Performance Analysis & Optimization/Tuning Tools

Zhengxiong Hou

Fall, 2022

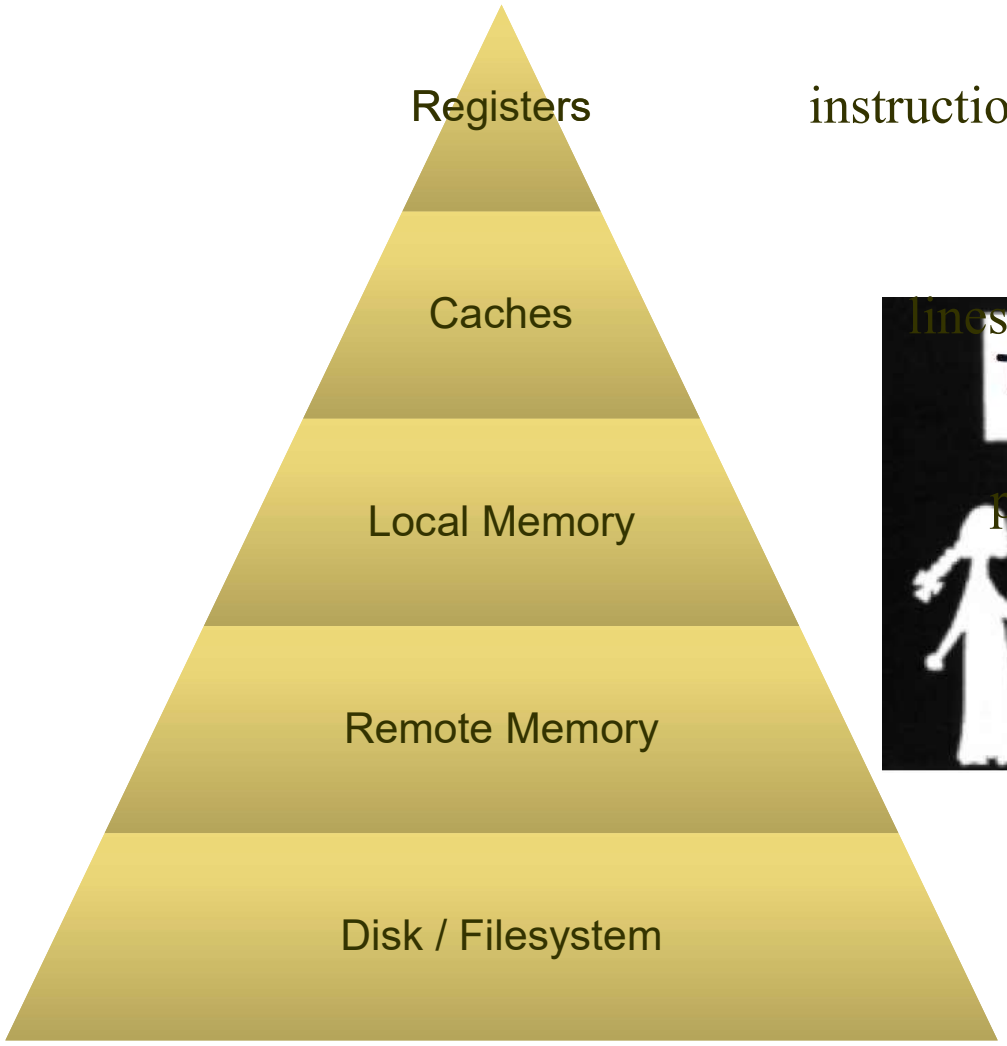
Topics Overview

- Performance Benchmarks
- Performance Debugging
- Performance Analysis Tools
- Performance Optimization

Specific Facets of Performance

- **Serial**
  - Leverage ILP (Instruction Level Parallelism) on the processor
  - Feed the pipelines
  - Reuse data in cache
  - Exploit data locality
- **Parallel**
  - Expose task level concurrency (or TLP thread level parallelism)
  - Minimizing latency effects (memory/cache access)
  - Maximizing work vs. communication

Performance is Hierarchical



instructions & operands

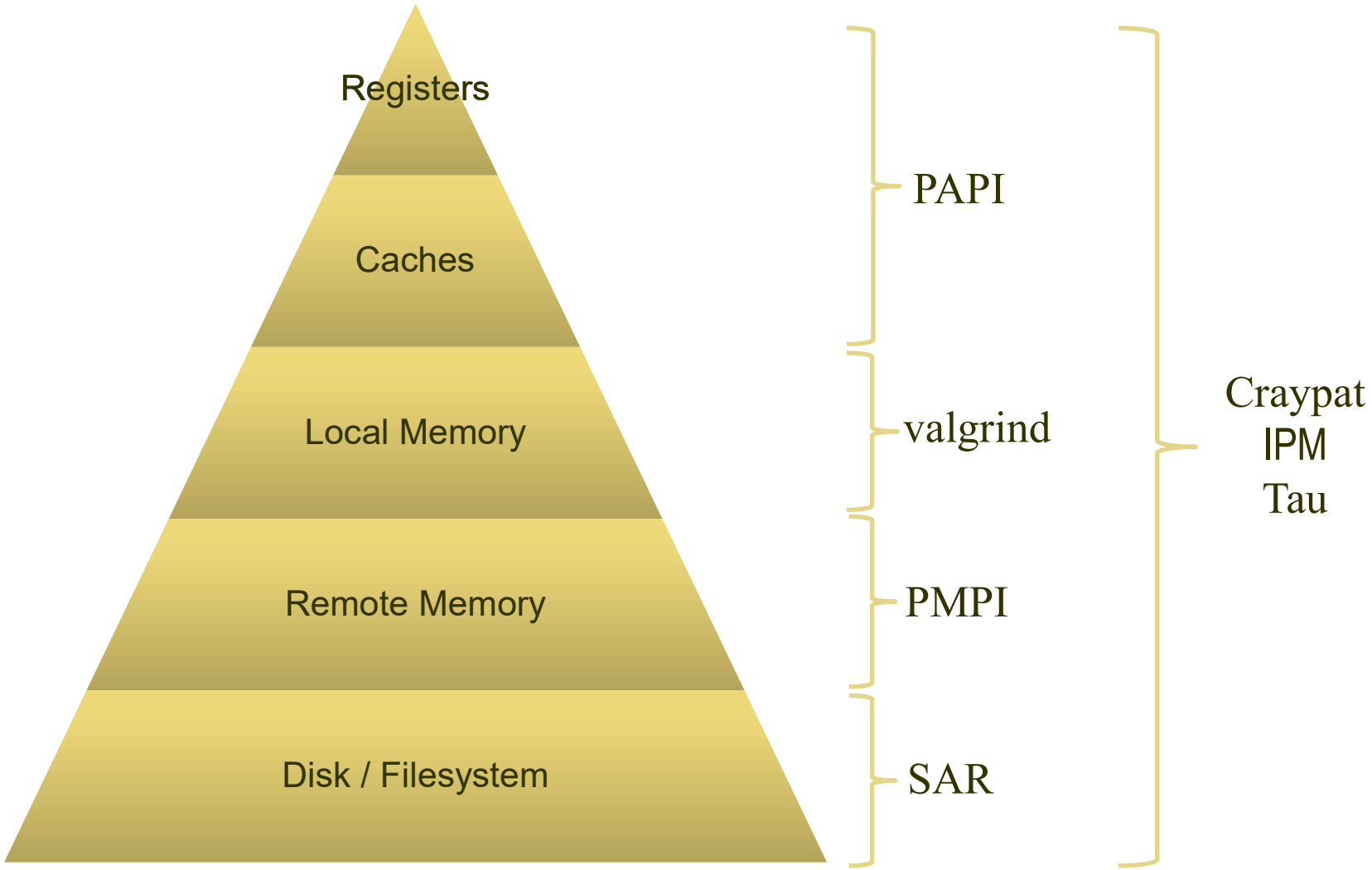
lines

pages

messages

blocks, files

Analysis Tools are Hierarchical



Performance Benchmarks

- **Basic benchmarks**

  -CPU: SPEC CPU

  -memory: Stream, LMbench

  -I/O: IOZone, IOR

  -network: Netperf

  point-to-point : OSU Micro-Benchmarks

  collective communication: Intel MPI Benchmarks **(**MPI_Allreduce，MPI_Barrier, MPI_Bcast, MPI_Allgather, MPI_Reduce, MPI_Alltoall)

- **Math Libraries**

  -FFT, LAPACK,BLAS,BLACS, ScaLAPACK

- **Parallel benchmarks**

  -Linpack, HPCG

  -NPB (Nasa Parallel Benchmark)

Memory benchmark

- A low-level benchmark is a program that tries to test some specific feature of the architecture like, e.g., peak performance or memory bandwidth.
- One of the prominent examples is the *vector triad*, introduced by Schönauer
  - It comprises a nested loop, the inner level executing a multiplyadd operation on the elements of three vectors and storing the result in a fourth.
  - The purpose of this benchmark is to measure the performance of data transfers between memory and arithmetic units of a processor.

```
1  double precision, dimension(N) :: A,B,C,D
2  double precision :: S,E,MFLOPS
3
4  do i=1,N                              !initialize arrays
5    A(i) = 0.d0; B(i) = 1.d0
6    C(i) = 2.d0; D(i) = 3.d0
7  enddo
8
9  call get_walltime(S)                  ! get time stamp
10 do j=1,R
11   do i=1,N
12     A(i) = B(i) + C(i) * D(i)         ! 3 loads, 1 store
13   enddo
14   if(A(2).lt.0) call dummy(A,B,C,D) ! prevent loop interchange
15 enddo
16 call get_walltime(E)                  ! get time stamp
17 MFLOPS = R*N*2.d0/((E-S)*1.d6)        ! compute MFlop/sec rate
```

- On the inner level, three *load streams* for arrays B, C and D and one *store stream* for A are active.
- Depending on N, this loop might execute in a very small time, which would be hard to measure. The outer loop thus repeats the triad R times so that execution time becomes large enough to be accurately measurable.

STREAM: Sustainable Memory Bandwidth in High Performance Computers

- John D. McCalpin, *"Dr. Bandwidth"*

name	kernel	bytes/iter	FLOPS/iter
COPY:	a(i) = b(i)	16	0
SCALE:	a(i) = q*b(i)	16	1
SUM:	a(i) = b(i) + c(i)	24	1
TRIAD:	a(i) = b(i) + q*c(i)	24	2

STREAM2 is an attempt to extend the functionality of the STREAM benchmark in two important ways:

- STREAM2 measures sustained bandwidth at all levels of the cache hierarchy, and
- STREAM2 more clearly exposes the performance differences between reads and writes

STREAM2 is based on the same ideas as STREAM, but uses a different set of vector kernels:

- FILL: similar to bzero(), but fills with a constant instead of zero
- COPY: similar to bcopy(), and the same as STREAM Copy
- DAXPY: similar to STREAM Triad, but overwrites one of the input vectors instead of writing results to a third vector
- SUM: sum reduction on a single vector -- reads only, no writes

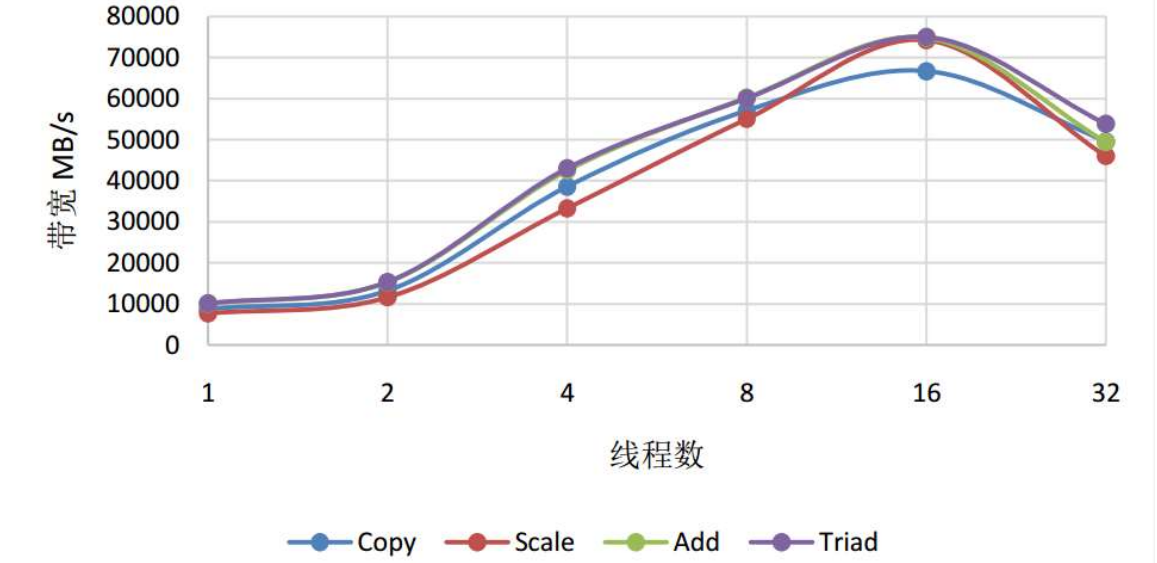
Kernel	Code	Bytes/iter read	Bytes/iter written	FLOPS/iter
FILL	a(i) = q	0 (+8)	8	0
COPY	a(i) = b(i)	8 (+8)	8	0
DAXPY	a(i) = a(i) + q*b(i)	16	8	2
SUM	sum = sum + a(i)	8	0	1

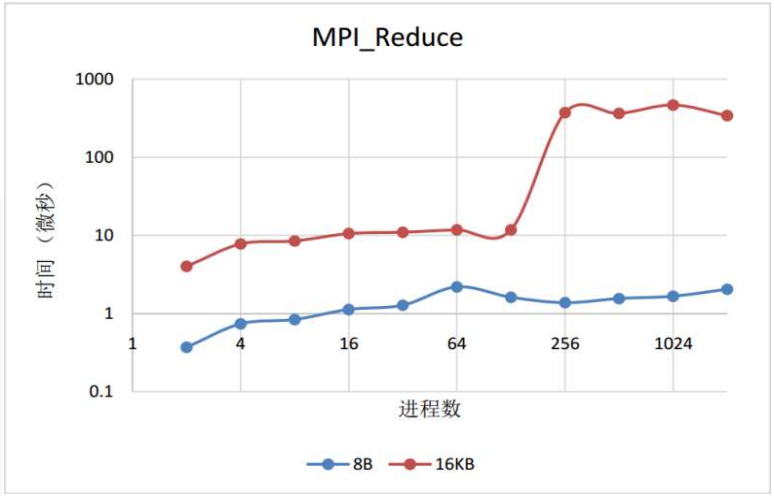
STREAM "Top20" results

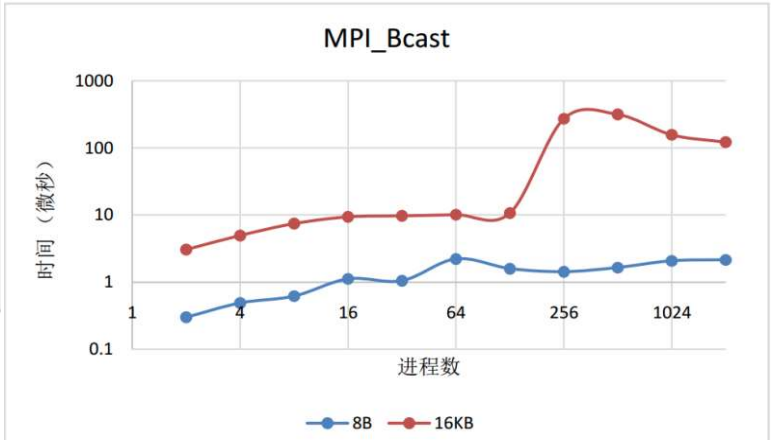
STREAM Memory Bandwidth -- John D. McCalpin, mccalpin@cs.virginia.edu
Revised to Mon Apr 3 19:18:55 CDT 2017

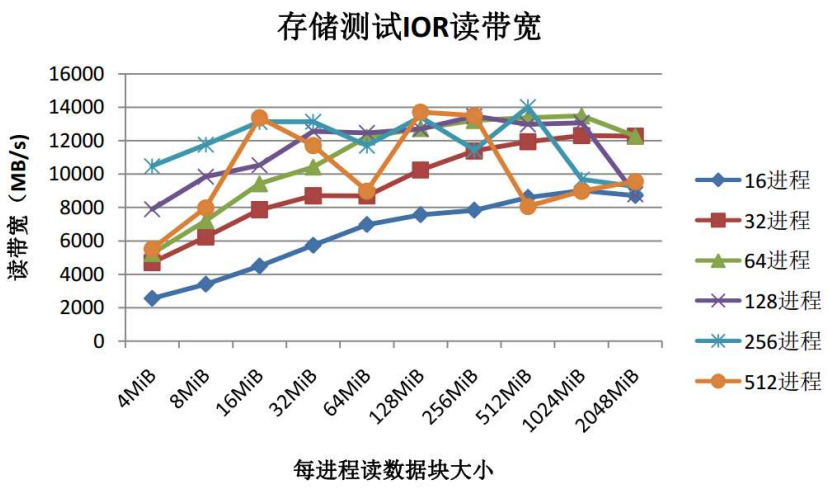
All results are in MB/s -- 1 MB=10^6 B, *not* 2^20 B

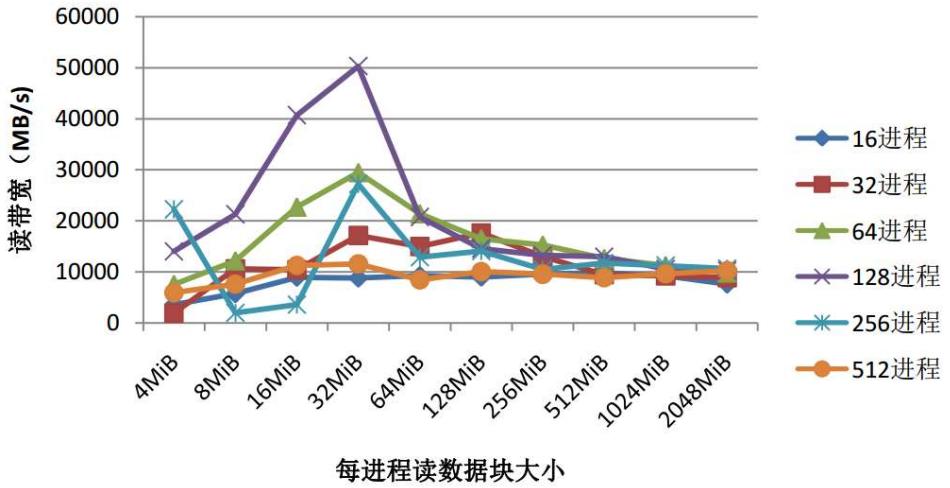
Sub. Date	Machine ID	ncpus	COPY	SCALE	ADD	TRIAD	
2015.07.10	SGI_UV_3000	3072	12799304.0	12815808.0	13838195.0	13826185.0	data
2012.08.14	SGI_Altix_UV_2000	2048	6591669.0	6592082.0	7128484.0	7139690.0	data
2016.01.13	ScaleMP_Xeon_E5-2680v3_64B	1534	5741247.0	5775190.0	6318785.0	6367015.0	data
2011.04.05	SGI_Altix_UV_1000	2048	5321074.0	5346667.0	5823380.0	5859367.0	data
2006.07.10	SGI_Altix_4700	1024	3661963.0	3677482.0	4385585.0	4350166.0	data
2013.03.26	Fujitsu_SPARC_M10-4S	1024	3474998.0	3500800.0	3956102.0	4002703.0	data
2011.06.06	ScaleMP_Xeon_X6560_64B	768	1493963.0	2112630.0	2252598.0	2259709.0	data
2017.04.04	Fujitsu_SPARC_M12-2S	192	1322423.0	1299737.0	1479182.0	1530865.0	data
2004.12.22	SGI_Altix_3700_Bx2	512	906388.0	870211.0	1055179.0	1119913.0	data
2003.11.13	SGI_Altix_3000	512	854062.0	854338.0	1008594.0	1007828.0	data
2003.10.02	NEC_SX-7	32	876174.7	865144.1	869179.2	872259.1	data
2008.04.07	IBM_Power_595	64	679207.2	624707.8	777334.8	805804.6	data
2013.09.12	Oracle_SPARC_T5-8	128	604648.0	611264.0	622572.0	642884.0	data
1999.12.07	NEC_SX-5-16A	16	607492.0	590390.0	607412.0	583069.0	data
2009.08.10	ScaleMP_XeonX5570_vSMP_16B	128	437571.0	431726.0	442722.0	445869.0	data
1997.06.10	NEC_SX-4	32	434784.0	432886.0	437358.0	436954.0	data
2004.08.11	HP_AlphaServer_GS1280-1300	64	407351.0	400142.0	437010.0	431450.0	data
1996.11.21	Cray_T932_321024-3E	32	310721.0	302182.0	359841.0	359270.0	data
2014.04.24	Oracle_Sun_Server_X4-4	60	221370.0	221944.0	244588.0	245068.0	data
2007.04.17	Fujitsu/Sun_Enterprise_M9000	128	224401.0	223113.0	224271.0	227059.0	data


80000
70000
60000
50000
40000
30000
20000
10000
0
带宽 MB/s
1
2
4
8
16
32
线程数
Copy
Scale
Add
Triad


MPI_Reduce
1000
100
10
1
0.1
时间（微秒）
1
4
16
64
256
1024
进程数
8B
16KB


MPI_Bcast
1000
100
10
1
0.1
时间（微秒）
1
4
16
64
256
1024
进程数
8B
16KB


存储测试IOR读带宽
16000
14000
12000
10000
8000
6000
4000
2000
0
读带宽（MB/s）
4MiB
8MiB
16MiB
32MiB
64MiB
128MiB
256MiB
512MiB
1024MiB
2048MiB
每进程读数据块大小
16进程
32进程
64进程
128进程
256进程
512进程


60000
50000
40000
30000
20000
10000
0
读带宽（MB/s）
4MiB
8MiB
16MiB
32MiB
64MiB
128MiB
256MiB
512MiB
1024MiB
2048MiB
每进程读数据块大小
16进程
32进程
64进程
128进程
256进程
512进程

How to run HPL/HPCG directly

- **HPL-CPU version**

  mpirun -np 2448 -machinefile ./nodes ./xhpl

```
ibc01b03
ibc01b08
```

```
ibc01b03:8
ibc01b08:8
```

```
ibc01b03
ibc01b03
ibc01b03
ibc01b03
ibc01b03
ibc01b03
ibc01b03
ibc01b03
ibc01b08
ibc01b08
ibc01b08
ibc01b08
ibc01b08
ibc01b08
ibc01b08
ibc01b08
```

- **HPL-GPU version**

  mpirun -np 8 --hostfile ./hosts ./linpack_GPU_dynamic

```
ibasc01   slots=2
ibasc02   slots=2
ibasc03   slots=2
ibasc04   slots=2
```

- **HPCG-GPU version**

  mpirun -np 8 --hostfile ./hosts ./HPCG*

```
================================================================================
HPLinpack 2.1  --  High-Performance Linpack benchmark  --   October 26, 2012
Written by A. Petitet and R. Clint Whaley,  Innovative Computing Laboratory, UTK
Modified by Piotr Luszczek, Innovative Computing Laboratory, UTK
Modified by Julien Langou, University of Colorado Denver
================================================================================

An explanation of the input/output parameters follows:
T/V    : Wall time / encoded variant.
N      : The order of the coefficient matrix A.
NB     : The partitioning blocking factor.
P      : The number of process rows.
Q      : The number of process columns.
Time   : Time in seconds to solve the linear system.
Gflops : Rate of execution for solving the linear system.

The following parameter values will be used:

N      : 1008784
NB     :     112
PMAP   : Row-major process mapping
P      :      16
Q      :     153
PFACT  :    Left
NBMIN  :       4       2
NDIV   :       2
RFACT  :   Crout
BCAST  :   1ring
DEPTH  :       0
SWAP   : Mix (threshold = 256)
L1     : no-transposed form
U      : no-transposed form
EQUIL  : no
ALIGN  :    8 double precision words

--------------------------------------------------------------------------------

- The matrix A is randomly generated for each test.
- The following scaled residual check will be computed:
      ||Ax-b||_oo / ( eps * ( || x ||_oo * || A ||_oo + || b ||_oo ) * N )
- The relative machine precision (eps) is taken to be           1.110223e-16
- Computational tests pass if scaled residuals are less than                16.0

Column=005152 Fraction=0.005 Mflops=40105430.61
Column=010192 Fraction=0.010 Mflops=41632483.51
Column=015232 Fraction=0.015 Mflops=41834078.68
Column=020272 Fraction=0.020 Mflops=40537624.87
Column=025312 Fraction=0.025 Mflops=40838240.26
Column=030352 Fraction=0.030 Mflops=40917928.72
Column=035392 Fraction=0.035 Mflops=40328128.68
Column=040432 Fraction=0.040 Mflops=40348740.97
Column=045472 Fraction=0.045 Mflops=40466855.09
Column=050512 Fraction=0.050 Mflops=40681170.80
Column=055552 Fraction=0.055 Mflops=40279725.82
Column=060592 Fraction=0.060 Mflops=40291462.39
Column=065632 Fraction=0.065 Mflops=40425071.70
Column=070672 Fraction=0.070 Mflops=40133714.79
Column=075712 Fraction=0.075 Mflops=40107715.94
```

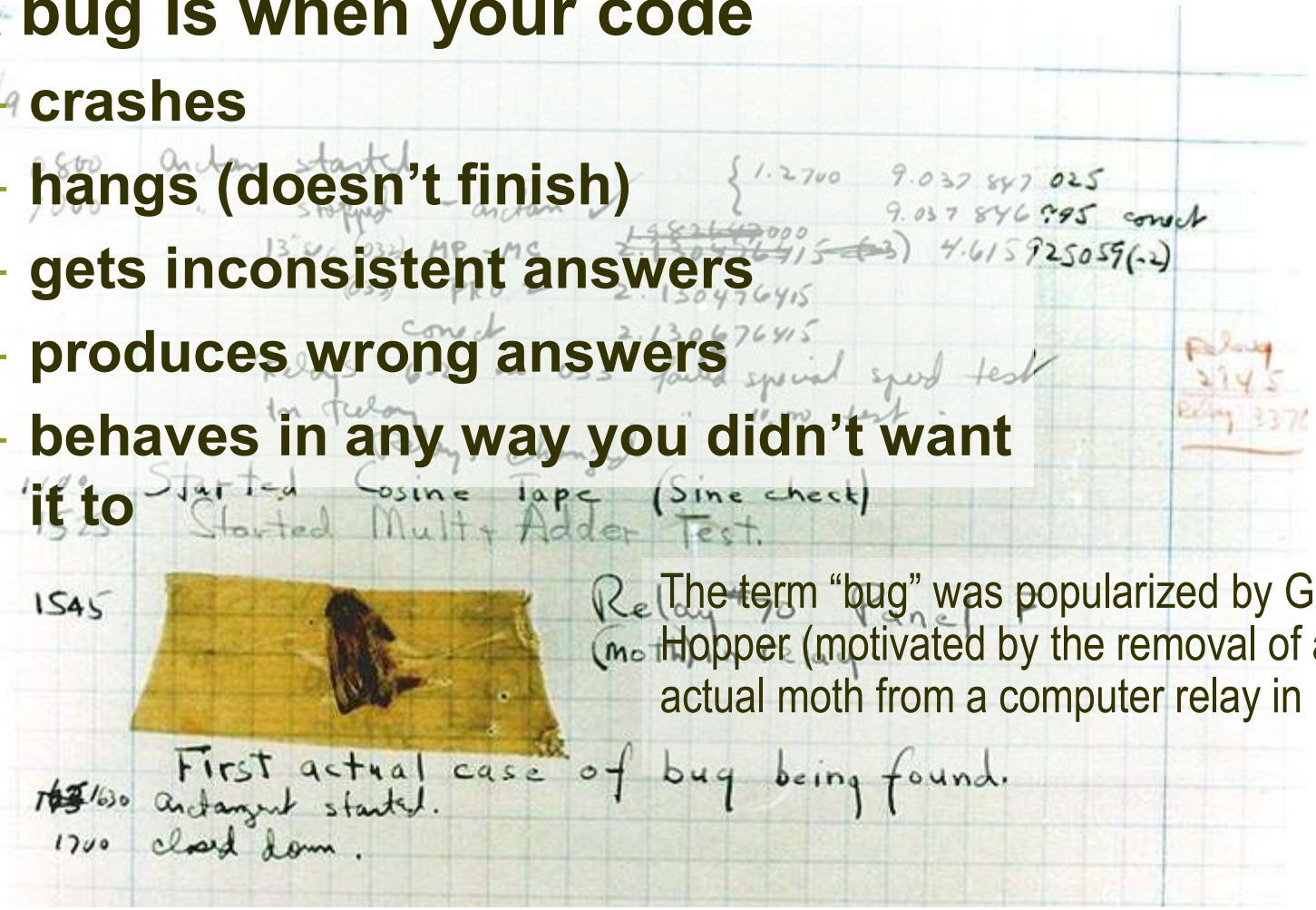
```
Column=539728 Fraction=0.535 Mflops=38437138.35
Column=559888 Fraction=0.555 Mflops=38400998.57
Column=580160 Fraction=0.575 Mflops=38362264.33
Column=600320 Fraction=0.595 Mflops=38299628.70
Column=620480 Fraction=0.615 Mflops=38253703.87
Column=640640 Fraction=0.635 Mflops=38210817.73
Column=660800 Fraction=0.655 Mflops=38160088.72
Column=680960 Fraction=0.675 Mflops=38113189.32
Column=701120 Fraction=0.695 Mflops=38064256.79
Column=802032 Fraction=0.795 Mflops=37806483.65
Column=902944 Fraction=0.895 Mflops=37553354.53
Column=1003744 Fraction=0.995 Mflops=37375309.15
================================================================================
T/V                N    NB     P     Q               Time                 Gflops
--------------------------------------------------------------------------------
WR00C2L4     1008784   112    16   153           18316.50           3.73647e+04
HPL_pdgesv() start time Thu Jul  4 12:30:12 2013

HPL_pdgesv() end time   Thu Jul  4 17:35:28 2013


Column=701120 Fraction=0.695 Mflops=39410203.10
Column=802032 Fraction=0.795 Mflops=39110003.61
Column=902944 Fraction=0.895 Mflops=38833239.33
Column=1003744 Fraction=0.995 Mflops=38644112.75
================================================================================
T/V                N    NB     P     Q               Time                 Gflops
--------------------------------------------------------------------------------
WR00C2L2     1008784   112    16   153           17714.84           3.86338e+04
HPL_pdgesv() start time Thu Jul  4 17:35:52 2013

HPL_pdgesv() end time   Thu Jul  4 22:31:06 2013

--------------------------------------------------------------------------------
||Ax-b||_oo/(eps*(||A||_oo*||x||_oo+||b||_oo)*N)=        0.0007179 ..... PASSED
================================================================================

Finished      2 tests with the following results:
              2 tests completed and passed residual checks,
              0 tests completed and failed residual checks,
              0 tests skipped because of illegal input values.
--------------------------------------------------------------------------------

End of Tests.
================================================================================
Done: Thu Jul  4 22:31:28 CST 2013
```

Topics Overview

- Performance Benchmarks
- Performance Debugging
- Performance Analysis Tools
- Performance Optimization

What is a Bug?

- A bug is when your code
  - ~~is~~ crashes
  - hangs (doesn't finish)
  - gets inconsistent answers
  - produces wrong answers
  - behaves in any way you didn't want it to



The term "bug" was popularized by Grace Hopper (motivated by the removal of an actual moth from a computer relay in 1947)

Common Causes of Bugs

- "Serial" (Sequential might be a better word)
  - Invalid memory references
  - Array reference out of bounds
  - Divide by zero
  - Use of uninitialized variables
- Parallel Let's concentrate on these
  - Unmatched sends/receives
  - Blocking receive before corresponding send
  - Out of order collectives
  - Race conditions
  - Unintentionally modifying shared memory structures

What to Do if You Have a Bug?

- **Find It**
  - **You want to locate the part of your code that isn't doing what it's designed to do**
- **Fix It**
  - **Figure out how to solve it and implement a solution**
- **Check It**
  - **Run it to check for proper behavior**

Find It: Tools

- printf, write
  - Versatile, sometimes useful
  - Doesn't scale well
  - Not interactive
  - Fishing expedition
- Compiler / Runtime
  - Bounds checking, exception handling
  - Dereferencing of NULL pointers
  - Function and subroutine interface checking
- Serial gdb + friends
  - GNU debugger, serial, command-line interface
  - See "man gdb"
- Parallel debuggers
  - DDT
  - Totalview
- Memory debuggers
  - Valgrind

See NERSC web site
https://www.nersc.gov/users/software/debugging-and-profiling/

Debugging

- Debugging parallel codes can be incredibly difficult, particularly as codes scale upwards.
- The good news is that there are some excellent debuggers available to assist:
  - Threaded - pthreads and OpenMP
  - MPI
  - GPU / accelerator
  - Hybrid
- **Parallel debugging tools**
  - **TotalView** from RogueWave Software
  - **DDT (Distributed Debugging Tool)** from Allinea
  - **Inspector** from Intel
  - All of these tools have a learning curve associated with them - some more than others.

Parallel Programming Bug

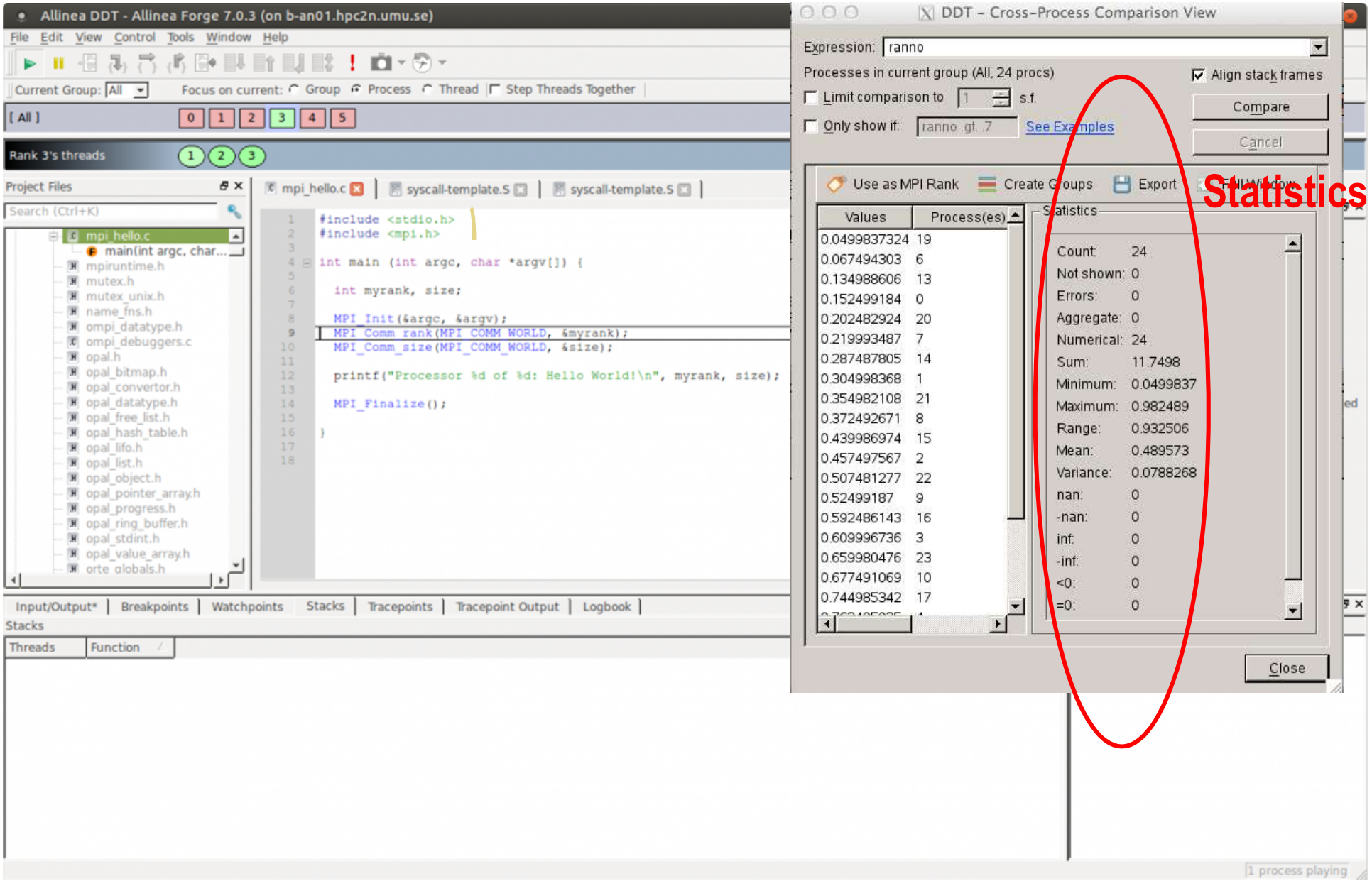
- **This code hangs because both Task 0 and Task N-1 are blocking on MPI_Recv**

```
if(task_no==0) {

     ret = MPI_Recv(&herBuffer, 50, MPI_DOUBLE,
totTasks-1, 0, MPI_COMM_WORLD, &status);
     ret = MPI_Send(&myBuffer, 50, MPI_DOUBLE, totTasks-
1, 0, MPI_COMM_WORLD);

} else if (task_no==(totTasks-1)) {

     ret = MPI_Recv(&herBuffer, 50, MPI_DOUBLE, 0, 0,
MPI_COMM_WORLD, &status);
     ret = MPI_Send(&myBuffer, 50, MPI_DOUBLE, 0, 0,
MPI_COMM_WORLD);

}
```

What About Massive Parallelism?

- With 10K+ tasks/threads/streams it's impossible to examine every parallel instance
- Make use of statistics and summaries
- Look for tasks that are doing something different
  - Amount of memory used
  - Number of calculations performed (from counters)
  - Number of MPI calls
  - Wall time used
  - Time spent in I/O
  - One or a few tasks paused at a different line of code

Distributed Debugging Tool (DDT)



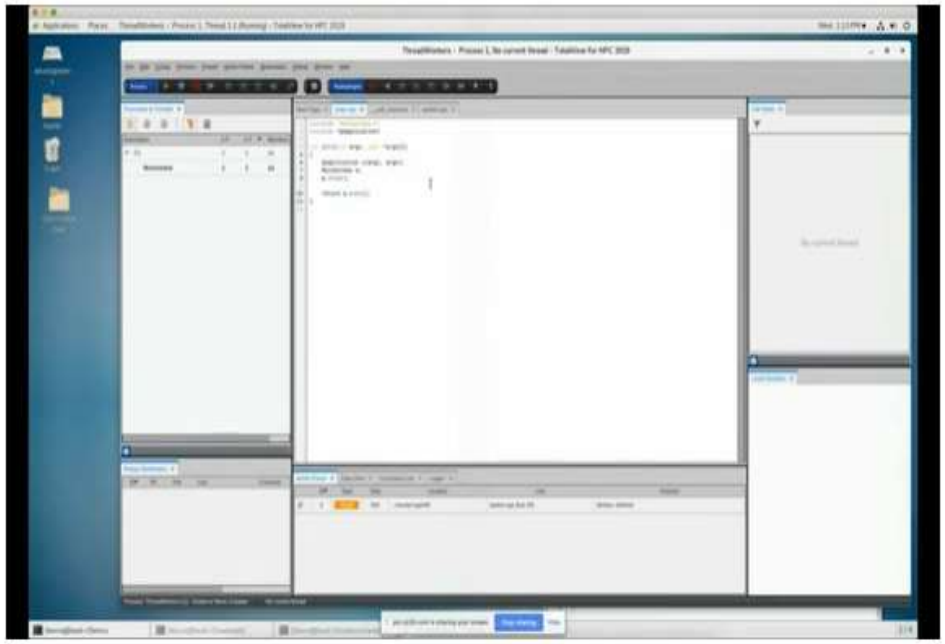
Explore TotalView's Features

Learn more about what you can accomplish with TotalView.

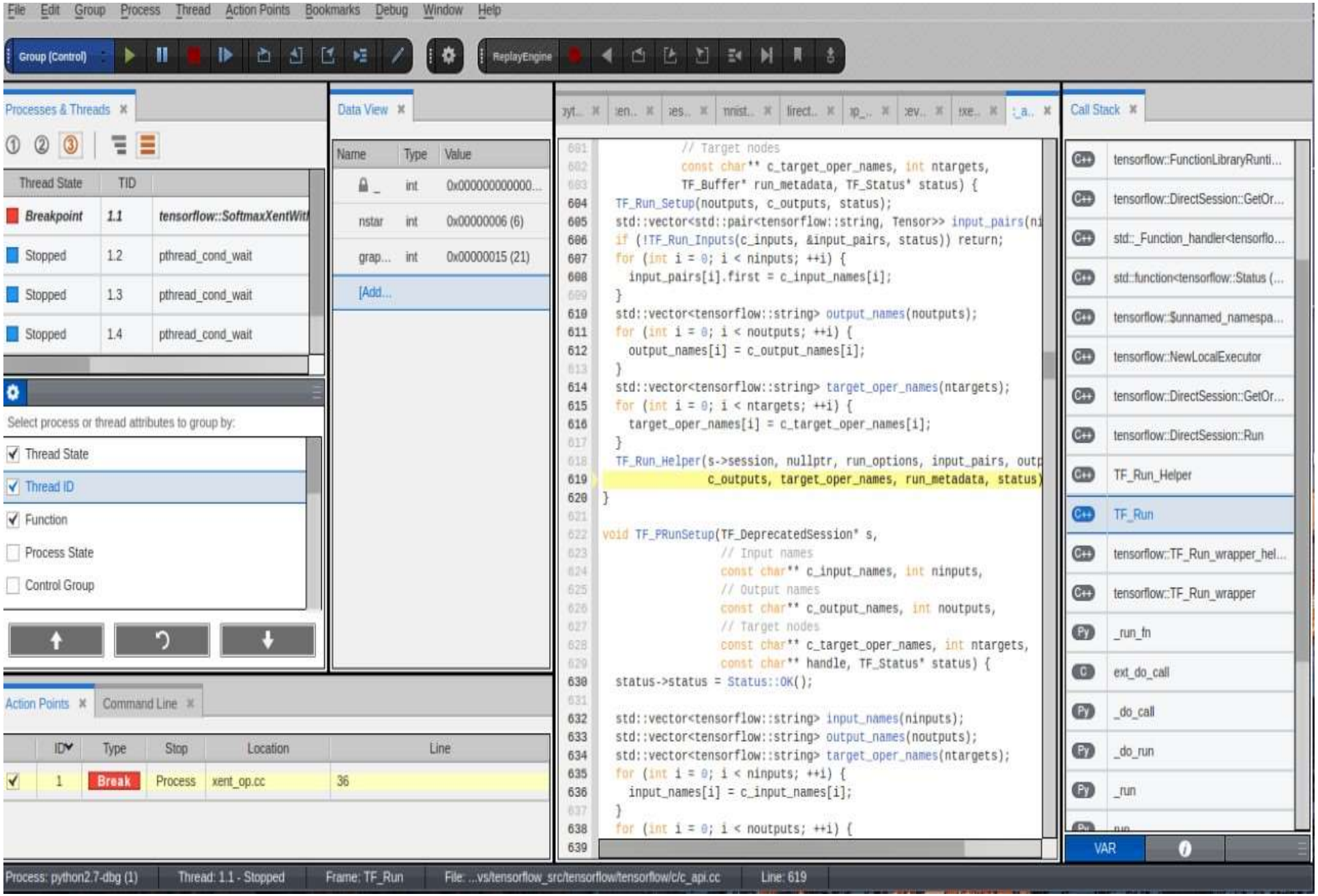
HPC Features | Reverse Debugging | Memory Debugging | Remote Debugging | Languages | Operating Systems | Processors | APIs

Debug C, C++, and Fortran applications running in your HPC environments faster with TotalView, by taking advantage of its abilities to:

- Support more than 20 different MPI implementations.
- Debug multithreaded applications with pthreads, OpenMP, and TBB.
- Easily establish interactive debugging sessions within your cluster — including through batch submission systems.
- Provide visualizations of program data and variables.
- Define how aggregate data in displayed, so you can see custom data structures in simpler forms.
- Advanced CUDA and OpenACC GPU debugging.
- Perform advanced memory debugging, including memory leaks and errors.



Total View


File Edit Group Process Thread Action Points Bookmarks Debug Window Help
Group (Control)
ReplayEngine
Processes & Threads
Thread State TID
Breakpoint 1.1 tensorflow::SoftmaxXentWit
Stopped 1.2 pthread_cond_wait
Stopped 1.3 pthread_cond_wait
Stopped 1.4 pthread_cond_wait
Select process or thread attributes to group by:
Thread State
Thread ID
Function
Process State
Control Group
Data View
Name Type Value
int 0x000000000000...
nstar int 0x00000006 (6)
grap... int 0x00000015 (21)
[Add...
Action Points
Command Line
ID Type Stop Location Line
1 Break Process xent_op.cc 36
// Target nodes
const char** c_target_oper_names, int ntargets,
TF_Buffer* run_metadata, TF_Status* status) {
TF_Run_Setup(noutputs, c_outputs, status);
std::vector<std::pair<tensorflow::string, Tensor>> input_pairs(ni
if (!TF_Run_Inputs(c_inputs, &input_pairs, status)) return;
for (int i = 0; i < ninputs; ++i) {
input_pairs[i].first = c_input_names[i];
}
std::vector<tensorflow::string> output_names(noutputs);
for (int i = 0; i < noutputs; ++i) {
output_names[i] = c_output_names[i];
}
std::vector<tensorflow::string> target_oper_names(ntargets);
for (int i = 0; i < ntargets; ++i) {
target_oper_names[i] = c_target_oper_names[i];
}
TF_Run_Helper(s->session, nullptr, run_options, input_pairs, outp
c_outputs, target_oper_names, run_metadata, status)
}
void TF_PRunSetup(TF_DeprecatedSession* s,
// Input names
const char** c_input_names, int ninputs,
// Output names
const char** c_output_names, int noutputs,
// Target nodes
const char** c_target_oper_names, int ntargets,
const char** handle, TF_Status* status) {
status->status = Status::OK();
std::vector<tensorflow::string> input_names(ninputs);
std::vector<tensorflow::string> output_names(noutputs);
std::vector<tensorflow::string> target_oper_names(ntargets);
for (int i = 0; i < ninputs; ++i) {
input_names[i] = c_input_names[i];
}
for (int i = 0; i < noutputs; ++i) {
Call Stack
tensorflow::FunctionLibraryRunti...
tensorflow::DirectSession::GetOr...
std::_Function_handler<tensorflo...
std::function<tensorflow::Status (...
tensorflow::$unnamed_namespa...
tensorflow::NewLocalExecutor
tensorflow::DirectSession::GetOr...
tensorflow::DirectSession::Run
TF_Run_Helper
TF_Run
tensorflow::TF_Run_wrapper_hel...
tensorflow::TF_Run_wrapper
_run_fn
ext_do_call
_do_call
_do_run
_run
VAR
Process: python2.7-dbg (1)
Thread: 1.1 - Stopped
Frame: TF_Run
File: ...vs/tensorflow_src/tensorflow/tensorflow/c/c_api.cc
Line: 619

Summary for performance debug

- Debugging and Parallel Code Optimization can be hard
- Tools can help
  - See NERSC web pages for recommendations
  - Use the ones that work for you
- Be aware of some of the more common errors and best practices
- Look for outliers in parallel programs

Topics Overview

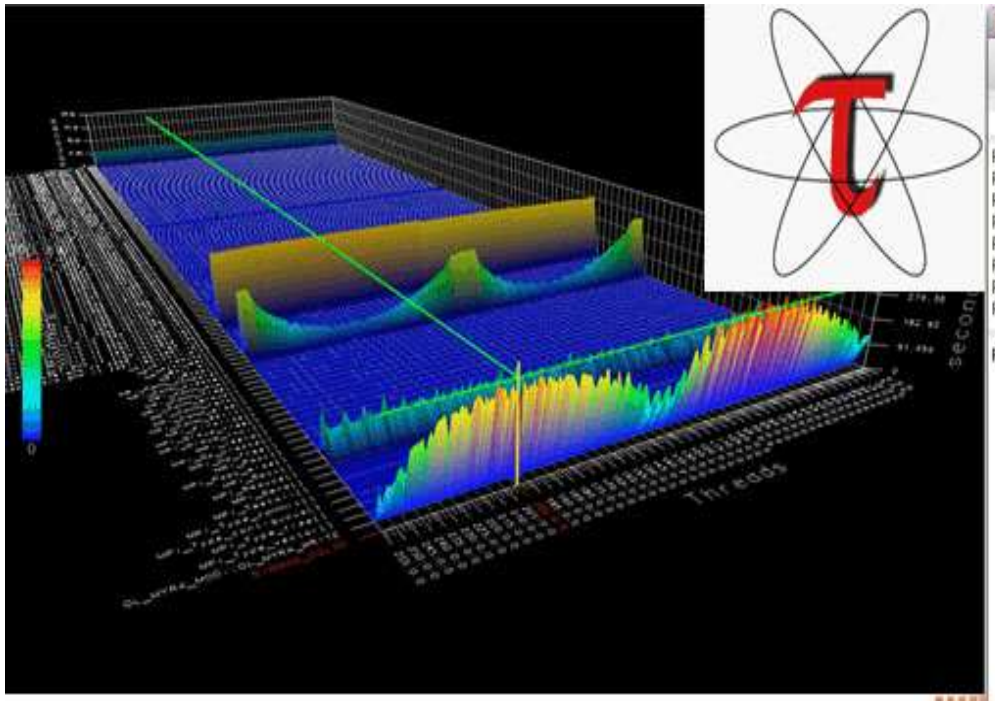
- Performance Benchmarks
- Performance Debugging
- Performance Analysis Tools
- Performance Optimization

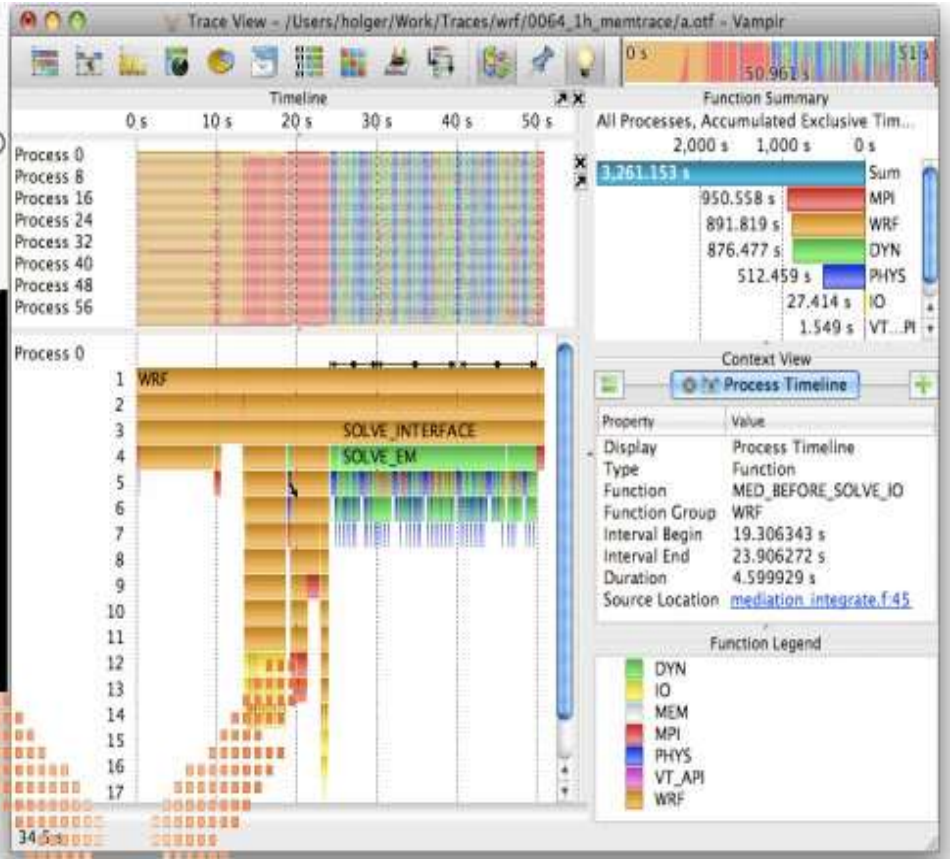
Performance Analysis and Tuning

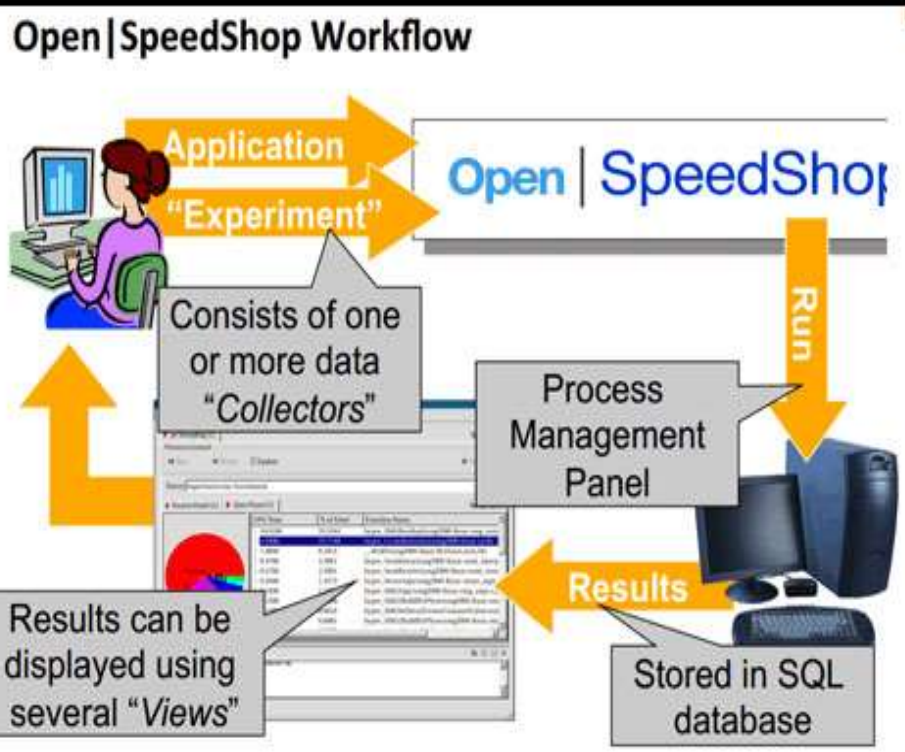
- As with debugging, analyzing and tuning parallel program performance can be much more challenging than for serial programs.
- Fortunately, there are a number of excellent tools for parallel program performance analysis and tuning.
  - Intel Vtune: https://software.intel.com/en-us/intel-vtune-amplifier-xe/
  - TAU(Tuning and Analysis Utilities) : http://www.cs.uoregon.edu/research/tau/docs.php
  - Gprof : http://www.sourceware.org/binutils/docs/gprof/
  - PAPI: http://icl.cs.utk.edu/papi/
  - mpiP: Lightweight, Scalable MPI Profiling http://mpip.sourceforge.net/
  - Sclascar: (SCalable performance Analysis of LArge SCale Applications) https://www.lrz.de/services/software/parallel/scalasca/
  - Vampir / Vampirtrace: http://vampir.eu/

- mpitrace https://computing.llnl.gov/tutorials/bgq/index.html#mpitrace
- memP: http://memp.sourceforge.net/
- Valgrind: http://valgrind.org/
- HPCToolkit: http://hpctoolkit.org/documentation.html
- IPM: Integrated Performance Monitoring
  http://projekt17.pub.lab.nm.ifi.lmu.de/ipm/
- Perf:

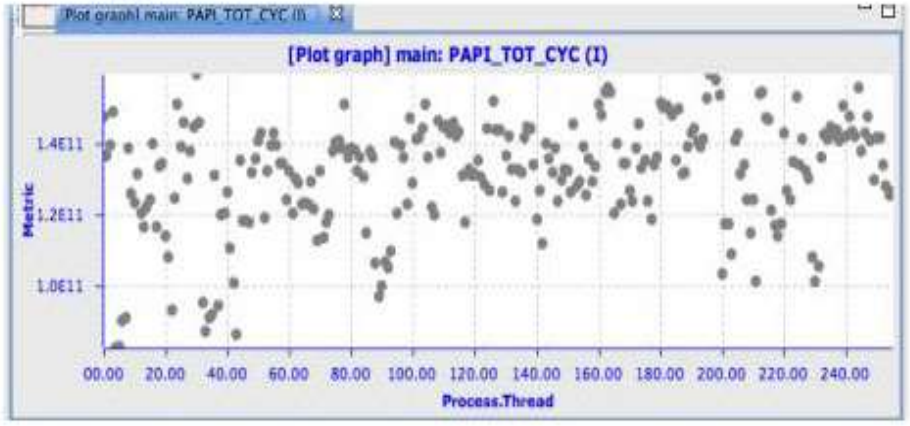
  A Performance Monitoring and Analysis Tool for Linux




Trace View - /Users/holger/Work/Traces/wrf/0064_1h_memtrace/a.otf - Vampir
Timeline
Function Summary
All Processes, Accumulated Exclusive Tim...
Process 0
Process 8
Process 16
Process 24
Process 32
Process 40
Process 48
Process 56
Sum
MPI
WRF
DYN
PHYS
IO
Context View
Process Timeline
Property
Value
Display
Process Timeline
Type
Function
Function
MED_BEFORE_SOLVE_IO
Function Group
WRF
Interval Begin
19.306343 s
Interval End
23.906272 s
Duration
4.599929 s
Source Location
mediation_integrate.f:45
Function Legend
DYN
IO
MEM
MPI
PHYS
VT_API
WRF
SOLVE_INTERFACE
SOLVE_EM


Open|SpeedShop Workflow
Application
"Experiment"
Open | SpeedShop
Run
Consists of one or more data "Collectors"
Process Management Panel
Results
Results can be displayed using several "Views"
Stored in SQL database

VAMPIR


[Plot graph] main: PAPI_TOT_CYC (I)
Metric
1.4E11
1.2E11
1.0E11
00.00
20.00
40.00
60.00
80.00
100.00
120.00
140.00
160.00
180.00
200.00
220.00
240.00
Process.Thread

What can HPC tools tell us?

- **CPU and memory usage**
  - FLOP rate
  - Memory high water mark
- **OpenMP**
  - OMP overhead
  - OMP scalability (finding right # threads)
- **MPI**
  - % wall time in communication
  - Detecting load imbalance
  - Analyzing message sizes

Things HPC tools may ask you to do

- (Sometimes) Modify your code with macros, API calls, timers

- Re-compile your code
- Transform your binary for profiling/tracing with a tool
- Run the transformed binary
  - A data file is produced
- Interpret the results with another tool

HPC Perf Tool Mechanisms (How)

- **Sampling**
  - **Regularly interrupt the program and record w**
  - **Build up a statistical profile**
- **Tracing / Instrumenting**
  - **Insert hooks into program to record time and**
- **Use Hardware Event/Performance Counte**
  - **Special registers count events on processor**

    E.g. floating point instructions
  - **Many possible events**
  - **Only a few (~4 counters)**

EVENT NAME
cpu-cycles
instructions
cache-references
cache-misses
branch-instructions
branch-misses
bus-cycles
L1-dcache-loads
L1-dcache-load-misses
L1-dcache-stores
L1-dcache-store-misses
L1-icache-loads
L1-icode-load-misses
LLC-loads
LLC-load-misses
LLC-stores
LLC-store-misses
dTLB-load-misses
dTLB-store-misses
iTLB-load-misses
branch-loads
branch-load-misses

PAPI (Performance Application Programming Interface)

- http://icl.utk.edu/papi/
- https://icl.utk.edu/exa-papi/

- Library that provides a **consistent interface** (and methodology) for hardware performance counters, found across the system:
  i. e., CPUs, GPUs, on-/off-chip Memory, Interconnects, I/O system, File System, Energy/Power, etc.
- PAPI enables software engineers to see, in near real time, the relation between **SW performance** and **HW events across the entire compute system**

SUPPORTED ARCHITECTURES:

- AMD, **GPU Vega**
- ARM Cortex A8, A9, A15, ARM64
- CRAY: Gemini and Aries interconnects, power/energy
- IBM Blue Gene Series, Q: 5D-Torus, I/O system, EMON power/energy
- IBM Power Series, **PCP for P9-uncore**
- Intel Sandy|Ivy Bridge, Haswell, **Broadwell, Skylake()-x, Kaby Lake**, KNC, KNL, Knights Mill
- Intel KNC, KNL, Knights Mill power/energy
- Intel RAPL (power/energy), **power capping**
- InfiniBand
- Lustre FS
- NVIDIA Tesla, Kepler, Maxwell, Pascal, **Volta**: support for multiple GPUs
- NVIDIA NVML (power/energy); **power capping**
- Virtual Environments: VMware, KVM

- **Papi and installation**

```
aix.c                    papi_memory.h
aix-context.h            papi_memory.o
aix.h                    papi.o
aix-lock.h               papi.pc
aix-memory.c             papi.pc.in
components               papi-perf_event
components_config.h      papi_preset.c
config.h                 papi_preset.h
config.h.in              papi_preset.o
config.log               papiStdEventDefs.h
config.status            papi-.tar
configure                papi_vector.c
configure.in             papi_vector.h
cpus.c                   papi_vector.o
cpus.h                   papivi.h
cpus.o                   pe_libpfm4_events.o
CreatePresetTbl.sh       perfctr-2.6.x
```

```
% ./configure

% make

% make test
```

```
% make install
```

- **Usage**

```
#include "papi.h"
#include<stdlib.h>

#include<stdio.h>
```

```
printf("TOT_INS:%lld\nL1_DCM: %lld\n",values[0], values[1]);
```

```
papi_avail          papi_decode          papi_native_avail
papi_clockres       papi_event_chooser   papi_version
papi_command_line   papi_mem_info        papi_xml_event_info
papi_cost           papi_multiplex_cost
```

```
[houzx@c01b03 ~]$ papi_avail
PAPI Error: pfm_find_full_event(RETIRED_MISPREDICTED_BRANCH_INSTRUCTIONS,0x7fffce57f9e0): event not found.
PAPI Error: 1 of 4 events in papi_events.csv were not valid.
Available events and hardware information.
--------------------------------------------------------------------------------
PAPI Version             : 4.1.3.0
Vendor string and code   : GenuineIntel (1)
Model string and code    : Intel(R) Xeon(R) CPU E5-2670 0 @ 2.60GHz (45)
CPU Revision             : 7.000000
CPUID Info               : Family: 6  Model: 45  Stepping: 7
CPU Megahertz            : 1200.000000
CPU Clock Megahertz      : 1200
Hdw Threads per core     : 1
Cores per Socket         : 8
NUMA Nodes               : 2
CPU's per Node           : 8
Total CPU's              : 16
Number Hardware Counters : 11
Max Multiplex Counters   : 512
--------------------------------------------------------------------------------
The following correspond to fields in the PAPI_event_info_t structure.

    Name        Code    Avail Deriv Description (Note)
PAPI_L1_DCM  0x80000000  No    No   Level 1 data cache misses
PAPI_L1_ICM  0x80000001  No    No   Level 1 instruction cache misses
PAPI_L2_DCM  0x80000002  No    No   Level 2 data cache misses
PAPI_L2_ICM  0x80000003  No    No   Level 2 instruction cache misses
PAPI_L3_DCM  0x80000004  No    No   Level 3 data cache misses
PAPI_L3_ICM  0x80000005  No    No   Level 3 instruction cache misses
PAPI_L1_TCM  0x80000006  No    No   Level 1 cache misses
PAPI_L2_TCM  0x80000007  No    No   Level 2 cache misses
PAPI_L3_TCM  0x80000008  No    No   Level 3 cache misses
PAPI_CA_SNP  0x80000009  No    No   Requests for a snoop
PAPI_CA_SHR  0x8000000a  No    No   Requests for exclusive access to shared cache line
PAPI_CA_CLN  0x8000000b  No    No   Requests for exclusive access to clean cache line
PAPI_CA_INV  0x8000000c  No    No   Requests for cache line invalidation
PAPI_CA_ITV  0x8000000d  No    No   Requests for cache line intervention
PAPI_L3_LDM  0x8000000e  No    No   Level 3 load misses
PAPI_L3_STM  0x8000000f  No    No   Level 3 store misses
```

3rd Party Tools applying PAPI

- PaRSEC (UTK) http://icl.cs.utk.edu/parsec/
- Caliper (LLNL) github.com/LLNL/caliper-compiler
- Kokkos (SNL) https://github.com/kokkos
- HPCToolkit (Rice University) http://hpctoolkit.org/
- Score-P http://score-p.org/
- TAU (U Oregon) http://www.cs.uoregon.edu/research/tau/
- Scalasca (FZ Juelich, TU Darmstadt) http://scalasca.org/
- VampirTrace and Vampir (TU Dresden) http://www.vamir.eu
- CrayPAT (Cray) https://pubs.cray.com/content/S-2474/6.5.2/cray-performance-measurement-and-analysis-tools-installation-guide/use-the-cray-performance-measurement-and-analysis-tools
- Open|Speedshop https://openspeedshop.org/
- ompP (LMU Munich) http://www.ompp-tool.com/

TAU
How to install TAU

- papi-To configure TAU with PAPI, we strongly recommend using the -MULTIPLECOUNTERS option.

- % configure -papi=<dir> -MULTIPLECOUNTERS -c++=icpc -cc=icc -fortran=intel -mpiinc=<dir> -mpilib=<dir>

- % make clean install

A real case of installing TAU

- ./configure -prefix=/export/opensource/tau-2_intel_mpi -papi=/export/opensource/papi-5.2-inst/ -MULTIPLECOUNTERS -c++=icpc -cc=icc -fortran=intel -pdt=/export/opensource/PDT_inst -mpiinc=/export/compiler/intel/impi/4.1.0.024/intel64/include -mpilib=/export/compiler/intel/impi/4.1.0.024/intel64/lib

- make clean install

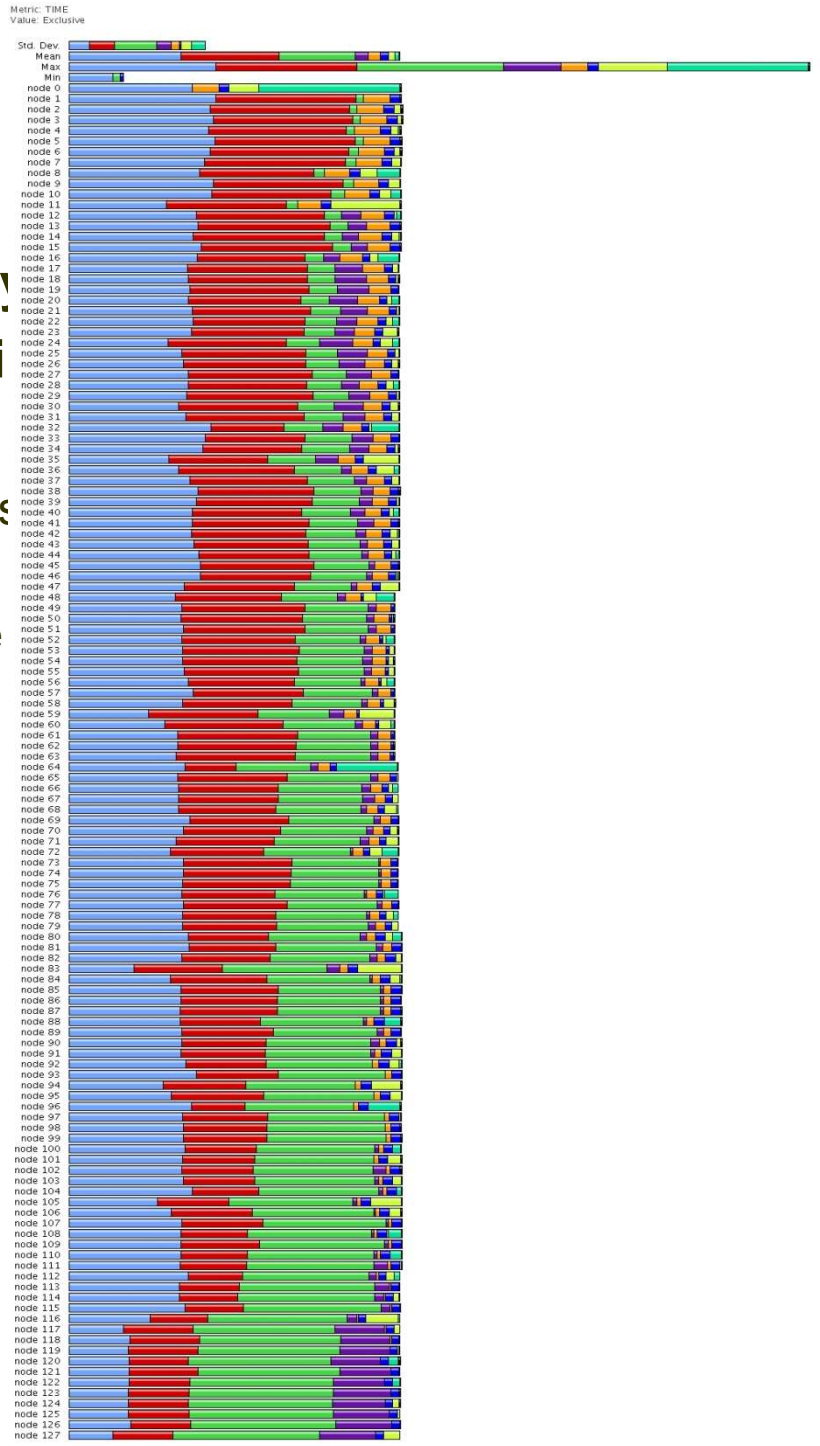
- export path=$path:/usr/local/packages/tau/x86_64/bin

Usage of TAU

- **To instrument source code automaticall**
- **Choose an appropriate TAU stub makefi**

- % export TAU_MAKEFILE=/home/sshende/pkgs
  pdt
  % export TAU_OPTIONS='-optVerbose …' (see
  % export PATH=<taudir>/<arch>/bin:$PATH
  % tau_f90.sh matrix.f90 -o matrix

- **Uninstrumented code:**
  - **% mpirun –np 8 ./a.out**
- **With TAU:**
  - **% mpirun –np 8 tau_exec ./matrix**
- **% pprof (for text based profile display)**
- **% paraprof (for GUI)**



profiling

- %export TAU_MAKEFILE=/export/home/houzx/tau/tau_mpich2_gcc/x86_64/lib/Makefile.tau-papi-mpi-pdt
- %export TAU_OPTIONS='-optTauSelectFile=select.tau -optVerbose'
- %export TAU_METRICS=TIME:PAPI_FP_INS:PAPI_FP_OPS:PAPI_L1_DCM

- % tau_cxx.sh jacobi_mpi.cpp -o jacobi_mpi_tau

- % mpirun –np 8 tau_exec ./ jacobi_mpi_tau

- % paraprof (for GUI)

Binary codes

- **Uninstrumented execution**
  - % mpirun –np 8 ./a.out
- **Track MPI performance**
  - % mpirun –np 8 tau_exec ./a.out
- **Track POSIX I/O and MPI performance (MPI enabled by default)**
  - % mpirun –np 8 tau_exec –io ./a.out
- **Track memory operations**
  - % setenv TAU_TRACK_MEMORY_LEAKS 1
  - % mpirun –np 8 tau_exec –memory ./a.out

- **Use event based sampling (compile with –g)**
  - % mpirun –np 8 tau_exec –ebs ./a.out
  - Also –ebs_source=<PAPI_COUNTER> -ebs_period=<overflow_count>
- **Load wrapper interposition library**
  - % mpirun –np 8 tau_exec –loadlib=<path/libwrapper.so> ./a.out
- **Track GPGPU operations**
  - % mpirun –np 8 tau_exec –cuda ./a.out
  - % mpirun –np 8 tau_exec –opencl ./a.out

Intel Vtune
How to use Intel Vtune

- amplxe-cl -result-dir test -collect hotspots

  mpirun -np 4 ./UGKS_test

- amplxe-cl -report hotspots -r my_result_openmp_50_new.1

- amplxe-cl -collect hpc-performance

Topics Overview

- Performance Benchmarks
- Performance Debugging
- Performance Analysis Tools
- Performance Optimization

Performance Questions

- (1) How can we tell if a program is performing well? Or isn't? What is "good"?
- (2) If performance is not "good," can we identify the causes?
- (3) What can we do about it?

Is Your Code Performing Well?

- **No single answer, but**
  - Does it scale well?
  - Is MPI time <20% of total run time?
  - Is I/O time <10% of total run time?
  - Is it load balanced?
  - If GPU code, does GPU+Processor perform better than 2 Processors?
- **"Theoretical" CPU performance vs. "Real World" performance in a highly parallel environment**
  - Cache-based x86 processors: >10% of theoretical is pretty good
  - GPUs: >1% in today's real full HPC applications pretty good?

This your challenge!

Can We Identify the Causes? Use Tools

- **Vendor Tools:**
  - **CrayPat on Crays**
  - **INTEL VTune**
- **Community Tools :**
  - **TAU (U. Oregon via ACTS)**
  - **PAPI (Performance API)**
  - **gprof**
- **NERSC "automatic" and/or easy-to-use tools**
  - **e.g. IPM, Darshan**

See NERSC web site
https://www.nersc.gov/users/software/debugging-and-profiling/

What can we do about it

- Minimize latency effects (aggregate messages)
- Maximize work vs. communication
- Minimize data movement (recalculate vs. send)
- Use the “most local” memory
- Use large-block I/O
- Use a balanced strategy for I/O
  - Avoid “too many” tasks accessing a single file, but “too many” files performs poorly ~1000s
  - Use “enough” I/O tasks to maximum I/O bandwidth, but “too many” causes contention 1/node

Overview of performance optimization/tuning

- **Courtesy to** Dr. Judy Gardiner, Ohio Supercomputer Center

Getting good performance

- Use parallelism effectively
- Keep data close to the processor using it
  - Cache utilization
  - Minimal data movement
  - Appropriate file system
- **Reduce communication overheads**

What would I do with your code?

- Profile it
- Experiment with compiler optimization flags
- Analyze data layout, memory access patterns
- Examine algorithms
  - Complexity
  - Availability of optimized version
- Look for potential parallelism, inhibitors to parallelism

A word about algorithms

- Problem-dependent – can't generalize
- Replace with an equivalent algorithm of lower complexity
  - computational geometry: change from vertex representation to half-plane representation
    - $O(2^n) \rightarrow O(n)$
  - replace 2-dimensional convolutions with 2-dimensional FFTs
    - $O(n^4) \rightarrow O(n^2 \log(n))$
- Replace home-grown algorithm with call to optimized library

High performance algorithms

- Considerations for algorithm selection
  - Speed
  - Scalability
  - Numerical stability
  - Availability of high quality library implementations
- Best sequential algorithm isn't always the best parallel algorithm
  - But it's a good place to start

A word about languages

- HPC software traditionally written in Fortran or C
- Workshop examples mostly Fortran
  - C/C++ programmers will have no trouble following the concept being demonstrated
- C++ code has many issues on HPC systems
  - Arrays of objects (inefficient use of cache)
  - Exception handling (difficult to optimize/parallelize)
  - Information hiding
  - Standard template library (STL) not thread-safe
- Fortran 90 user-defined types have the same issues as C++ objects
  - Arrays of objects (inefficient use of cache)

Main steps/approaches for performance tuning

- (0) hotspot analysis with Vtune/Tau/gprof
- Compiling Optimization
  - (1) Compiling options
  - (2) Mathematical Library (Intel MKL)
- (3) (*Parallel) Algorithm Optimization*
- Code optimization
  - (4) common optimization: to reduce redundant computation
    Loop optimization
  - (5) Memory access/cache optimization: data alignment
  - (6) Vectorization
  - (7) Parallel computing (concurrency)
    one node: Multi-threads with OpenMP
    multi-nodes: MPI, hybrid MPI and OpenMP
  - (8) communication optimization
  - (9) load balance
  - (10) (parallel) IO/data storage and fetch
- (11) Runtime parameters optimization

Single Processor Performance Tuning Techniques

- **Courtesy to** Dr. Judy Gardiner, Ohio Supercomputer Center

Single Processor Performance Tuning

The best way to improve parallel performance often is to improve sequential performance!

- High performance algorithms
- Compiler options
- Code modifications
- Optimized mathematical libraries

Single Processor Performance Measurement and Analysis Methods

- Part of assessing the performance of an application is understanding how well the compiler is doing at optimizing your code and what regions of your code are most in need of optimization.
- There are four generally available techniques for measuring and analyzing code performance:
  - Timing
  - Compiler reports and listings
  - Profiling
  - Hardware performance counters

Profiling

- Method for determining in which routines (or lines) code spends the most time
  - Allows you to identify areas to focus on for optimization
- Two types of profiling
  - Time-based statistical sampling
  - Compiler-generated instrumentation of the code (entry/exit of a code block)
- Can done at varying levels of granularity
  - Subroutine
  - Basic block
  - Source code line

Profiling (cont.)

- Requires special compilation
  - Executable will generate a file containing execution profile data as it runs.
  - This data file can be analyzed after the code is run using a profiling analysis program.
- For profiling serial code
  - gprof
  - pgprof
- For profiling MPI code
  - jumpshot

Profiling Compiler Options

GNU compilers

`-pg`	Enable function-level profiling using `gprof`

Portland Group compilers

`-Mprof=func`	Enable function-level profiling using `pgprof`
`-Mprof=lines`	Enable source code line-level profiling using `pgprof`; **overhead may be extremely high!**
`-Mprof=time`	Enable profiling using time-based statistical sampling; similar to gnu `-pg`
`-Mpfi, -Mpfo`	Generate / use profile feedback instrumentation

Intel compilers

`-p`	Enable function-level profiling using `gprof`
`-prof-gen, -prof-use`	Generate / use profile-guided optimization data

Optimization	Intel	GNU	PGI
-O	Same as O2	Same as O1	O1 + global optimizations. No SIMD.
-O0	No optimization	DEFAULT. No optimization. Same as omitting any -O flag.	No optimization
-O1	Optimize for size: basic optimizations to create smallest code	Reduce code size and execution time, without performing any optimizations that take a great deal of compilation time.	Local optimizations, block scheduling and register allocation.
-O2	DEFAULT. Optimize for speed: O1 + additional optimizations such as basic loop and vectorization	Optimize even more. O1 + nearly all supported optimizations that do not involve a space-speed tradeoff.	
-O3	O2 + aggressive loop optimizations. Recommended for loop dominated codes.	O2 + further optimizations	O2 + aggressive global optimizations
-O4	n/a	n/a	O3 + hoisting of guarded invariant floating point expressions
-Ofast	Same as O3 (mostly)	Same as O3 + optimizations that disregard strict standards compliance.	n/a
-fast	O3 + several additional optimizations	n/a	Generally specifies global optimization. Actual optimizations vary from release to release.
-Og	n/a	Enables optimizations that do not interfere with debugging.	n/a
Optimization / Vectorization report	-opt-report -vec-report	-ftree-vectorizer-verbose=*[1-7]* -ftree-vectorizer-verbose=7	-Minfo=*[option]* -Minfo=all

Profiling: gprof

- GNU profiler – available everywhere
- Compile and link your code
  - Use the `-pg` option flag with GNU or PGI compilers
  - Use the `-p` option flag with Intel compilers
- Run your code in the typical manner
- Creates `gmon.out`
  - Contains the profile data
- Analyze the profile data

```
gprof progname gmon.out
```

Profiling: gprof example

```
[opt0867] > make
g77 -O2 -pg  -c cdnz3d.f
g77 -O2 -pg  -c sdbdax.f
g77 -O2 -pg -o cdnz3d cdnz3d.o sdbdax.o

[opt0867] > ./cdnz3d
(…gmon.out created…)
```

Profiling: gprof example (cont.)

```
[opt0867] > gprof cdnz3d gmon.out | more
Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls   s/call   s/call  name
 24.67    942.76   942.76  4100500     0.00     0.00  lxi_
 23.51   1841.45   898.69  4100500     0.00     0.00  leta_
 20.10   2609.66   768.21  4100500     0.00     0.00  damping_
 12.64   3092.90   483.24  4100500     0.00     0.00  lzeta_
 11.55   3534.28   441.38  4100500     0.00     0.00  sum_
  4.12   3691.73   157.45      250     0.63    14.83  page_
  2.91   3802.84   111.11      250     0.44     0.44  tmstep_
  0.41   3818.62    15.78      500     0.03     0.03  bc_
  0.03   3819.59     0.97                             pow_dd
 (…output continues…)
```

Profiling – What do I look for?

- Hot spots – where most of the time is spent
  - This is where we'll focus our optimization effort
- Excessive number of calls to short functions
  - Use inlining! (compiler flags)
- Memory usage
  - Swapping, thrashing
- CPU time vs. wall time (% CPU)
  - Low CPU utilization may mean excessive I/O delays

Hardware Performance Counters

- Processors have one or more event counters
  - Count low level processor events such as
    - Floating point operations
    - Cache line misses
    - Total instructions executed
- Output from counters can be used to infer how well a program utilizes the processor

General Performance Tuning Techniques

- Inlining – eliminate function calls
- Loop Optimization
  - Vectorization
  - Unrolling
  - Splitting
- Memory Optimization
  - Cache alignment
  - Stride-One memory access
  - Avoiding cache thrashing
- Floating Point Behavior
  - Instruction set extensions
  - Division
  - Exception handling
- Optimized Mathematical Libraries

Inlining

- Replace a subroutine or function call with the actual body of the subprogram
  - Particularly effective for subprograms that are relatively small and are called a large number of times
- Advantages
  - Overhead of calling the subprogram is eliminated
  - More loop optimizations are possible if calls are eliminated
- Tried and true optimization technique
  - Applicable to all computers from workstations to supercomputers
  - Implemented with compiler options or directives
- Drawbacks:
  - “Code bloat" (memory needed to hold code grows)
  - Code for a loop may not fit in instruction cache (if call is in a loop)

Optimization Compiler Options – Intel compilers

- Many of these flags have parameters
- Many other optimization options are available
- See **man** pages for details
- Recommended options:
  `-O2`
- Example:
  `ifort -O2 program.f90`

`-fast`	Common optimizations
`-On`	Set optimization level (0, 1, 2, 3)
`-ip, -ipo`	Interprocedural optimization, inlining
`-O3`	Loop transforms
`-funroll-loops`	Loop unrolling
`-parallel`	Loop auto-parallelizaton

Inlining Compiler Options

GNU compilers

`-fno-inline`	Disable inlining
`-finline-functions`	Enable inlining of functions

Portland Group compilers

`-Minline=option[,option,…]`	Perform inlining using *option*; *option* may be `lib:`*filename.ext*, `name:`*function*, `size:`*N*, or `levels:`*P*
`-Mipa`	Enable InterProcedural Analysis

Intel compilers

`-ip`	Enable single-file interprocedural optimization, including enhanced inlining
`-ipo`	Enable interprocedural optimization across files

Loop Optimizations

- Vectorization / streaming
- Loop unrolling
- Loop splitting

Vectorization

- Vectorization is a process by which code is structured to operate on lists of operands.
  - Most vectorized loops end up looking like matrix/vector arithmetic or linear algebra, hence the name.
  - Some languages (e.g. Fortran 90 and after, MATLAB) have this idea built in.
- On vector processors like the old Crays, vectorization was absolutely critical to performance.
  - Most modern architectures have some SIMD/vector capability, though the maximum vector length is comparatively short.
- Code written to be vectorizable is good for multithreading.
- Streaming is conceptually similar to vectorization; both are SIMD.

Vectorization Compiler Options

GNU compilers

`-mfpmath=sse`	Use scalar floating point instructions in the SSE instruction set

Portland Group compilers

`-Mvect`	Enable vectorization
`-Mvect=sse`	Use SSE and prefetch instructions in loops where possible

Intel compilers

`-O3`	Aggressive optimization, including vectorization and other loop transformations

Vectorization Inhibitors

- Not unit stride
  - Loops in wrong order (column-major vs. row major)
    - Usually fixed by the compiler
  - Loops over derived types
- Function calls
  - Sometimes fixed by inlining
  - Can split loop into two loops

Loop Unrolling

- Loop unrolling: Optimization technique applied to loops which perform calculations on array elements
- Replicates the body of the loop so that calculations are performed on several array elements during each iteration
- Reason: Take advantage of multiple functional units. Consecutive elements of the arrays can be in the functional units simultaneously.
- Don't unroll loops "by hand". – Use compiler options and directives.
  - Code is more readable and portable to other systems
  - Code is self-documenting and easier to read
- Loops with small trip counts or data dependencies should not be unrolled!

Loop Unrolling Example

- Normal loop

```
do i=1,N
   a(i)=b(i)+x*c(i)
enddo
```

- Performance (compiled -02)
  - Opteron 2.6GHz: 425 MFLOPS
  - Xeon 2.4GHZ: 356 MFLOPS
  - Itanium 2 1.3GHz: 190 MFLOPS

- Manually unrolled loop

```
do i=1,N,4
   a(i)=b(i)+x*c(i)
   a(i+1)=b(i+1)+x*c(i+1)
   a(i+2)=b(i+2)+x*c(i+2)
   a(i+3)=b(i+3)+x*c(i+3)
enddo
```

- Performance (compiled -02)
  - Opteron 2.6GHz: 430 MFLOPS
  - Xeon 2.4GHz: 355 MFLOPS
  - Itanium 2 1.3GHz: 191 MFLOPS

Loop Unrolled by 4

```
for(i=0; i<SIZE;i++){
   x[i] = y[i];
}
```

```
int lastBlock = 4*SIZE/4;
for(i=0; i<lastBlock;i+=4){
   x[i] = y[i];
   x[i+1] = y[i+1];
   x[i+2] = y[i+2];
   x[i+3] = y[i+3];
}

for(i=lastBlock;i<SIZE;i++){
   X[i] = y[i];
}
```

Loop Unrolling Compiler Options

GNU compilers

`-funroll-loops`	Enable loop unrolling
`-funroll-all-loops`	Unroll all loops; not recommended

Portland Group compilers

`-Munroll`	Enable loop unrolling
`-Munroll=c:`*N*	Completely unroll loops with trip counts of at most *N*
`-Munroll=n:`*M*	Unroll loops up to *M* times

Intel compilers

`-funroll-loops`	Enable loop unrolling

Loop Optimizations

- Break one loop into multiple loops
  - Extra overhead connected with the new loops is typically offset by increased overall performance
  - Loop splitting
    - Split the index range into multiple ranges to eliminate data dependencies (`if` statements). The new loops can be unrolled or vectorized.
  - Loop fission
    - Split loop body into two parts with same index range to eliminate register pressure (use of too many FP registers in unrolled loop).
  - Loop tiling
    - Split the index range into smaller chunks that fit in L2 cache
- Combine loops
  - If two loops over the same range don't make full use of hardware

Loop splitting/peeling

- For (i=0; i< n; i++)
  {
  }

```
i=0;

i=1;


For (I=2; i< n; i++)
{
 }
```

Loop fission<-->loop fusion

- For (i=1; i<n; i++)

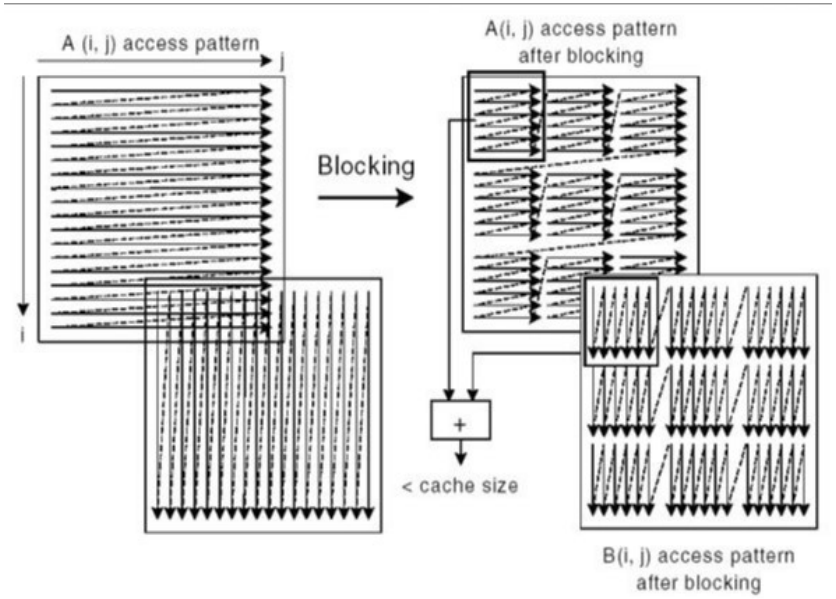
```
{
    do something A

    do something B

    do something C

}
```

- For (i=1; i<n; i++)

```
{
    do something A
}
For (i=1; i<n; i++)
{
    do something B
}
For (i=1; i<n; i++)
{
 do something C
}
```

Loop tilling/blocking

```
for(i=0; i<N; ++i){
  ...
}
```




```
for(j=0; j<N; j+=B)
  for(i=j; i<min(N, j+B); ++i){
    ....
  }
```

where min() is a function returning the minimum of its arguments.

Loop tilling/blocking-
matrix vector multiplication

```
int i, j, a[100][100], b[100], c[100];
int n = 100;
for (i = 0; i < n; i++) {
  c[i] = 0;
  for (j = 0; j < n; j++) {
    c[i] = c[i] + a[i][j] * b[j];
  }
}
```

After we apply loop tiling using 2 * 2 blocks, our code looks like:

```
int i, j, x, y, a[100][100], b[100], c[100];
int n = 100;
for (i = 0; i < n; i += 2) {
  c[i] = 0;
  c[i + 1] = 0;
  for (j = 0; j < n; j += 2) {
    for (x = i; x < min(i + 2, n); x++) {
      for (y = j; y < min(j + 2, n); y++) {
        c[x] = c[x] + a[x][y] * b[y];
      }
    }
  }
}
```

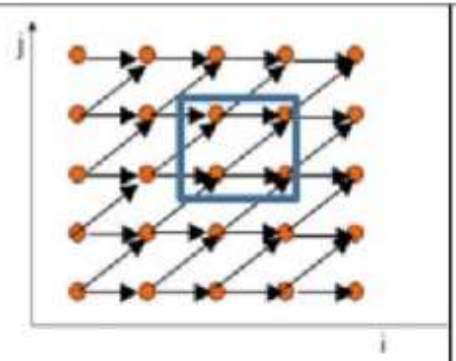
- The original loop iteration space is *n* by *n*. The accessed chunk of array a[i, j] is also *n* by *n*.
- When *n* is too large and the cache size of the machine is too small, the accessed array elements in one loop iteration may cross cache lines, causing cache misses

```
For i=0, N
  For j=0, N
    A(i,j)=A(i-1, j-1)+A(i-1,j)
```

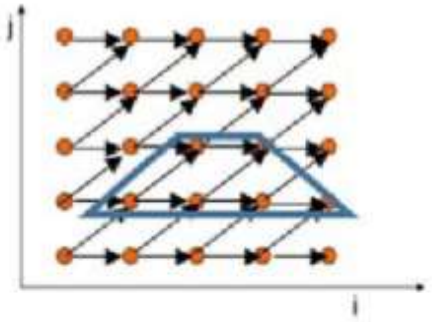
Initial code

```
For ii=0, N, 2
  For jj=0, N, 2
    For i=ii, min (ii+1,N)
      For j=jj, min(jj+1,N)
        A(i,j)=A(i-1, j-1)+A(i-1,j)
```

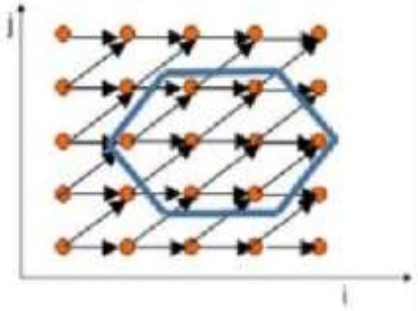
Tiled code



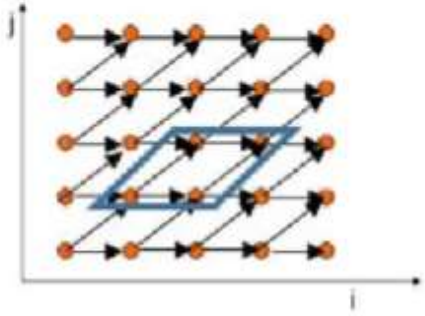
(a). Rectangular Tiling



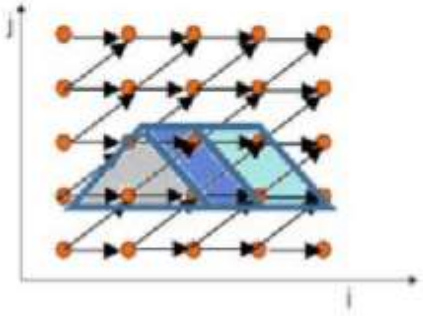
(b). Trapezoïdal Tiling



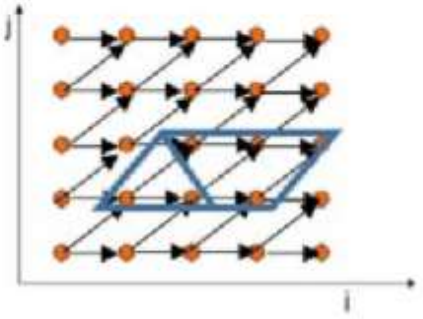
(c). Hexagonal Tiling



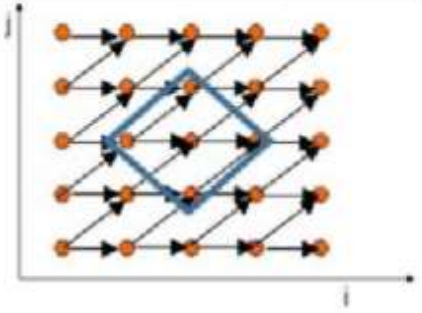
(d). Molecular Tiling



(e). Overlapped Tiling



(f). Split Tiling



(g). Diamond Tiling

Loop Optimizaton Compiler Options

GNU compilers	
	None(?)
Portland Group compilers	
`-Mvect`	Enable vectorization, including loop optimizations
Intel compilers	
`-O3`	Aggressive optimization, including loop transformations

Cache Line Alignment of Arrays

- One optimization technique offered on most systems is a way to align arrays or common blocks to begin on cache line boundaries. This allows the user to
  - first, better understand the exact layout of their arrays in the data cache; and
  - second, to make sure cache lines are "full" of actual array elements and not extraneous data.
- As with many optimizations, compilers can usually do automatic cache line alignment of arrays, although they sometimes need help in the form of directives.

Cache Alignment Compiler Options

GNU compilers

`-malign-double`	Align double precision variables on 64-bit boundaries (not really cache alignment)

Portland Group compilers

`-Mcache_align`	Align arrays on cache line boundaries

Intel compilers

`-align`	Add padding to align data items in common blocks, derived types, and record structures on their natural boundaries (not really cache alignment)

Stride-One Memory Access

- The most important factor in your code's performance!!!
- Loops that work with arrays should use a stride of one whenever possible
- C, C++ are ***row-major***; in a 2D array, they store elements consecutively by row:
  - First array index should be outermost loop
  - Last array index should be innermost loop
- Fortran is ***column-major***, so the reverse is true:
  - Last array index should be outermost loop
  - First array index should be innermost loop
- Avoid arrays of derived data types, structs, or classes

Loop Interchange Compiler Options

GNU compilers

None (?)

Portland Group compilers

`-Mvect` — Enable vectorization, including loop interchange

Intel compilers

`-O3` — Enable aggressive optimization, including loop transformations

Optimized Math Libraries

- By default the algorithms used to calculate intrinsic math functions ("calculator functions") are scalar in nature. On the other hand, loops which use some intrinsic function calls on array elements used can be "vectorized" so that the functions calls can be optimized. The vectorization of the loop occurs in several steps.
  - First the intrinsic calls are separated into their own loop where the results are assigned to a temporary array and
  - an optimized algorithm is used to make use of vectorization.
  - The resulting temporary array is then used in the original loop instead of the actual function calls.
- The functions which can be vectorized in this manner include `log`, `alog`, `exp`, `pow`, `**`, `sin`, `cos`, `sqrt`, and `1.0/sqrt`.
- Manufacturers of high performance computers also typically supply highly optimized versions of commonly used numerical libraries such as BLAS and LAPACK.

Optimized Math Libraries (cont.)

- ACML (AMD Core Math Library)
  - BLAS
  - LAPACK
  - FFT
  - Fast and/or vectorized transcendental functions (sin, cos, exp)
  - Random number generators
- MKL (Intel Math Kernel Library)
- FFTW
- ScaLAPACK
- SuperLU
- … and many others

Intel® oneAPI Math Kernel Library

- For C and Fortran on CPU
  - **Linear algebra**
  - **Fast Fourier Transforms (FFT)**
  - **Vector math**
  - **Direct and iterative sparse solvers**
  - **Random number generators**
- For DPC++ on CPU and GPU
  - **Linear algebra**
    - **BLAS**
    - **Selected Sparse BLAS functionality**
    - **Selected LAPACK functionality**
  - **Fast Fourier Transforms (FFT)**
    - **1D, 2D, and 3D**
  - **Random number generators**
    - **Selected functionality**
  - **Selected Vector Math functionality**

- Step 1: Install Intel® oneAPI Math Kernel Library

```
icc app.obj -L${MKLROOT}/lib/intel64 -lmkl_intel_lp64-lmkl_intel_thread -lmkl_core -liomp5 -lpthread -lm -ldl
```

- Step 2: Select a Function or Routine
- Step 3: Link Your Code

Multi-Core and Parallel Performance Tuning Techniques

- **Courtesy to** Dr. Judy Gardiner, Ohio Supercomputer Center

Parallel Performance Metrics

- As with single processor performance, the simplest parallel performance metric is always wallclock time.
  - Represents the time you wait for your results
  - CPU time is even less useful here than in the single processor case.
- Hardware performance counters can still be used to assess overall performance as well.
- However, the addition of parallelism in the application introduces the concept of scalability and two new metrics to consider:
  - Speedup
  - Parallel efficiency

Measuring the Performance of Parallel Applications

- For parallel applications with strong scaling characteristics, timing is absolutely critical to assessing parallel performance.
  - Timings needed to compute speedups
  - Speedups needed to compute parallel efficiency
- As with serial applications, profiling can be used to identify performance bottlenecks in parallel applications.
  - In parallel, performance bottlenecks are often ***explicitly*** or ***implicitly serialized*** parts of the code.
    - ***Explicit serialization***: something intended to be done serially.
    - ***Implicit serialization***: something intended to be done in parallel but implemented in a way that operates serially or scales very poorly.
- In addition, hardware performance counters can also be used to assess performance of both strong and weak scaling parallel applications, in largely the same way as serial applications.

Timing of Parallel Programs

- As with serial programs, the easiest and most basic way to assess a program's performance is by wallclock timing it externally with something like `/usr/bin/time`.
- However, the CPU time and other metrics reported by `time` are typically not that useful.
  - This is particularly the case for message passing applications, which usually do not have a direct parent-child relationship with the `time` command.
  - Moreover, message passing applications have non-zero startup times that may or may not be desirable to include in timings.
- As a result, it is usually a good idea to instrument your application with timing calls around important sections of the code.
  - C/C++: `time`(2), `difftime`(3), `getrusage`(2)
  - Fortran 77/90: `SYSTEM_CLOCK`(3)
  - MPI (C/C++/Fortran): `MPI_Wtime`(3)

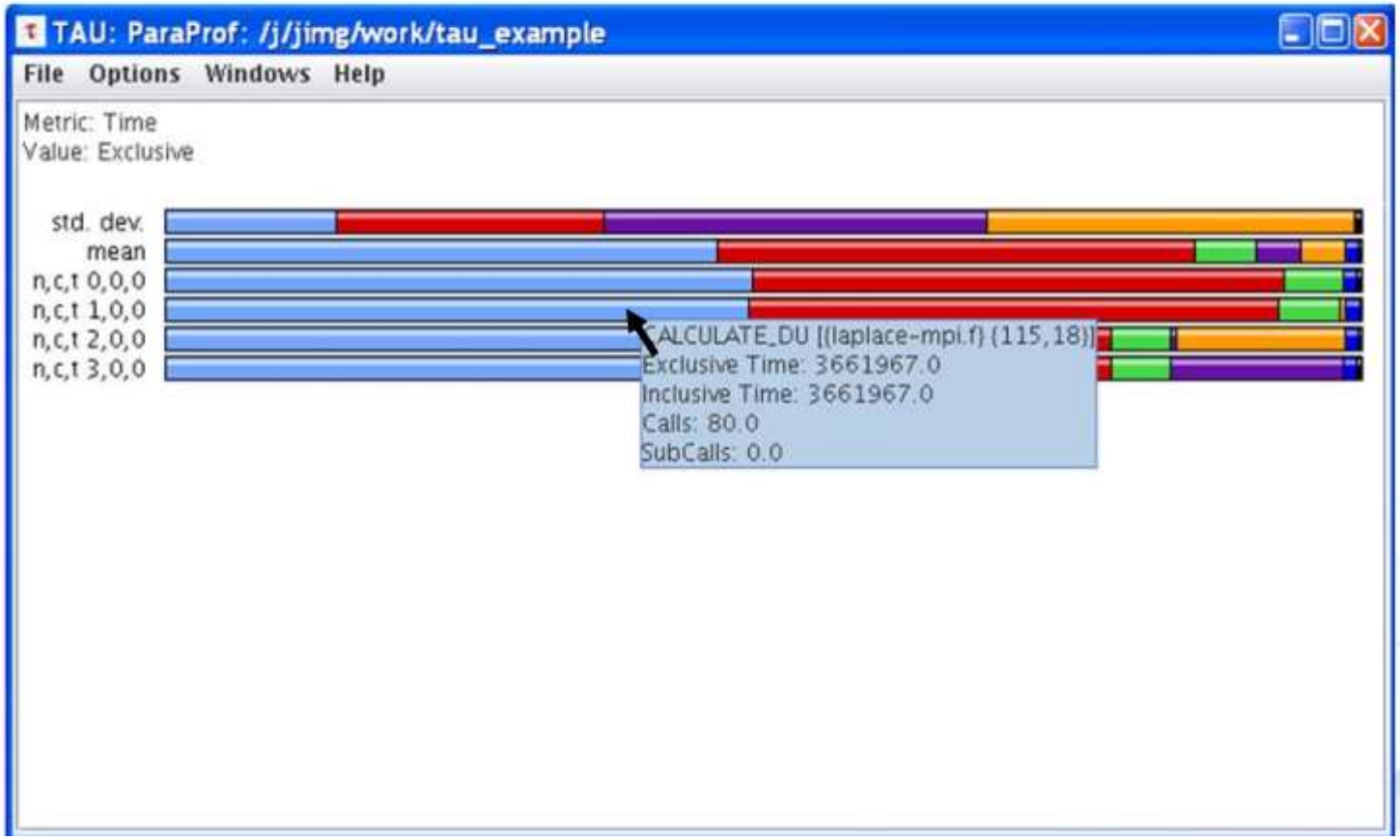
Profiling of Parallel Programs

- Profiling parallel program is often difficult.
  – Profiling tools included with compilers typically not designed for use with concurrent programs.
  – As a result, parallel profiling tools are often purposebuilt.

  – Tau
    - Supports both threaded and MPI applications
    - Flexible
    - Good at traditional profiling
  – Jumpshot
    - Supports only MPI applications in C and C++
    - Good for visualizing communication patterns

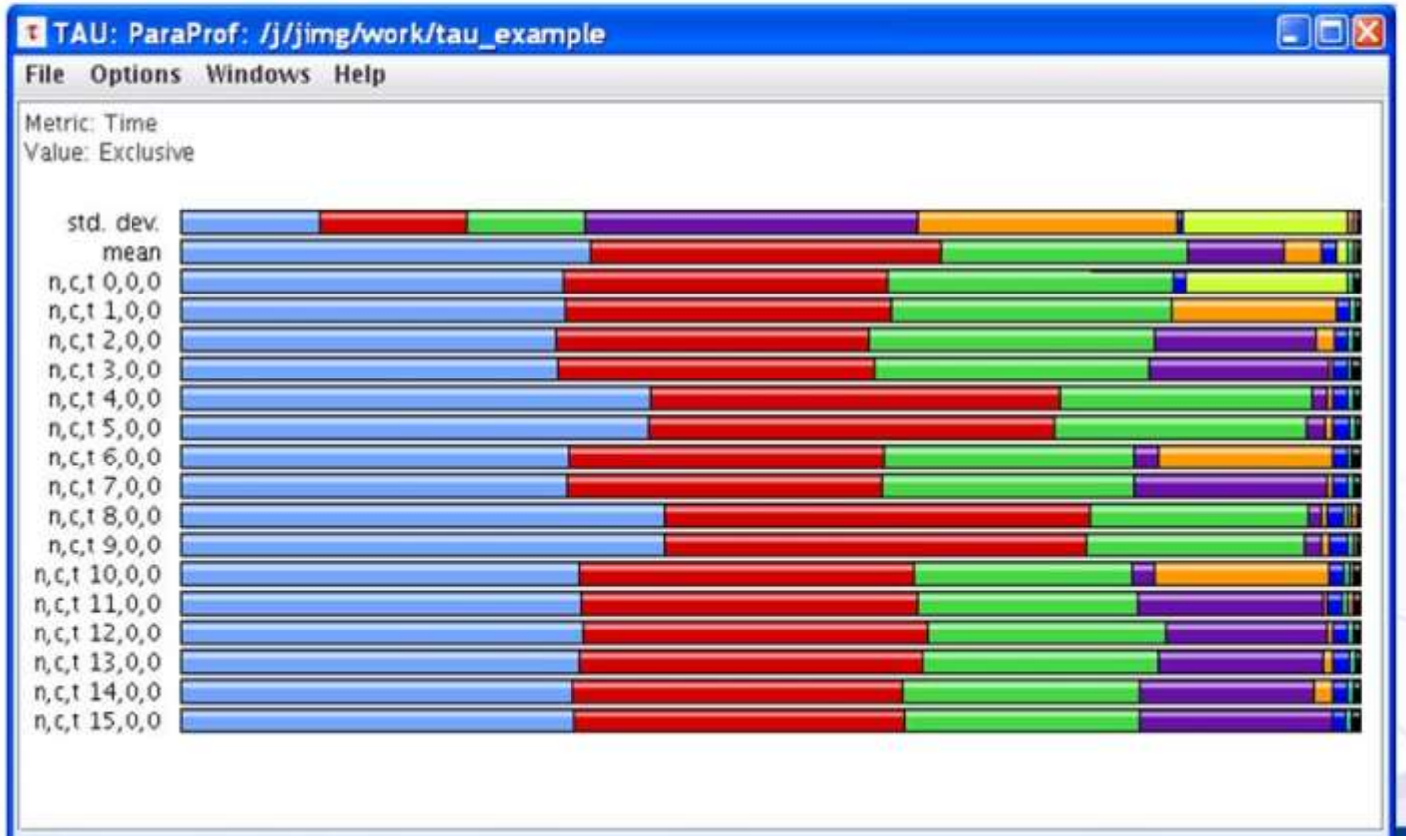
TAU visualizer paraprof

- To analyze the profile information generated during program execution, run the paraprof application from an interactive login session



TAU visualizer paraprof

- Here is an example of a parallel program that has MPI Communication problems
  - Purple MPI_WAIT and orange MPI_RECV have large standard deviations



Common Threaded Performance Bottlenecks

- Data dependency and recurrence relationships
- Unbalanced loops
- Excessive synchronization

- **OpenMP**

Common Message Passing Performance Bottlenecks

- Excessive synchronization
- Stairstepping
- Homegrown collectives
- Contention for interconnect device

Excessive Synchronization

- Barrier synchronization such as `MPI_Barrier()` has a non-trivial cost associated with it.
  - Some MPP style systems (e.g. Cray T3E, IBM BlueGene) have dedicated barrier networks for very low latency barriers.
  - However, most clusters use commodity networks where a barrier costs *at least* ~1μs on a single node, which then scales like $O(\log_2(N))$. On a system with a GHz scale clock speed, a barrier can therefore take thousands of clock cycles to clear, if not tens or even hundreds of thousands of clock cycles.
- As a result, the first thing to examine when trying to tune the performance of an MPI code is see if any of its barrier synchronizations can be removed.

Stairstepping

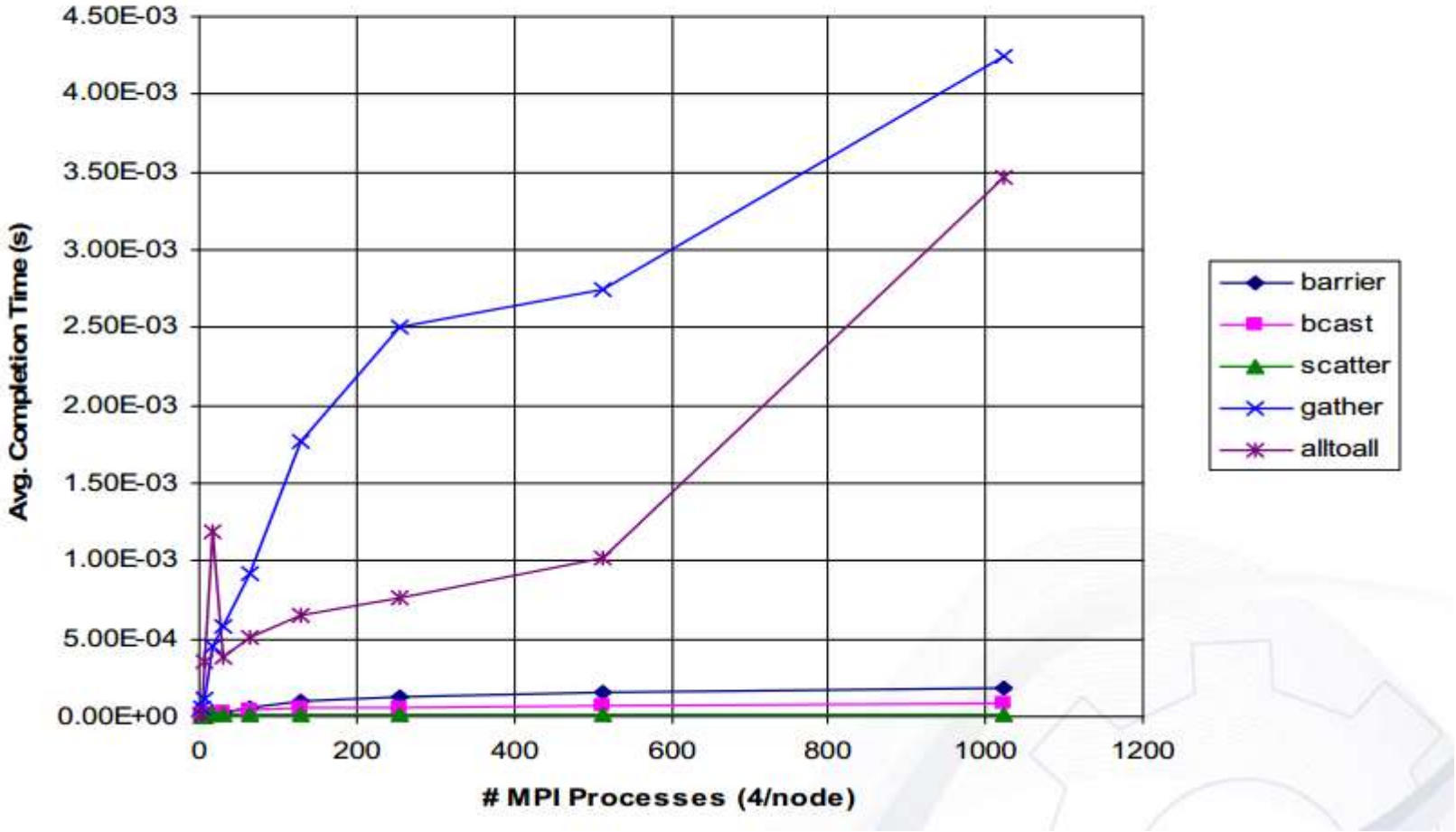
- Stairstepping is a very common performance issue in message passing applications, resulting from common misunderstanding of MPI semantics.
  - `MPI_Send()` is **not** guaranteed to return until the receiver invokes `MPI_Recv()` – it is closer to `MPI_Ssend()` than `MPI_Bsend()` or `MPI_Isend()`!
  - In codes with a conditional `MPI_Send()` followed by a conditional `MPI_Recv()` (such as in a ghost-cell exchange), the transfers can become serialized on the `MPI_Recv()`s.
    - Called "stairstepping" because of the resulting appearance of profile graphs
  - The solution to this is either to make the send non-blocking using `MPI_Isend()`, or to make the receive a pre-posted non-blocking `MPI_Irecv()`.


TAU: ParaProf: /c/troy/perftune/mpi
File Options Windows Help
Metric: Time
Value: Exclusive
std. dev.
mean
n,c,t 0,0,0
n,c,t 1,0,0
n,c,t 2,0,0
n,c,t 3,0,0
n,c,t 4,0,0
n,c,t 5,0,0
n,c,t 6,0,0
n,c,t 7,0,0
n,c,t 8,0,0
n,c,t 9,0,0
n,c,t 10,0,0
n,c,t 11,0,0
n,c,t 12,0,0
n,c,t 13,0,0
n,c,t 14,0,0
n,c,t 15,0,0
TAU: ParaProf: Functio
File Filter Windows Help
MPI_Allgather()
MPI_Allreduce()
MPI_Attr_put()
MPI_Barrier()
MPI_Bcast()
MPI_Comm_create()
MPI_Comm_group()
MPI_Comm_rank()
MPI_Comm_size()
MPI_Comm_split()
MPI_Errhandler_set()
MPI_Group_incl()
MPI_Init()
MPI_Recv()
MPI_Send()
MPI_Type_commit()
MPI_Type_contiguous()
MPI_Type_struct()
STAIRSTEP
TAU: ParaProf: /c/troy/perftune/mpi
File Options Windows Help
Metric: Time
Value: Exclusive
std. dev.
mean
n,c,t 0,0,0
n,c,t 1,0,0
n,c,t 2,0,0
n,c,t 3,0,0
n,c,t 4,0,0
n,c,t 5,0,0
n,c,t 6,0,0
n,c,t 7,0,0
n,c,t 8,0,0
n,c,t 9,0,0
n,c,t 10,0,0
n,c,t 11,0,0
n,c,t 12,0,0
n,c,t 13,0,0
n,c,t 14,0,0
n,c,t 15,0,0
TAU: ParaProf: Function Le
File Filter Windows Help
MPI_Allgather()
MPI_Allreduce()
MPI_Attr_put()
MPI_Barrier()
MPI_Bcast()
MPI_Comm_create()
MPI_Comm_group()
MPI_Comm_rank()
MPI_Comm_size()
MPI_Comm_split()
MPI_Errhandler_set()
MPI_Group_incl()
MPI_Init()
MPI_Isend()
MPI_Recv()
MPI_Type_commit()
MPI_Type_contiguous()
MPI_Type_struct()
MPI_Wait()
STAIRSTEP2

Homegrown Collectives

- Inexperienced MPI programmers sometimes implement their own versions of collective operations.
  - Usually naïve O(N) implementations
  - May actually outperform the implementation's collectives on small processor counts.
  - However, rarely outperform the implementation's collectives on larger processor counts.
- Any application which has homegrown collectives should be examined to see if those homegrown collectives are in fact necessary, particularly if collective performance is inhibiting scalability.

MPI Collective Performance on 8MB Payload



Contention for Interconnect Device

- As multi-core processors proliferate, it is increasingly common to have 4, 8, or even 16 MPI processes sharing the same network device.
- Contention for the interconnect device can have a significant impact on performance.
- In *some* cases, reducing the number of MPI processes to 1 or 2 per node and then having each MPI process spawn multiple threads can extend scalability at the high end.
  - Large amount of per-process computation
  - Communication pattern whose expense increases with process count
    - Heavy use of collectives (e.g., `MPI_Alltoall()`)
    - Large volume of small messages
  - **Not a scalability panacea!** Some codes see significant benefit from this, others see none at all.

- **Hybrid MPI and OpenMP**

Thank you!