IPM - A Tutorial

Nicholas J Wright
Karl Fuerlinger

nwright @ sdsc.edu
fuerling@eecs.berkeley.edu

Allan Snavely, SDSC
David Skinner LBNL
Katherine Yelick LBNL & UCB

SDSC **SAN DIEGO SUPERCOMPUTER CENTER**

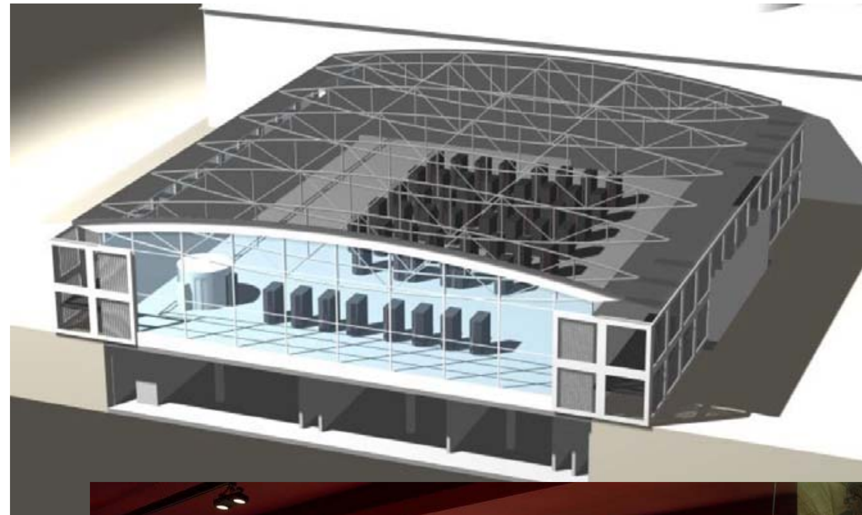
PMaC
Performance Modeling and Characterization

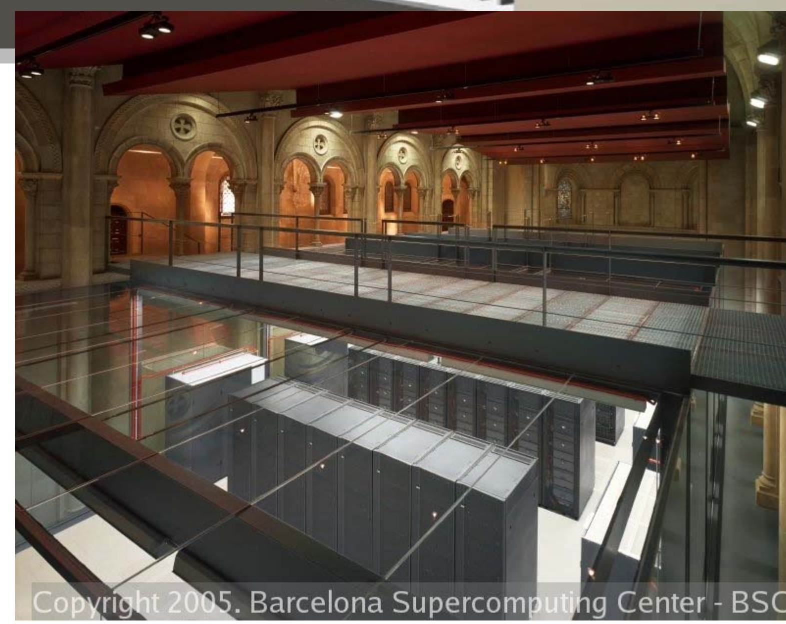
Menu

- **Performance Analysis Concepts and Definitions**
  - Why and when to look at performance
  - Types of performance measurement
- **Examining typical performance issues today using IPM**
- **Summary**

Motivation

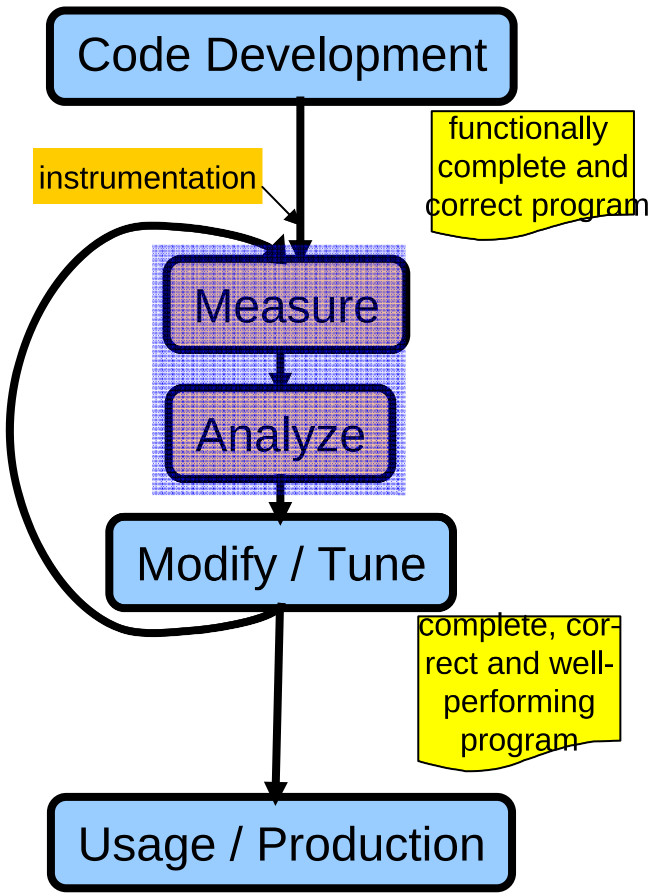
- **Performance Analysis is important**
  - Large investments in HPC systems
    - Procurement: ~$40 Mio
    - Operational costs: ~$5 Mio per year
    - Electricity: 1 MWyear ~$1 Mio
  - Goal: solve **larger** problems
  - Goal: solve problems **faster**





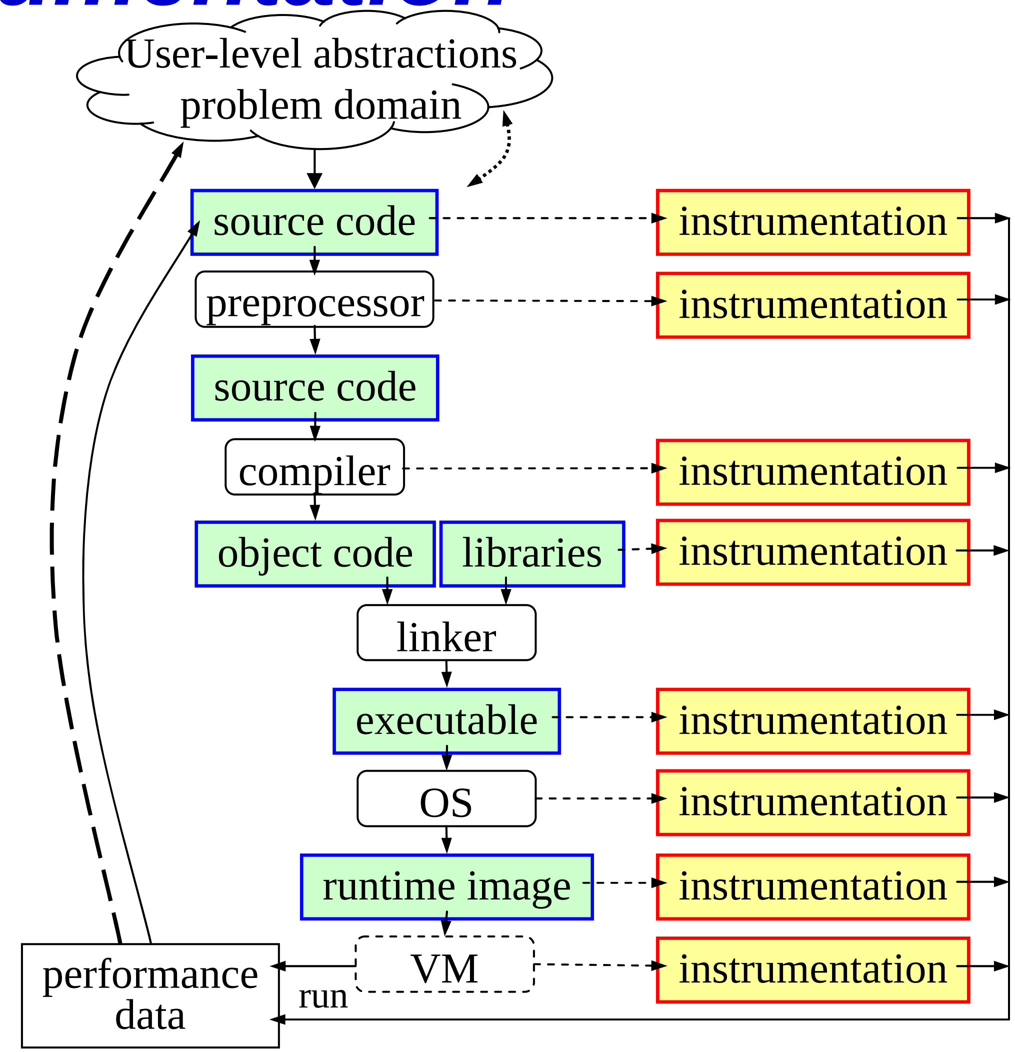
Concepts and Definitions

- **The typical performance optimization cycle**



Instrumentation

- Instrumentation = adding measurement probes to the code to observe its execution
- Can be done on several levels
- Different techniques for different levels
- Different overheads and levels of accuracy with each technique
- No instrumentation: run in a simulator. E.g., Valgrind



Instrumentation – Examples (1)

- **Source code instrumentation**
  - **User added** time measurement, etc. (e.g., **`printf`**(), **`gettimeofday`**())
  - Many **tools** expose mechanisms for source code instrumentation in addition to automatic instrumentation facilities they offer
  - Instrument program phases:
    - initialization/main iteration loop/data post processing
  - Pramga and pre-processor based

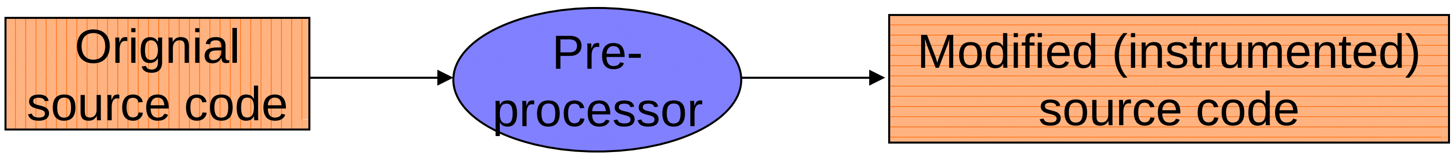
```
#pragma pomp inst begin(foo)
#pragma pomp inst end(foo)
```

  - Macro / function call based

```
ELG_USER_START("name");
...
ELG_USER_END("name");
```

Instrumentation – Examples (2)

- **Preprocessor Instrumentation**
  - Example: Instrumenting OpenMP constructs with Opari
  - Preprocessor operation



  - Example: Instrumentation of a parallel region

```
POMP_Parallel_fork [master]
#pragma omp parallel {
    POMP_Parallel_begin [team]
        /* ORIGINAL CODE in parallel region */
        POMP_Barrier_Enter [team]
        #pragma omp barrier
        POMP_Barrier_Exit [team]
    POMP_Parallel_end [team]
}
POMP_Parallel_join [master]
```

This is used for OpenMP analysis in tools such as KOJAK/Scalasca/ompP

Instrumentation added by Opari

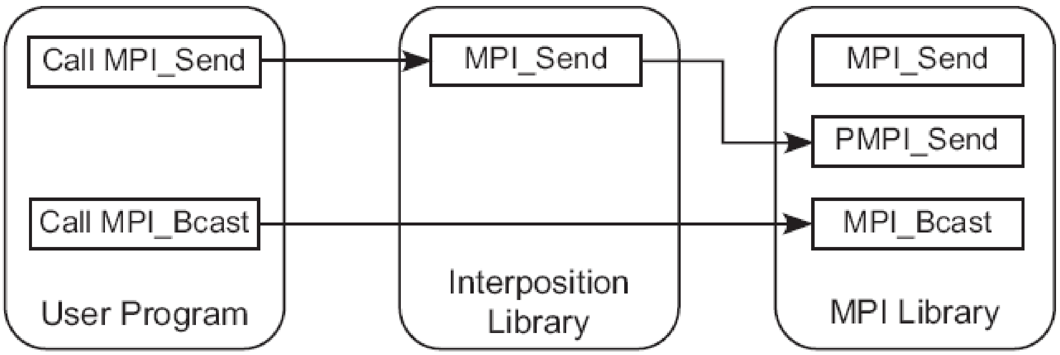
Instrumentation – Examples (3)

- **Compiler Instrumentation**
  - Many compilers can instrument functions automatically
  - GNU compiler flag: **`-finstrument-functions`**
  - Automatically calls functions on function entry/exit that a tool can capture
  - Not standardized across compilers, often undocumented flags, sometimes not available at all
  - GNU compiler example:

```
void __cyg_profile_func_enter(void *this, void *callsite)
{
    /* called on function entry */
}

void __cyg_profile_func_exit(void *this, void *callsite)
{
    /* called just before returning from function */
}
```

Instrumentation – Examples (4)

- **Library Instrumentation:**



- **MPI library interposition**
  - All functions are available under two names: **MPI_xxx** and **PMPI_xxx**, **MPI_xxx** symbols are **weak**, can be over-written by interposition library
  - Measurement code in the interposition library measures begin, end, transmitted data, etc… and calls corresponding PMPI routine.
  - Not all MPI functions need to be instrumented

Measurement

- **Profiling vs. Tracing**

- **Profiling**
  - Summary statistics of performance metrics
    - Number of times a routine was invoked
    - Exclusive, inclusive time/hpm counts spent executing it
    - Number of instrumented child routines invoked, etc.
    - Structure of invocations (call-trees/call-graphs)
    - Memory, message communication sizes

- **Tracing**
  - When and where events took place along a global timeline
    - Time-stamped log of events
    - Message communication events (sends/receives) are tracked
    - Shows when and from/to where messages were sent
    - Large volume of performance data generated usually leads to more perturbation in the program

Measurement: Profiling

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

Profiling: Inclusive vs. Exclusive

```
int main( )
{ /* takes 100 secs */
  f1(); /* takes 20 secs */
  /* other work */
  f2(); /* takes 50 secs */
  f1(); /* takes 20 secs */
  /* other work */
}

/* similar for other metrics,
such as hardware performance
counters, etc. */
```

- Inclusive **time for** main
  - 100 secs
- Exclusive **time for** main
  - 100-20-50-20=10 secs
- **Exclusive time sometimes called “self time”**

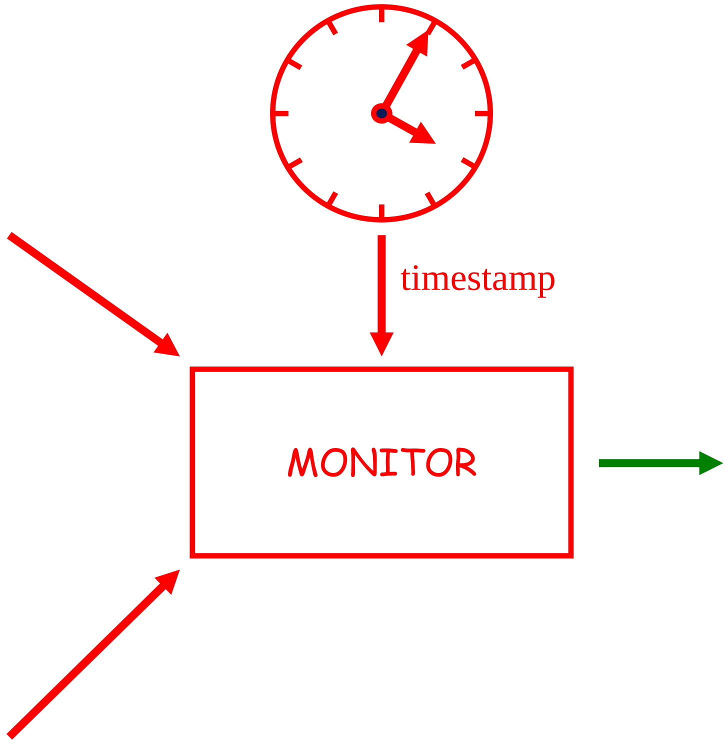
Tracing Example: Instrumentation, Monitor, Trace

CPU A:

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

CPU B:

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



Event definition

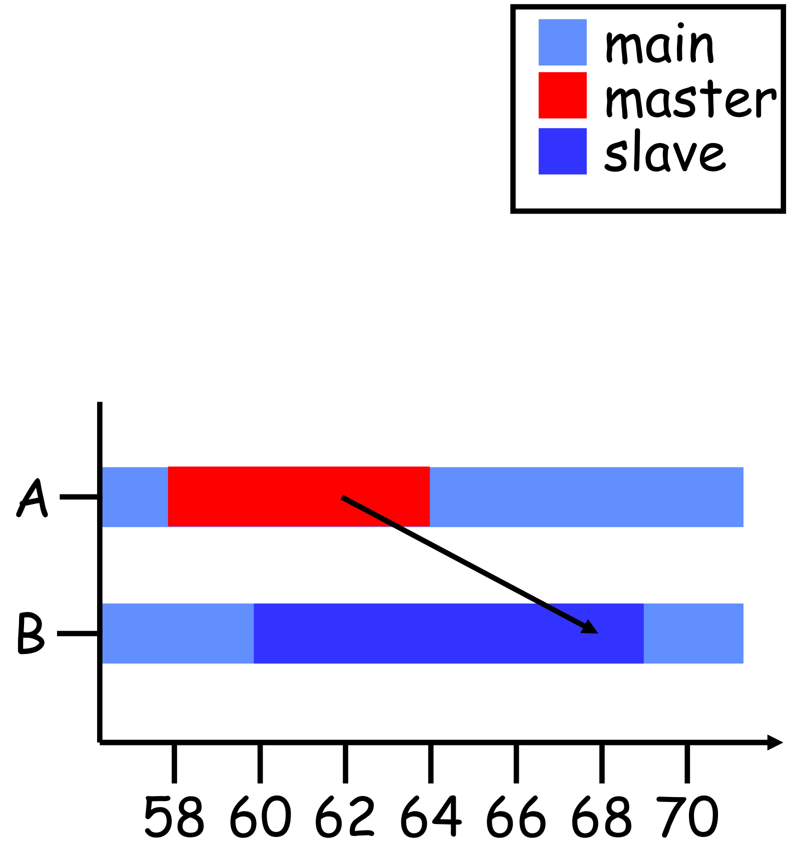
1	master
2	slave
3	...

...			
58	A	ENTER	1
60	B	ENTER	2
62	A	SEND	B
64	A	EXIT	1
68	B	RECV	A
69	B	EXIT	2
...			

Tracing: Timeline Visualization

1	master
2	slave
3	...

...			
58	A	ENTER	1
60	B	ENTER	2
62	A	SEND	B
64	A	EXIT	1
68	B	RECV	A
69	B	EXIT	2
...			



Measurement: Tracing

- **Tracing**
  - Recording of information about significant points (events) during program execution
    - entering/exiting code region (function, loop, block, …)
    - thread/process interactions (e.g., send/receive message)
  - Save information in event record
    - timestamp
    - CPU identifier, thread identifier
    - Event type and event-specific information
  - Event trace is a time-sequenced stream of event records
  - Can be used to reconstruct dynamic program behavior
  - Typically requires code instrumentation

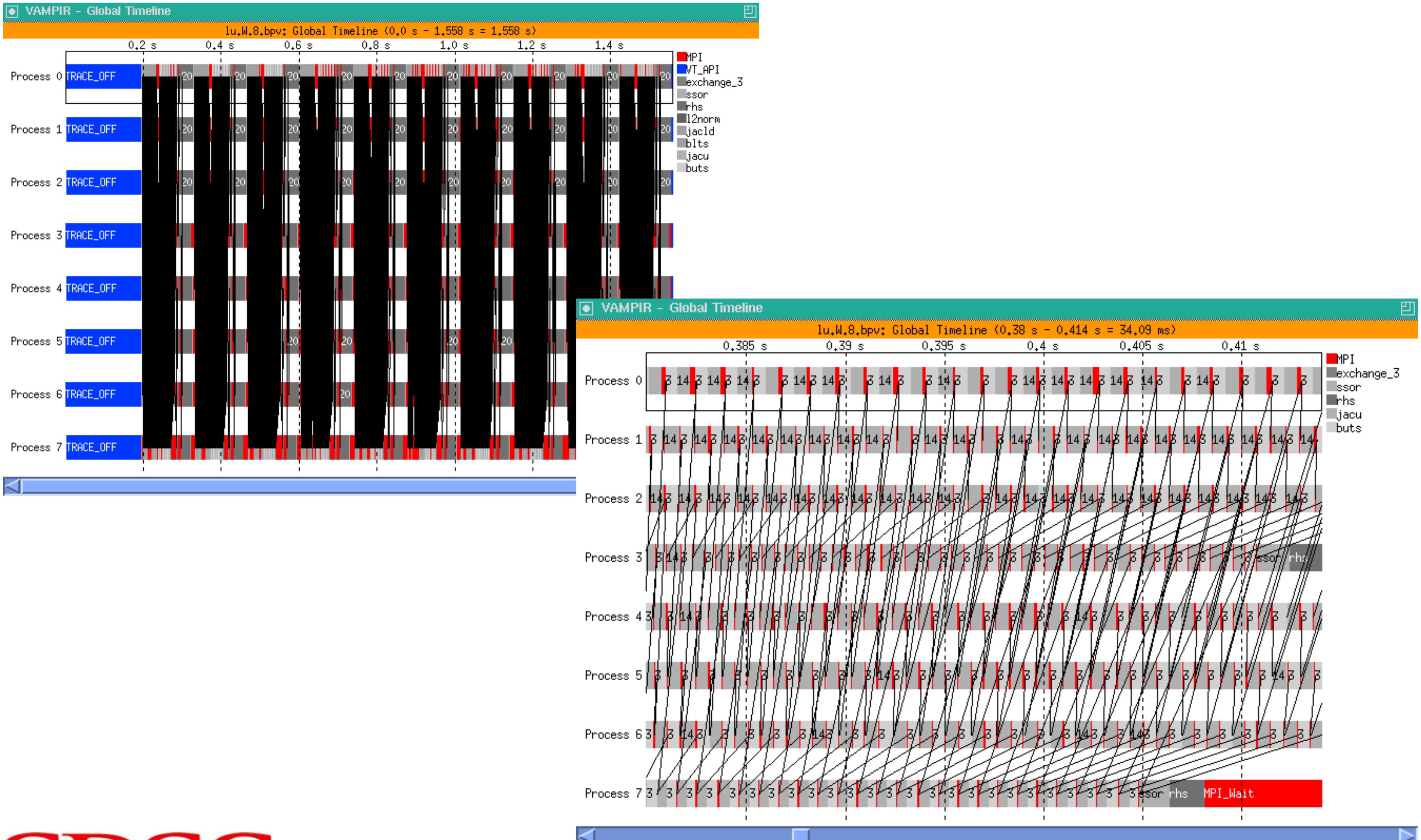
Performance Data Analysis

- **Draw conclusions from measured performance data**

- **Manual analysis**
  - Visualization
  - Interactive exploration
  - Statistical analysis
  - Modeling

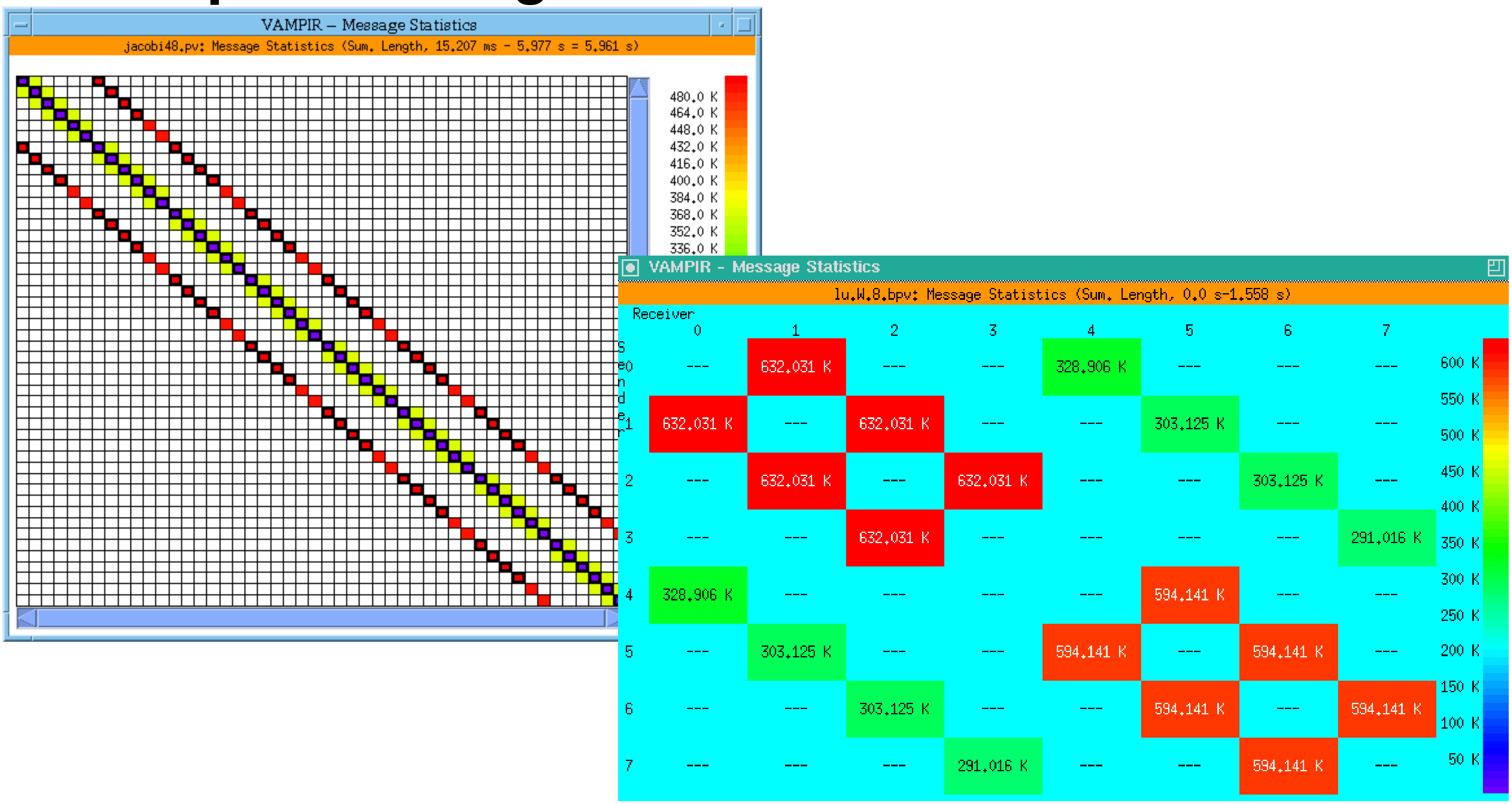
- **Automated analysis**
  - Try to cope with huge amounts of performance by automation
  - Examples: Paradyn, KOJAK, Scalasca

Trace File Visualization



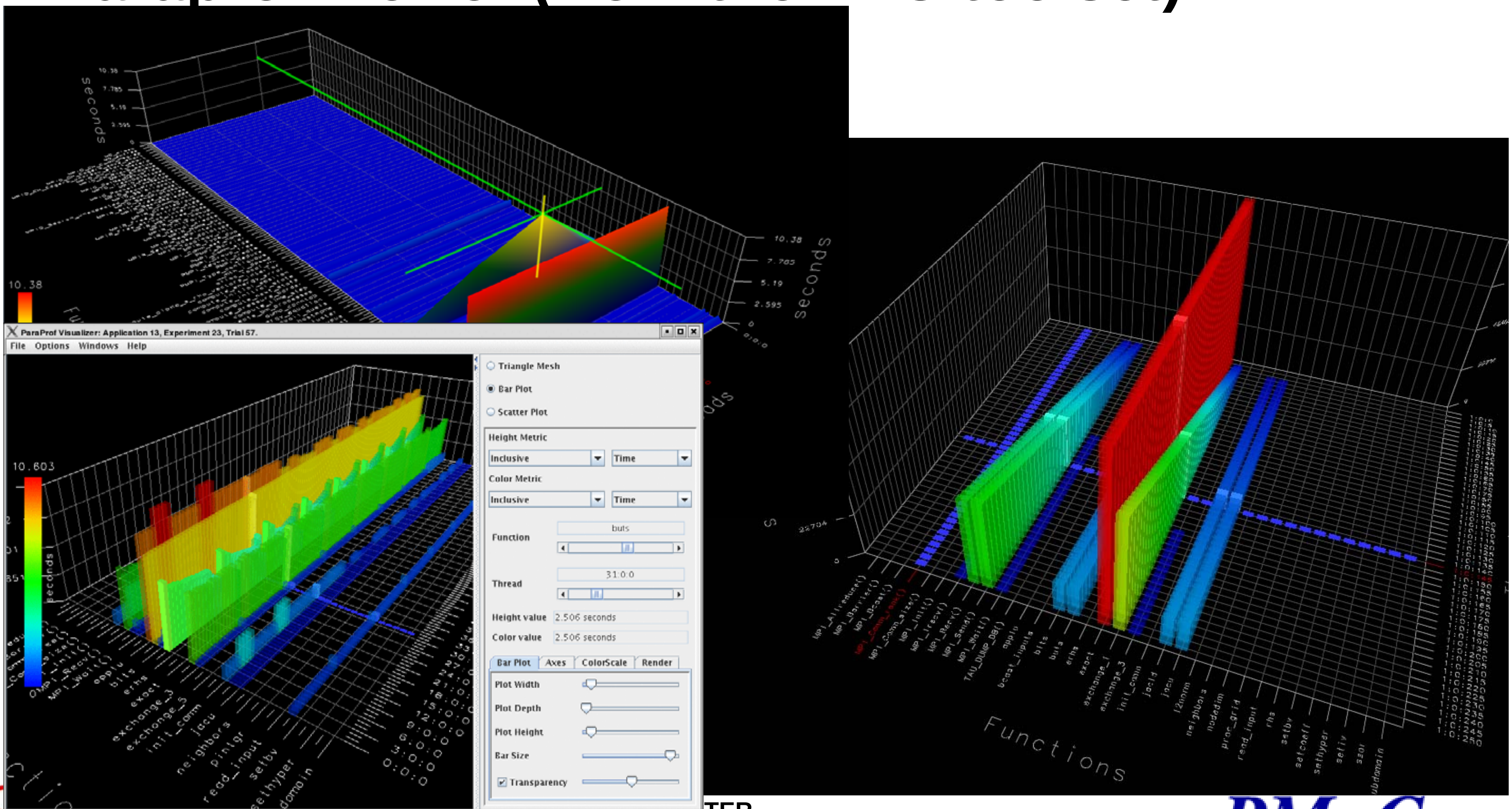
Trace File Visualization

- **Vampir: message communication statistics**



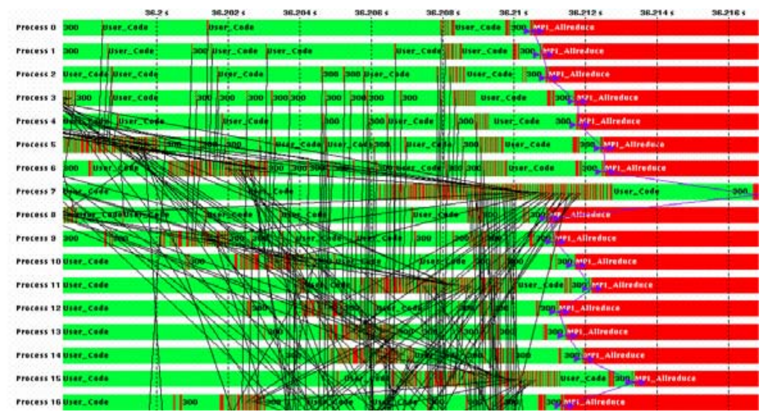
3D performance data exploration

- **Paraprof viewer (from the TAU toolset)**

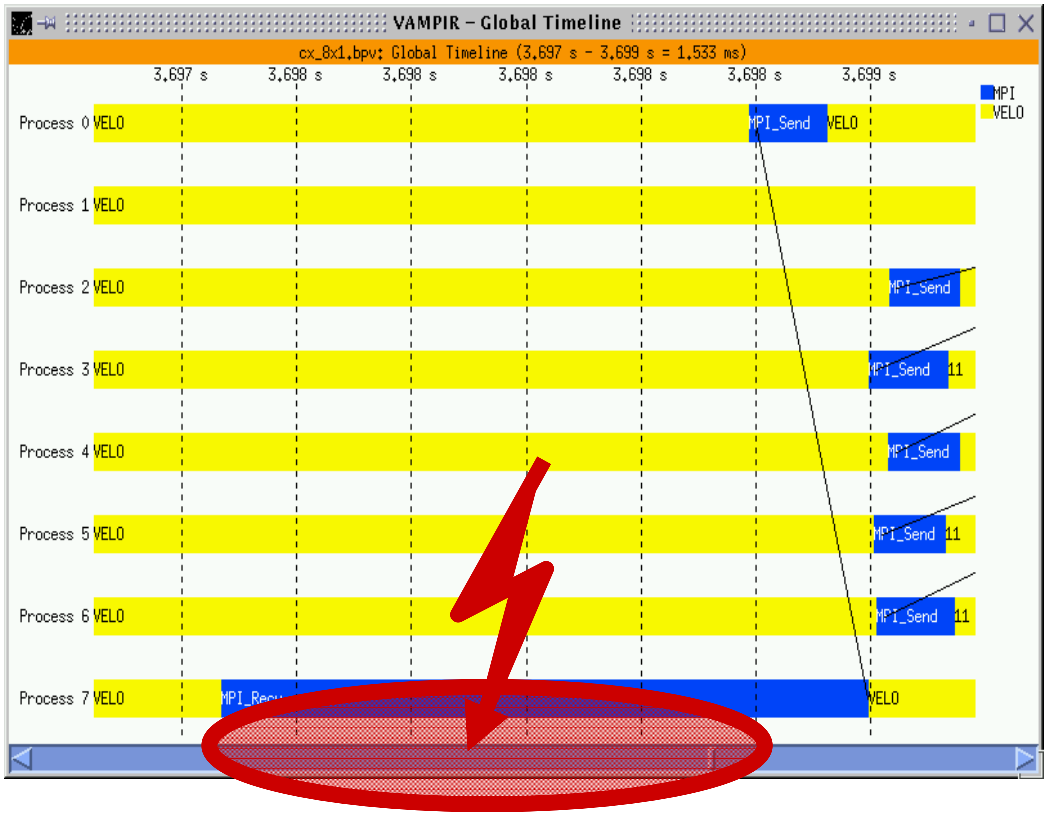


Automated Performance Analysis

- **Reason for Automation**
  - Size of systems: several tens of thousand of processors
  - LLNL Sequoia: ~1.6 million cores
  - Trend to multi-core
- **Large amounts of performance data when tracing**
  - Several gigabytes or even terabytes
  - Overwhelms user
- **Not all programmers are performance experts**
  - Scientists want to focus on their domain
  - Need to keep up with new machines
- **Automation can solve some of these issues**



Automation - Example



This is a situation that can be detected *automatically* by analyzing the trace file

-> late sender pattern

Menu

- **Performance Analysis Concepts and Definitions**
  - Why and when to look at performance
  - Types of performance measurement
- **Examining typical performance issues today using IPM**
- **Summary**

“Premature optimization is the root of all evil.” - Donald Knuth

- Before attempting to optimize make sure your code works correctly !
  - **Debugging before tuning**
  - **Nobody really cares how fast you can compute**
  - **the wrong answer**
- 80/20 Rule
  - **Program spends 80 % of its time in 20% of the code**
  - **Programmer spends 20% effort to get 80% of the total speedup possible**
  - **Know when to stop !**
  - **Don’t optimize what does not matter**

Practical Performance Tuning

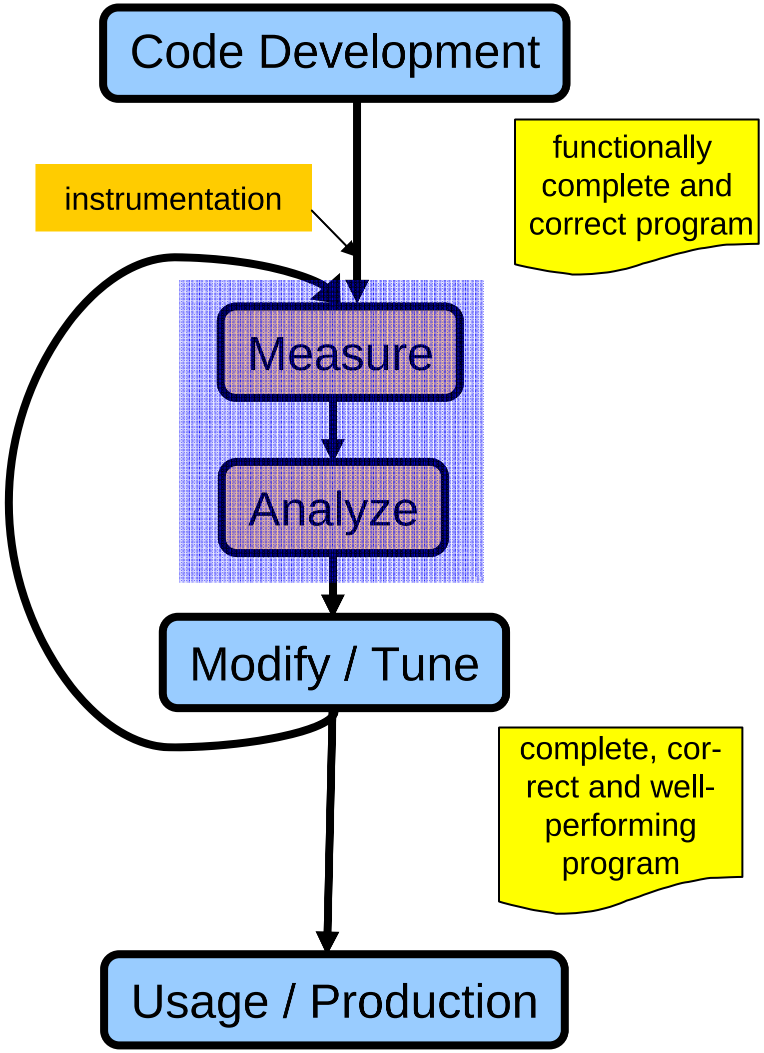
Successful tuning is combination of

- **Right algorithm and libraries**
- **Compiler flags and pragmas / directives (Learn and use them)**

- **THINKING**

Measurement > intuition (~guessing !)

- **To determine performance problems**
- **To validate tuning decisions / optimizations (after each step!)**



Typical Performance Analysis Procedure

- **Do I have a performance problem at all? What am I trying to achieve ?**
  - Time / hardware counter measurements
  - Speedup and scalability measurements
- **What is the main bottleneck (computation/communication...) ?**
  - Flat profiling (sampling / prof)
  - Why is it there?

Users Perspective: I Just Want to do My Science ! - Barriers to Entry Must be Low

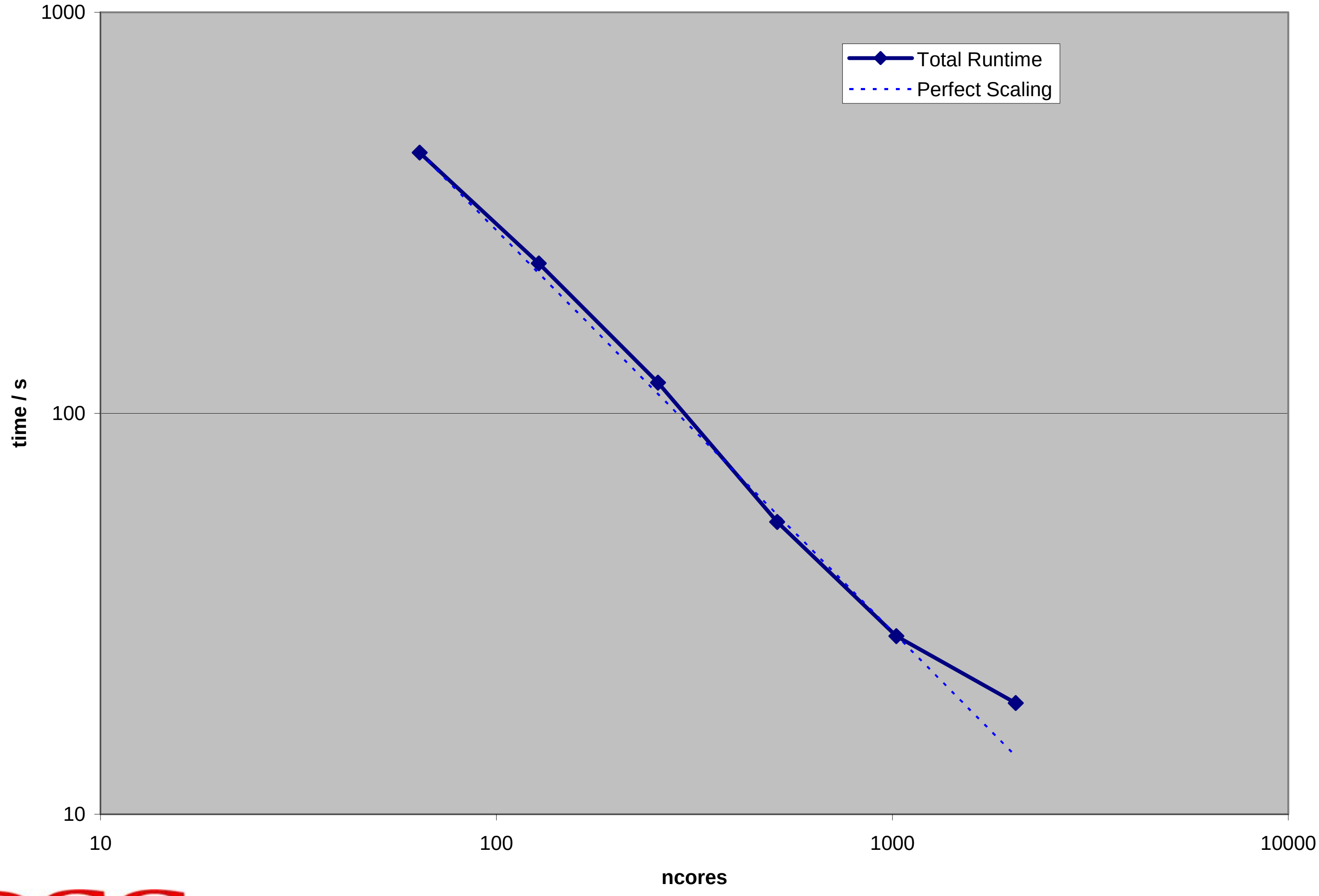
- **"Yea, I tried that tool once, it took me 20 minutes to figure out how to get the code to compile, then it output a bunch of information, none of which I wanted, so I gave up."**
- **Is it easier than this ?**

  Call timer

  Code_of_interest

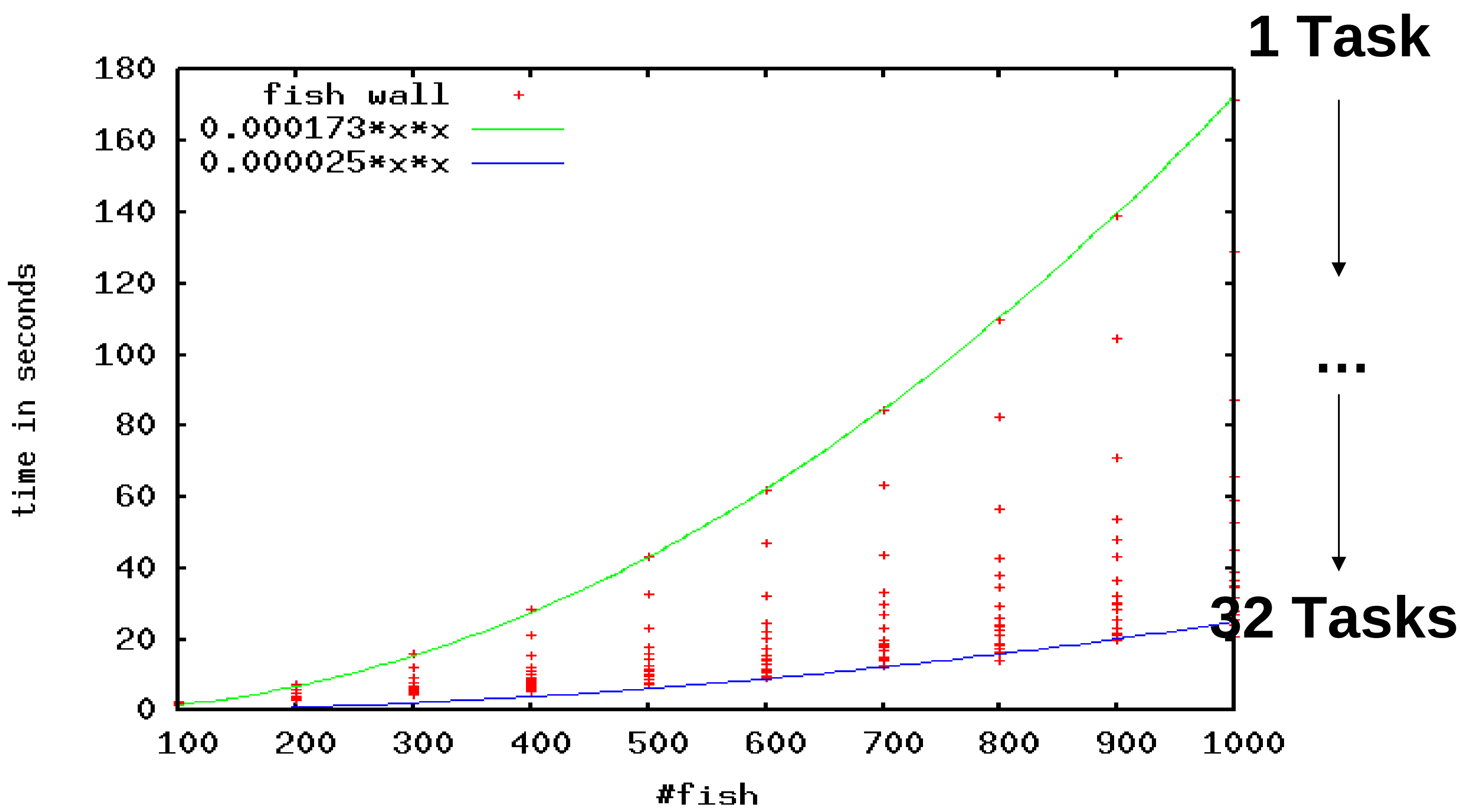
  Call timer
- **The carrot works. The stick does not.**

MILC on Ranger –Runtime Shows Perfect Scalability



Scaling: Good 1st Step: Do runtimes make sense?

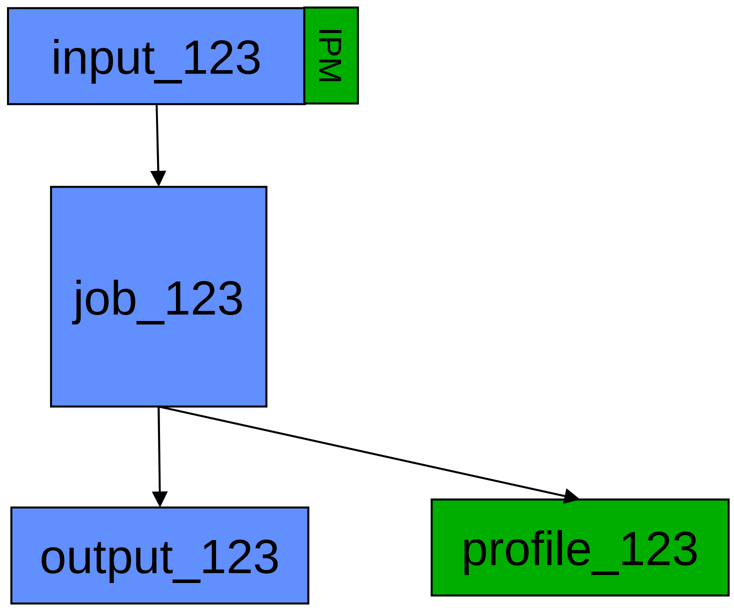
Running fish_sim for 100-1000 fish on 1-32 CPUs we see



time ~ fish^2 ✓

What is Integrated Performance Monitoring?

IPM provides a performance profile on a batch job



How to use IPM : basics

1) Do "module load ipm", then "setenv LD_PRELOAD …"
2) Upon completion you get

```
##IPMv0.85################################################################
#
# command : ../exe/pmemd -O -c inpcrd -o res (completed)
# host    : s05405                          mpi_tasks : 64 on 4 nodes
# start   : 02/22/05/10:03:55               wallclock : 24.278400 sec
# stop    : 02/22/05/10:04:17               %comm     : 32.43
# gbytes  : 2.57604e+00 total               gflop/sec : 2.04615e+00 total
#
###########################################################################
```

Maybe that's enough. If so you're done.

Have a nice day.

Want more detail? IPM_REPORT=full

```
##IPMv0.85#####################################################################
#
# command : ../exe/pmemd -O -c inpcrd -o res (completed)
# host    : s05405                         mpi_tasks : 64 on 4 nodes
# start   : 02/22/05/10:03:55              wallclock : 24.278400 sec
# stop    : 02/22/05/10:04:17              %comm     : 32.43
# gbytes  : 2.57604e+00 total              gflop/sec : 2.04615e+00 total
#
#                          [total]       <avg>          min          max
# wallclock                1373.67     21.4636      21.1087      24.2784
# user                      936.95     14.6398        12.68         20.3
# system                     227.7     3.55781         1.51            5
# mpi                      503.853      7.8727       4.2293      9.13725
# %comm                                32.4268        17.42       41.407
# gflop/sec                2.04614   0.0319709      0.02724      0.04041
# gbytes                   2.57604   0.0402507    0.0399284    0.0408173
# gbytes_tx               0.665125   0.0103926  1.09673e-05    0.0368981
# gbyte_rx                0.659763   0.0103088  9.83477e-07    0.0417372
#
```

Want more detail? IPM_REPORT=full

```
# PM_CYC            3.00519e+11  4.69561e+09  4.50223e+09  5.83342e+09
# PM_FPU0_CMPL      2.45263e+10  3.83223e+08   3.3396e+08  5.12702e+08
# PM_FPU1_CMPL      1.48426e+10  2.31916e+08  1.90704e+08   2.8053e+08
# PM_FPU_FMA        1.03083e+10  1.61067e+08  1.36815e+08  1.96841e+08
# PM_INST_CMPL      3.33597e+11  5.21245e+09  4.33725e+09  6.44214e+09
# PM_LD_CMPL        1.03239e+11  1.61311e+09  1.29033e+09  1.84128e+09
# PM_ST_CMPL        7.19365e+10  1.12401e+09  8.77684e+08  1.29017e+09
# PM_TLB_MISS       1.67892e+08  2.62332e+06  1.16104e+06  2.36664e+07
#
#                        [time]      [calls]       <%mpi>      <%wall>
# MPI_Bcast             352.365         2816        69.93        22.68
# MPI_Waitany           81.0002       185729        16.08         5.21
# MPI_Allreduce         38.6718         5184         7.68         2.49
# MPI_Allgatherv        14.7468          448         2.93         0.95
# MPI_Isend             12.9071       185729         2.56         0.83
# MPI_Gatherv           2.06443          128         0.41         0.13
# MPI_Irecv               1.349       185729         0.27         0.09
# MPI_Waitall          0.606749         8064         0.12         0.04
# MPI_Gather          0.0942596          192         0.02         0.01
###############################################################################
```

Want More? – You'll Need a Webbrowser

Which problems should be tackled with IPM?

- **Performance Bottleneck Identification**
  - Does the profile show what I expect it to?
  - Why is my code not scaling?
  - Why is my code running 20% slower than I expected?
- **Understanding Scaling**
  - Why does my code scale as it does ? (MILC on Ranger)
- **Optimizing MPI Performance**
  - Combining Messages

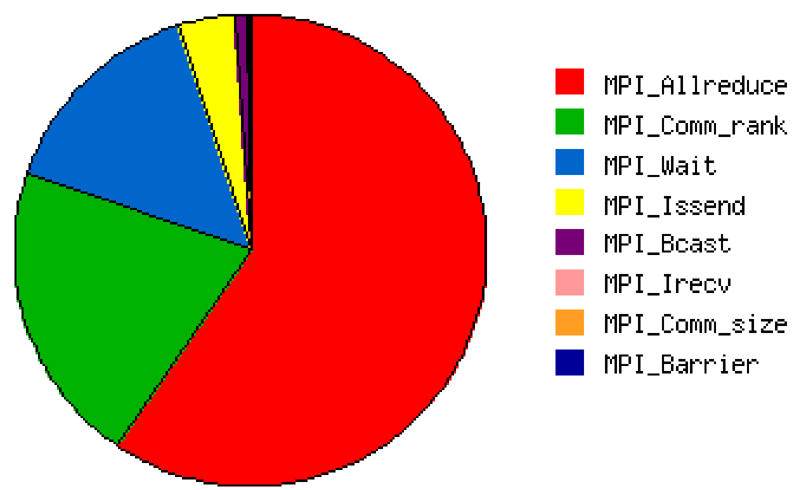
Using IPM to Understand Common Performance Issues

- **Dumb Mistakes**
- **Load balancing**
- **Combining Messages**
- **Scaling behavior**
- **Amdahl (serial) fractions**
- **Optimal Cache Usage**

What's wrong here?

Communication

% of MPI Time

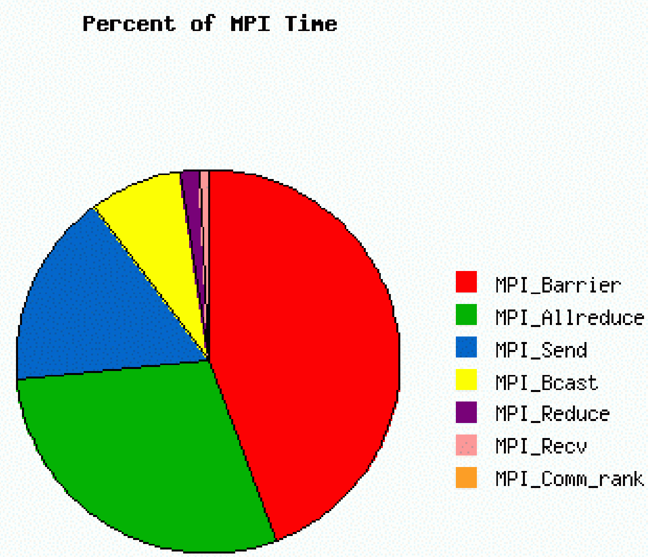


Communication Event Statistics (100.00% detail)

	Buffer Size	Ncalls	Total Time	Min Time	Max Time	%MPI	%Wall
MPI_Allreduce	8	3278848	124132.547	0.000	114.920	59.35	16.88
MPI_Comm_rank	0	35173439489	43439.102	0.000	41.961	20.77	5.91
MPI_Wait	98304	13221888	15710.953	0.000	3.586	7.51	2.14
MPI_Wait	196608	13221888	5331.236	0.000	5.716	2.55	0.72
MPI_Wait	589824	206848	5166.272	0.000	7.265	2.47	0.70

MPI_Barrier

Function	Total calls	Total time (sec)		Total buffer size (MB)	Avg. Buffer Size/call (Bytes)
MPI_Barrier	6.02e+05	3.48e+05	44.23%	0	0
MPI_Allreduce	3.18e+07	2.31e+05	29.33%	3.61e+05	11,936
MPI_Send	1.29e+08	1.29e+05	16.36%	5.24e+04	426
MPI_Bcast	5.73e+07	6.08e+04	7.73%	5.39e+04	987
MPI_Reduce	1.08e+08	1.24e+04	1.58%	1.66e+05	1,620
MPI_Recv	1.29e+08	6.11e+03	0.78%	5.24e+04	426
MPI_Comm_rank	1.14e+03	5.92e-01	7.52e-05%	0	0
MPI_Comm_size	6.66e+02	0	0%	0	0



Is MPI_Barrier time bad? Probably. Is it avoidable?

~three cases:

1) The stray / unknown / debug barrier
2) The barrier which is masking compute balance
3) Barriers used for I/O ordering

Often very easy to fix

120708.nid03588

- Load Balance
- Communication Balance
- Message Buffer Sizes
- Communication Topology
- Switch Traffic
- Memory Usage
- Executable Info
- Host List
- Environment
- Developer Info

Powered by IPM

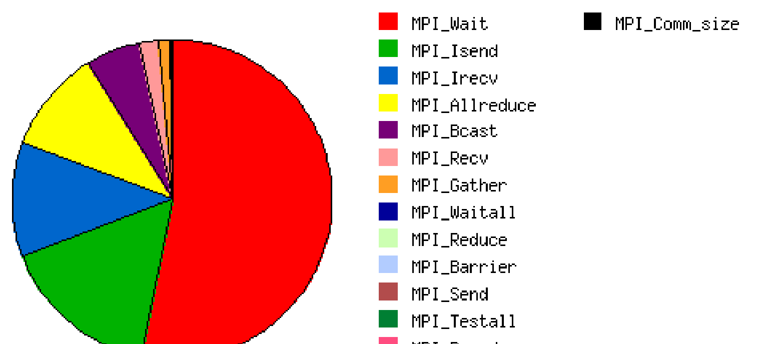
command: unknown			
codename:	unknown	state:	running
username:	unknown	group:	unknown
host:	yodjag12 (x86_64_catamount)	mpi_tasks:	256 on 1 hosts
start:	08/17/07/07:51:08	wallclock:	1192.54 sec
stop:	08/17/07/08:11:01	%comm:	36.2
total memory:	0.000 gbytes	total gflop/sec:	616.168120
switch(send):	-0.000 gbytes	switch(recv):	-0.000 gbytes

COMPUTATION

Event	Count	Pop
PAPI_FP_OPS	734805154331357	*
PAPI_TOT_CYC	687097407373206	*
PAPI_TOT_INS	815378888957961	*
PAPI_VEC_INS	525829751778865	*

COMMUNICATION

% of MPI Time



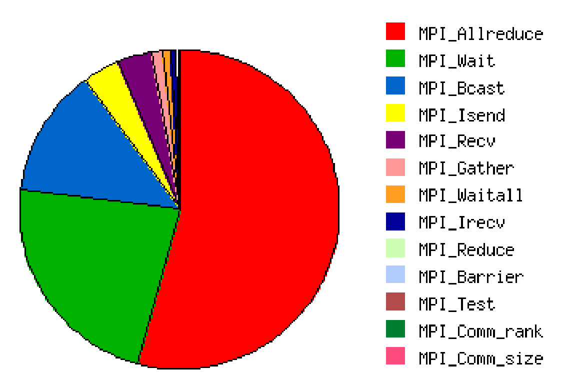
star/paratec.mpi.opt			
	unknown	state:	running
	nwright	group:	CSD102
	ds155)020A67A4C00_AIX)	mpi_tasks:	256 on 32 hosts
	08/14/07/19:24:11	wallclock:	1286.31 sec
	08/14/07/19:45:37	%comm:	20.0
	200.904 gbytes	total gflop/sec:	555.396942
	1183.325 gbytes	switch(recv):	1183.325 gbytes

COMPUTATION

Event	Count	Pop
PM_CYC	469586794358386	*
PM_FPU0_FIN	184606070199636	*
PM_FPU1_FIN	183964050563598	*
PM_FPU_FDIV	19710887051	*
PM_FPU_FMA	353749004950530	*
PM_FPU_STF	7906415712146	*
PM_INST_CMPL	618304210821913	*
PM_LSU_LDF	140544138144501	*

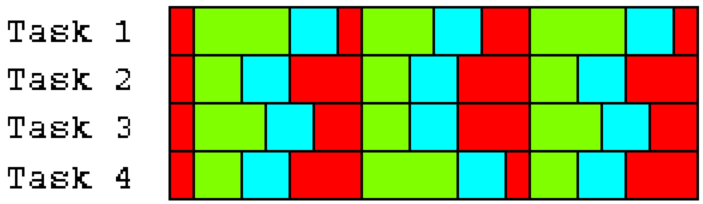
COMMUNICATION

% of MPI Time

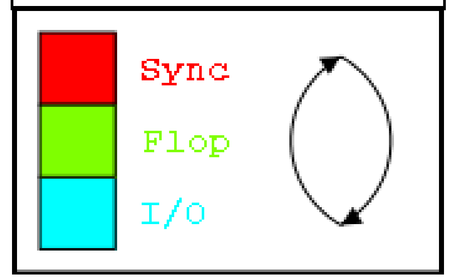


Load Balance : Application Cartoon

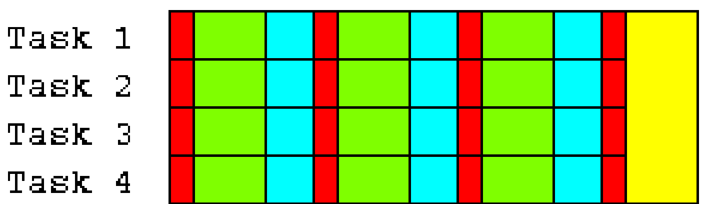
Unbalanced:



Universal App



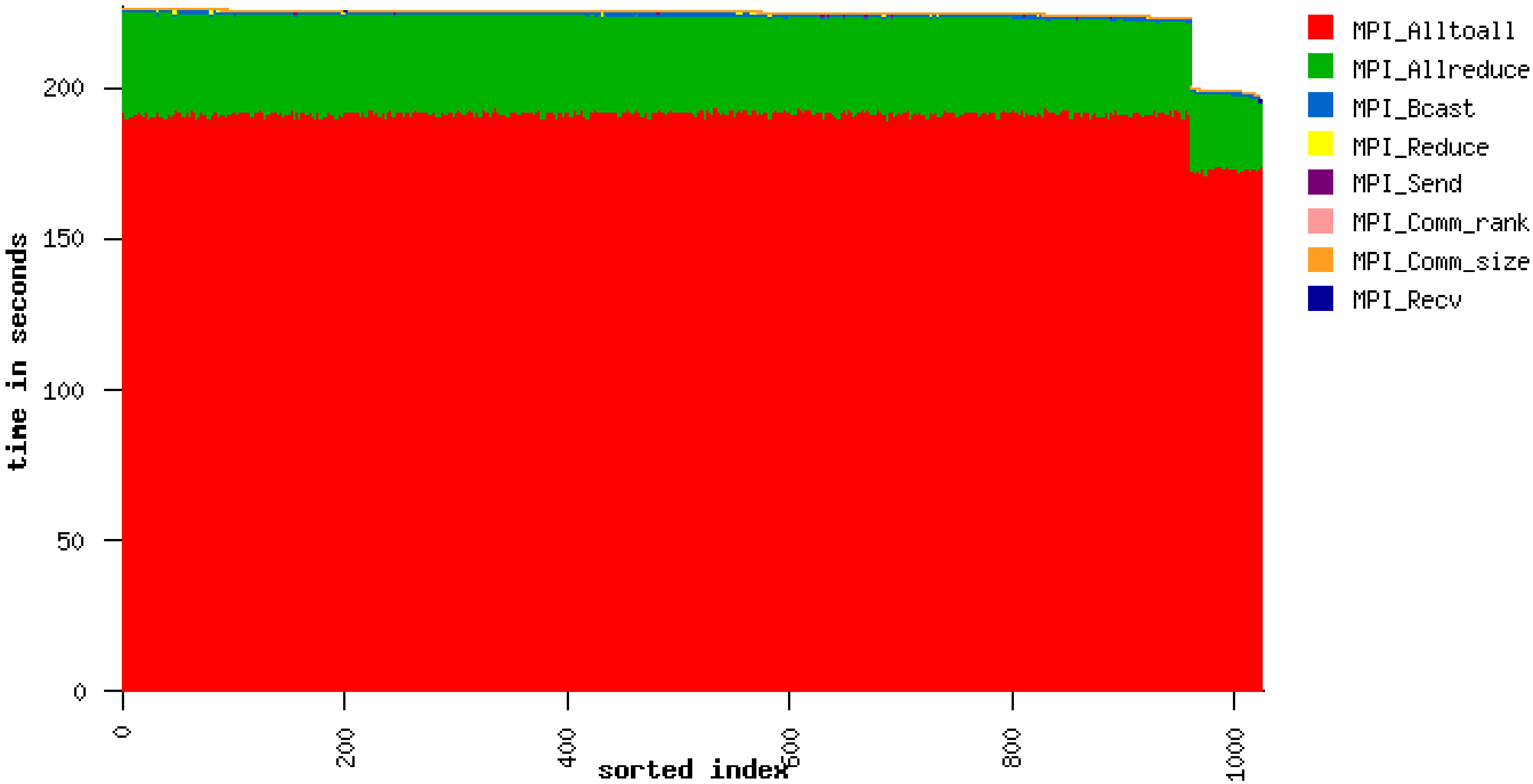
Balanced:



Time saved by load balance

Load Balance : performance data

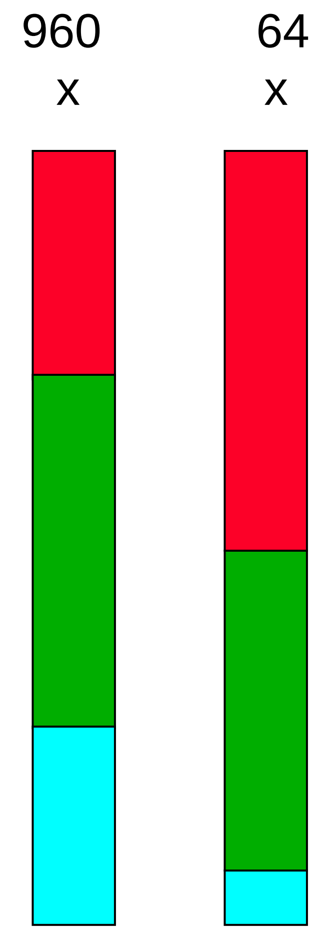
Communication Time: 64 tasks show 200s, 960 tasks show 230s



MPI ranks sorted by total communication time

Load Balance: ~code

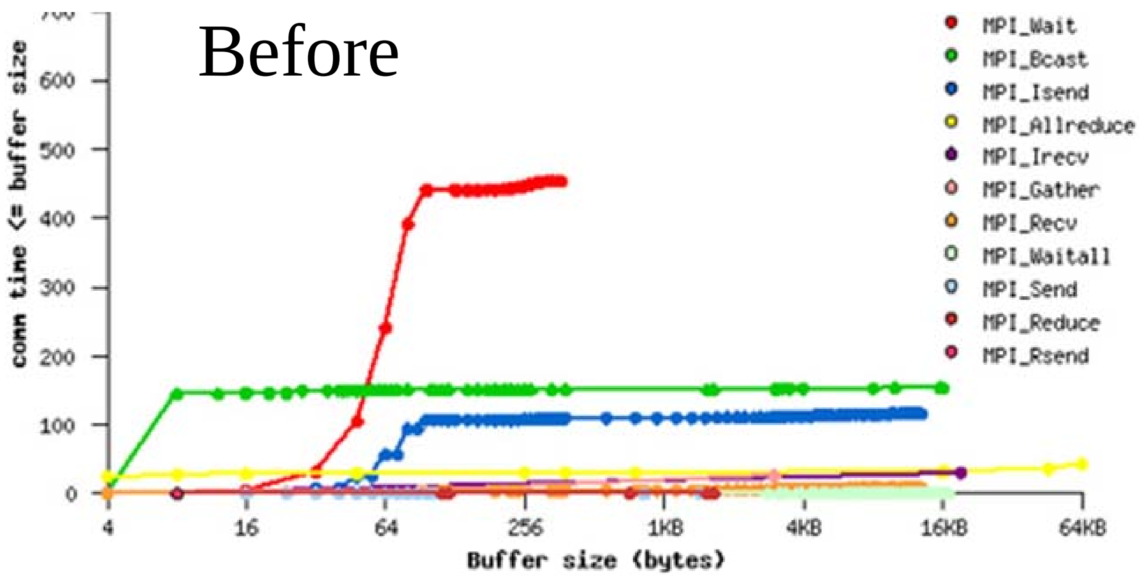
```
while(1) {
  do_flops(N_i);

   MPI_Alltoall(
   );

   MPI_Allreduce
   ();
}
```

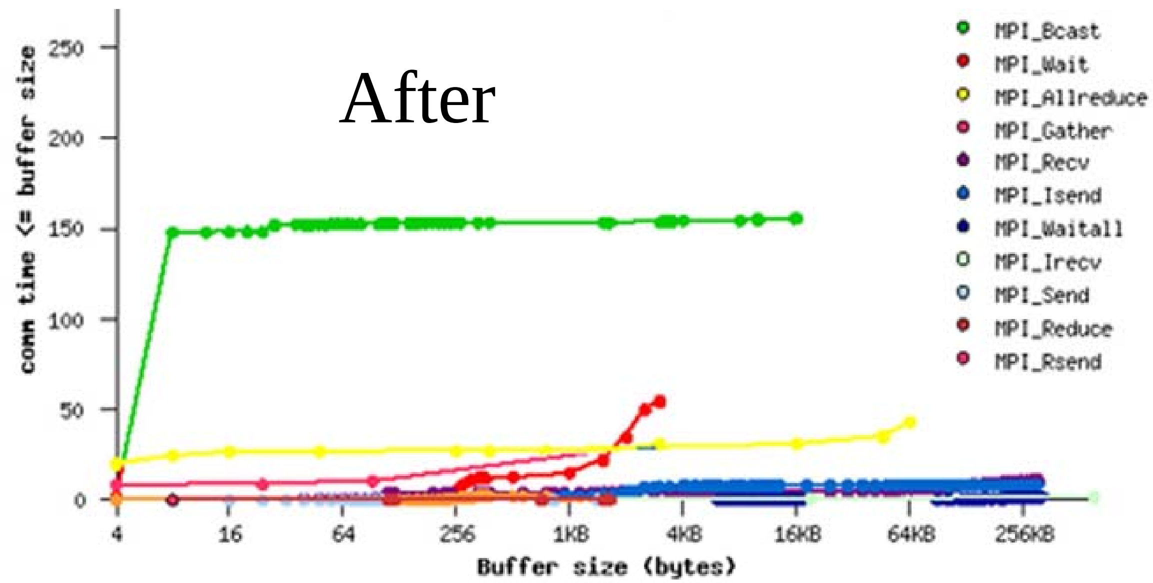


Load Balance : analysis

- The 64 slow tasks (with more compute work) cause 30 seconds more "communication" in 960 tasks
- This leads to 28800 CPU*seconds (8 CPU*hours) of unproductive computing
- All load imbalance requires is one slow task and a synchronizing collective!
- Pair well problem size and concurrency.
- Parallel computers allow you to waste time faster!

Message Aggregation Improves Performance





Ideal Scaling Behavior

- **Strong Scaling**
  - Fix the size of the problem and increase the concurrency
    - # of grid points per mpi task decreases as 1/P
    - Ideally runtime decreases as 1/P
  - Run out of parallel work

- **Weak Scaling**
  - Increase the problem size with the concurrency
    - # of grid points per mpi task remains constant
    - Ideally runtime remains constant as P increases
  - Time to solution

Scaling Behavior : MPI Functions

- Local : leave based on local logic
  - **MPI_Comm_rank, MPI_Get_count**
- Probably Local : try to leave w/o messaging other tasks
  - **MPI_Isend/Irecv**
- Partially synchronizing : leave after messaging M<N tasks
  - **MPI_Bcast, MPI_Reduce**
- Fully synchronizing : leave after every else enters
  - **MPI_Barrier, MPI_Allreduce**

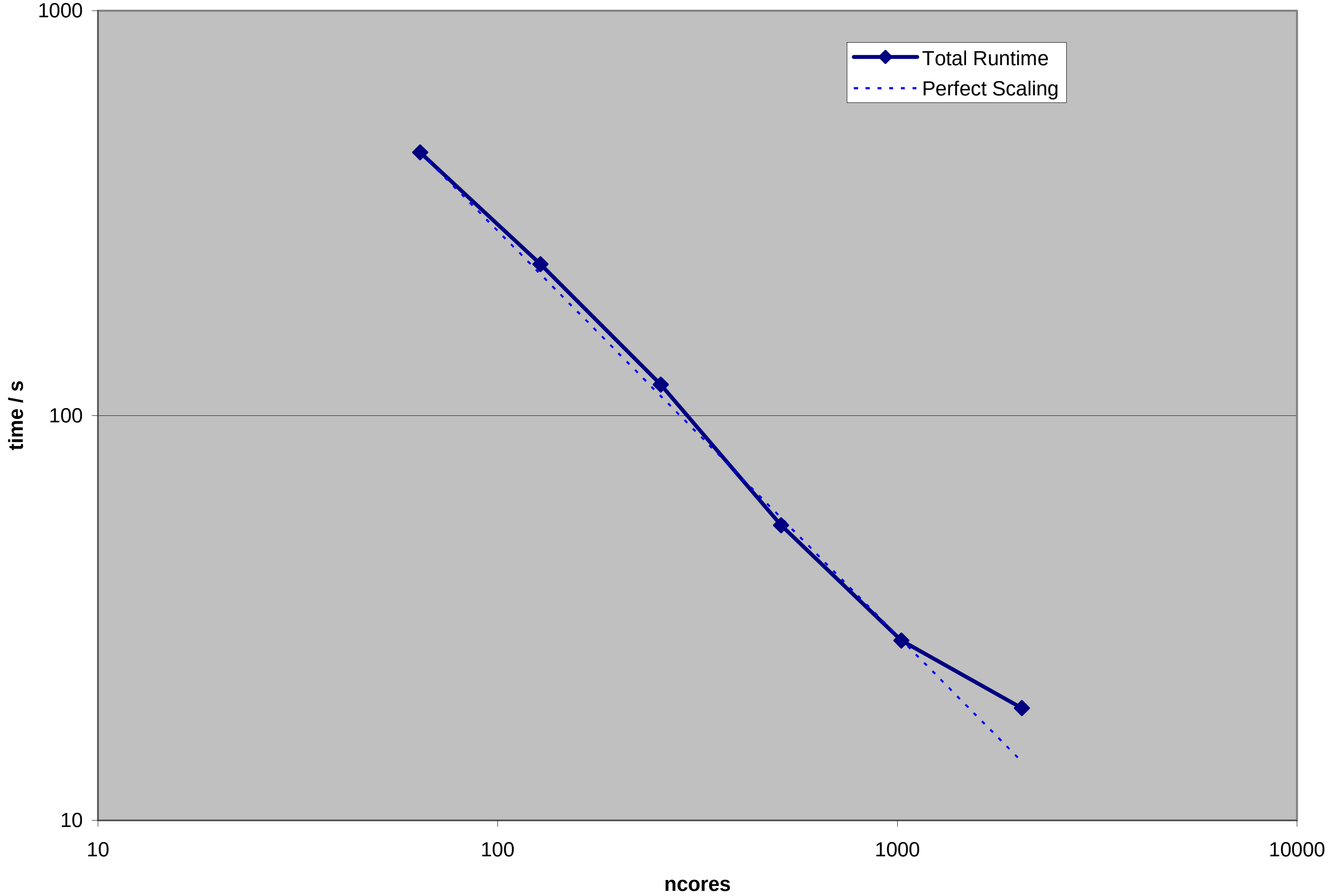
Strong Scaling: Communication Bound

MPI_Allreduce buffer size is 32 bytes.

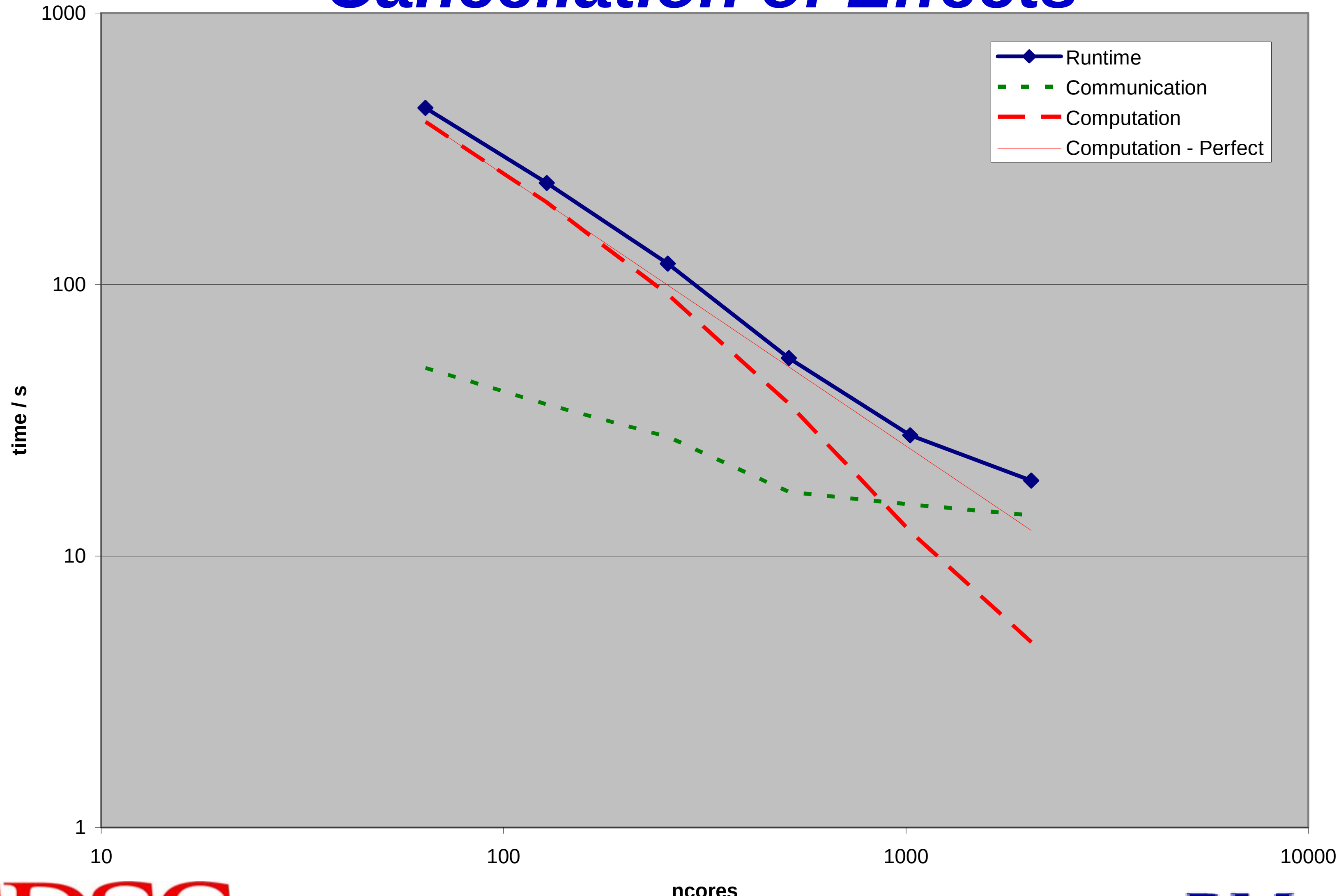
Q: What resource is being depleted here?
A: Small message latency

1) Compute per task is decreasing
2) Synchronization rate is increasing
3) Surface:Volume ratio is increasing

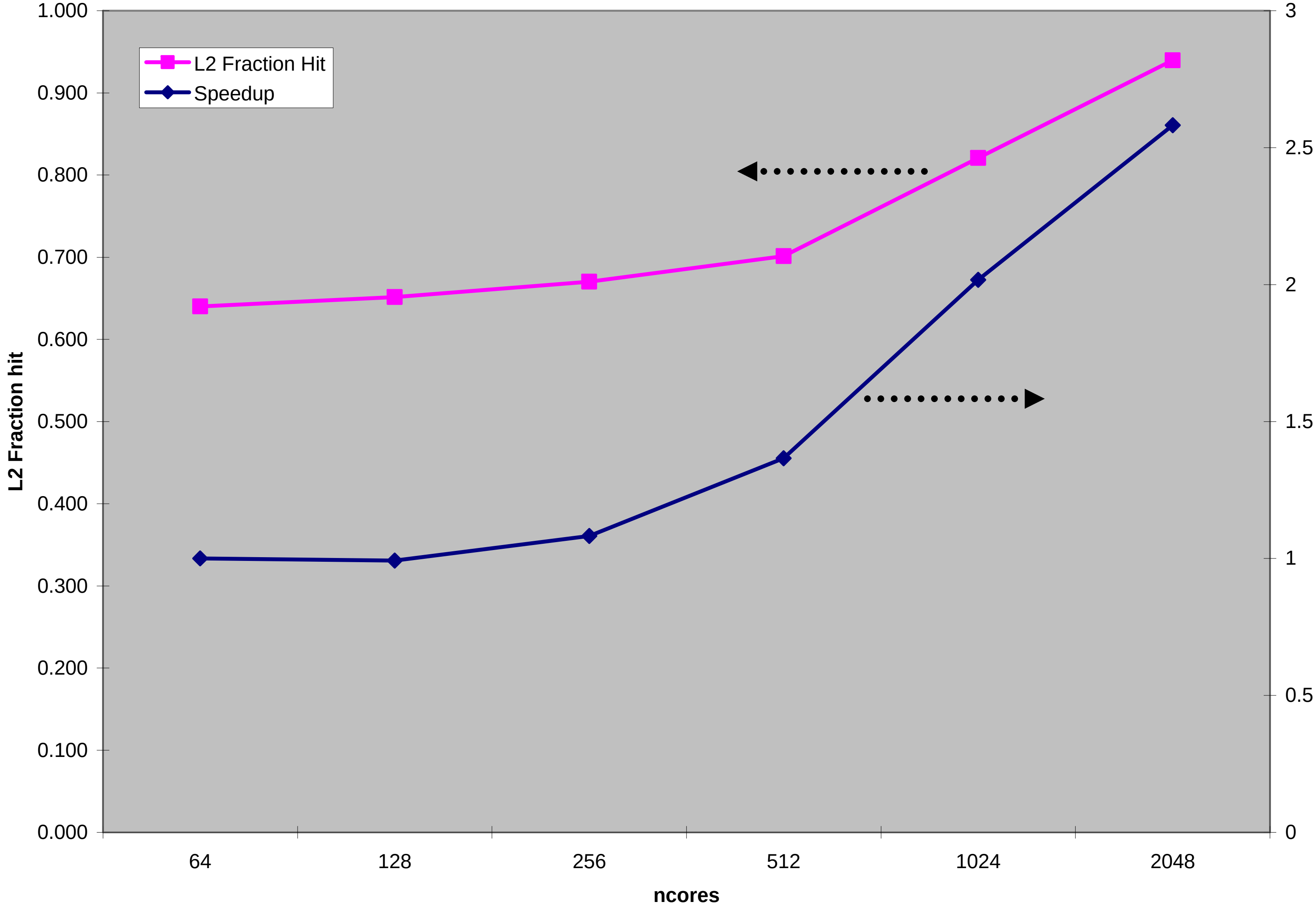
MILC on Ranger –Runtime Shows Perfect Scalability



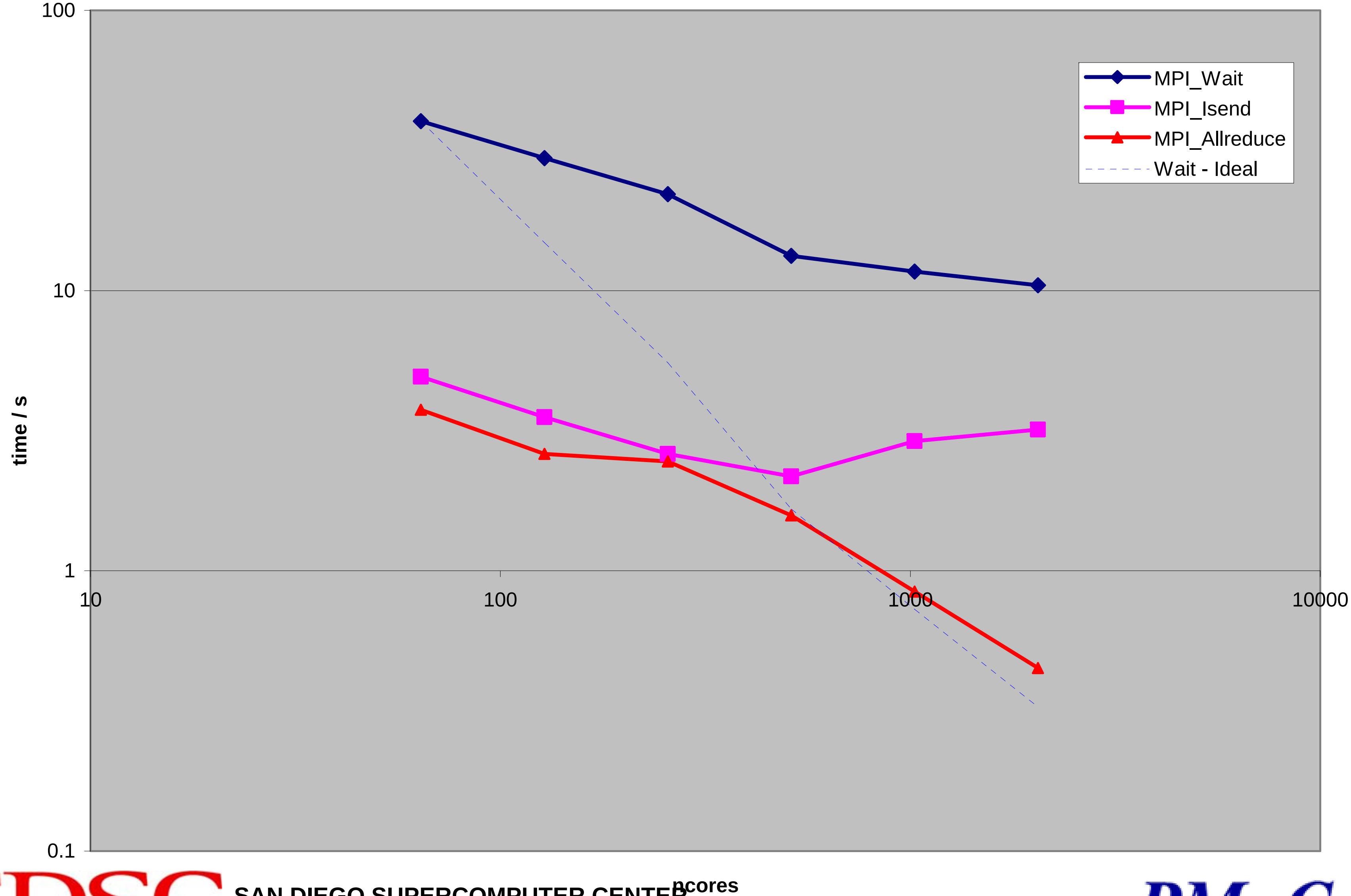
MILC – Perfect Scalability due to Cancellation of Effects



MILC – Superlinear Speedup Cache Effect

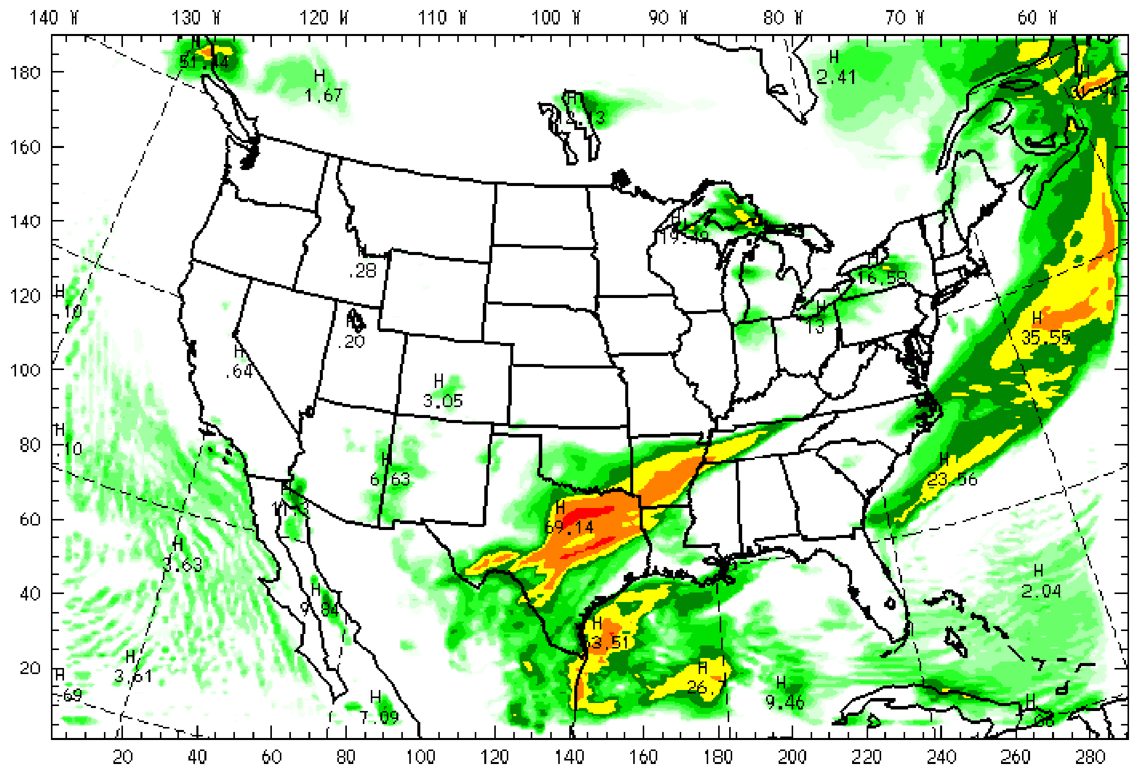


MILC Communication

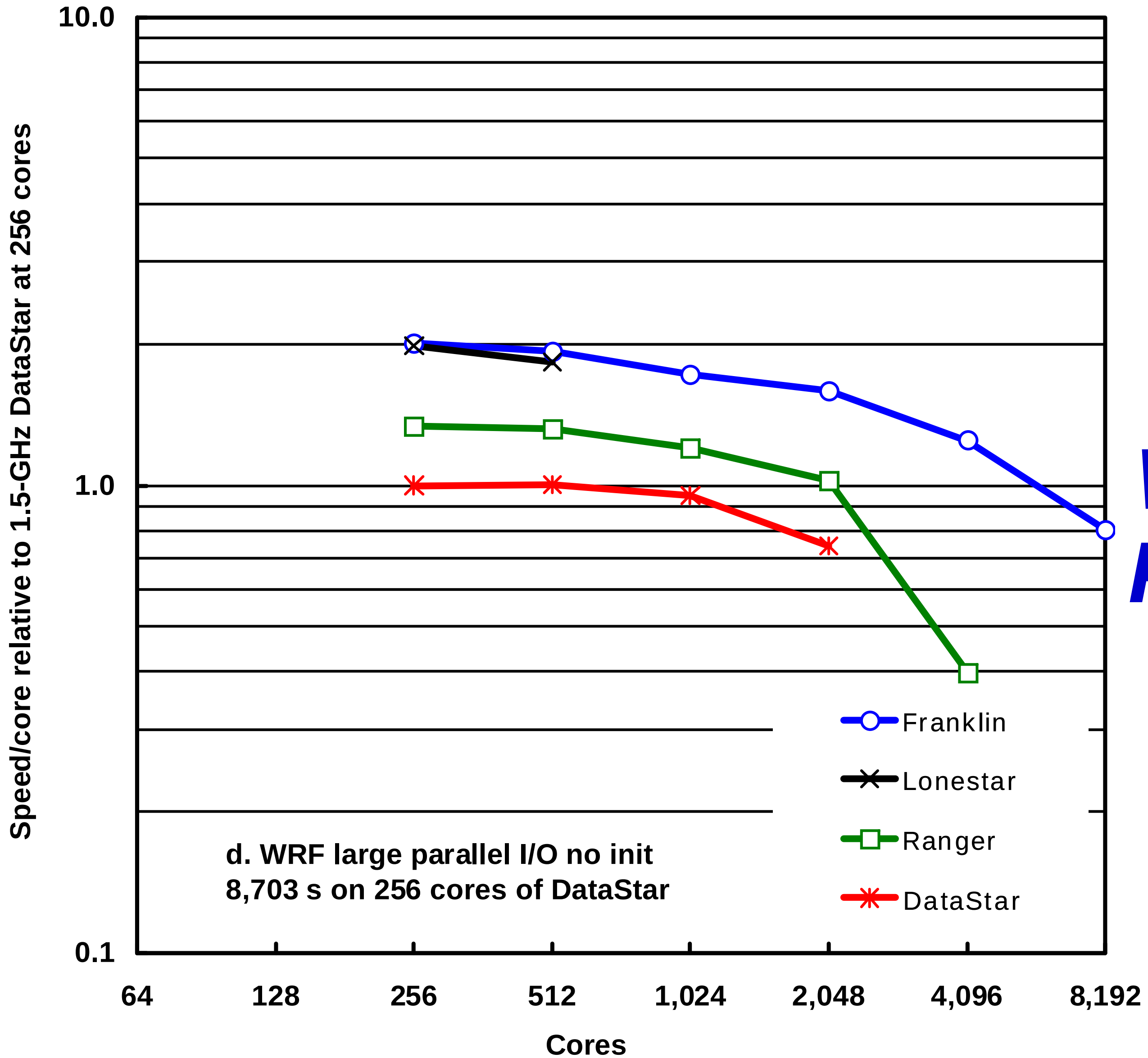


WRF – Problem Definition

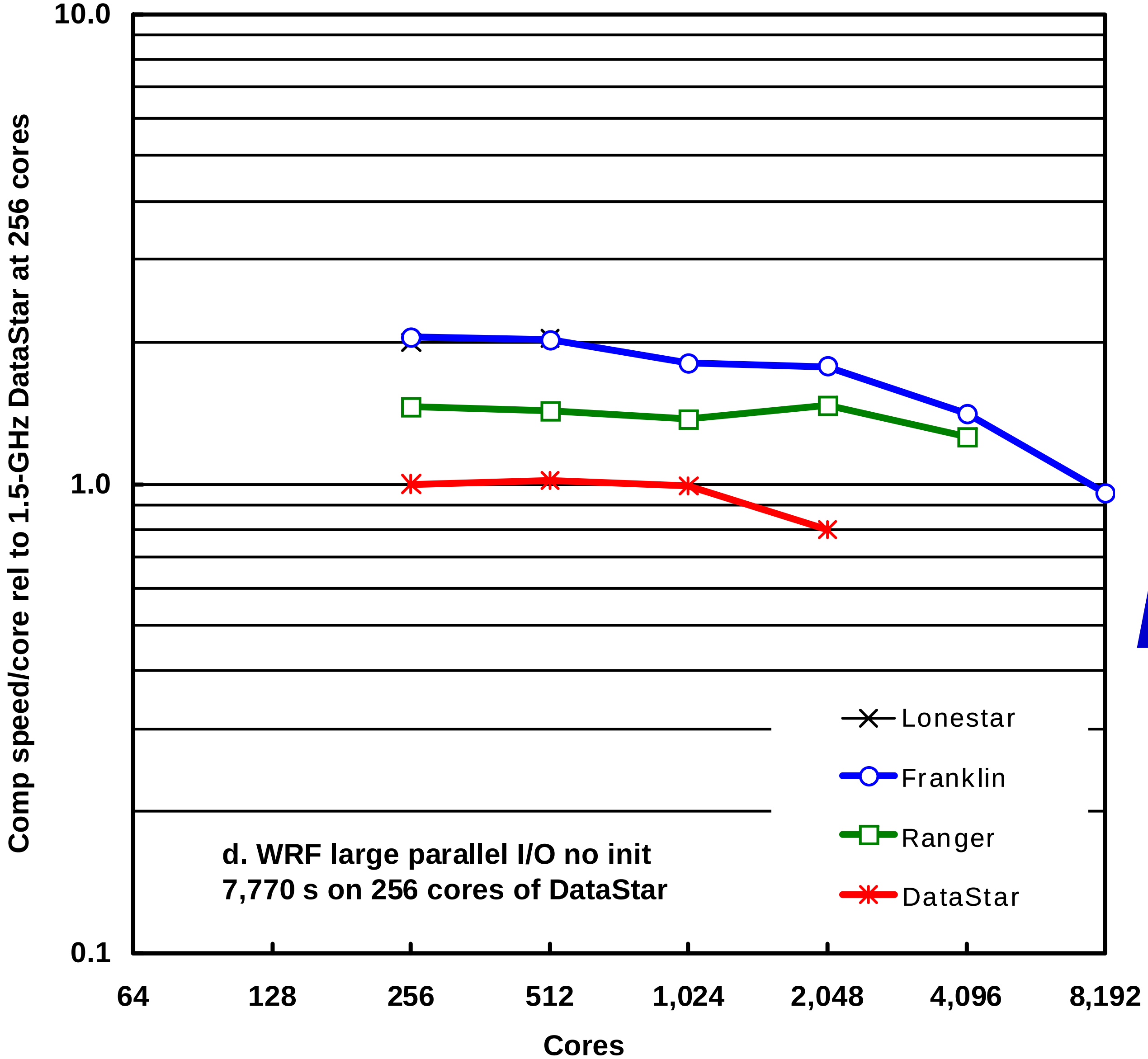
- **WRF – 3D numerical weather prediction**
- **Explicit Rugga-Kutta solver in 2 dimensions**
- **Grid is spatially decomposed in X & Y**
- **Version 2.1.2**
- **2.5 km Continental US 1501 x 1201 x 35 grid**
- **9 simulated hours**
- **parallel I/O turned on**

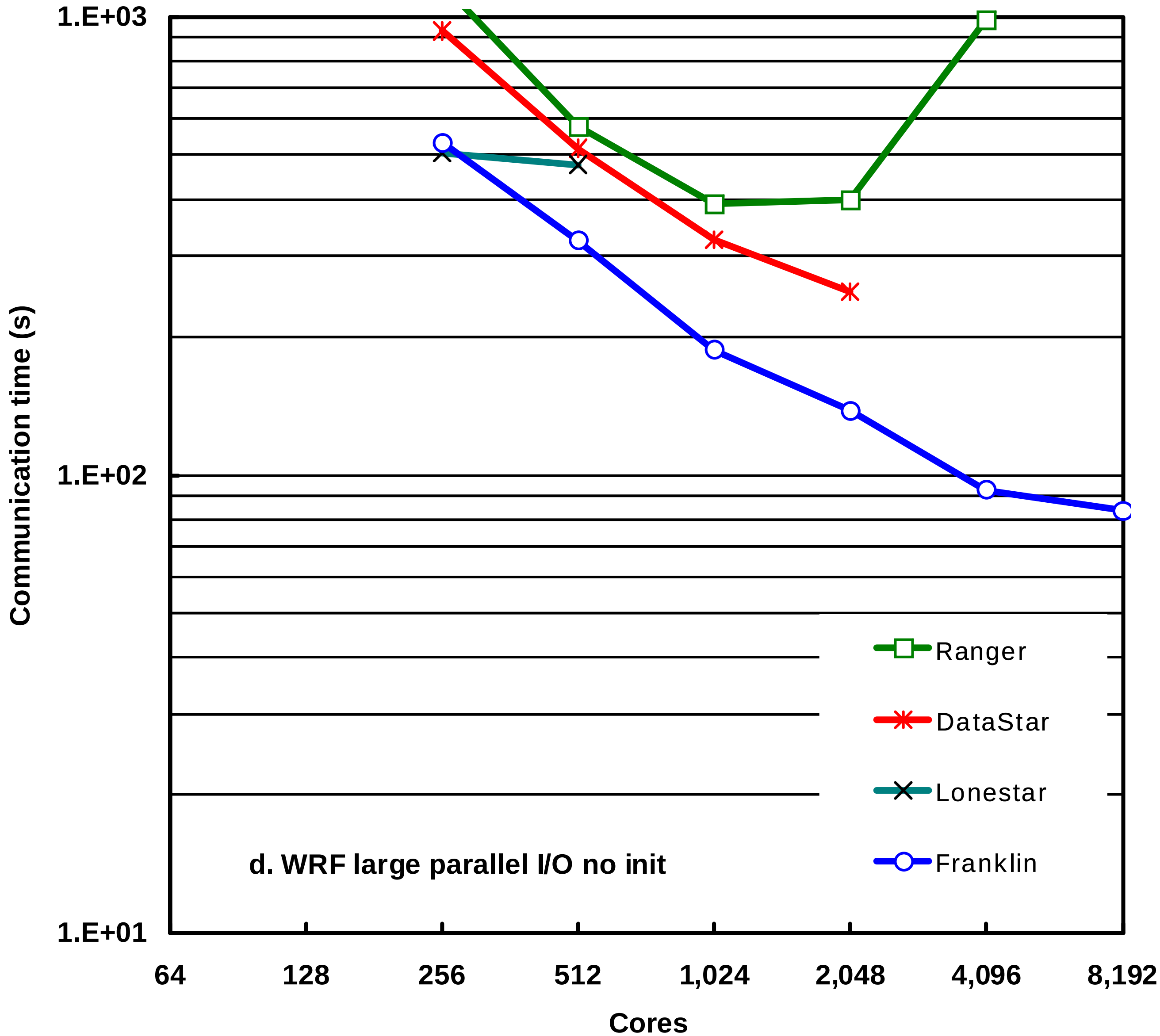


WRF Overall Performance



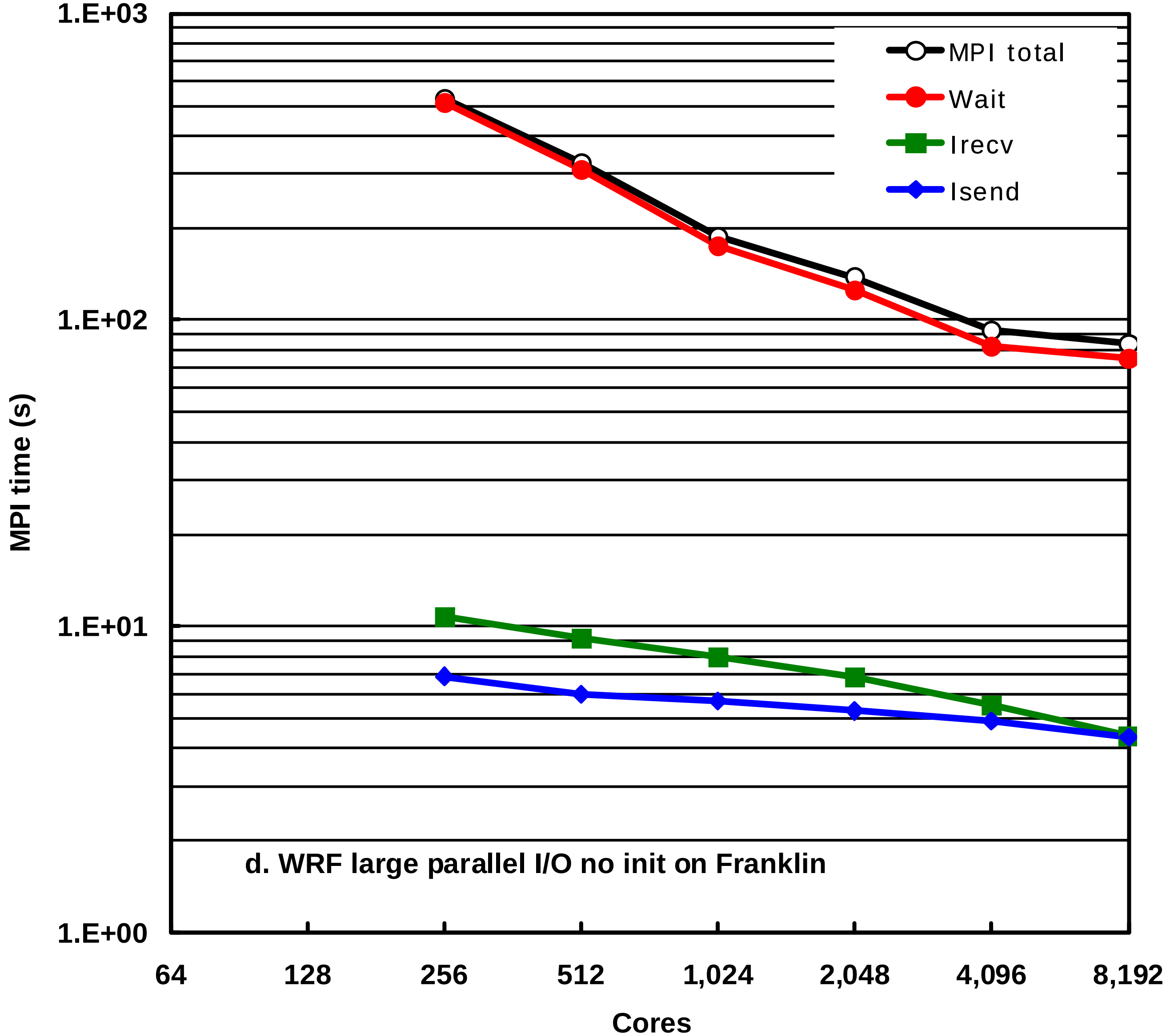
WRF-Compute Performance



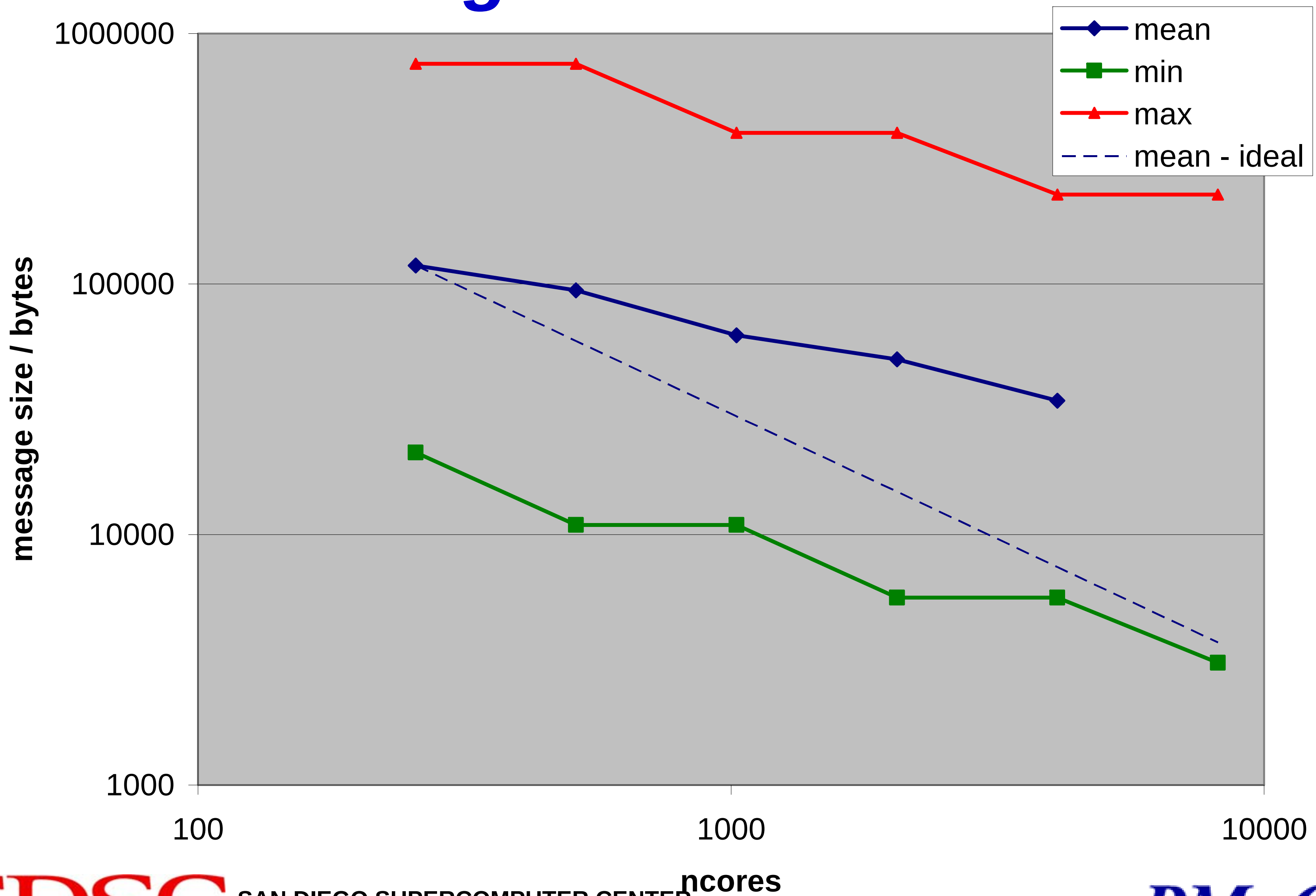


WRF Communication times

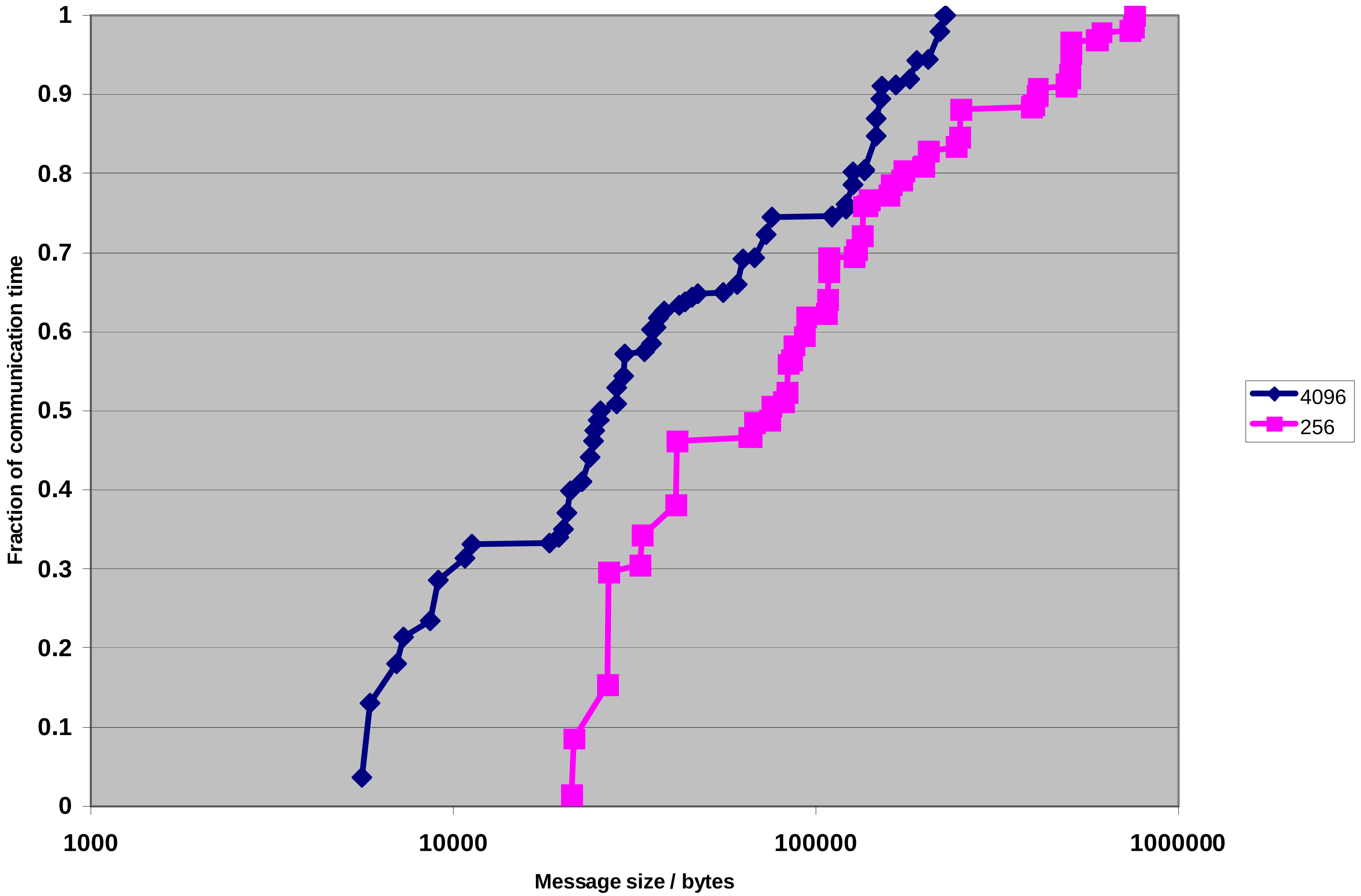
WRF - MPI Breakdown



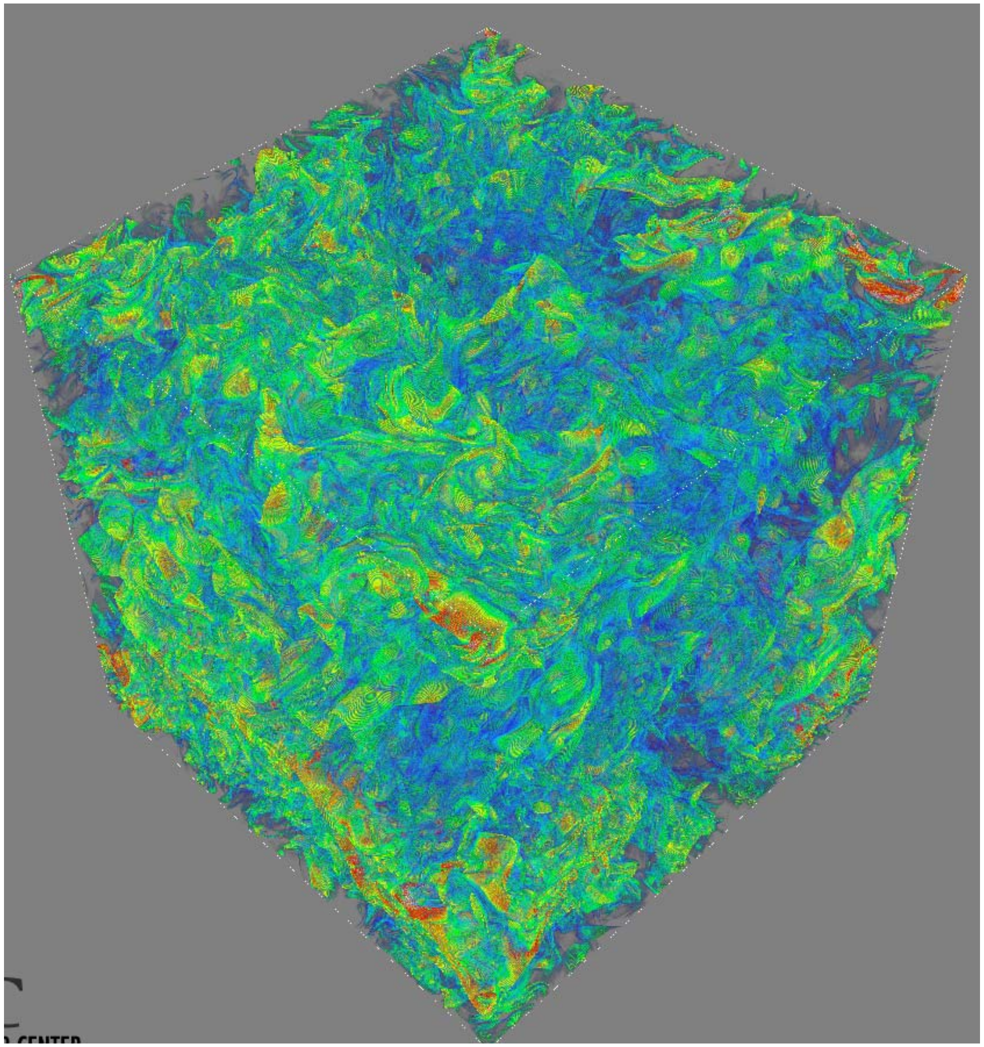
WRF – Message Sizes Decrease Slowly



WRF – Latency and Bandwidth Dependence



Direct Numerical Simulation (DNS)

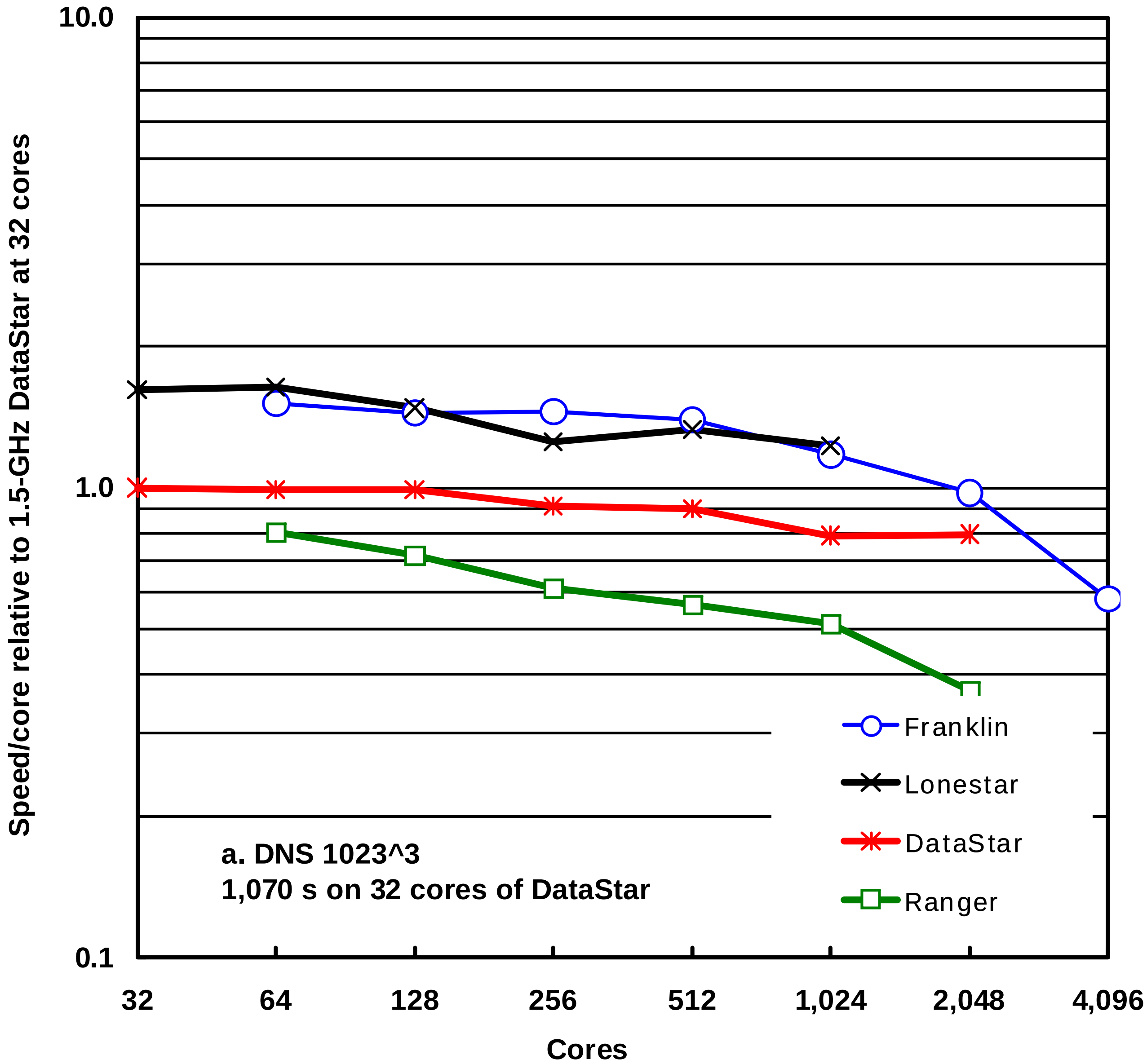


Direct Numerical Simulation of turbulent flows

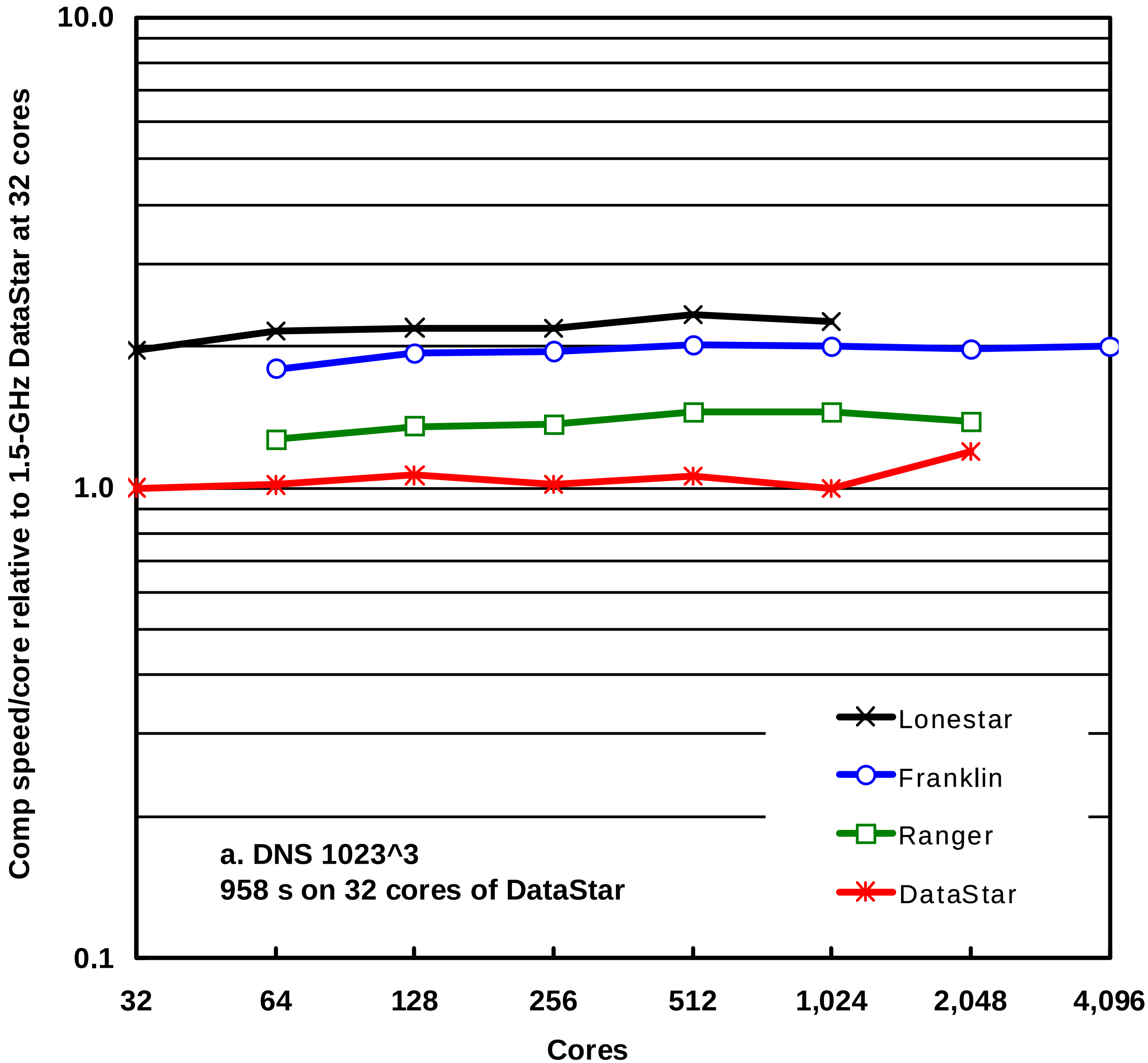
Uses pseudospectral method - 3D FFT's

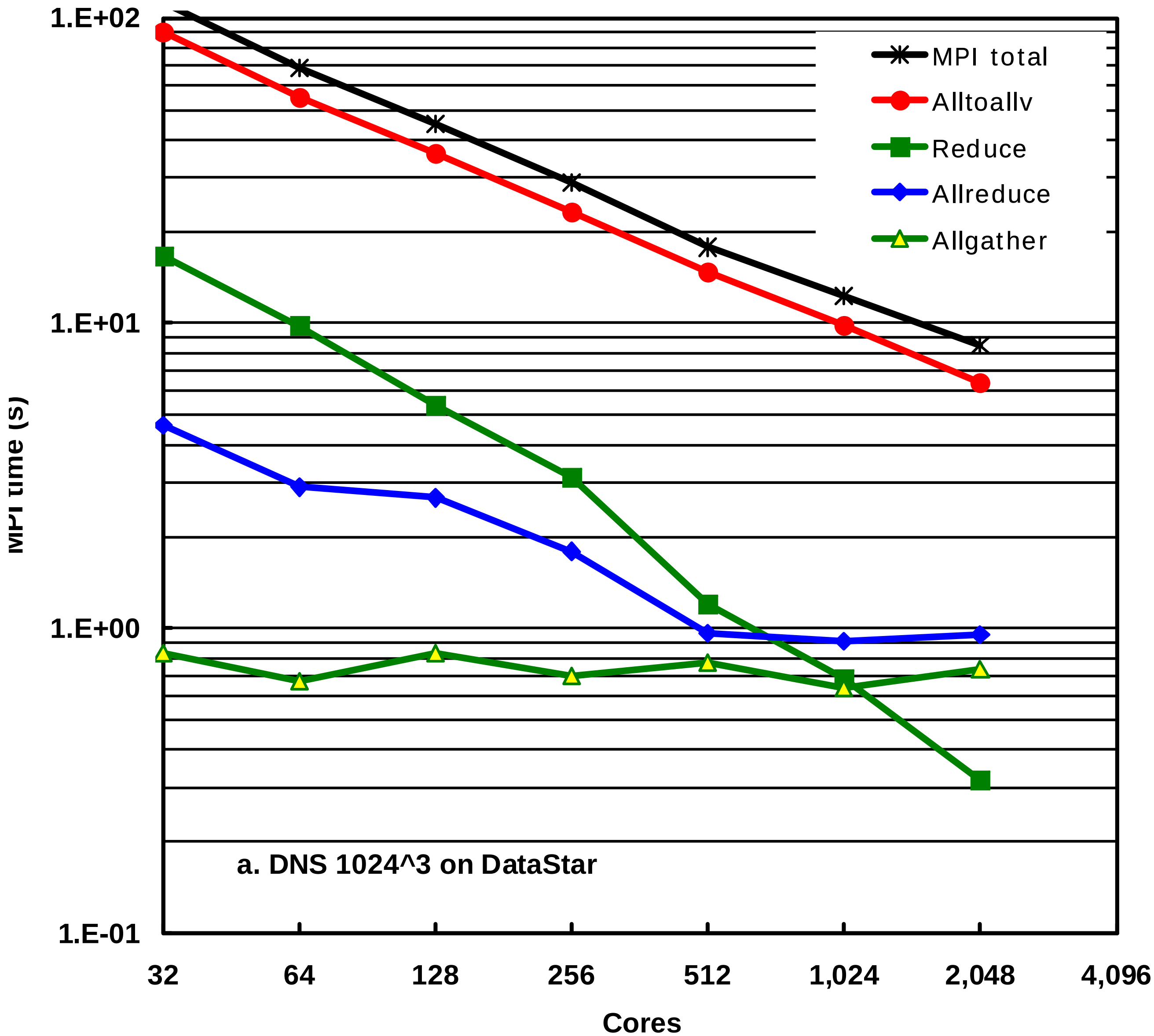
1024^3 problem – 10 timesteps

DNS – Overall Performance

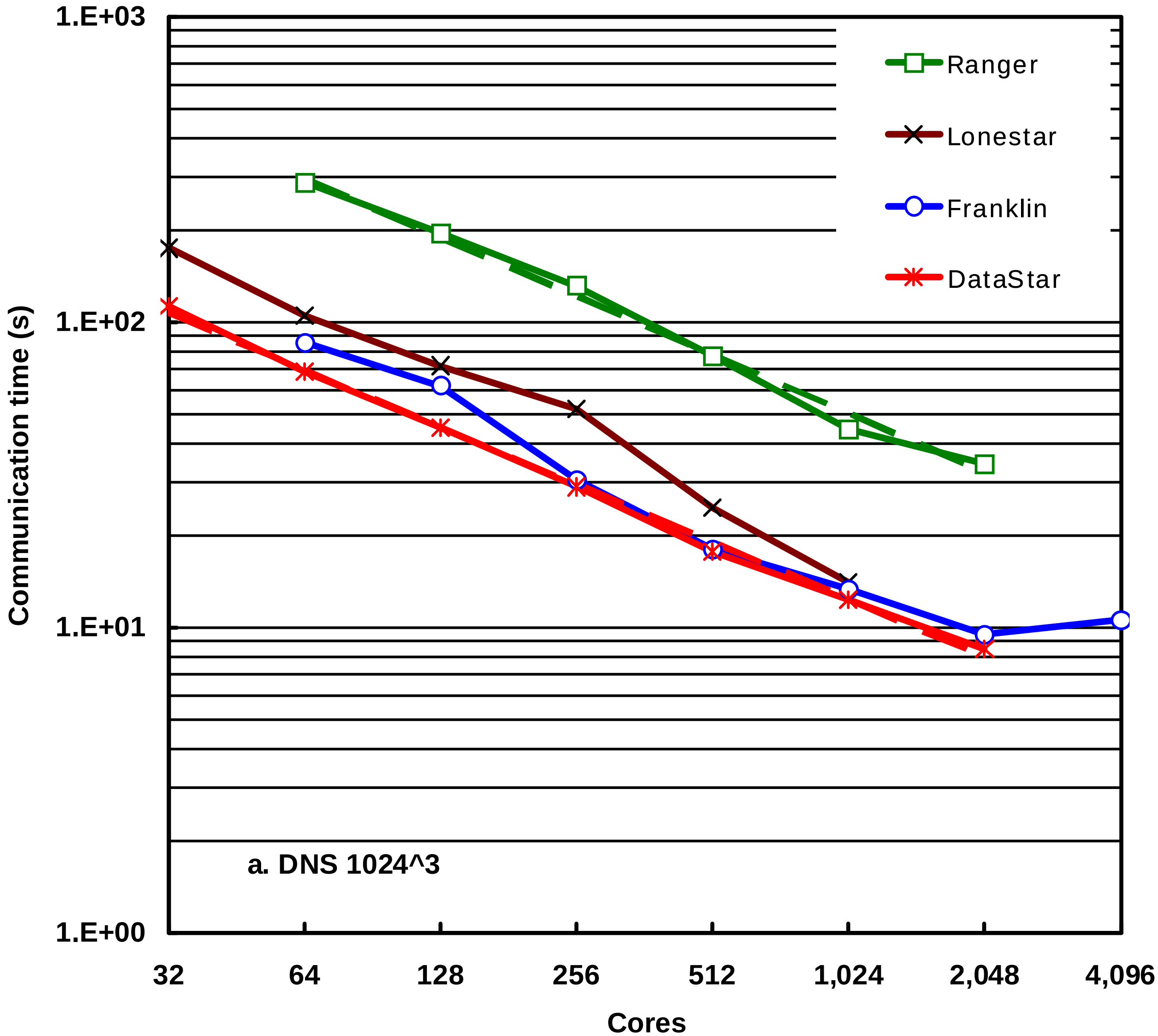


DNS - Compute Performance





DNS – MPI Breakdown



DNS communication time

Theory: $1/P^{0.67}$

Measured $1/P^{0.62-0.71}$

Overlapping Computation and Communication

MPI_ISend()

MPI_IRecv()

some_code()

MPI_Wait()

- **Basic idea – make the time in MPI_Wait goto zero**
- **In practice very hard to achieve**

More Advance Usage: Regions

Uses MPI_Pcontrol Interface

The first argument to MPI_Pcontrol determines what action will be taken by IPM.

Arguments	Description
1,"label"	start code region "label"
-1,"label"	exit code region "label"

Defining code regions and events:

C	FORTRAN
MPI_Pcontrol(1,"proc_a");	call mpi_pcontrol(1,"proc_a"//char(0))
MPI_Pcontrol(-1,"proc_a");	call mpi_pcontrol(-1,"proc_a"//char(0))

More Advanced Usage: Chip Counters –

AMD (Ranger & Kraken) Intel(Abe & Lonestar)

- **Default set:**

  PAPI_FP_OPS

  PAPI_TOT_CYC

  PAPI_VEC_INS

  PAPI_TOT_INS
- **Alternative (setenv IPM_HPM 2)**

  PAPI_L1_DCM

  PAPI_L1_DCA

  PAPI_L2_DCM

  PAPI_L2_DCA

- **Default set:**

  PAPI_FP_OPS

  PAPI_TOT_CYC
- **Alternative (setenv IPM_HPM)**

  2 PAPI_TOT_IIS, PAPI_TOT_INS

  3 PAPI_TOT_IIS, PAPI_TOT_INS

  4 PAPI_FML_INS, PAPI_FDV_INS

User defined counters also possible – setenv IPM_HPM PAPI_FP_OPS PAPI_TOT_CYC,…

User is responsible for choosing a valid set

See PAPI documentation and papi_avail command for more information

Matvec: Regions & Cache Misses

- **What is wrong with this fortran code ?**

```
…
call mpi_pcontrol(1,"main"//char(0))
do i = 1,natom
      sum=0.0d0
      do j = 1, natom
         sum=sum+coords(i,j)*q(j)
       end do
        p(i)=sum
end do
 call mpi_pcontrol(-1,"main"//char(0))
…
```

setenv IPM_HPM 2

Regions and Cache Misses cont.

```
…
##############################################################################
# region  : main       [ntasks] =      1
#
#                            [total]         <avg>           min           max
# entries                          1             1             1             1
# wallclock                0.0185561     0.0185561     0.0185561     0.0185561
# user                      0.016001      0.016001      0.016001      0.016001
# system                           0             0             0             0
# mpi                              0             0             0             0
# %comm                                          0             0             0
# gflop/sec                0.0190196     0.0190196     0.0190196     0.0190196
#
# PAPI_L1_DCM                 352929        352929        352929        352929
# PAPI_L1_DCA            8.01278e+06   8.01278e+06   8.01278e+06   8.01278e+06
# PAPI_L2_DCM                 126097        126097        126097        126097
# PAPI_L2_DCA                 461965        461965        461965        461965
#
###############################################################################
```

27% cache misses !

Matvec: Regions & Cache Misses - 3

- **What is wrong with this fortran code ?**

```
…
do i = 1,natom
      sum=0.0d0
      do j = 1, natom
        sum=sum+coords(i,j)*q(j)
       end do
        p(i)=sum
end do
…
```

Indices transposed!

Regions and Cache Misses - 4

```
##############################################################################
# region  : main       [ntasks] =      1
#
#                          [total]          <avg>          min          max
# entries                        1              1            1            1
# wallclock             0.00727696     0.00727696   0.00727696   0.00727696
# user                       0.008          0.008        0.008        0.008
# system                         0              0            0            0
# mpi                            0              0            0            0
# %comm                                         0            0            0
# gflop/sec            0.000636804    0.000636804  0.000636804  0.000636804
#
# PAPI_L1_DCM                 4634           4634         4634         4634
# PAPI_L1_DCA          8.01436e+06    8.01436e+06  8.01436e+06  8.01436e+06
# PAPI_L2_DCM                 4609           4609         4609         4609
# PAPI_L2_DCA               126108         126108       126108       126108
#
###############################################################################
```

3.6% cache misses – Problem solved - Runtime doubled !

Using IPM on Ranger – 1 Running

- **In submission script:**
- **(csh syntax)**

module load ipm

setenv LD_PRELOAD $TACC_IPM_LIB/libipm.so

ibrun ./a.out

- **(bash syntax)**

module load ipm

export LD_PRELOAD=$TACC_IPM_LIB/libipm.so

ibrun ./a.out

Using IPM on Ranger – 2 Postprocessing

- **Text summary should be in stdout**
- **IPM also generates an XML file (username.1235798913.129844.0) that can be parsed to produce webpage**

  module load ipm
  ipm_parse -html tg456671.1235798913.129844.0
- **This generates a directory with the html content in**

  tar zxvf ipmoutput.tgz <directory> eg. a.out_2_tg456671…

  scp tar file to your local machine; untar and view with your favorite browser

Summary

- **Understanding the performance characteristics of your code is essential for good performance**
- **IPM is a lightweight, easy-to-use profiling interface (with very low overhead <2%).**
- **It can provide information on**
  - An individual jobs performance characteristics
  - Comparison between jobs
  - Workload characterization
- **IPM allows you to gain a basic understanding of *why* your code performs the way it does.**
- **IPM is installed on various TG machines: Ranger, BigBen, Pople, (Abe, Kraken) see instructions on IPM website http://ipm-hpc.sf.net**