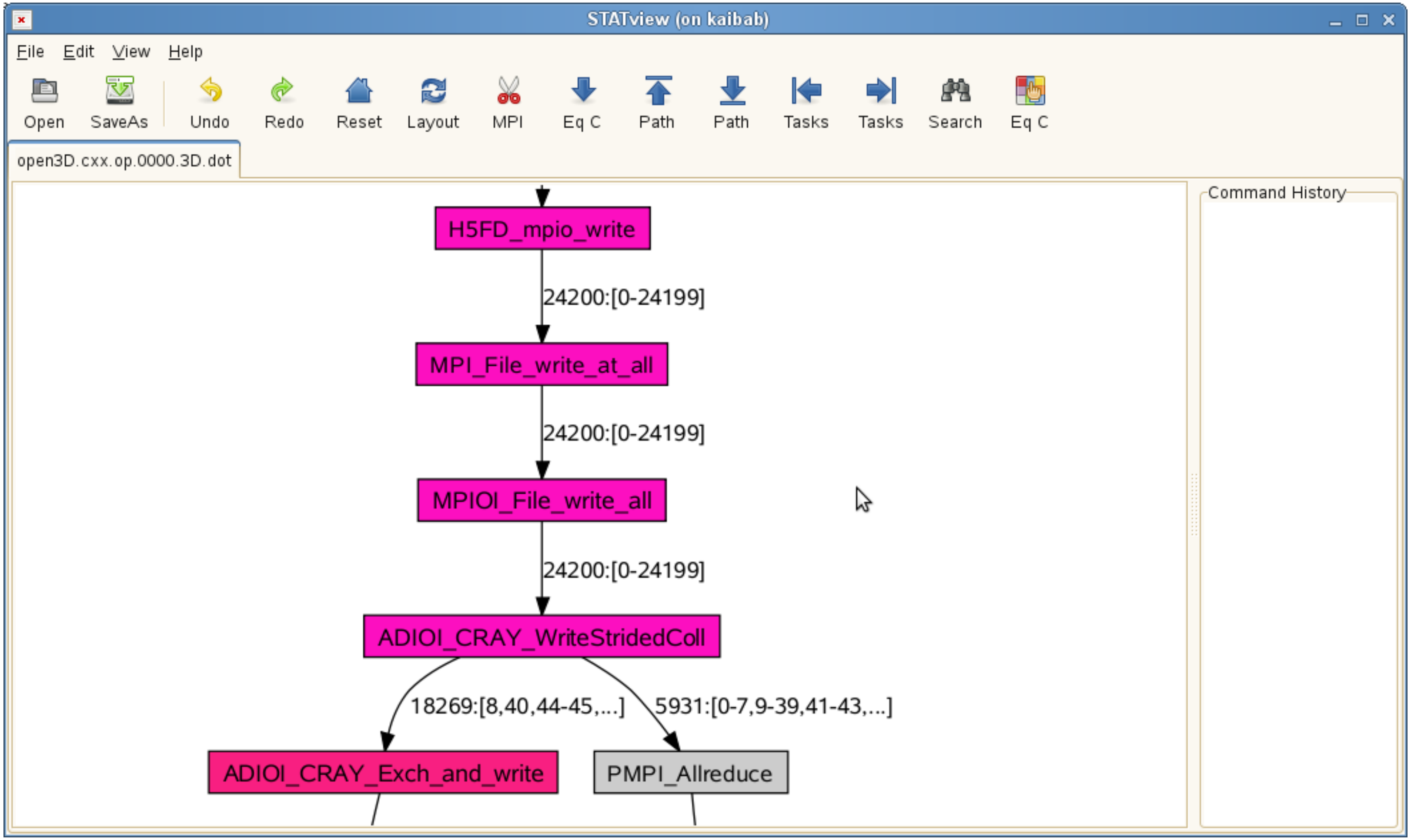
Example: NERSC Plasma Physics Application

- 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

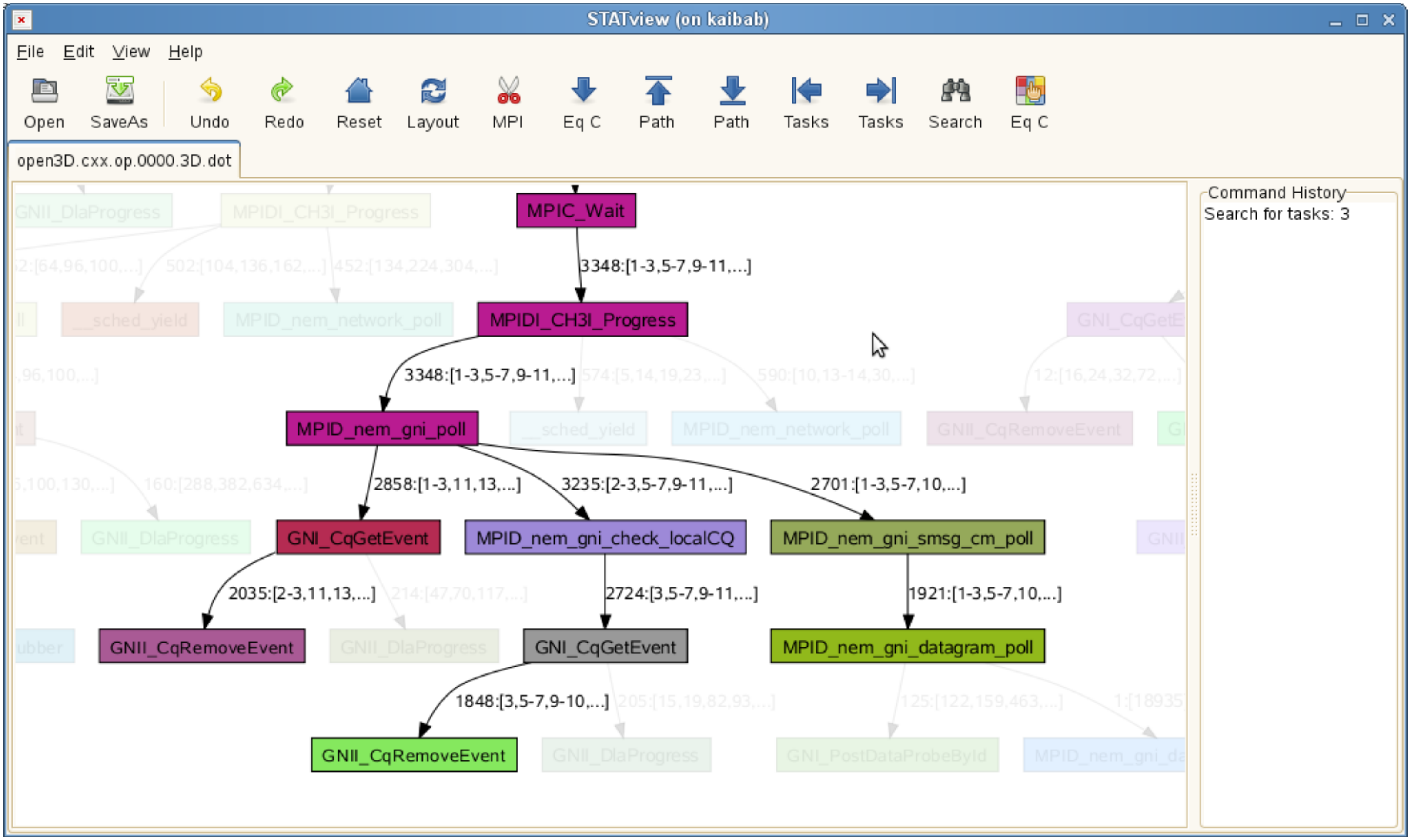
Example Continued - STATview



Example Continued - STATview



Example Continued - STATview



How to Use STAT

Current Release: STAT 1.2.1.3

- **module load stat**
- **man STAT**
- **STAT <pid_of_aprun>**
  - Creates STAT_results/<app_name>/<merged_st_file>
  - apstat command can help find aprun PID
- **statview <merged_st_file>**
- **statgui**

How to Use STAT (2)

- **STATGUI – GUI that drives STAT**
  - Work bench for repeated requests
  - Allows you to
    - Change granularity
    - Change sampling
    - Continue then resample
  - Launches or attaches to application

Abnormal Termination Processing (ATP)

My application crashes!

The Problem Being Solved

- Applications on Cray systems use hundreds of thousands of processes
- On a crash one, many, or all processes might trap
- No one wants that many core files
- No one wants that many stack backtraces
- Accessing and comprehending many stack backtraces is a daunting task

Current Release: ATP 1.6.0

- **Automatic**
  - ATP module loaded by default
    - Signal handler added to application and registered
  - aprun launches ATP in parallel with application launch
  - Enabled/disabled at runtime via ATP_ENABLED environment variable (can be set by site and/or by user)

- **Provides:**
  - backtrace of first crash to stderr
  - merged backtrace trees
  - dumps core file set (if limit/ulimit allows)
  - Summary of ranks that faulted included in tree

- **Tested at 15K PEs**