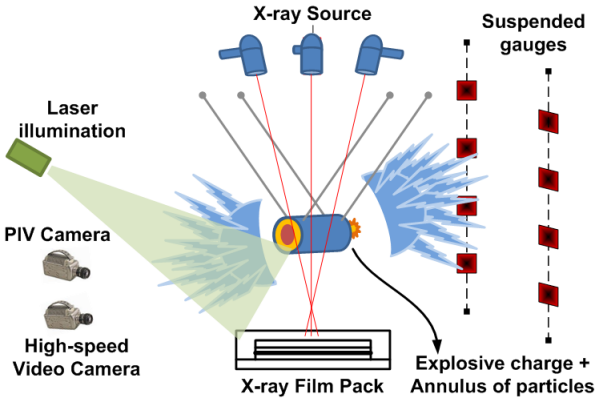
CCMT

CENTER FOR COMPRESSIBLE MULTIPHASE TURBULENCE

CCMT Workshop

February 19, 2015



UF UNIVERSITY of FLORIDA

NNSA

UF UNIVERSITY of FLORIDA

CCMT Workshop

University of Florida
February 19, 2015

Attendee List

Jackson,Thomas L
Balachandar, Sivaramakrishnan
Subramanian Annamalai
Shringarpure,Mrugesh S
Hackl,Jason
Park,Chanyoung
Fernandez,Maria G
Neal,Christopher R
Mehta,Yash A
Durant,Bradford A
Saptarshi Biswas
Ouellet,Frederick
Koneru,Rahul Babu
Sridharan,Prashanth
Cook,Charles R

CCMT agenda

- Lunch (pizza and drinks) - 12:00 - 12:15pm
- Basics of Good Software Engineering - Bertrand - 12:45am-1pm
- Paraview and Python Scripting for Paraview - Chris and Brad - 1pm-1:30pm
- A quick guide through the DoE supercomputers - Chris - 1:30-2:00pm
- Code Memory Profiling with Valgrind - Charles - 2:00-2:30pm
- Code Debugging with Totalview - Bertrand - 2:30-3:00pm
- Code Testing and Validation - Mrugesh - 3:00-3:30pm
- Version Control with CVS - Tania - 30min - 3:30-4:00pm
- Code Performance Profiling with Tau - Tania - 4:00-4:30pm
- UQ with Dakota - Chanyoung - 4:30-5:00pm

CCMT

Good Software Engineering Practices

Bertrand Rollin

CCMT – 02/19/15

Disclaimer: This presentation is largely inspired from presentations of ATPESC 2013 & 2014. This presentation is intended for CCMT internal use only.

Today's program

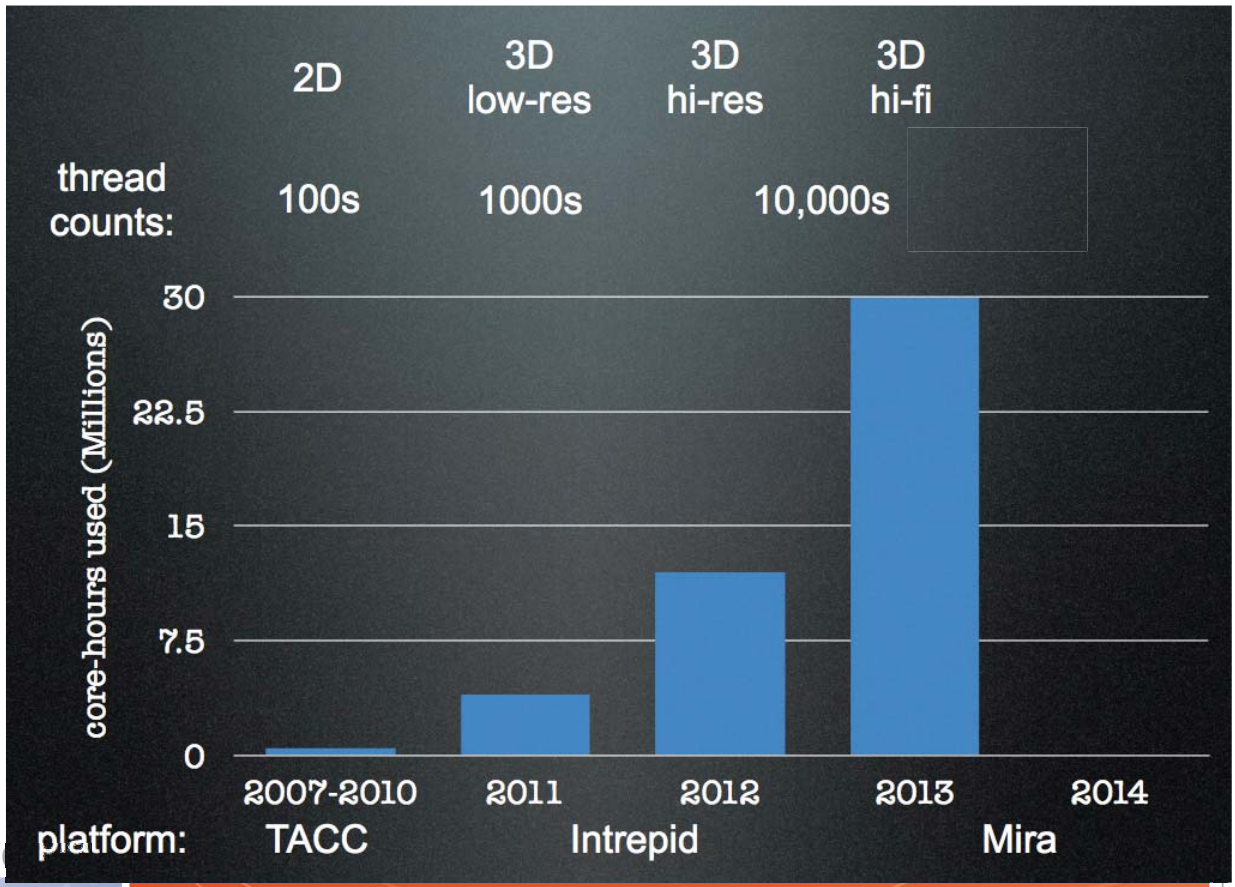
- Basics of Good Software Engineering - Bertrand – 12:15am-1pm
- Paraview and Python Scripting for Paraview - Chris and Brad - 1pm-1:30pm
- A quick guide through the DoE supercomputers - Chris - 1:30-2:00pm
- Code Memory Profiling with Valgrind - Charles - 2:00-2:30pm
- Code Debugging with Totalview - Bertrand - 2:30-3:00pm
- Code Testing and Validation - Mrugesh - 3:00-3:30pm
- Version Control with CVS - Tania - 30min - 3:30-4:00pm
- Code Performance Profiling with Tau - Tania - 4:00-4:30pm
- UQ with Dakota - Chanyoung - 4:30-5:00pm

Overview

The purpose of this seminar is to make you aware of what a modern research scientist should do/"master" to be productive in his research effort:

- I am going to introduce the some important practices of software construction and management
- making specific suggestions for practices that you should be following

What you should aim for (S.M. Couch)



The 8 Essential Practices

1. Write Programs for People, Not Computers
2. Let the Computer Do the Work
3. Make Incremental Changes
4. Don't Repeat Yourself
5. Plan for Mistakes
6. Document Design and Purpose, Not Mechanics
7. Design Flexibly for Performance, Build Accessibly for Correctness
8. Collaborate

Write Programs for People, Not Computers

- A program should not require its readers to hold more than a handful of facts in memory at once.
- Make names consistent, distinctive, and meaningful.
- Make code style and formatting consistent.

```
! ************************************************************************
!
! Purpose: Read in user input for various specific tasks.
!
! Description: None.
!
! Input:
!   global      Pointer to global data
!
! Notes: None.
!
! ************************************************************************
!
! $Id: ReadMiscSection.F90,v 1.2 2015/02/10 18:24:22 brollin Exp $
!
! Copyright: (c) 2007-2008 by the University of Illinois
!
! ************************************************************************

SUBROUTINE ReadMiscSection(global)

  USE ModDataTypes
  USE ModGlobal, ONLY: t_global
  USE ModError
  USE ModParameters

  USE ModInterfaces, ONLY: ReadSection

  IMPLICIT NONE

! ************************************************************************
! Declarations and definitions
! ************************************************************************
```

Let the Computer Do the Work

- Make the computer repeat tasks.
- Save recent commands in a file for re-use.
- Use a build tool to automate workflows

```
#!/bin/bash

################################################################################
#
# Purpose: Add zero to rflupost output files in the exponential part of the file
#          name.
#
#
# Description: Adds an additional zero to the exponential part of the rflupost
#              output in order for output to be readable by Tecplot macros.
#              Example: cylds_3.00035E-02.plt ---> cylds_3.00035E-002.plt
#
#
#
#
#
# Author: Christopher Neal
#
################################################################################

echo "NOTICE: A different version of rflupost was used for this script"
echo "Change rflupost call to the name of the executable that you use"
sleep 30s

#--------------------------RFLUPOST SECTION-------------------------------------
echo "Running RFLUpost on Solution Files"
sleep 10s

#Put all files in current directory into a text file
for f in *; do echo "$f"; done >temp.txt

#Print the occurences of the first processor solution files to a new file
sed -n "/.mixt.cva_00001_/p" temp.txt >temp2.txt
```


Let the Computer Do the Work

Other than
Googling for things,
the majority of scientists
do not use computers
more effectively today
than they did 28 years ago.

A. Ahmadia

Make Incremental Changes

- Work in small steps with frequent feedback and course correction.
- Use a version control system.
- Put everything that has been created manually in version control.

- organize your personal notes into a personal wiki (gollum, gitit, instiki)
- organize your shared notes into a group wiki (gollum, gitit, instiki)

- use local version control software to checkpoint personal code development
- use **distributed version control** to collaborate with others

See Tania's talk on CVS

Don't Repeat Yourself

- Every piece of data must have a single authoritative representation in the system.
- Modularize code rather than copying and pasting.
- Re-use code instead of rewriting it.

- **Automate common actions by saving simple blocks of code into scripts**
- **Refactor commonly used blocks of code into functions**
- **Group commonly used functions into libraries**

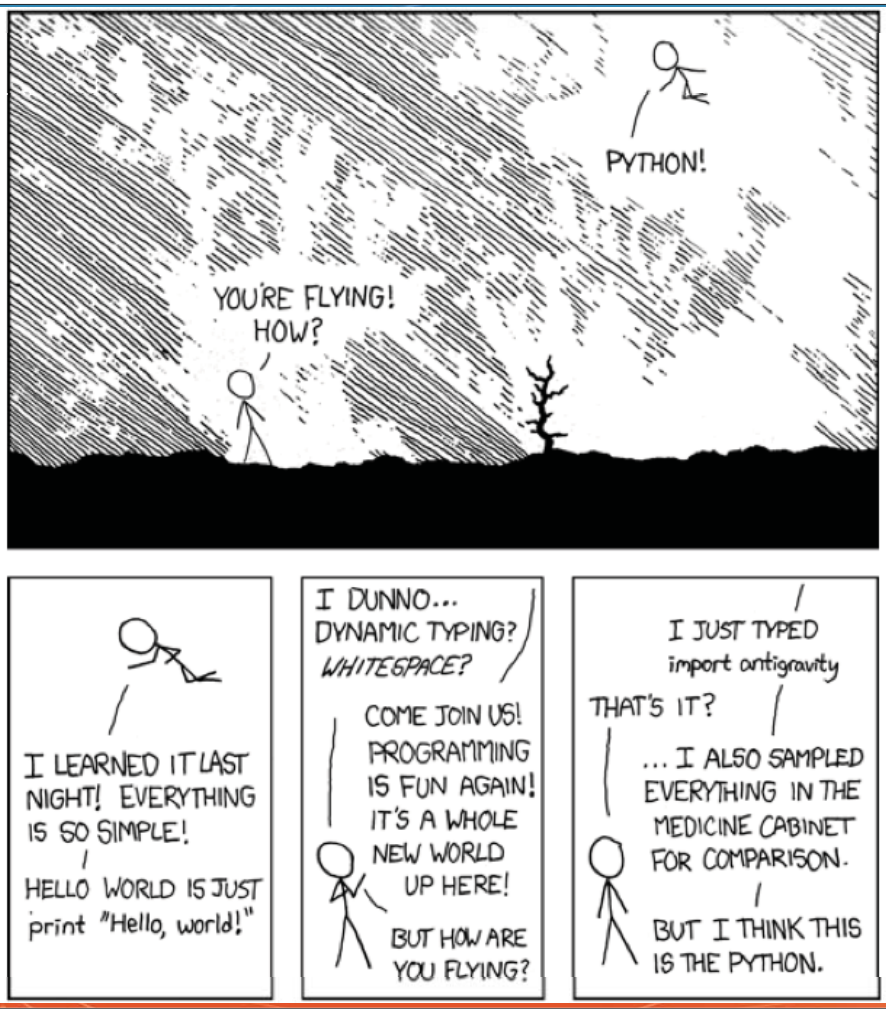
```
RFLU_ModConstraintUtils.F90            RFLU_ModGENXAdmin.F90
RFLU_ModConvertCv.F90                  RFLU_ModGENXIO.F90
RFLU_ModConvertCv.F90-SAVE-2012-05-16  RFLU_ModGENXTools.F90
RFLU_ModCopyData.F90                   RFLU_ModGENXUtils.F90
RFLU_ModDeallocateMemory.F90           RFLU_ModGeometry.F90
RFLU_ModDifferentiationBFaces.F90      RFLU_ModGeometryTools.F90
RFLU_ModDifferentiationCells.F90       RFLU_ModGFM.F90
RFLU_ModDifferentiationFaces.F90       RFLU_ModGlobalIds.F90
```

Share your code with others, and use their code

Resource for unix shell scripting, python, etc:

Software-carpentry.org

See also Chris's and Brad's talk on python scripting



CC

Plan for Mistakes

- Add assertions to programs to check their operation.
- Use an off-the-shelf unit testing library.
- Turn bugs into test cases.
- Use a symbolic debugger.

```
    iRegMax = MAX(iReg1,iReg2)
    iRegMin = MIN(iReg1,iReg2)

    !RFLU_MPI_SetTag = iRegMin + (iRegMax-1)*global%nRegions &
    !                + (iRegMax-1)*(iMsg-1)*global%nRegions*global%nRegions

    RFLU_MPI_SetTag = iRegMin*(2*(global%nRegions-1) - iRegMin + 1)/2 &
                    + (iRegMax - iRegMin) &
                    + (iMsg-1)*global%nRegions*(global%nRegions-1)/2


!   IF ( RFLU_MPI_SetTag > tagMax ) THEN
    IF ( RFLU_MPI_SetTag > tagMax .OR. RFLU_MPI_SetTag < 0 ) THEN
      CALL ErrorStop(global,ERR_MPI_TAGMAX,__LINE__)
    END IF ! RFLU_MPI_SetTag
```

- **verification** - is your code correctly written?
- **validation** - do your computations accurately model the physical phenomena in question?

See Mrugesh's talks

- test frameworks help you verify your code, but validation is usually a manual process
- although it is desirable to write regression tests that verify previously validated results hold true when the code has been modified!
- use the **method of manufactured solutions** to verify correctness of code
- use **comparisons to experiment** to validate code
- use **comparisons to similar software** as an additional check

Document Design and Purpose, Not Mechanics

- Document interfaces and reasons, not implementations.
- Refactor code in preference to explaining how it works.
- Embed the documentation for a piece of software in that software.

```
! ********************************************************************************
!   Get maximum allowed value of tag. NOTE must always use MPI_COMM_WORLD -
!   cannot use global%mpiComm because may be split off in GENx computations
!   and get zero tagMax as a result.
! ********************************************************************************

    !BBR - Begin - Swap CALL name becasue of deprecated MPI function
    ! MPI_Attr_get --> deprecated but, working on all machine, in particular
    ! VULCAN
    CALL MPI_Attr_get(MPI_COMM_WORLD,MPI_TAG_UB,tagMax,dummyLogical,errorFlag)
    ! MPI_MPI_Comm_get_attr --> up to date call, but does not work with MPI
    ! version on VULCAN
    !CALL MPI_Comm_get_attr(MPI_COMM_WORLD,MPI_TAG_UB,tagMax &
    !                      ,dummyLogical,errorFlag)
```

Principles of documentation:

- Save every bit of code you use for generating publishable results
- Document and comment your code for yourself as if you will need to understand it in 6 months
- use README files liberally, as well as file-level, function-level, and inline documentation
- If any piece of code is too complex to easily describe, consider refactoring it

Design Flexibly for Performance, Build Accessibly for Correctness

- Better algorithms beat better architectures
- Write code in the highest-level language possible.
- Use a profiler to identify bottlenecks.

See Tania and Charles talks

- **Be fluent in multiple languages**
  - **You will learn faster by observing and working with others who are more skilled than you**
- **Use domain specific languages and libraries to increase your expressivity**
  - **Aim for languages and tools that allow you to express your models simply.**
  - **Minimize the coupling to external libraries so it is easier to upgrade, replace, or remove them.**

Collaborate

- Use pre-merge code reviews.
- Use pair programming when bringing someone new up to speed and when tackling particularly tricky problems.
- Use an issue tracking tool.

Additional Advice

- **Reduce Complexity:**
  - **Use languages and libraries that reduce the complexity of your work**
  - **It is worth installing a complicated or expensive software tool if your computations are naturally expressed with it**
  - **Always look for opportunities to write less code *you will have to do less initial work (sometimes)***
    - ***you will introduce less bugs***
    - ***your code will be easier to understand and maintain***
  - **When writing software, try to keep individual functions short, single- purpose, and avoid excessive nesting**
- **Aim for reproducibility:**
  - **The goals of non-review scientific publications are to:**
    - ***Describe a new result or approach***
    - ***Convince others of its usefulness***
  - **The reproducibility of your publication will greatly benefit both of these goals**

References

- A. Ahmadia and C. Kees, US Army Engineer Research and Development Center, "**Software Carpentry in High Performance Computing ATPESC 2014",** ATPESC 2013
- Anshu Dubey, LBNL**, "Software Engineering and Process for HPC Scientific Software", ATPESC 2013**
- Sean M. Couch, University of Chicago, **"Petascale Post-doctoral Computing or: How I Learned to Stop Worrying and Blow Up Stars"**, ATPESC 2013
- A. Ahmadia, Software Carpentry, **"Scientific Computing while Supercomputing"**, ATPESC 2013

ATPESC

ATPESC 2015 will be held August 2 – 14.
Application deadline is April 3, 2015.

Apply Now | 2014 ATPESC Videos

Subscribe to our ATPESC 2015 mailing list for important dates and announcements.

Subscribe

ATPESC Buzz

Argonne Training Program on Extreme-Scale Computing Scheduled for August 2-14, 2015

ATPESC helps groom a new generation of supercomputer users

A Q&A with Paul Messina, ALCF Director of Science

Video: Exascale, Data and Biology

CCMT Computing Resources

Chris Neal

Bradford Durant

02/19/2015

What DOE machines do we use?

Lawrence Livermore National Lab
- Vulcan
- Cab
- Surface

Los Alamos National Lab
- Mustang

University of Florida
- HiPerGator

*Others are available, but these are the most commonly used by the center.

LLNL Machines - Vulcan

Vulcan – A BG/Q (PowerPC IBM Architecture) supercomputer

Information

- 393,216 compute cores
- 16GB RAM per node
- 24,576 nodes (16 cores per node)

For more information about computing resources at LLNL visit

- https://computing.llnl.gov/

CCMT https://computing.llnl.gov/tutorials/bgq/images/sequoia5.462pix.jpg

LLNL Machines - Vulcan

Queues

Queue Name	Node Limit	Time Limit	Notes
pdebug	1K	1 hr	Minimum node size =1
psmall	2K	12 hr	Minimum node size =1
pbatch	8K	12 hr	Minimum job size >1K nodes
pall	24K	scheduled	Available through DAT only

DAT = Dedicated Access Time. A user is given control of a large portion of the machine(for a few days at max).

CCMT Table derived from info at: https://computing.llnl.gov/tutorials/bgq/index.html

LLNL Machines - Cab

Cab– An Intel Xeon supercomputer

Information

- 20,736 compute cores
- 32GB RAM per node
- 1296 nodes (16 cores per node)

For more information about Cab and Intel Xeon systems at LLNL visit

- https://computing.llnl.gov/?set=resources&page=OCF_resources#cab

CCMT https://computing.llnl.gov/tutorials/bgq/images/sequoia5.462pix.jpg

LLNL Machines - Cab

Queues

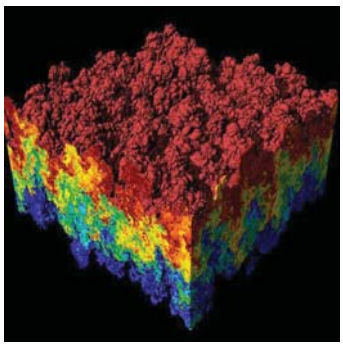
Queue Name	Node Limit	Time Limit (primetime/off primetime	Notes
pdebug	32	30 m / 2 hr	Primetime : 6AM-6PM, M-F
pbatch	258	16 hr / 24 hr	

To see this information for your machine login and type: news job.lim.<machinename> , example: news job.lim.cab

CCMT Table derived from info at: https://computing.llnl.gov/tutorials/bgq/index.html

LLNL Machines - Surface

Surface– A heterogeneous supercomputer with Intel Xeon CPUs and Nvidia Tesla GPUs for scientific visualizations



Information

- 2592 CPU cores (5198 w/hyperthreading)
- 256GB RAM per node + 2 Tesla GPUs per node
- 156 nodes (16 cores per node)

For more information about Surface at LLNL visit

- https://computing.llnl.gov/?set=resources&page=OCF_resources#surface

CCMT http://upload.wikimedia.org/wikipedia/commons/5/54/Rayleigh-Taylor_instability.jpg

LLNL Machines - Surface

Queues

Queue Name	Node Limit	Time Limit	Notes
pbatch	50	24 hr	Try to use less than half of machine

This machine is special because it can see the scratch spaces of Vulcan and Cab.

CCMT Table derived from info at: https://computing.llnl.gov/tutorials/bgq/index.html

LLNL Machines – All of them

Purge Policy

The LLNL File systems are periodically purge of files that haven't been accessed in a while or if the storage space becomes too full.

The lscratch# space where simulations are run is subject to the purge policy.

Possible reasons for purging files
- When system becomes >80% full
- When files are older than 60 days
- For any reason without notification

BACK UP YOUR DATA!

More about LLNL purge policy: https://computing.llnl.gov/tutorials/lc_resources/#PurgePolicies

CCMT http://i.ytimg.com/vi/w12Nm5_Vm98/maxresdefault.jpg

LANL Machines - Mustang

Mustang– An AMD Opteron (x86 Architecture) supercomputer

Information
- 38,400 compute cores
- 64 GB RAM per node
- 1600 nodes (24 cores per node)

For more information about computing resources at LANL visit
- https://ssl-portal.lanl.gov/
- You will need your Z-number and password to view this information.

CCMT http://www.hpcwire.com/2011/10/12/appro_corrals_another_win_at_los_alamos_with_mustang_super/

LANL Machines - Mustang

Queues

Queue Name	Node Limit	Time Limit	Notes
pdebug	1K	1 hr	Minimum node size =1
psmall	2K	12 hr	Minimum node size =1
pbatch	8K	12 hr	Minimum job size >1K nodes
pall	24K	scheduled	Available through DAT only

DAT = Dedicated Access Time. A user is given control of a large portion of the machine(for a few days at max).

CCMT Table derived from info at: https://computing.llnl.gov/tutorials/bgq/index.html

LANL Machines – Mustang

Purge Policy

LANL file systems are also subject to file purging.

The /scratch# space is subject to the purge policy. Files that were last modified 14 days ago will be flagged for deletion, and you should receive an email about which files are being targeted.

\scratch3 will preserve files for 40 days.

Save your files to /archive. This stores the files on the General Parallel File System(GPFS) servers. Access the files using /archive/<username>. If you do not have an archive account request one from the ICN consulting team.

More about LLNL purge policy: https://computing.llnl.gov/tutorials/lc_resources/#PurgePolicies

CCMT http://i.ytimg.com/vi/w12Nm5_Vm98/maxresdefault.jpg

UF Machines - HiPerGator

HiPerGator– An AMD Opteron (x86 Architecture) supercomputer

Information

- 16,384 compute cores
- 256 GB RAM per node
- 256 nodes (64 cores per node)

For more information about computing resources at UF visit

- http://hpc.ufl.edu/
- For troubleshooting issues on HiPerGator use: http://support.rc.ufl.edu/

UF Machines - HiPerGator

Queues

HiPerGator queues are handled by the scheduler for regular jobs. The queues are based on your group's investment status for the machine.

Queue Name	Node Limit	Time Limit	Notes
test	Not listed	30 minutes	Troubleshooting queue
bigmem	1	30 days	1 TB of RAM, 80 cores

Info on HiPerGator's bigmem queue: http://wiki.hpc.ufl.edu/doc/Large-Memory_SMP_Servers

UF Machine – HiPerGator

Purge Policy

The HiPerGator has no apparent data purge policy.

They will send you an email if your home directory becomes larger than 20GB and also if your scratch space uses for than your set quota.

To check your disk space using use: quota -s

```
Disk quotas for user neal (uid      ):
     Filesystem  blocks   quota   limit   grace   files   quota   limit   grace
nas2-data:/volumes/v0/home
                16005M  20480M  20480M              0       0       0
```


User Experience

I have used all of the systems in this presentation. Some of my experiences with using them are:

- Getting codes to work on Vulcan can be much more difficult than on x86 machines i.e. Cab, Mustang, Surface
- I think of the HiPerGator as the Wild West of HPC. They have very loose regulations i.e. wall-times up to a month, an almost non-existent purge policy
  - I have files on HiPerGator for 2 years that have not been purged yet.
  - HiPerGator is great for running quick tests with your code before moving to DOE machines.
- Getting in the queue on Vulcan is very easy because it has ~400,000 cores. Other DOE computers can put your job in a queue for days depending on the machine usage.

Resources

LLNL Linux Cluster Computing Resources(Great place to start):
https://computing.llnl.gov/tutorials/lc_resources/

LLNL General Computing Resources(Use left side to navigate to resources):
https://computing.llnl.gov/

LLNL Purge Policies:
https://computing.llnl.gov/tutorials/lc_resources/#PurgePolicies

CCMT

Profiling with Valgrind

Charles Cook, February 19th, 2015,
crcook@ufl.edu

Overview

- Valgrind is a virtual machine
  - No instrumentation in code
  - Allows various tools to access an 'Intermediate Representation' to inject instrumentation at run time (toolchain)
- Memcheck is the most popular tool
- Other tools exist, with Callgrind being particularly useful (callgraph analyzer)
- Linux

Overview

- Uses debug symbols and binary directly
  - Language independent
- Function level callgraph
  - More granular functions will have more detailed profiling
- To install:

```
sudo apt-get install valgrind kcachegrind graphviz
```

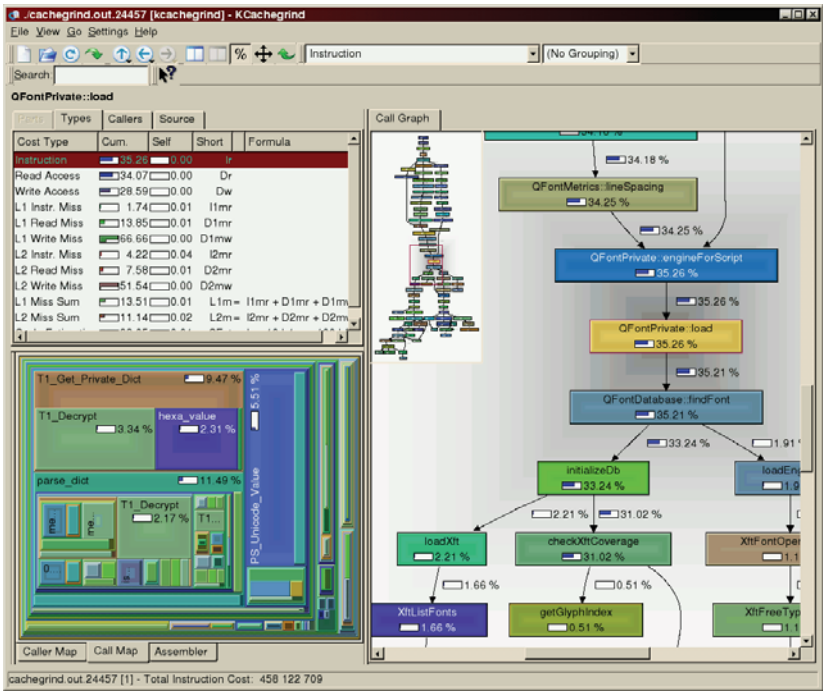

Memcheck

- Problems Memcheck can detect
  - Use of uninitialized memory
  - Reading/writing memory after it has been freed
  - Reading/writing off the end of malloc'd blocks
  - Memory Leaks
- Decreased performance due to added memory checks (~30 times)

Callgrind

- Records the call history among functions
  - Number of instructions
  - Relation to source files
  - Caller/callee relationship
  - Inclusive function costs
- Can use kcachegrind to visualize the call graph.

Kcachegrind.sourceforge.net



Using Memcheck

- Compile with debug symbols (-g)
- Run the program through valgrind
  - "valgrind --leak-check=yes prog arg1 arg2"
  - Memcheck is the default tool
  - --leak-check enables memory leak detection
- Valgrind executes the application (VM)

Using Memcheck

- From Valgrind's Quick Start (in c++)

```
#include <stdlib.h>

void f(void)
{
    int* x = new int(10);
    x[10] = 0;        // problem 1: heap block overrun
}                     // problem 2: memory leak -- x not freed

int main(void)
{
    f();
    return 0;

}
```

Using Memcheck

- Compile with debug symbols
  `g++ –g sample.c –o sample`
- Execute with valgrind
  `valgrind –leak-check=yes ./sample`
- Finds the invalid write

```
==38882== Invalid write of size 4
==38882==    at 0x40054B: f (sample.c:6)
==38882==    by 0x40055B: main (sample.c:11)
==38882==  Address 0x51fd068 is 0 bytes after a block of size 40 alloc'd
==38882==    at 0x4C2AB80: malloc (in /usr/lib/valgrind/vgpreload_memcheck-amd64-linux.so)
==38882==    by 0x40053E: f (sample.c:5)
==38882==    by 0x40055B: main (sample.c:11)
```


Using Memcheck

- Finds the memory leak (allocation point)

```
==3045== HEAP SUMMARY:
==3045==     in use at exit: 40 bytes in 1 blocks
==3045==   total heap usage: 1 allocs, 0 frees, 40 bytes allocated
==3045==
==3045== 40 bytes in 1 blocks are definitely lost in loss record 1 of 1
==3045==    at 0x4C2AB80: malloc (in /usr/lib/valgrind/vgpreload_memcheck-amd64-linux.so)
==3045==    by 0x40053E: f (sample.c:5)
==3045==    by 0x40055B: main (sample.c:11)
```

Using Memcheck

- Obvious fix is to add delete(x) to f() and fix the index on the assignment.
- Since x was created on the heap it is not automatically de-allocated with scope
- Another option is to use smart pointers which automatically free the memory

Using Memcheck

- Update f() to use a smart pointer
- std::unique_ptr is a c++ 11 standard
- boost::unique_ptr could be used instead

```
void f(void)
{
        std::unique_ptr<int[]> x(new int[10]);
        x[10] = 0;
}
```

Using Memcheck

- Compile

```
g++ –g sample.c –o sample -std=c++11
```

- Run with valgrind

```
valgrind –leak-check=yes ./sample
```

- Memory leak has been fixed
- Particularly useful for larger problems where it's not clear when to de-allocate
  - Std::shared_ptr de-allocates when all uses leave scope

Using Callgrind

- Let's add a little complexity:

```
#include <stdlib.h>
#include <memory>
#include <vector>
#include <algorithm>
void f(int N)
{
            std::unique_ptr<std::vector<int>> x(new std::vector<int>());
            for (int i = 0; i < N; ++i){
                        (*x).push_back(rand());
            }
            std::sort(begin(*x), end(*x));
}
int main(void)
{
            f(1000);
            f(10000);
            f(100000);
            return 0;
}
```

Using Callgrind

- Compile

```
g++ –g sample.c –o sample -std=c++11
```

- Run valgrind, specifying callgrind

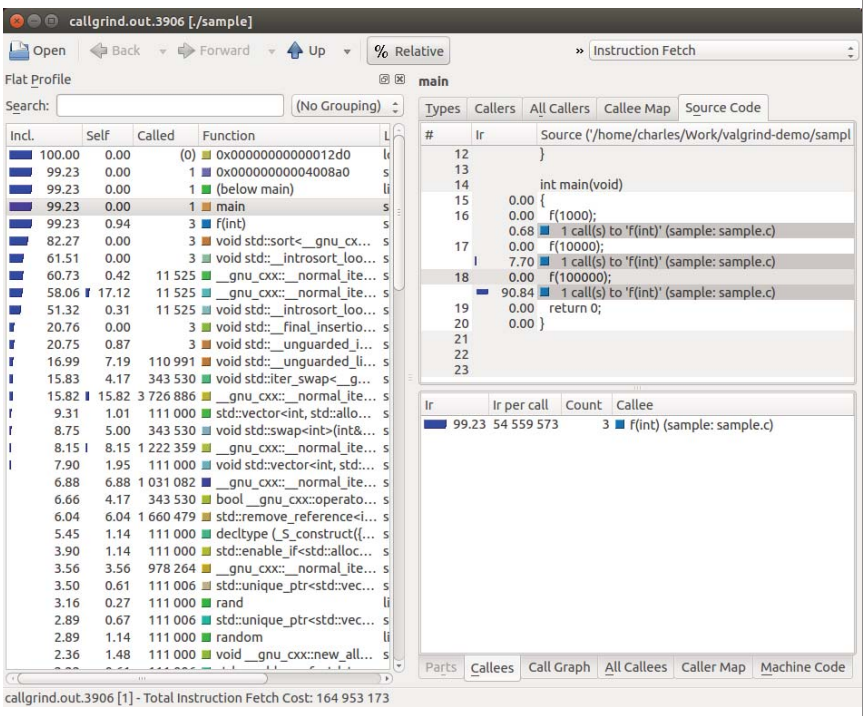
```
valgrind --tool=callgrind ./sample
```

- This generates an output file, callgrind.out.XXXX where XXXX is the process ID.
- Open the output with kcachegrind

```
kcachegrind callgrind.out.XXXX
```


Using Callgrind

- Sort 82%
- f(1E3) 0.68%
- f(1E4) 7.70%
- f(1e5) 90.8%
- N log2(N) as expected from the sort docs.
- $10\uparrow 4\ log\downarrow 2\ 10\uparrow 4 / 10\uparrow 5\ log\downarrow 2\ 10\uparrow 5 = .08$



Summary

- Valgrind contains several tools
- Memcheck is useful for finding memory errors, particularly memory leaks
- Callgrind is useful for quick profiling

CCMT

Debugging with Totalview: An Introduction

Bertrand Rollin

CCMT – 02/19/15

Disclaimer: This presentation is largely inspired from presentations of ATPESC 2014 and LLNL Totalview tutorial. This presentation is intended for CCMT internal use only.

What is Totalview?

- TotalView is a sophisticated software debugger product from Rogue Wave Software Inc.
- Used for debugging and analyzing both **serial and parallel programs**.
- Especially designed for use with complex, multi-process and/or multi-threaded applications.
- The most popular HPC debugger to date.
- Has been selected as the Department of Energy's ASC Program's debugger

Overview

Key Features of TotalView:

- Designed to handle most types of HPC parallel coding
- Supported on most HPC platforms.
- Provides both a GUI and command line interface
- Can be used to debug programs, running processes, and core files.
- Includes memory debugging features
- Provides graphical visualization of array data
- Includes a comprehensive built-in help system
- etc

Supported Platforms and Languages

- TotalView is supported on most major U.S. HPC platforms, and also, Apple Mac OS X.
- Supported languages include the usual HPC application languages:
  - C/C++
  - Fortran77/90
  - Mixed C/C++ and Fortran
  - Assembler

Before Starting Totalview

Compiling your code:

-g:

- Compile your program with the appropriate flag to enable generation of symbolic debug information. For most compilers, the **-g** option is used for this.
- TotalView will allow you to debug executables which were not compiled with the **-g** option. However, only the assembler code can be viewed.

Beyond -g:

- Don't compile your program with optimization flags while you are debugging it. Compiler optimizations can "rewrite" your program and produce machine code that doesn't necessarily match your source code.
- Parallel programs may require additional compiler flags.
- For Rocflu, compile with the flag “DEBUG=1”
- **Make sure “precious” and “secondary” are uncommented in Makefile.Linux**

Starting Totalview

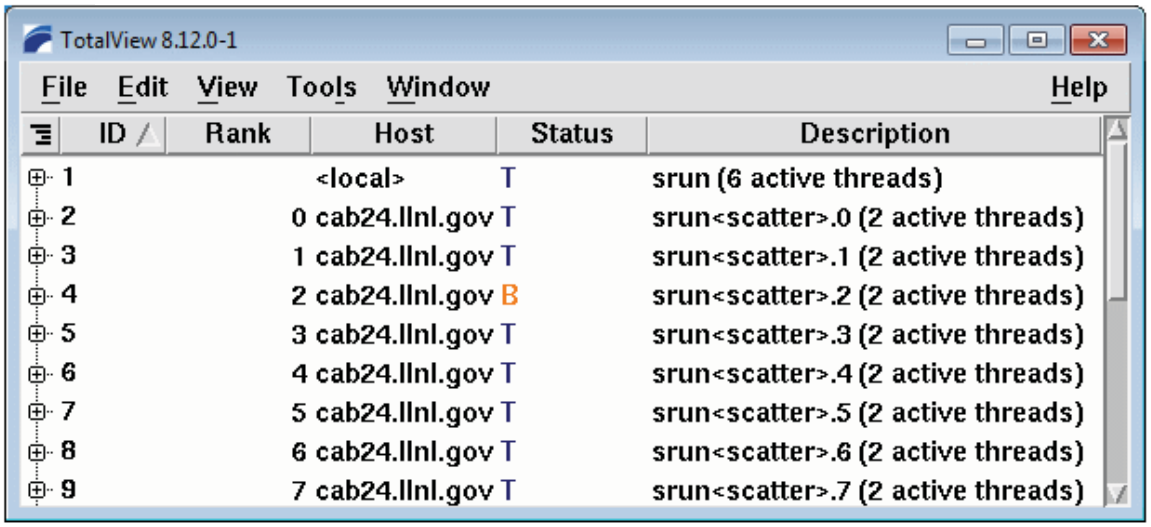
TotalView can be started in a number of different ways, depending upon whether you want to:

- debug an executable file
- attach to a running process
- debug a core file
- change the debugger's default appearance/behavior

Starting Totalview

Command / Action
totalview Starts the debugger with the Session Manager. You can then load a program, corefile, or attach to a running process.
totalview *filename* Starts the debugger and loads the program specified by *filename*.
totalview *filename corefile* Starts the debugger and loads the program specified by *filename* and its core file specified by *corefile*.
totalview *filename* -a *args* Starts the debugger and passes all subsequent arguments (specified by *args*) to the program specified by *filename*. The **-a** option must appear after all other TotalView options on the command line.
totalview *filename* -remote *hostname[:portnumber]* Starts the debugger on the local host and the totalview debugger server on the remote host *hostname*. Loads the program specified by *filename* for remote debugging. You can specify a host name or TCP/IP address for *hostname*, and optionally, a TCP/IP port number for *portnumber*.

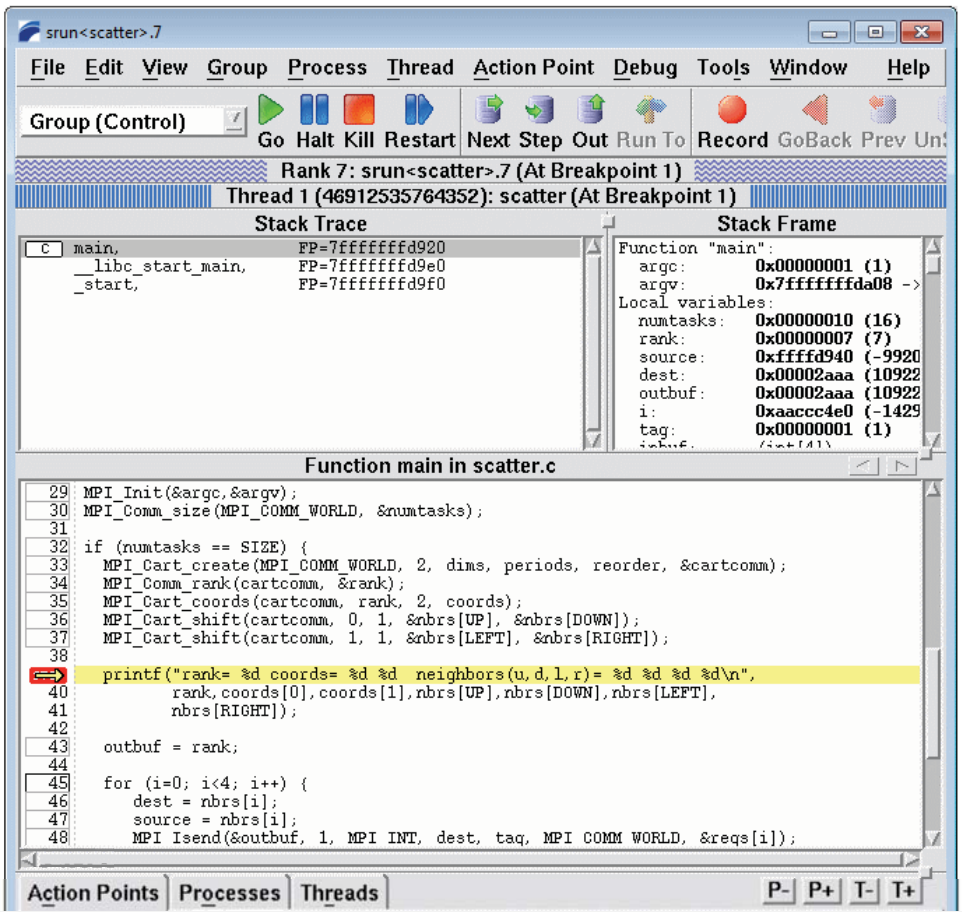
Root Window



- Will always appear when the TotalView GUI is started.
- Provides an overview of all processes and threads, showing the TotalView assigned ID, MPI rank, host, status and brief description/ name for each.
- Allows sorting by ID, rank, host and status.
- Provides the ability to expand/collapse each process to view/hide any included threads in that process.

CCIvi I

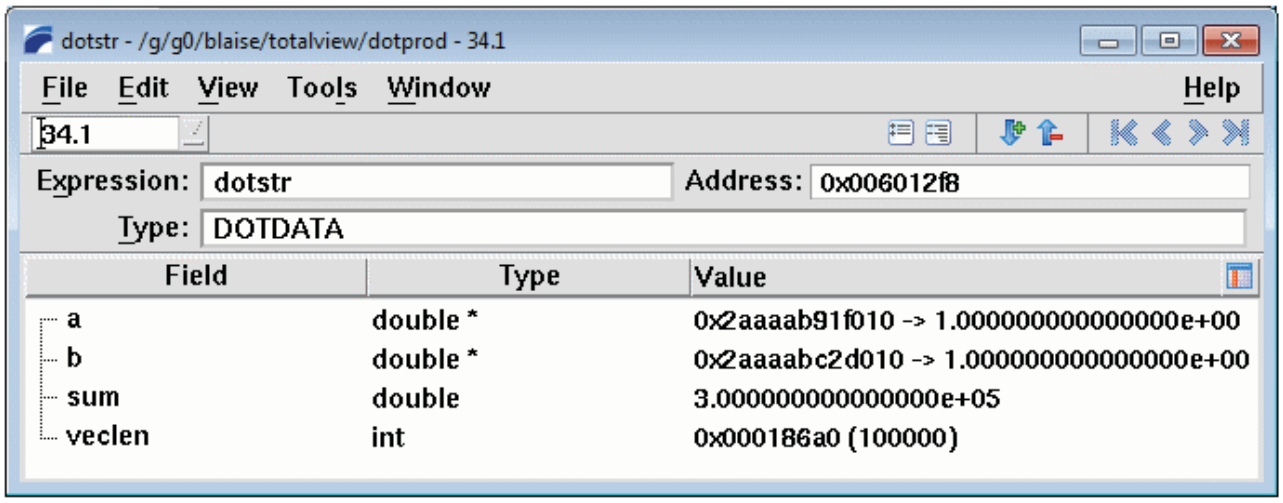
Process Window



4 “Panes”:

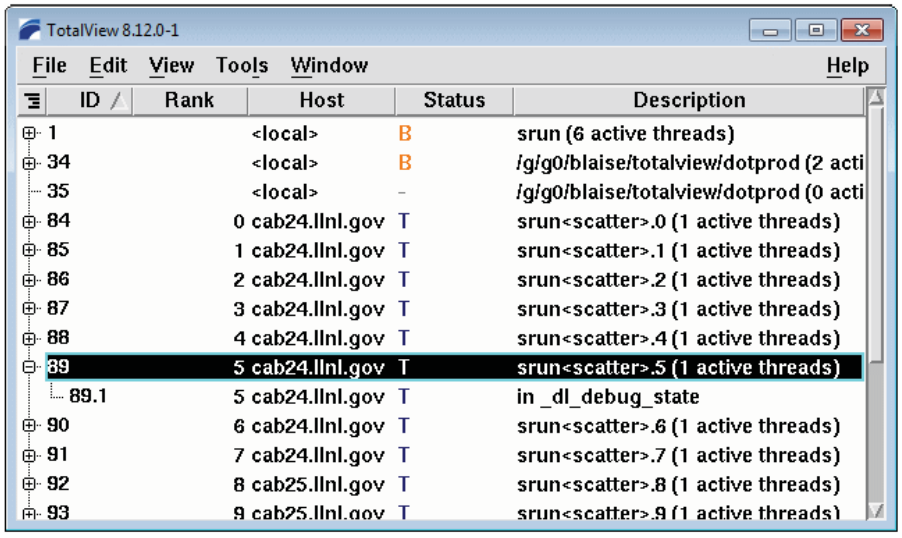
Stack Trace Pane,
Stack Frame Pane,
Source Pane
Action points Pane

Variable Window



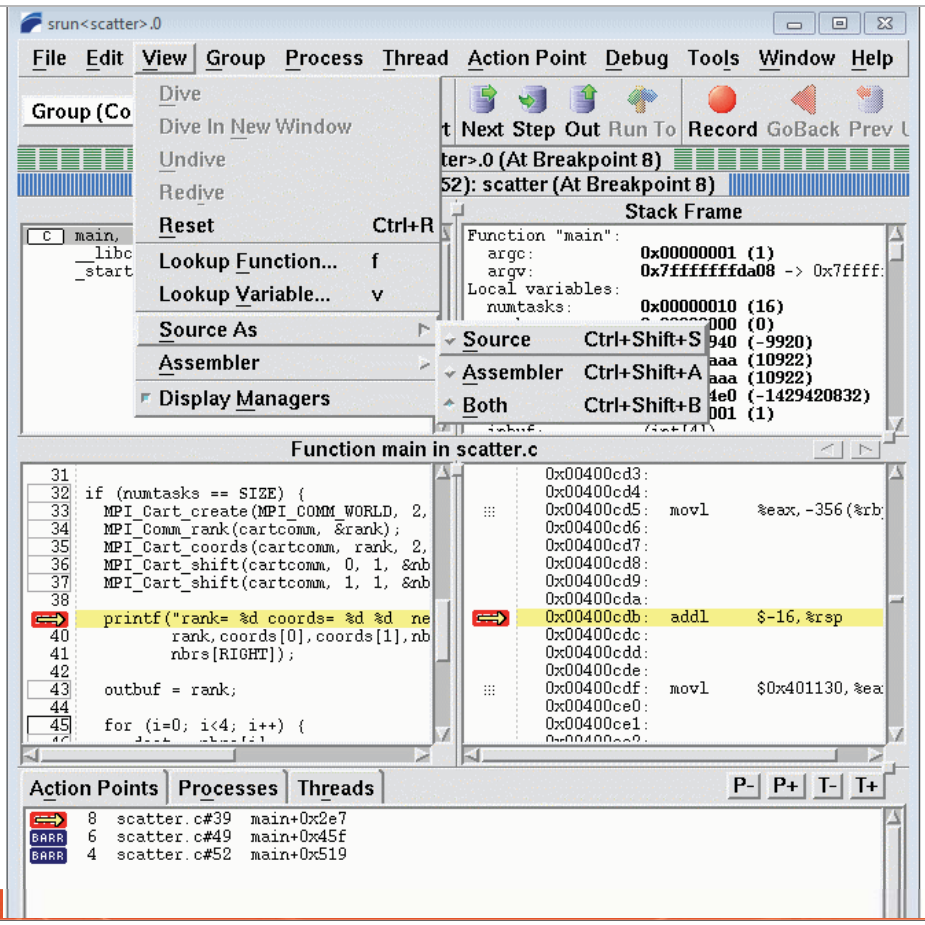
- Appears when you dive (covered later) on a variable or select a menu item to view variable information.
- Displays detailed information about selected program variables. Also permits editing, diving, filtering and sorting of variable data.
- Comprised of a single pane, pull-down menus, data field boxes and several action buttons.

Process and Thread State Codes



State Code	Description
B	Stopped at a breakpoint
E	Stopped because of an error
H	In a Hold state
K	Thread is executing within the kernel
M	Mixed - some threads in a process are running and some not
R	Running
T	Thread is stopped
W	At a watchpoint

Viewing Source Code



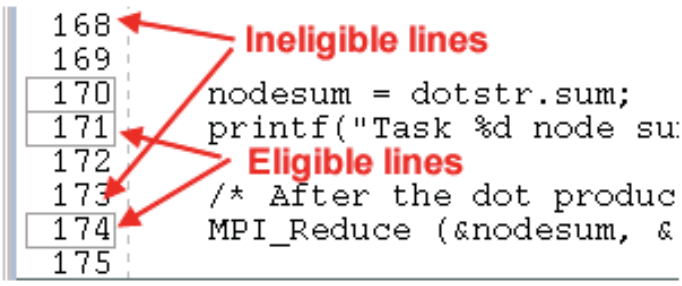
Setting a Breakpoint

- A breakpoint is the most basic of TotalView's action points used to control a program's execution. It causes a process/thread to halt execution at the line number - prior to executing that line number

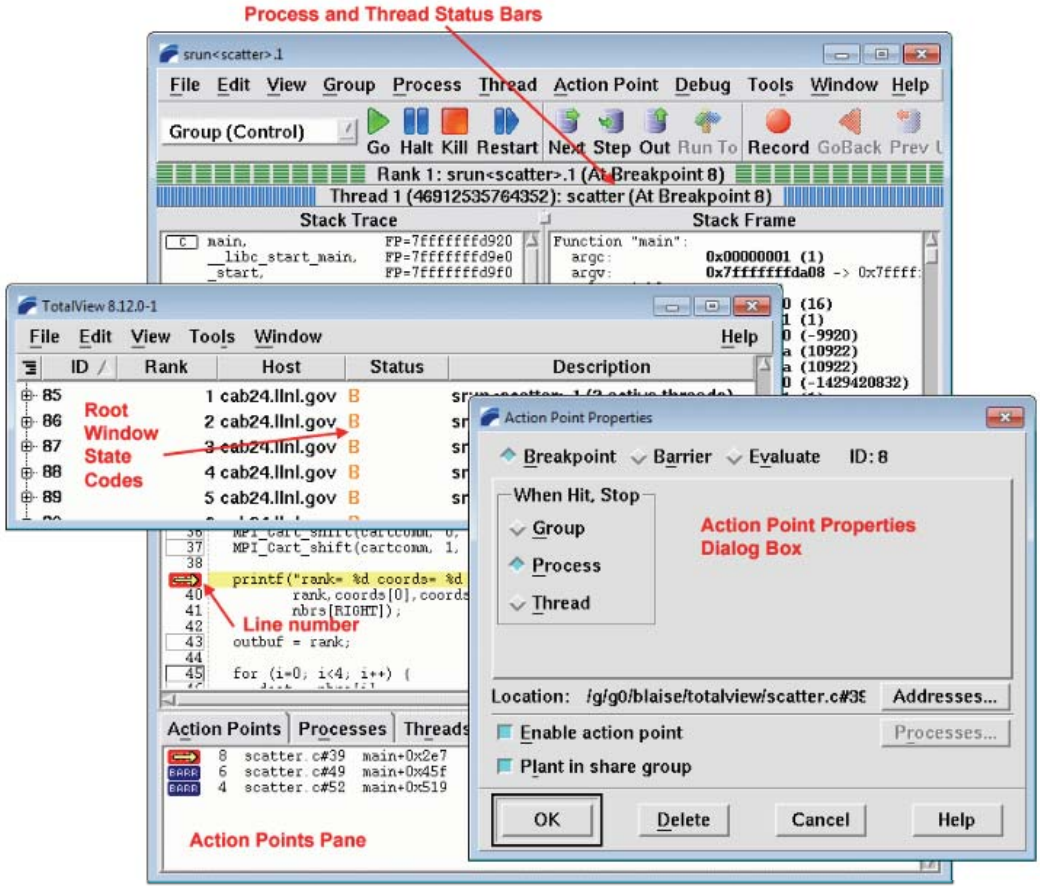
.

- Breakpoints can be set in source code and assembler code.

- For regular source, only "boxed" line numbers are eligible for breakpoints. For assembler, only assembler instructions displaying a "gridget" are eligible.



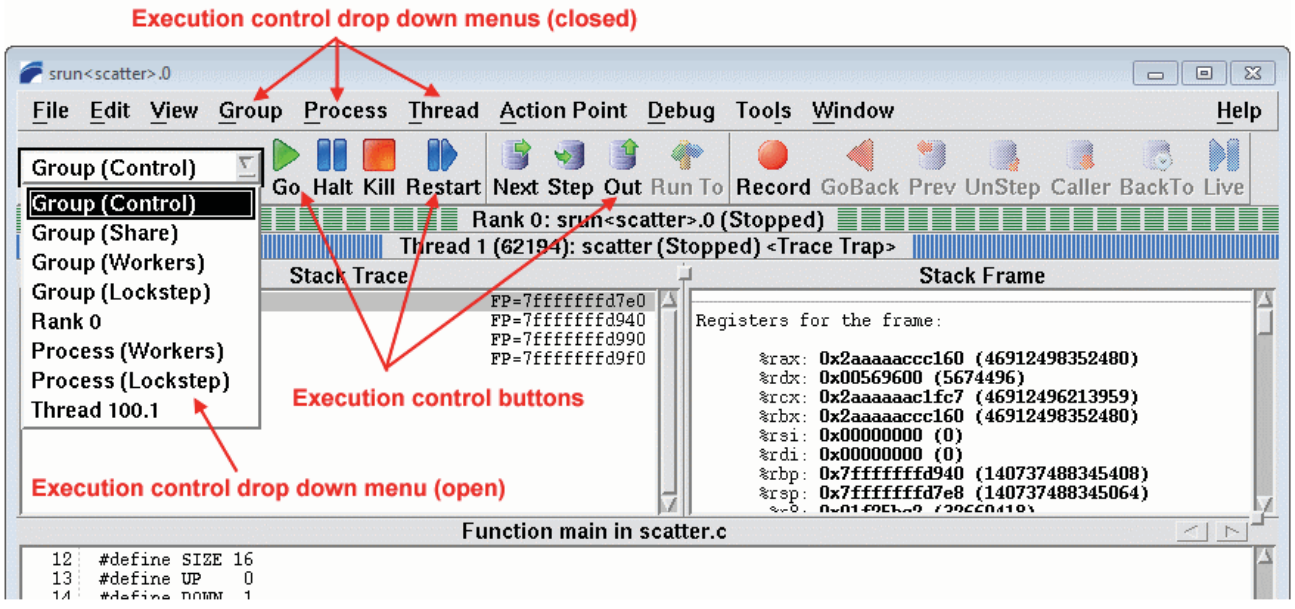
Viewing Breakpoints



Viewing Breakpoints

Controlling the execution of a program within TotalView involves two decisions:

1. Selecting the appropriate command
2. Deciding upon the scope of the chosen command



Most of TotalView's execution control commands can be applied at the Group, Process or Thread scoping level. The right scope depends upon what you want to effect.

Setting Source Code Search Paths

In cases where your source code and executables are not co-located, you may need to tell TotalView where to search for the various components.

By default, the debugger will search the following directories (in order):

1. Current working directory
2. Path of an executable started with a full path name
3. Directories specified in your **PATH** environment variable.

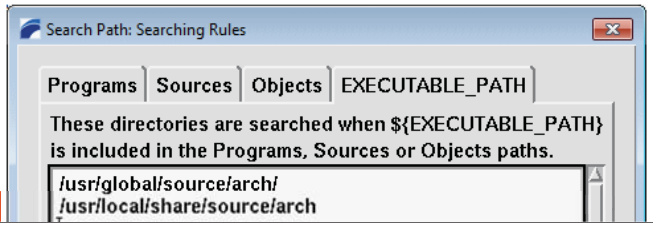
How to Add Additional Search Paths:

1. Select either: PATH: Process Window > File Menu > Search Path
   PATH: Root Window > File Menu > Search Path

2. In the **Search Path Dialog Box** (shown below), click on the **EXECUTABLE_PATH** tab and then enter the directories that should be searched, in order. They can be separated with a space or a new line. Relative path names are permitted (relative to the current working directory).

3.To browse for directories to add, click on the **Insert** button to open a Select Directory Dialog Box.

4.Note that these paths will be persistent from session to session.



CCMT

References and further tutorial material

- C. Gottbrath, RogueWave, **"Debugging Scalable MPI, Hybrid and/or Accelerated Applications with TotalView"**, ATPESC 2014
- **https://computing.llnl.gov/tutorials/totalview**

CCMT

Code testing and Validation

Mrugesh Shringarpure
Jason Hackl

UF UNIVERSITY of FLORIDA

Verification and Validation

- Verification
  - Are you building the product right?
  - Software must conform to its specification
- Validation
  - Are you building the right product?
  - Software should do what the user really requires

Verification and Validation Process

- Must applied at each stage of the software development process to be effective
- Objectives
  - Discovery of system defects
  - Assessment of system usability in an operational situation

Static and Dynamic Verification

- Software inspections (static)
  - Concerned with analysis of static system representations to discover errors
  - May be supplemented by tool-based analysis of documents and program code
- Software testing (dynamic)
  - Concerned with exercising product using test data and observing behavior

Types of Testing

- Defect testing
  - Tests designed to discover system defects
  - A successful defect test reveals the presence of defects in the system
- Statistical testing
  - Tests designed to reflect the frequency of user inputs
  - Used for reliability estimation

Verification and Validation Goals

- Establish confidence that software is fit for its intended purpose
- The software may or may not have all defects removed by the process
- The intended use of the product will determine the degree of confidence in product needed

UF UNIVERSITY of FLORIDA

Testing and Debugging

- These are two distinct processes
- Verification and validation is concerned with establishing the existence of defects in a program
- Debugging is concerned with locating and repairing these defects
- Debugging involves formulating a hypothesis about program behavior and then testing this hypothesis to find the error

UF UNIVERSITY of FLORIDA

Planning

- Careful planning is required to get the most out of the testing and inspection process
- Planning should start early in the development process
- The plan should identify the balance between static verification and testing
- Test planning must define standards for the testing process, not just describe product tests

Overview of CMT-Nek code development

- Code development process involves adding specific capabilities.
- Capabilities are divided into unique elemental tasks.
- These elemental tasks are Matrix-Matrix multiplication, scaling, dot-product, transpose operations, copy/swap operations etc.

CMT-Nek: Testing and Validatiom

- Step 1: Test the individual elemental tasks (outside the code).
- Step 2: Test the tasks in-situ
- Step 3: Test the capability
- Step 4: Test the entire system

- How do you set up test cases?
- Use synthetic testing, i.e., provide data such that the output of each task is known a priori.

CMT-Nek: Testing and Validation

- Testing and validation - Step 4: Entire system
- Plan in advance to know various test cases.
- Choose cases such that simple, so that defect can be identified quickly.

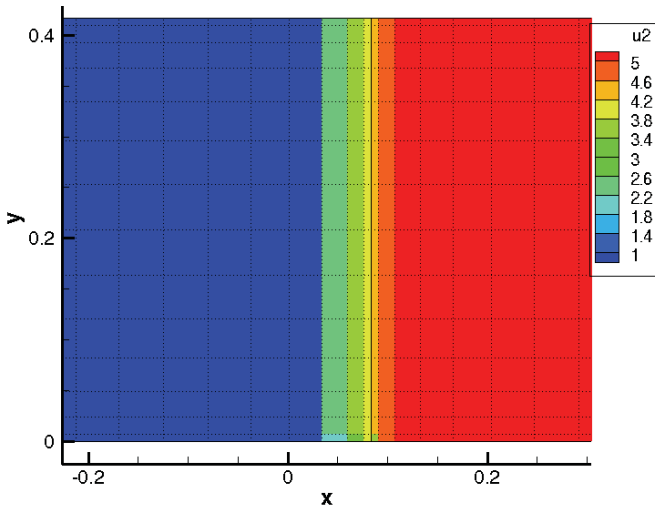
CMT-Nek: Testing and Validation

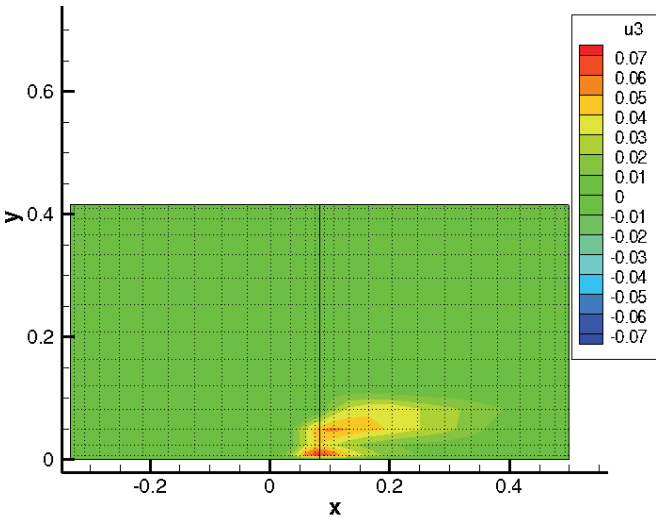
- Following was the full system test plan
- Test 1: Uniform subsonic and supersonic flow in a box.
  - Time integration, derivatives, AUSM ..etc.
- Test 2: Flip direction of the flow
  - Time integration, derivatives, AUSM ..etc.
- Code verified for trivial cases, needed an evolving solution !

CMT-Nek: Testing and Validation

- Test 3: Rarefaction - Hot compressed fluid released into ambient.
  - We identified algorithmic issues with the BC treatment.
- Test 4: Nozzle flow- converging nozzle for subsonic flows and diverging nozzle for supersonic flows
  - Identified sign error in adding throttling term to the governing equations

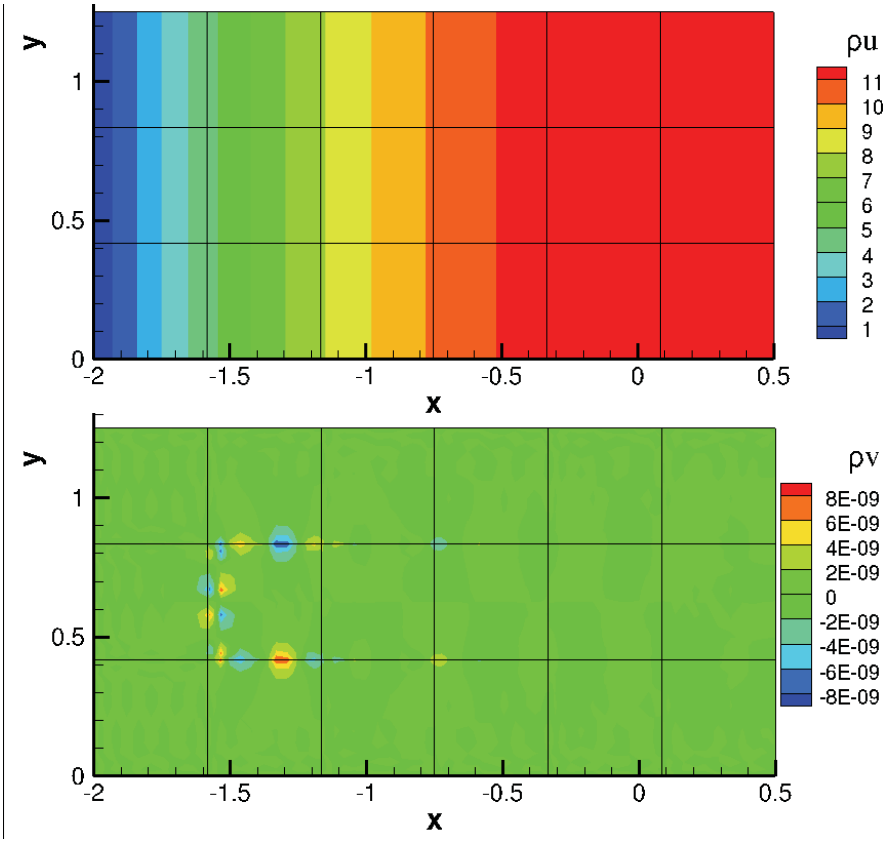
Test cases - rarefaction





- Rarefaction waves cannot smoothly pass from one element to another. Spike in v and incorrect u velocity is observed.

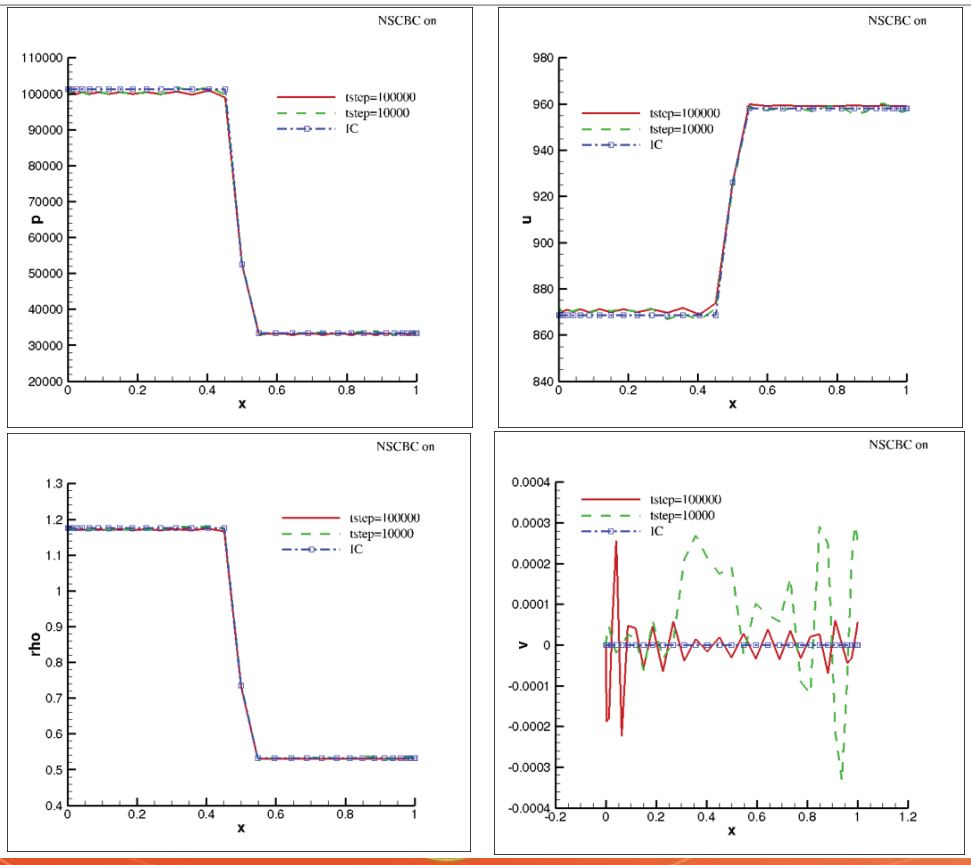
Rarefaction test case



CCMT

1D Nozzle - supersonic flow

Case3

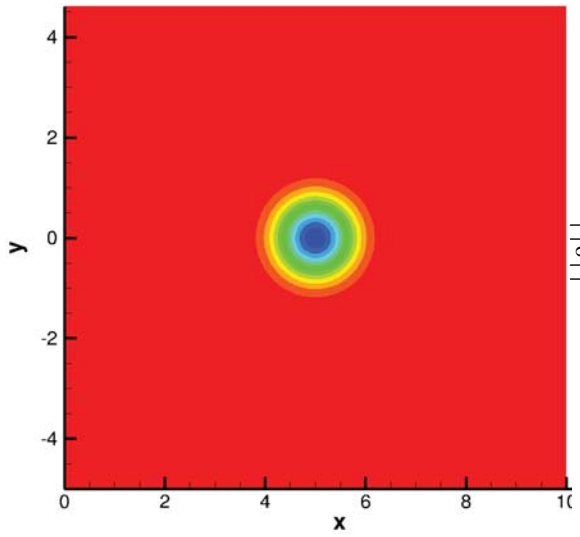


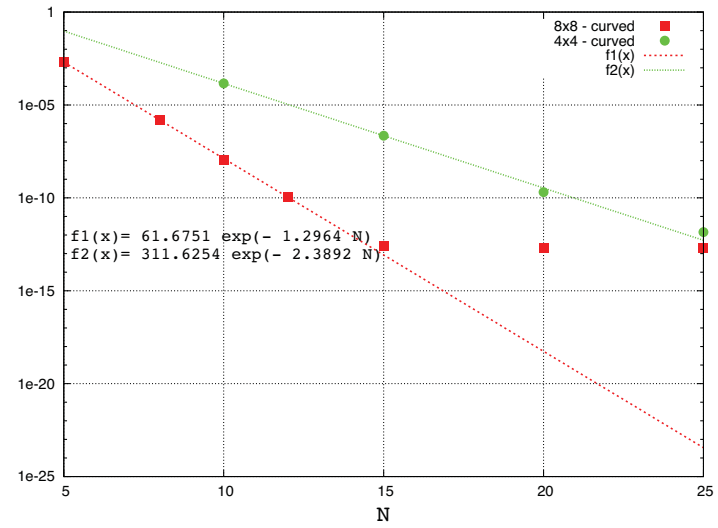
Results for case 4 are similar

CCMT

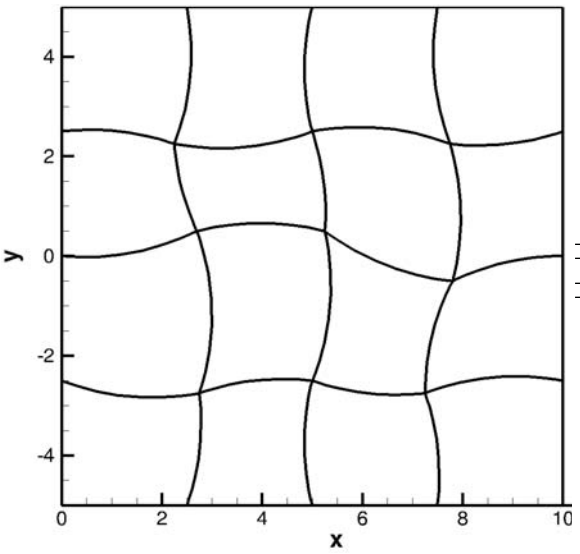
CMT-Nek: Testing and Validation

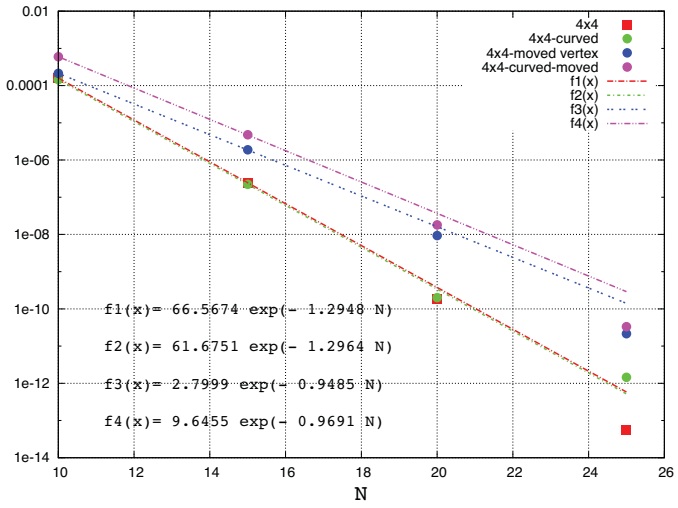
- Test 5: Verify convergence rate of your code. We used isotropic vortex advected in a periodic domain.





- Test 6: Ensure the convergence is retained for deformed meshes





CVS and TAU

Tania Banerjee
Computer and Information Science and Engineering

UF UNIVERSITY of FLORIDA

CVS Tutorial

An Introduction to Using

CVS

Concurrent

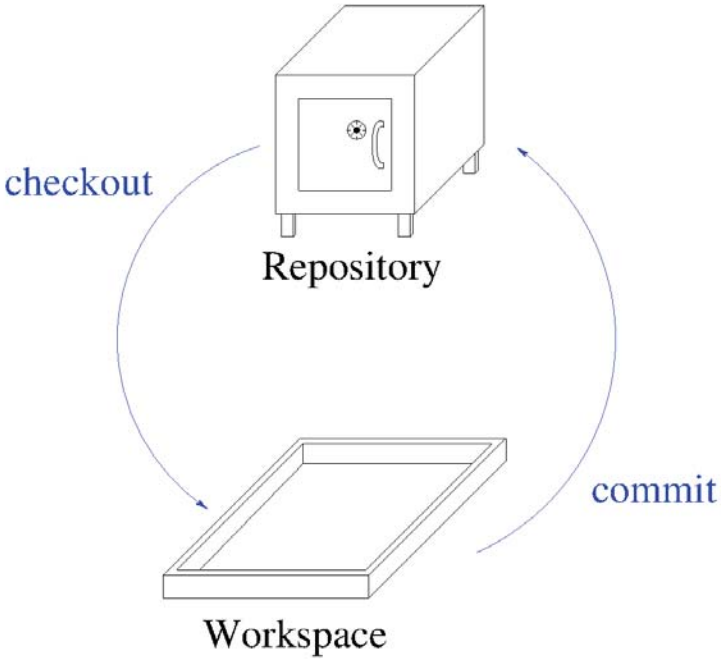
Versions System

* collaboration

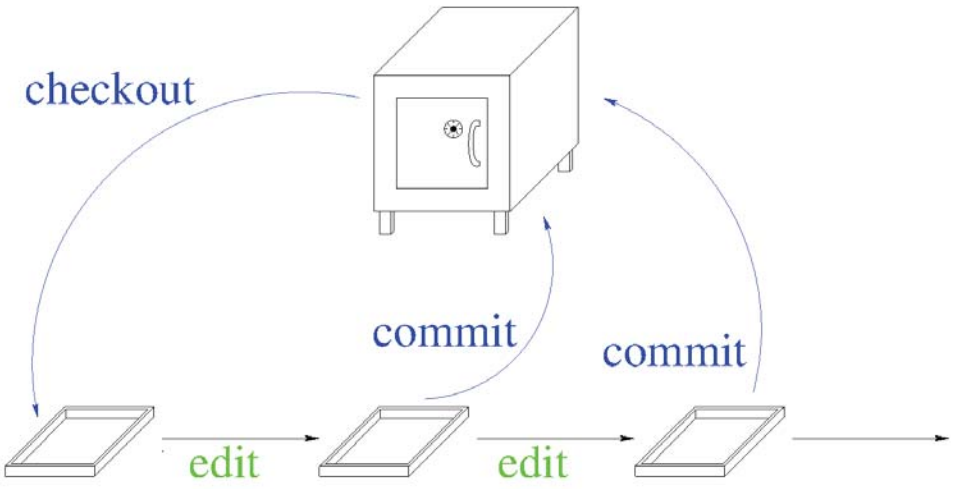
* history: when? why?
* examine old revisions
* bugfix releases

Thanks to Steve Robbins for the slides

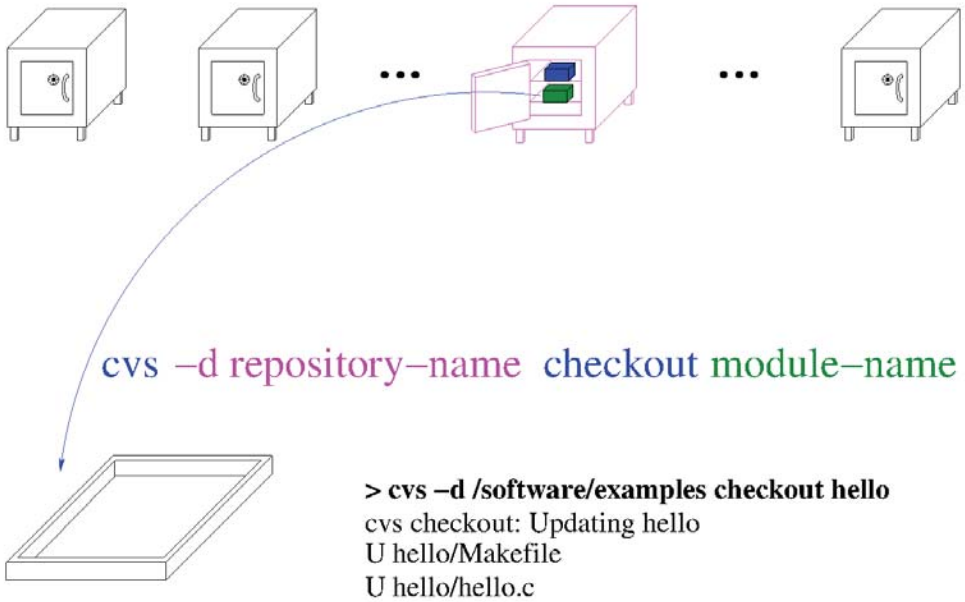
CVS Tutorial



CVS Tutorial



CVS Tutorial



> cvs –d /software/examples checkout hello
cvs checkout: Updating hello
U hello/Makefile
U hello/hello.c
U hello/world.c
U hello/world.h

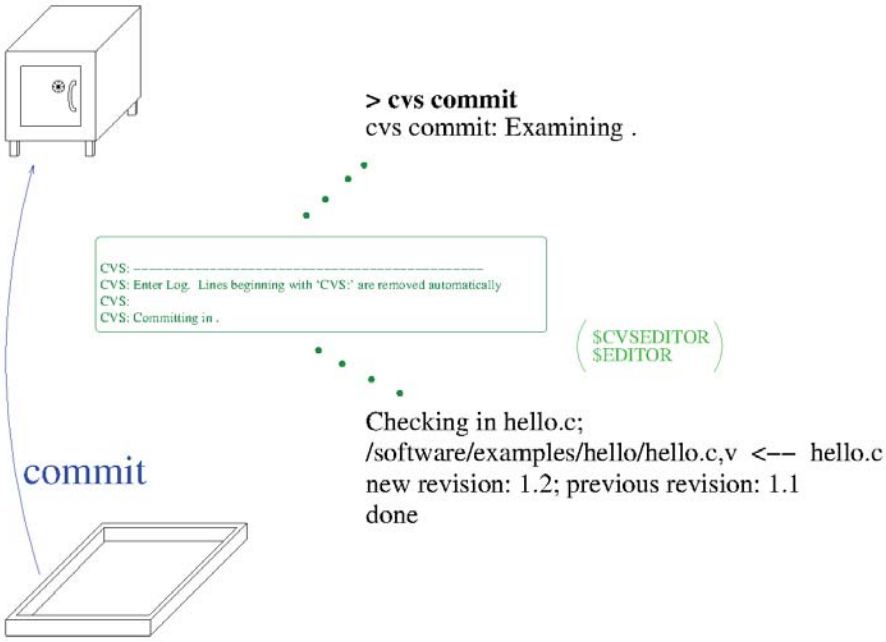
CVS Tutorial

> cd ~/src
> cvs –d /software/examples checkout hello
cvs checkout: Updating hello
U hello/Makefile
U hello/hello.c
U hello/world.c
U hello/world.h

> cd hello
> ls
CVS Makefile hello.c world.c world.h

> ls CVS
Entries Repository Root

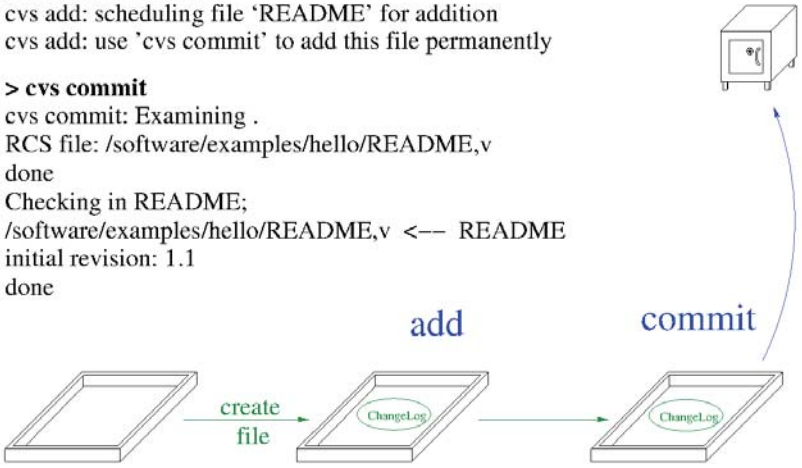
CVS Tutorial



CVS Tutorial

> cvs add README
cvs add: scheduling file 'README' for addition
cvs add: use 'cvs commit' to add this file permanently

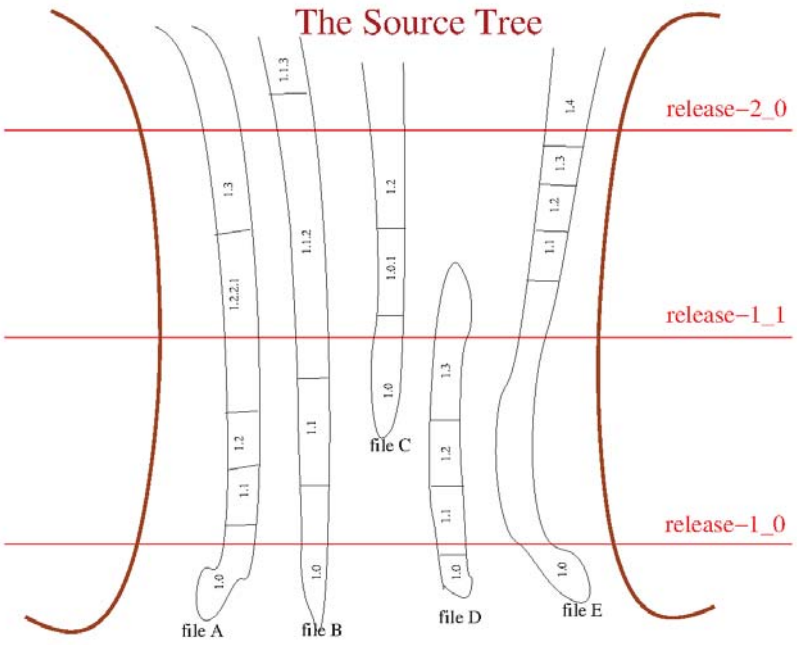
> cvs commit
cvs commit: Examining .
RCS file: /software/examples/hello/README,v
done
Checking in README;
/software/examples/hello/README,v <-- README
initial revision: 1.1
done



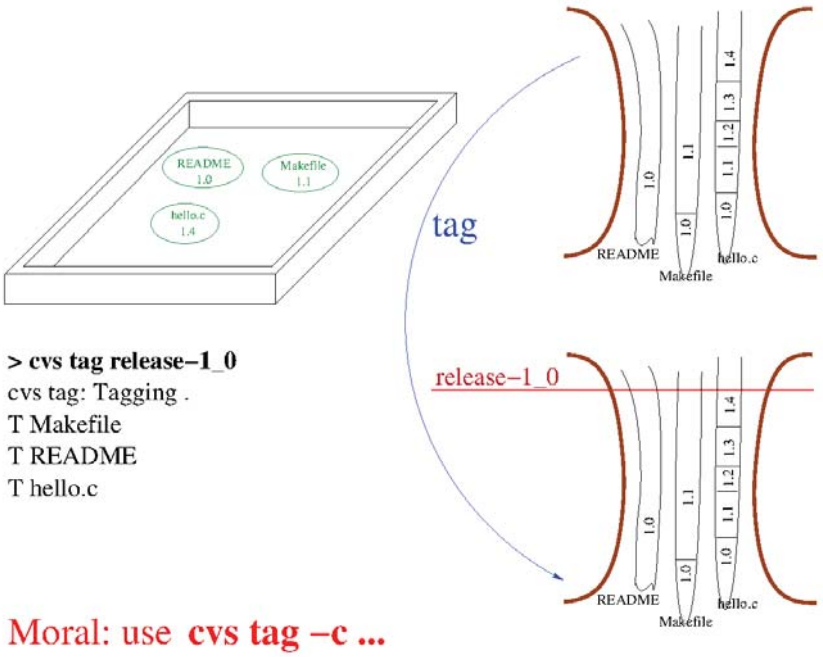
Remove files: rm filename; cvs remove filename

Rename files: remove then add

CVS Tutorial



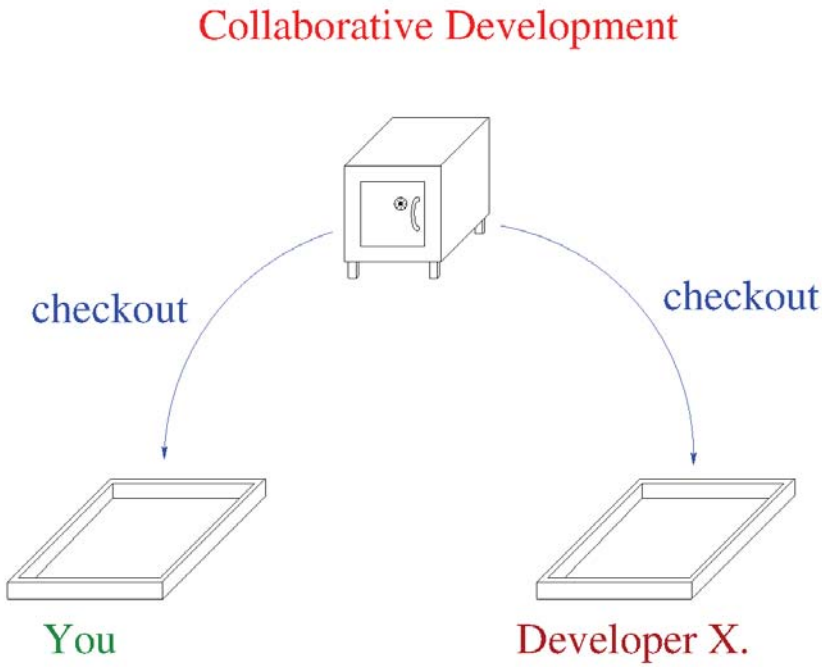
CVS Tutorial



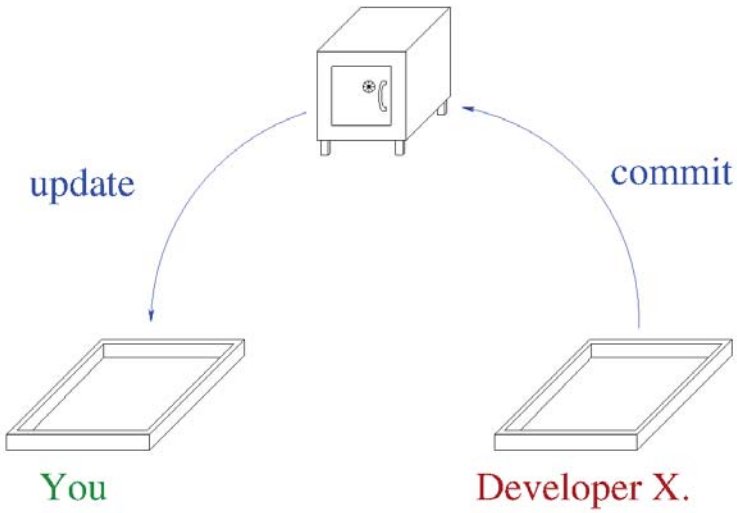
> cvs tag release–1_0
cvs tag: Tagging .
T Makefile
T README
T hello.c

Moral: use **cvs tag –c ...**

CVS Tutorial



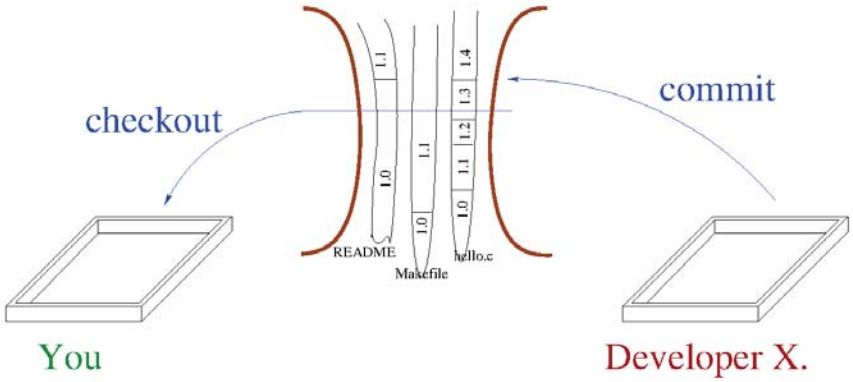
CVS Tutorial




```
> cvs update
cvs update: Updating .
U ChangeLog
U README
```

CVS Tutorial

> cvs commit
cvs commit: Examining .
cvs commit: Up–to–date check failed for 'README'
cvs commit: Up–to–date check failed for 'hello.c'
cvs [commit aborted]: correct above errors first!

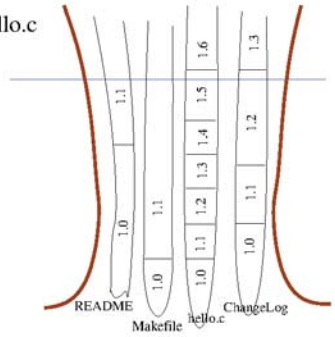


Moral: update before commit.

CVS Tutorial

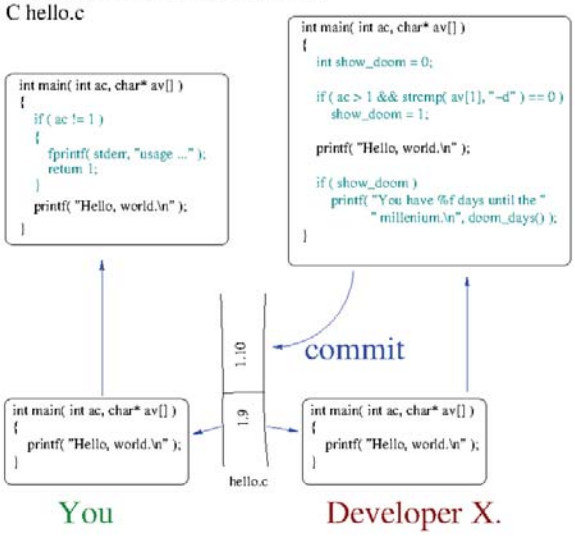
Update (revisited)

> cvs update
cvs update: Updating .
U ChangeLog
M Makefile
RCS file: /software/examples/hello/hello.c,v
retrieving revision 1.5
retrieving revision 1.6
Merging differences between 1.5 and 1.6 into hello.c
M hello.c
? hello



CVS Tutorial

> cvs update
cvs update: Updating .
RCS file: /software/examples/hello/hello.c,v
retrieving revision 1.9
retrieving revision 1.10
Merging differences between 1.9 and 1.10 into hello.c
rcsmerge: warning: conflicts during merge
cvs update: conflicts found in hello.c
C hello.c



CCMT

CVS Tutorial

Conflict Resolution

```
int main( int ac, char* av[] )
{
  if ( ac != 1 ) {
    fprintf( stderr, "usage ..." );
    return 1;
  }
  printf( "Hello, world.\n" );
}
```

hello.c before update

```
int main( int ac, char* av[] )
{
  int show_doom = 0;

  if ( ac > 1 && strcmp( av[1], "-d" ) == 0 )
    show_doom = 1;

  printf( "Hello, world.\n" );

  if ( show_doom )
    printf( "You have %f days until the "
            " millenium.\n", doom_days() );
}
```

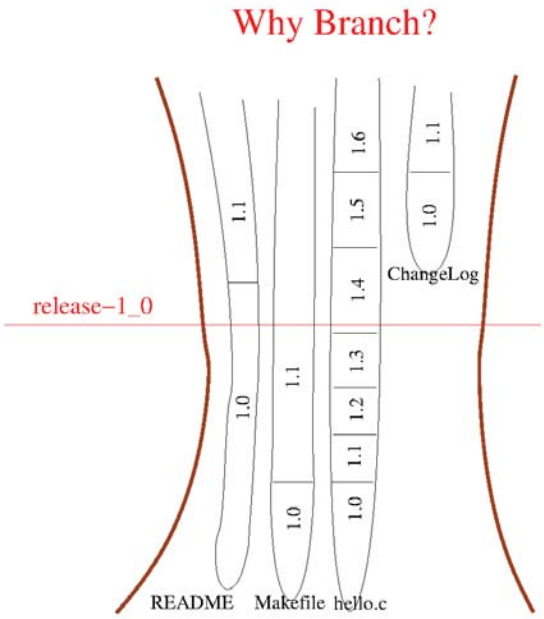
revision 1.10

```
int main( int ac, char* av[] )
{
<<<<<<< hello.c
  if ( ac != 1 ) {
    fprintf( stderr, "usage: hello\n" );
    return 1;
  }

=======
  int show_doom = 0;

  if ( ac > 1 && strcmp( av[1], "-d" ) == 0 )
    show_doom = 1;

>>>>>>> 1.10
  printf( "Hello, world.\n" );

  if ( show_doom )
    printf( "You have %f days until the "
            " millenium.\n", doom_days() );
}
```

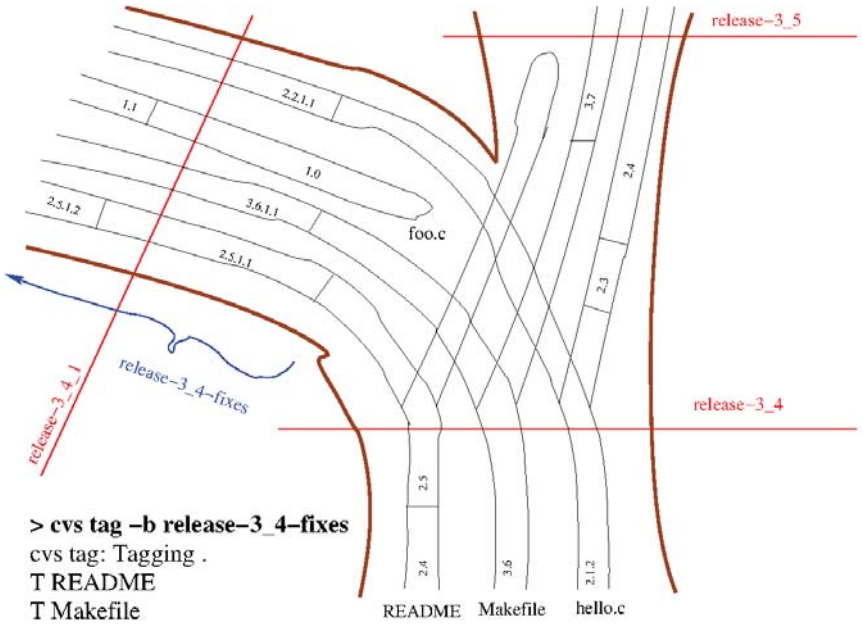
hello.c after update

CCMT

CVS Tutorial



CVS Tutorial



> cvs tag –b release–3_4–fixes
cvs tag: Tagging .
T README
T Makefile
T hello.c

CVS Tutorial

checkout (revisited)

cvs –d repository–name checkout module–name

= latest revisions on HEAD branch (main trunk)

cvs –d repository–name checkout –r release–tag module–name

= revisions selected by tag name

cvs –d repository–name checkout –r branch–tag module–name

= latest revisions on specified branch

also: –D date examples: –D "13:45 1 December 1997"
–D "5 days ago"

UF UNIVERSITY of FLORIDA

CVS Tutorial

Command Summary

cvs –d repository init
= create a new repository

cvs –d repository import module vendor–tag release–tag
= create a new module

cvs –d repository checkout [–r tag] module

cvs update [–r tag]

cvs add file ...

cvs remove file ...

cvs commit

cvs tag –c [–b] tag

cvs log
= list log messages, tags, etc

cvs status
= up–to–date, locally–modified, etc

cvs diff –r rev1 –r rev2 file
= difference between specified revisions

Questions?

TAU Tutorial

- **Tuning and Analysis Utilities** (18+ year project)
- **Comprehensive performance profiling and tracing**
  - Integrated, scalable, flexible, portable
  - Targets all parallel programming/execution paradigms
- **Integrated performance toolkit**
  - Instrumentation, measurement, analysis, visualization
  - Widely-ported performance profiling / tracing system
  - Open source (BSD-style license)
- **Integrates with application frameworks**

Thanks to Dr. Sameer Shende for the slides

TAU Tutorial : Understanding application performance using TAU

- **How much time** is spent in each application routine and outer *loops*? Within loops, what is the contribution of each *statement*?
- **How many instructions** are executed in these code regions? Floating point, Level 1 and 2 *data cache misses*, hits, branches taken?
- **What is the memory usage** of the code? When and where is memory allocated/de-allocated? Are there any memory leaks?
- **What are the I/O characteristics** of the code? What is the peak read and write *bandwidth* of individual calls, total volume?
- **What is the contribution of each *phase*** of the program? What is the time wasted/spent waiting for collectives, and I/O operations in Initialization, Computation, I/O phases?
- **How does the application *scale***? What is the efficiency, runtime breakdown of performance across different core counts?

TAU Tutorial : What can TAU do?

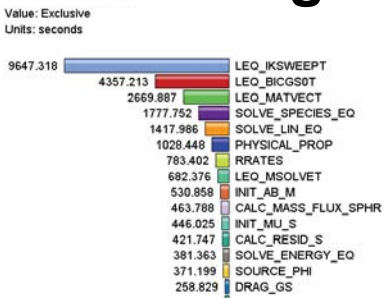
- Profiling and tracing
  - **Profiling** shows you **how much** (total) time was spent in each routine
  - **Tracing** shows you **when** the events take place on a timeline
- Multi-language debugging
  - Identify the source location of a crash by unwinding the system callstack
  - Identify memory errors
- Profiling and tracing can measure time as well as hardware performance counters (cache misses, instructions) from your CPU
- TAU can automatically instrument your source code using a package called PDT for routines, loops, I/O, memory, phases, etc.
- TAU runs on all HPC platforms and it is free (BSD style license)
- TAU includes instrumentation, measurement and analysis tools

TAU Tutorial : What does TAU support?

C/C++ CUDA UPC Python GPI Fortran OpenCL Java MPI pthreads Intel MIC OpenMP Intel GNU LLVM PGI Cray Sun MinGW Linux Windows AIX Insert yours here BlueGene Fujitsu ARM NVIDIA Kepler OS X

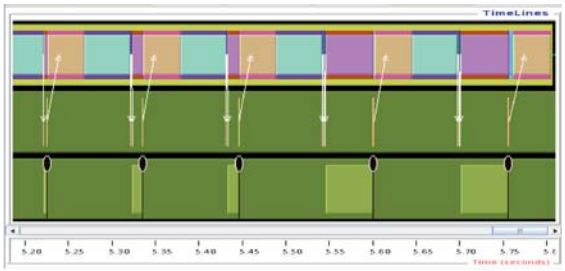
TAU Tutorial : Profiling and Tracing

Profiling



- **Profiling** shows you **how much** (total) time was spent in each routine

Tracing

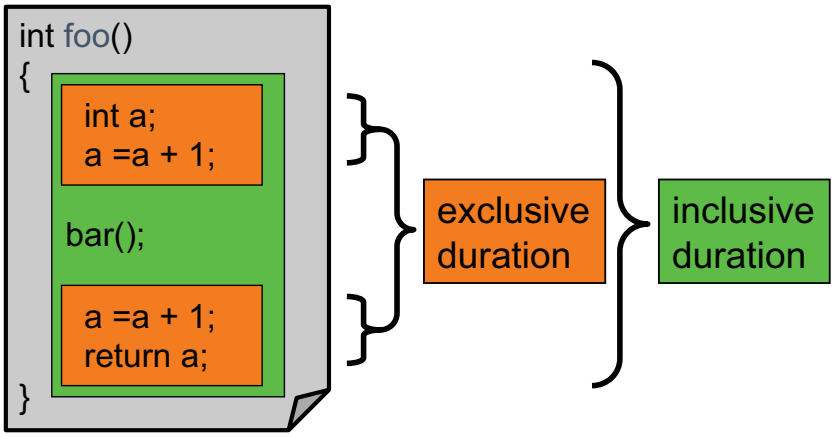


- Tracing shows you when the events take place on a timeline

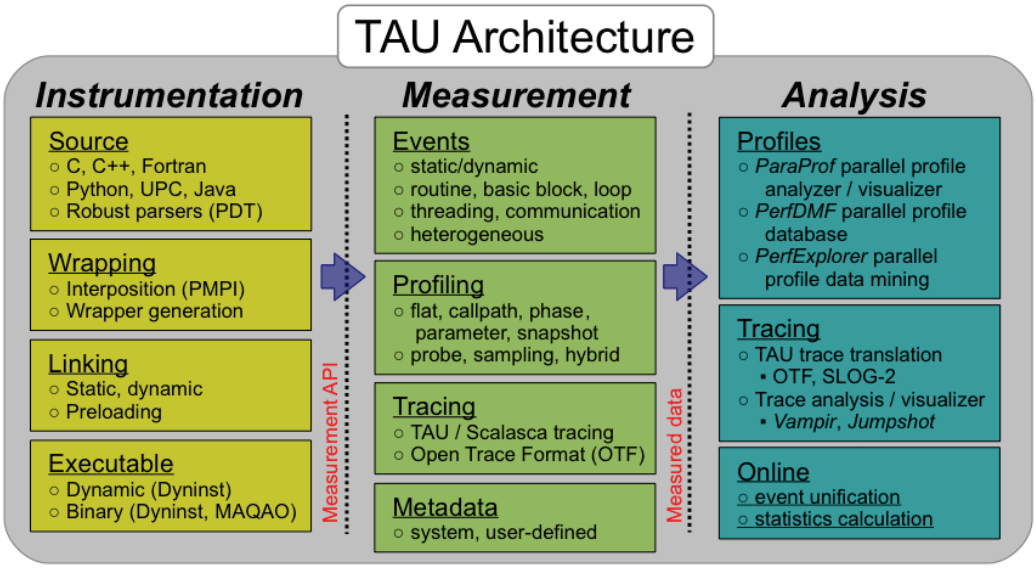
- Metrics can be time or hardware performance counters (cache misses, instructions)
- TAU can automatically instrument your source code using a package called PDT for routines, loops, I/O, memory, phases, etc.

TAU Tutorial : Inclusive Vs Exclusive Measurements

- Performance with respect to code regions
- Exclusive measurements for region only
- Inclusive measurements includes child regions



TAU Tutorial : TAU Architecture and Wo



TAU Tutorial : Using TAU

- To instrument source code automatically using PDT

  Choose an appropriate TAU stub makefile:

  % module load tau

  or

  % export TAU_MAKEFILE=$TAU/Makefile.tau-papi-mpi-pdt-pgi

  % export TAU_OPTIONS= '-optVerbose …' (see tau_compiler.sh)

  % export PATH=$TAUDIR/craycnl/bin:$PATH

  Use tau_f90.sh, tau_cxx.sh, tau_upc.sh, or tau_cc.sh as F90, C++, UPC, or C compilers respectively:

  % mpif90 foo.f90 changes to

  % tau_f90.sh foo.f90

- Set runtime environment variables, execute application and analyze performance data:

  % pprof (for text based profile display)

  % paraprof (for GUI)

TAU Tutorial : Automatic Instrumentatio

- Use TAU's compiler wrappers
  - Simply replace `CXX` with `tau_cxx.sh, etc.`
  - Automatically instruments source code, links with TAU libraries.
- Use `tau_cc.sh` for C, `tau_f90.sh` for Fortran, `tau_upc.sh` for UPC, etc.

Before

```
CXX = mpicxx
F90 = mpif90
CXXFLAGS =
LIBS = -lm
OBJS = f1.o f2.o f3.o … fn.o

app: $(OBJS)
     $(CXX) $(LDFLAGS) $(OBJS) -o $@
     $(LIBS)
.cpp.o:
     $(CXX) $(CXXFLAGS) -c $<
```

After

```
CXX = tau_cxx.sh
F90 = tau_f90.sh
CXXFLAGS =
LIBS = -lm
OBJS = f1.o f2.o f3.o … fn.o

app: $(OBJS)
     $(CXX) $(LDFLAGS) $(OBJS) -o $@
     $(LIBS)
.cpp.o:
     $(CXX) $(CXXFLAGS) -c $<
```

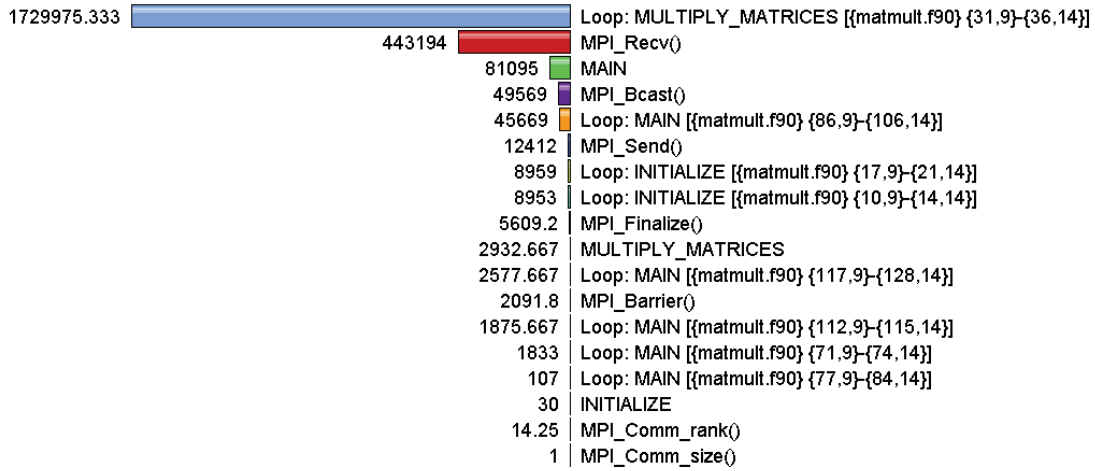
TAU Tutorial : Generating a loop level p

```
% export TAU_MAKEFILE=$TAU/Makefile.tau-intel-papi-mpi-pdt
% export TAU_OPTIONS='-optTauSelectFile=select.tau -optVerbose'
% cat select.tau
  BEGIN_INSTRUMENT_SECTION
  loops routine="#"
  END_INSTRUMENT_SECTION

% module load tau
% make F90=tau_f90.sh
(Or edit Makefile and change F90=tau_f90.sh)

% paraprof --pack app.ppk
  Move the app.ppk file to your desktop.

% paraprof app.ppk
```


TAU Tutorial : Loop Level Instrumentati

- Goal: What loops account for the most time? How much?
- Flat profile with wallclock time with loop instrumentation:

Metric: GET_TIME_OF_DAY
Value: Exclusive
Units: microseconds



TAU Tutorial

```
% source /pdc/vol/tau/tau.bashrc
% export TAU_MAKEFILE=$TAU/Makefile.tau-intel-papi-mpi-pdt
% export TAU_OPTIONS='-optTauSelectFile=select.tau -optVerbose'
% cat select.tau
  BEGIN_INSTRUMENT_SECTION
  loops routine="#"
  END_INSTRUMENT_SECTION
% make F90=tau_f90.sh
% msub -I
% export TAU_METRICS=TIME:PAPI_FP_INS:PAPI_L1_DCM
 OR
% export TAU_METRICS=TIME,PAPI_FP_INS,PAPI_L1_DCM
% mpirun -np 4  ./matmult
% paraprof --pack app.ppk
  Move the app.ppk file to your desktop.
% paraprof app.ppk
  Choose Options -> Show Derived Panel -> Click PAPI_FP_INS,
 Click "/",  Click TIME, Apply, Choose new metric by double
clicking.
```


TAU Tutorial : Computing FLOPS per lo

- Goal: What is the execution rate of my loops in MFLOPS?
- Flat profile with PAPI_FP_INS and time with loop instrumentation:

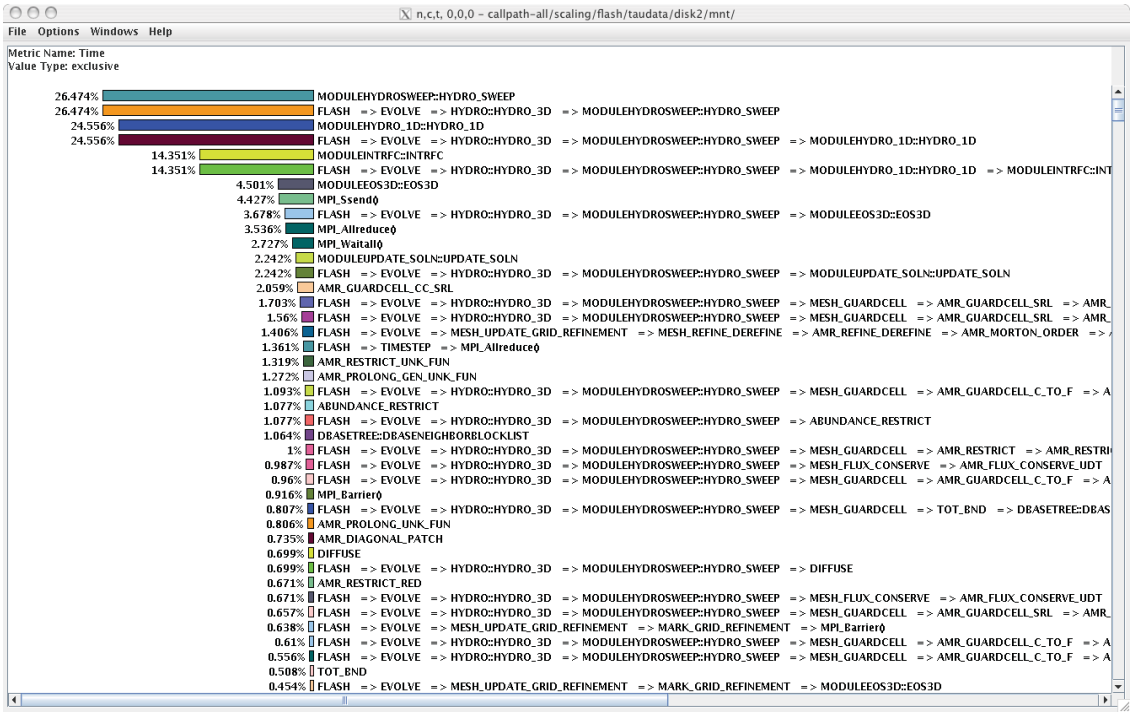
Metric: PAPI_FP_INS / GET_TIME_OF_DAY
Value: Exclusive
Units: Derived metric shown in microseconds format

770.699	Loop: MULTIPLY_MATRICES [{matmult.f90} {31,9}-{36,14}]
223.39	Loop: INITIALIZE [{matmult.f90} {10,9}-{14,14}]
223.24	Loop: INITIALIZE [{matmult.f90} {17,9}-{21,14}]
171.855	Loop: MAIN [{matmult.f90} {71,9}-{74,14}]
170.862	Loop: MAIN [{matmult.f90} {112,9}-{115,14}]
122.96	Loop: MAIN [{matmult.f90} {117,9}-{128,14}]
37.549	MULTIPLY_MATRICES
21.367	INITIALIZE
13.795	Loop: MAIN [{matmult.f90} {86,9}-{106,14}]
11	MPI_Comm_size()
8.935	Loop: MAIN [{matmult.f90} {77,9}-{84,14}]
1.131	MPI_Send()
0.794	MPI_Comm_rank()
0.647	MPI_Bcast()
0.355	MPI_Recv()
0.171	MPI_Barrier()
0.115	MPI_Finalize()
0.023	MAIN

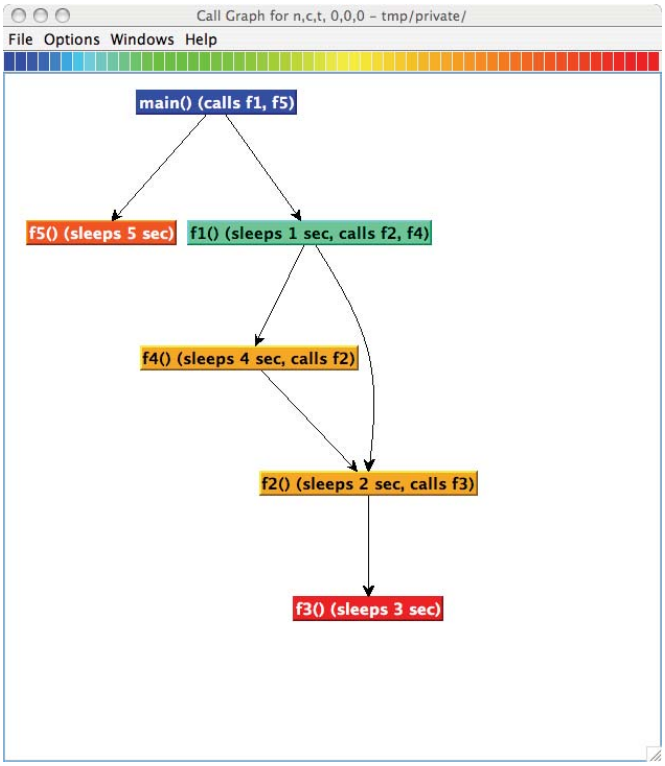
TAU Tutorial : Generate Callpath Profile

```
% source /pdc/vol/tau/tau.bashrc
% export TAU_MAKEFILE=$TAU/Makefile.tau-intel-papi-mpi-pdt
% make F90=tau_f90.sh
(Or edit Makefile and change F90=tau_f90.sh)

% msub -I
% export TAU_CALLPATH=1
% export TAU_CALLPATH_DEPTH=100
(truncates all calling paths to a specified depth)
% mpirun -np 4    ./a.out
% paraprof --pack app.ppk
  Move the app.ppk file to your desktop.
% paraprof app.ppk
(Windows -> Thread -> Call Graph)
```


TAU Tutorial : Callpath Profile



TAU Tutorial : ParaProf Call Graph Wind



TAU Tutorial

Example

Thank you

CCMT

UQ with DAKOTA

Chanyoung Park

UB Team

(02/19/2015)

UF UNIVERSITY of FLORIDA

Outlines

- **Why Validation and why UQ?**
- **Overview of DAKOTA**
- **Coupling DAKOTA and Simulation**
- **Concurrent Simulation Executions**

Why Validation?

- Blind predictions of lateral midpoint displacement

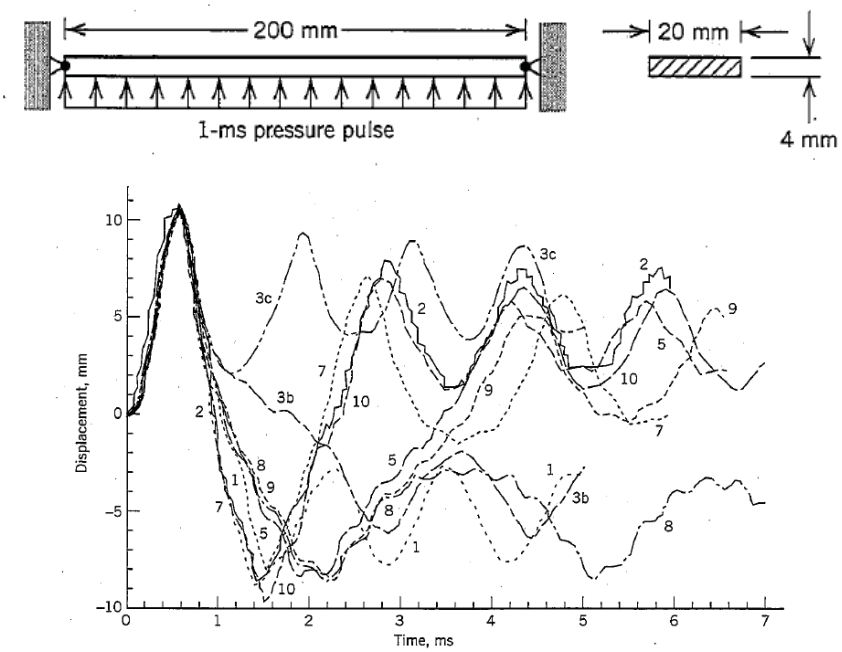
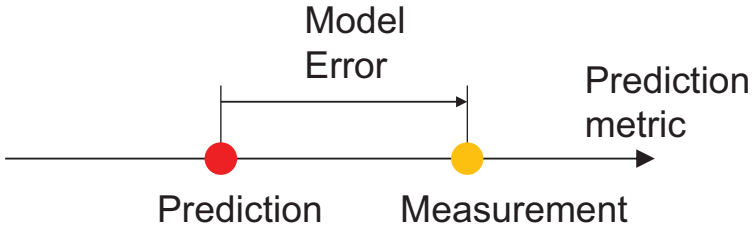


Fig. 1.5-1. Lateral midpoint displacement versus time for a beam loaded by a pressure pulse [1.6] The material is elastic–perfectly plastic. Plots were generated by various users and various codes.

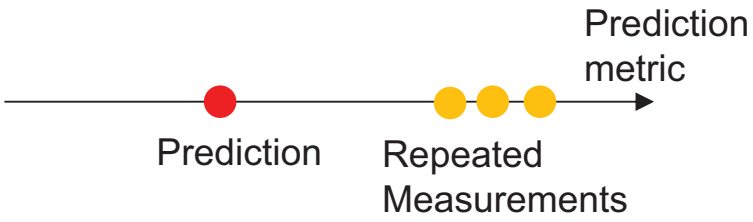
The Goal of Validation

- Figure out model error
  - Model error
    = physical model error + numerical model error



Why UQ?

- **Typical uncertainty sources in validation**
  - Model uncertainty
  - Measurement uncertainty in prediction metrics

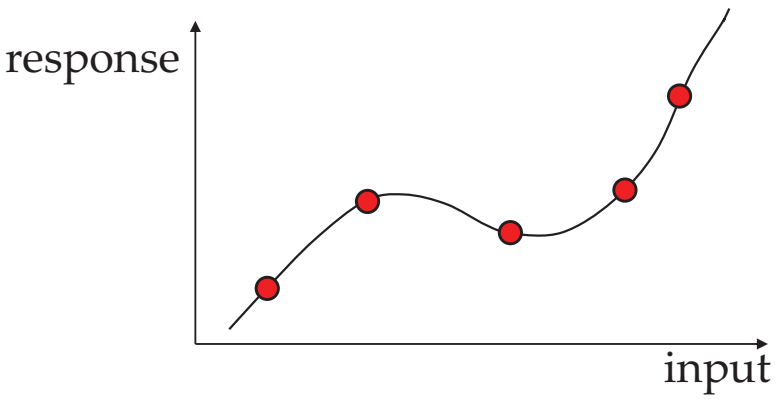


  - Measurement uncertainty in inputs
    - Thickness of very thin panel
    - Volume fraction of particles

Why UQ?

- **Inherent uncertainty (variability)**
  - What is variation in response?

  - Surrogate model (curve fitting)

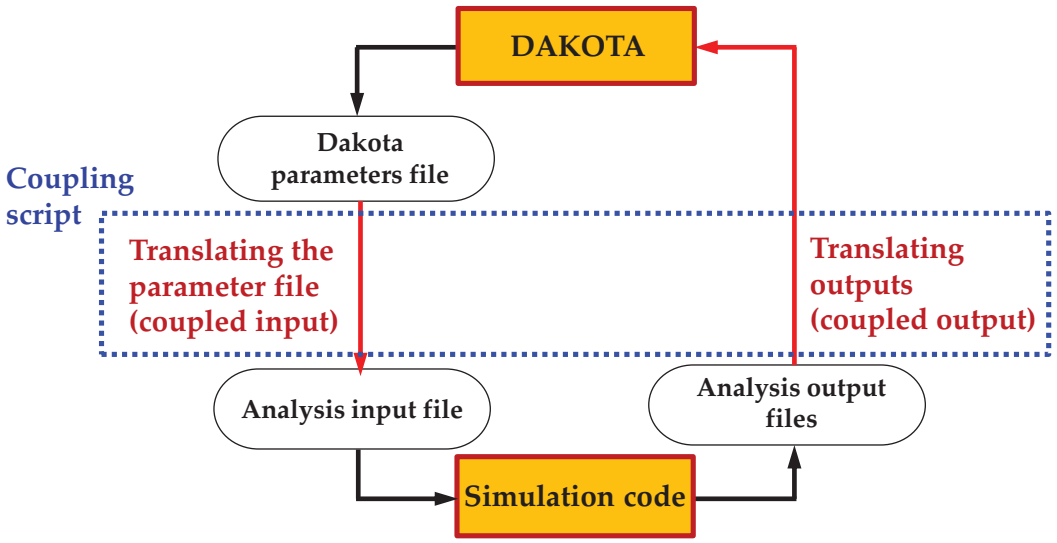


Coupling DAKOTA and Simulation

- Coupling types
  - **Closely coupled interface**: Automated analysis code control
  - **Loosely coupled interface**: Needs user control between Dakota and Analysis codes

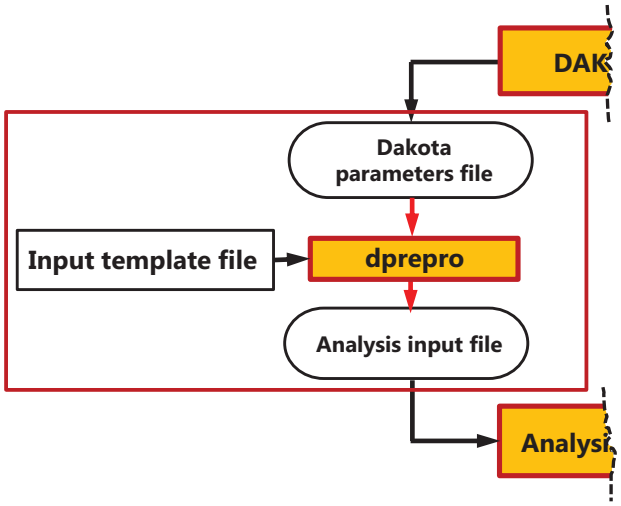
Closely Coupled Interface

- Closely coupled interface between Dakota and analysis codes
  - Dakota communicates with the simulation through files with predefined format



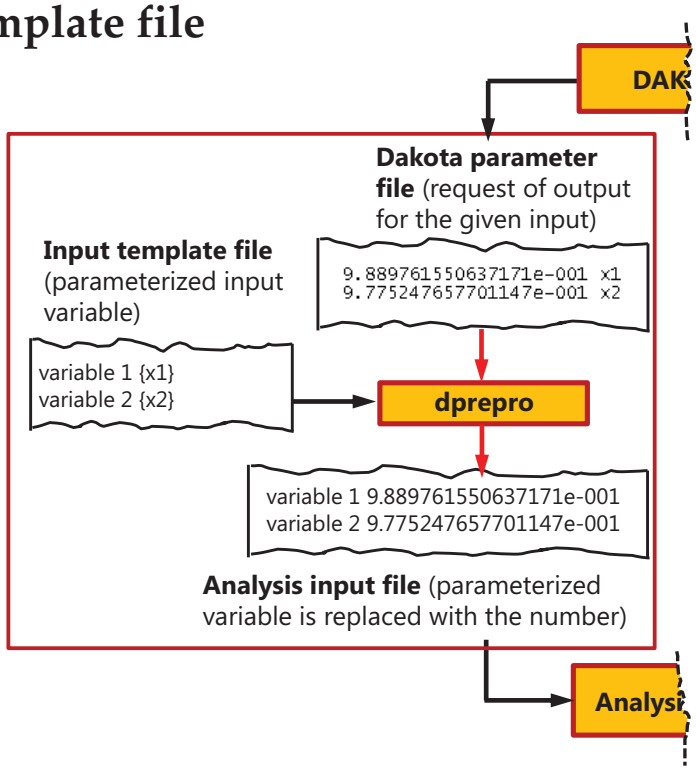
Coupled Input

- Input coupling with "dprepro" and a input template file
  - Dakota parameter files deliver information in a pre-defined format
  - dprepro script translates the Dakota parameter file into an analysis input file using the input template file



Coupled Input Example

- Input coupling with "dprepro" and a input template file



Coupled Input Example

Examples of a Dakota parameter file and a template input file

- A dakota parameter file and template input file for solving an optimization problem of the Rosenbrock function

A Dakota parameter file

```
                        2 variables
9.889761550637171e-001 x1
9.775247657701147e-001 x2
                        1 functions
                        1 ASV_1:obj_fn
                        2 derivative_variables
                        1 DVV_1:x1
                        2 DVV_2:x2
                        0 analysis_components
                      123 eval_id
```

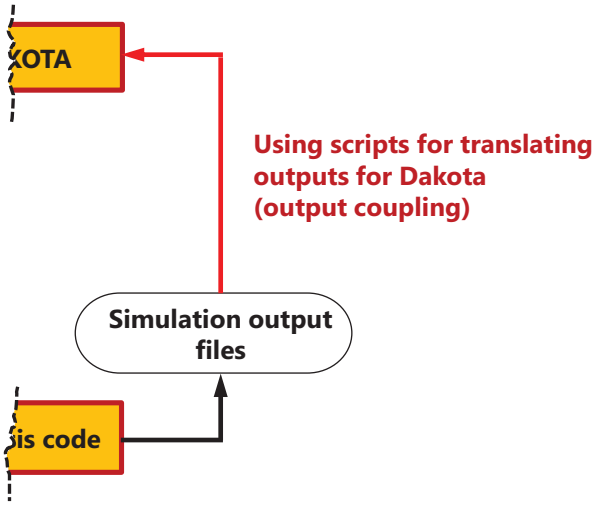
```
Title of Model: Rosenbrock black box
#################################################################################
# Description:  This is an input file to the Rosenbrock black box
#               Fortran simulator.  This simulator is structured so
#               as to resemble the input/output from an engineering
#               simulation code, even though Rosenbrock's function
#               is a simple analytic function.  The node, element,
#               and material blocks are dummy inputs.
#
# Input:  x1 and x2
# Output: objective function value
#################################################################################
node 1 location 0.0 0.0
node 2 location 0.0 1.0
node 3 location 1.0 0.0
node 4 location 1.0 1.0
node 5 location 2.0 0.0
node 6 location 2.0 1.0
node 7 location 3.0 0.0
node 8 location 3.0 1.0
element 1 nodes 1 3 4 2
element 2 nodes 3 5 6 4
element 3 nodes 5 7 8 6
element 4 nodes 7 9 10 8
material 1 elements 1 2
material 2 elements 3 4
variable 1 {x1}
variable 2 {x2}
end
```

A template input file

Coupling Output

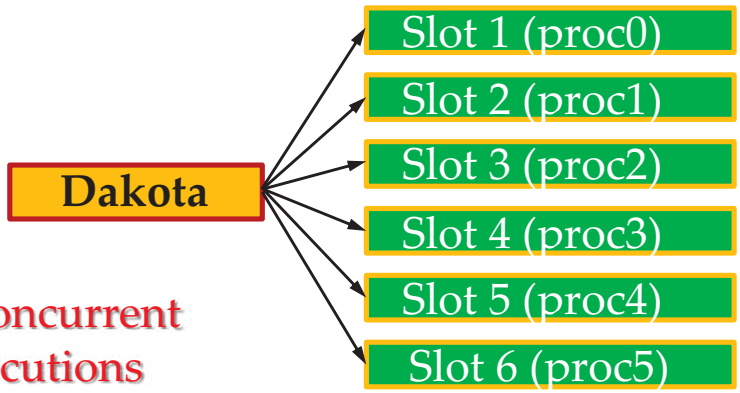
Output coupling

- Output coupling is composed of two steps:
  1) extracting required outputs from simulation output files
  2) write them on DAKOTA input file
- Output coupling is open for all possible ways



Case 1

- For the case of a single simulation run is cheap
  - Dakota manages "executions" of simulation
  - Dakota controls simulation execution schedule (because not every simulations requires exactly the same amount of time)
  - Users are supposed to separate directories for different executions to prevent sharing one input file for different simulation calls

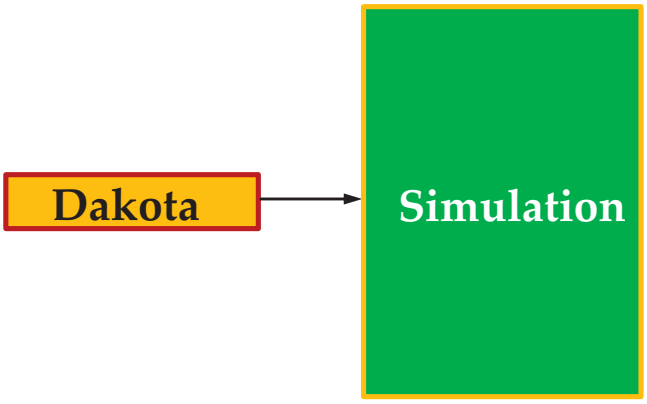


6 concurrent executions

Processors in parenthesis are relative process indices

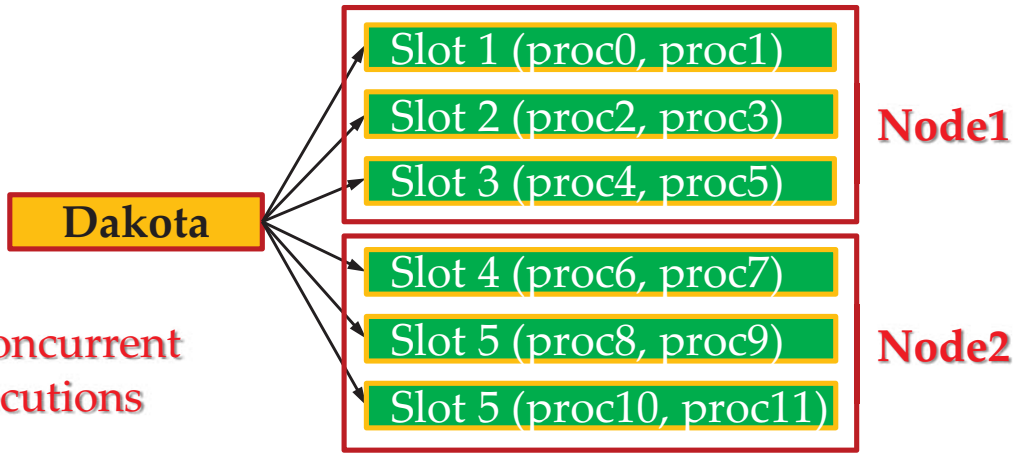
Case 2

- When Users want to directly control simulation execution schedule
  - Dakota provides parameter files
  - No "execution" schedule control from Dakota
  - The least beneficial approach of using Dakota



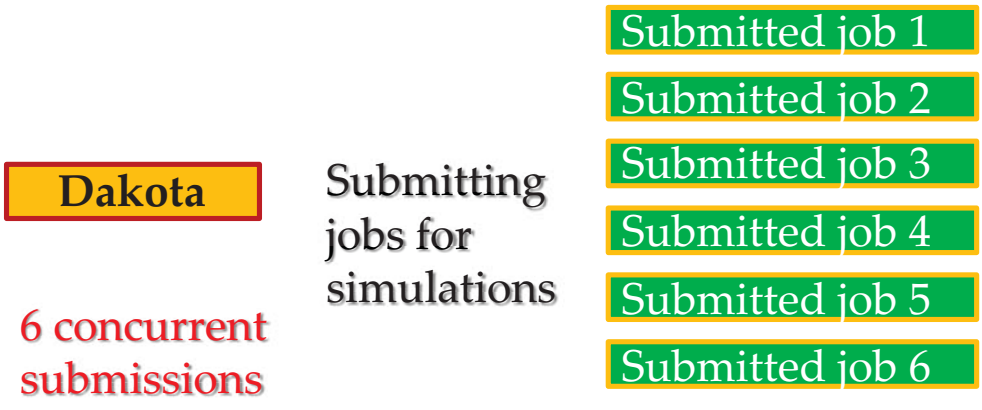
Case 3

- **For the case of a single simulation run is not expensive**
  - Dakota manages "executions" of simulation
  - 1) Dakota control which processors will be used for which task 2) allocate processors for a particular simulation
  - openMpi may cause troubles



Case 4

- **For the case of a single simulation run is very expensive (?)**
  - Dakota controls "submissions" for simulation
  - Dakota keep submitting another job for keeping the number of submitted jobs is 6 at the same time



Exemplary Coupling Process

- **Minimum required items before coupling**
  - Input files (for templates)
  - Files except input files to run a simulation
  - Simulation end signal

Install DAKOTA on HPC

- **Install Dakota on HPC (for Dakota 5.4)**
  - Download and extract Dakota source file
  - Check if the HPC system has libraries that DAKOTA requires:
    cmake (≥2.8.9)
    boost (≥1.49.0)
    LAPACK (Linear Algebra PACKage)
    BLAS(Basic Linear Algebra Solver)
    MPI / OpenMPI
  - Load required modules
  - Modify DakotaBuildTemplate.cmake as needed
  - Compile / build / install

UF UNIVERSITY of FLORIDA

Thank you!

UF UNIVERSITY of FLORIDA

Overview of DAKOTA

- **Applications**
  - **Optimization**: to minimize cost or maximize system performance subject to constraints
  - **Uncertainty quantification**: to compute probabilistic information about response
  - **Design of experiments**: to generate sampling points for good coverage of the input parameter space
  - **Calibration** (parameter estimation): to maximize agreement between simulation outputs and experimental data
  - **Parameter studies**: to study the effect of parametric changes within simulation models

Concurrent Simulation Executions

Application parallelism

- There are four cases of concurrent simulation executions

Case	Dakota	Simulation	Notes
1	parallel	serial	M-1 simultaneous simulation instances
2	serial	parallel	A simultaneous simulation instance on N processors
3	serial	parallel	(M-1)/N simultaneous N processor per job
4	serial	parallel	Submit *expensive* N processor application jobs to a scheduler

- "1 processor" for case 1 and 3 is for Dakota

Coupling Output Example

Examples of an analysis output file and a result file for Dakota

- Output coupling is composed of two steps:
  1) extracting required outputs from simulation output files
  2) write them on DAKOTA input file

A simulation output file

```
Beginning execution of model: Rosenbrock black box
Set up complete.
Reading nodes.
Reading elements.
Reading materials.
Checking connectivity...OK
************************************************************

Input value for x1 =  0.9832748447306406E+00
Input value for x2 =  0.9673642199304130E+00

Computing solution...Done
************************************************************
Function value =  0.3083318853853630E-03
Function gradient = [ -0.2437923278017054E+00  0.1069599300695057E+00 ]
```

A Dakota input file

```
 0.3083318853853630E-03
 [ -0.2437923278017054E+00  0.1069599300695057E+00 ]
```