Parallel Performance and Optimization

Gregory G. Howes
Department of Physics and Astronomy
University of Iowa

PHYS 5905: Numerical Simulation of Plasmas
Department of Physics and Astronomy
University of Iowa
Spring 2019

Thank you

THE UNIVERSITY OF IOWA

Ben Rogers	Information Technology Services
Glenn Johnson	Information Technology Services
Mary Grabe	Information Technology Services
Sai Ramadugu	Information Technology Services
Brenna Miller	Information Technology Services
Tino Kaltsis	Information Technology Services
Ben Rothman	Information Technology Services

and

National Science Foundation

Outline

- General Comments on Optimization
- Measures of Parallel Performance
- Debugging Parallel Programs
- Profiling and Optimization

General Comments on Optimization

- Always keep in mind the Rule of Thumb

  Computation is FAST Communication is SLOW

- If you can do extra work in an initialization step to reduce the work done in each timestep, it is generally well worth the effort

- Collect all of your communication at one point in your program timestep

Measures of Parallel Performance

THE UNIVERSITY OF IOWA

- When you apply for time on a supercomputer, it is critical to provide quantitative data on the parallel performance of your application

- Algorithm vs. Parallel Scaling
  - Algorithm scaling measures the increased computational time as the size of computation is increased
  - Parallel scaling measures the decrease in wallclock time as more processors are used for the calculation

- Common measures of parallel scaling
  - Strong Scaling: Time for fixed problem size as number of processors is increased
  - Weak Scaling: Time for fixed computational work per processor as problem size is increased

Strong Scaling

THE UNIVERSITY OF IOWA

- Measure time for fixed problem size as number of processors increases

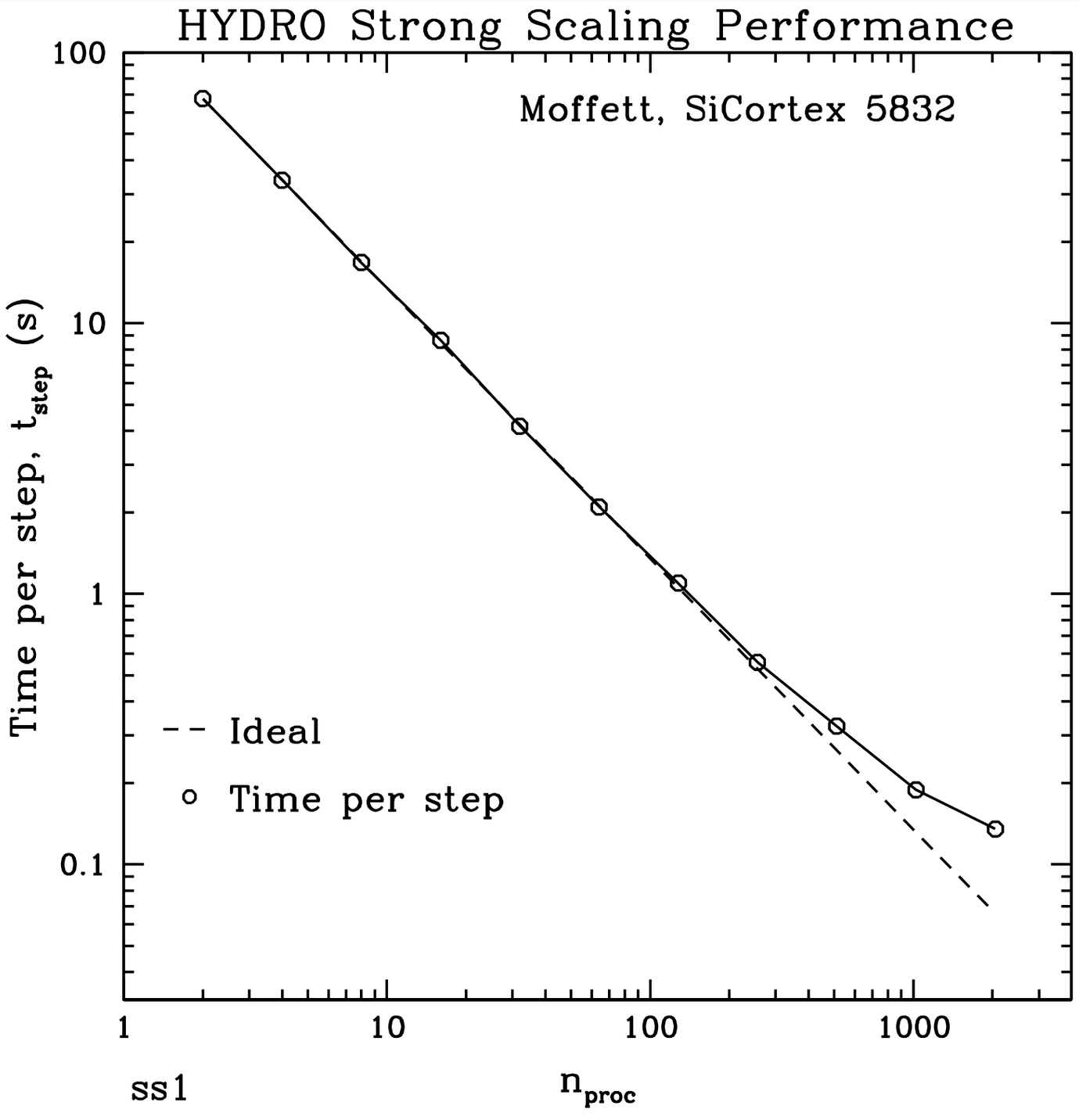
- To get the best strong scaling, you want to find the largest problem size that will fit on one processor

- Eventually, all but embarrassingly parallel applications will lead to a turnover in the strong scaling plot:
  - -As number of processors increases, computational work per processor decreases, and communication time typically increases.
  - -This reduces the granularity (time for local computation vs. time for communication), and generally degrades parallel performance

- To get an impressive strong scaling curve, it will often take some experimentation to identify parameters that allow ideal performance over the widest range of processor number.

- Multi-core processors often lead to some complications in strong scaling behaviors due to bandwidth limitations of memory access.

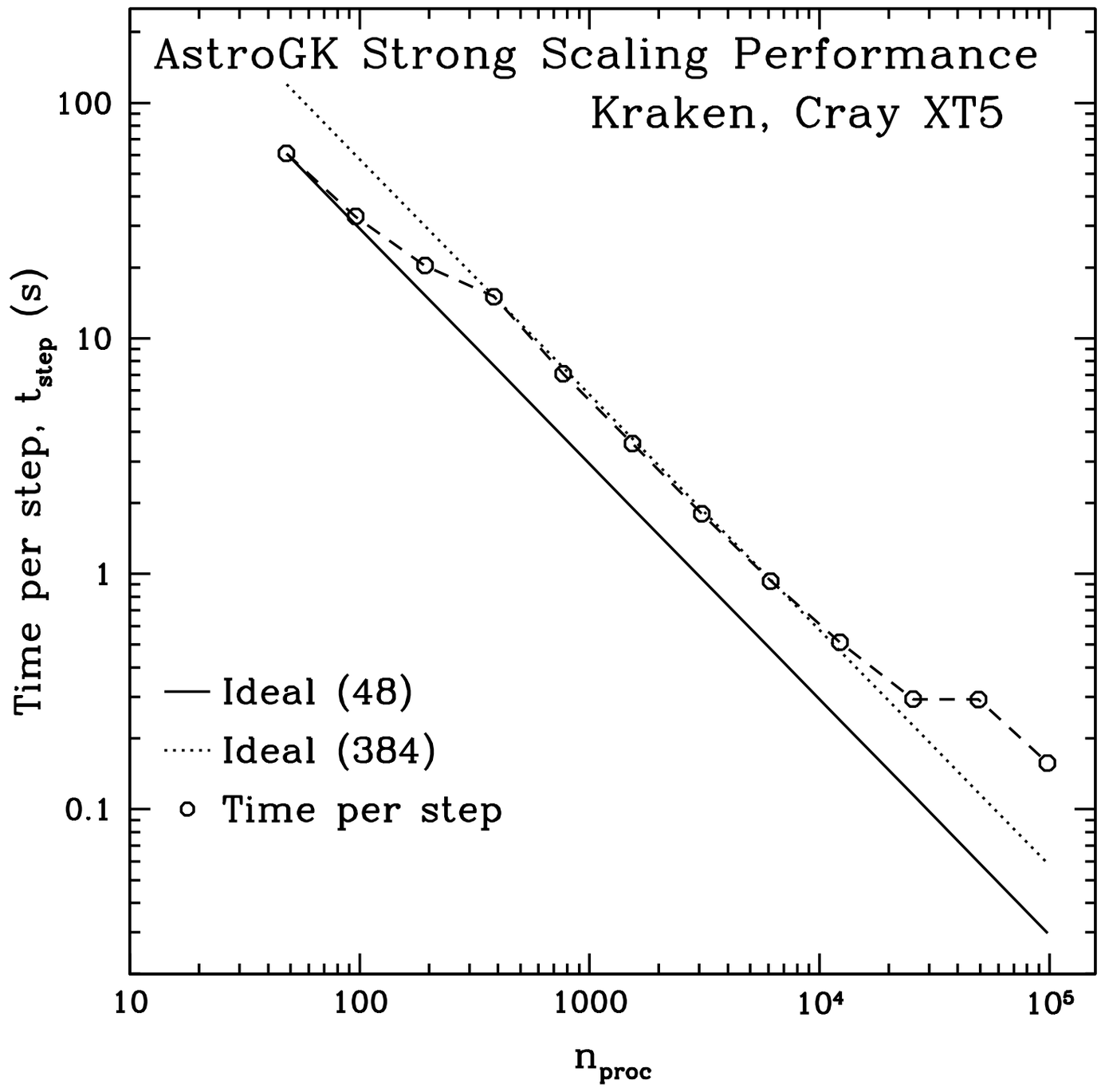
Strong Scaling for **HYDRO**



- Note: For ideal behavior, computational time is inversely proportional to the number of processors.

Strong Scaling for **AstroGK**

THE UNIVERSITY OF IOWA



- Note: Kraken (Cray XT5) nodes have dual hex-core processors, so performance degrades as more cores/node are used due to memory bandwidth limitations

Weak Scaling

THE UNIVERSITY OF IOWA

- Measure time for fixed computational work per processor as problem size is increased

- Again, you want to find the largest problem size that will fit on one processor

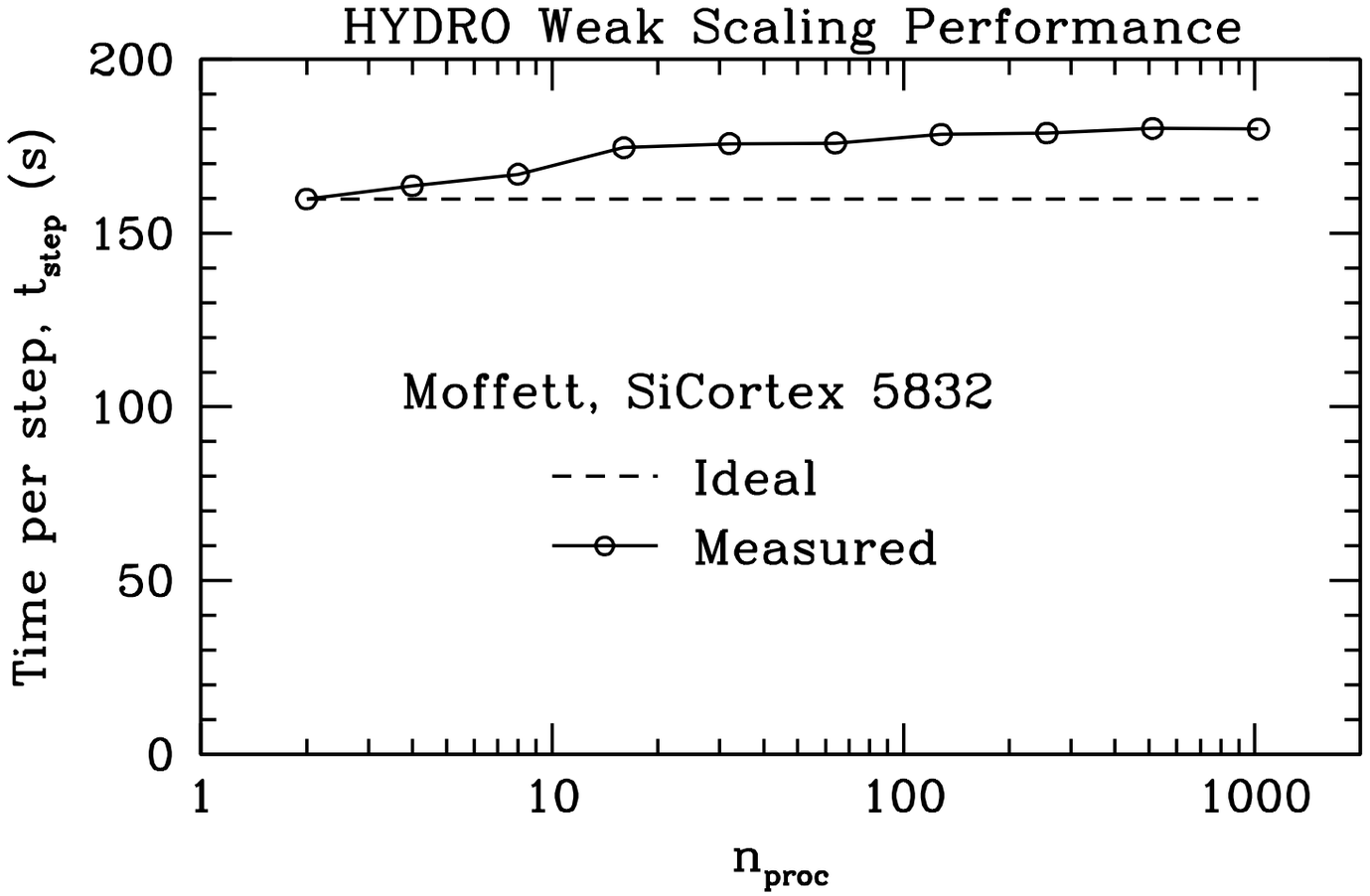
- It is usually easier to get a good weak scaling than a good strong scaling
  -Since computational work per processor is constant, granularity only decreases due to increased communication time

- Since the total problem size must increase, one has to choose how to increase it.
  - Ex: In **`HYDRO`**, you can increase either nx or ny
  - Often weak scaling performance will depend on which choice you make

- Again, some exploration of parameters may be necessary to yield the most impressive weak scaling curve

Weak Scaling for **HYDRO**

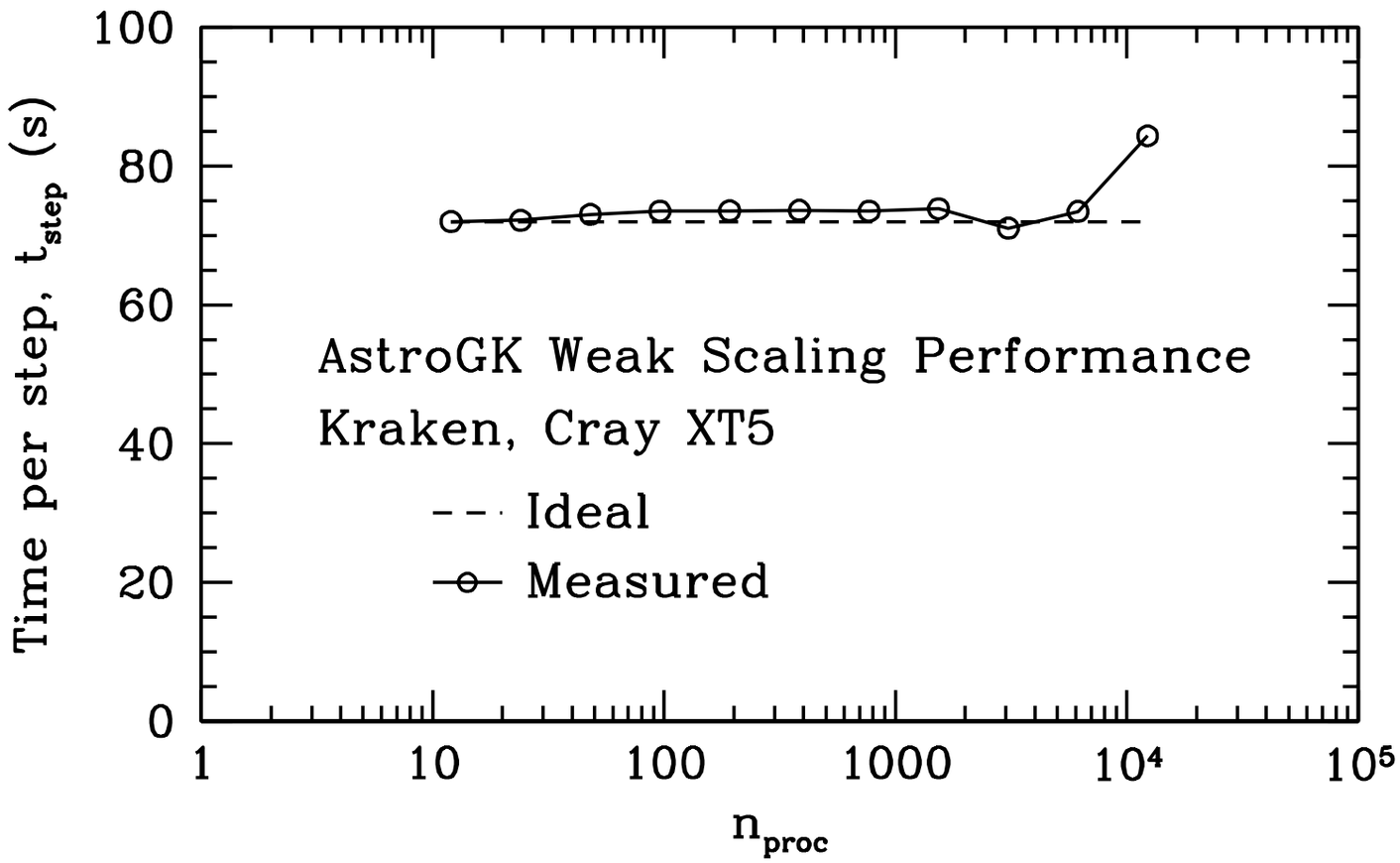
THE UNIVERSITY OF IOWA



- Note: For ideal behavior, computational time should remain constant.

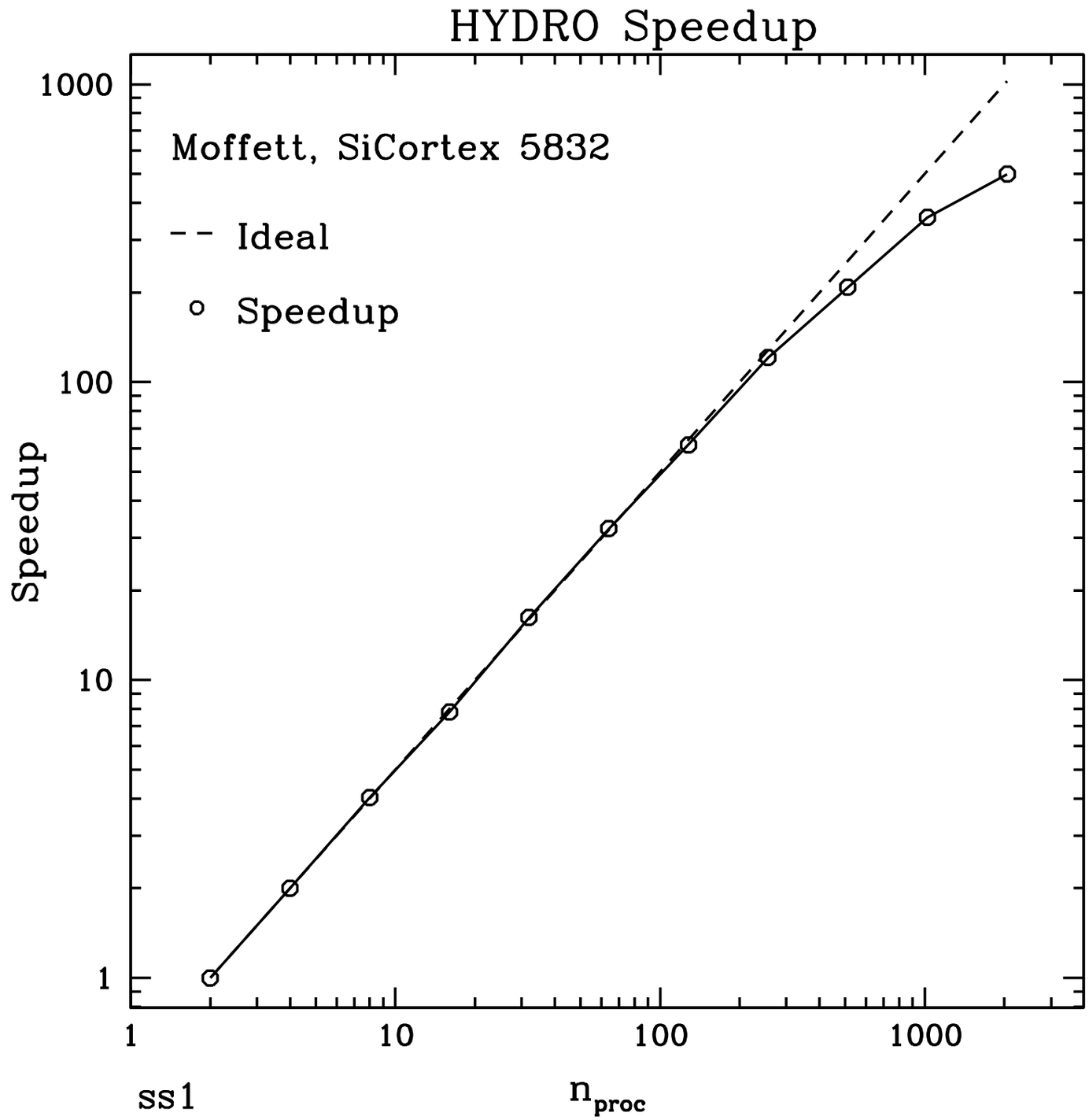
Weak Scaling for **AstroGK**

THE UNIVERSITY OF IOWA

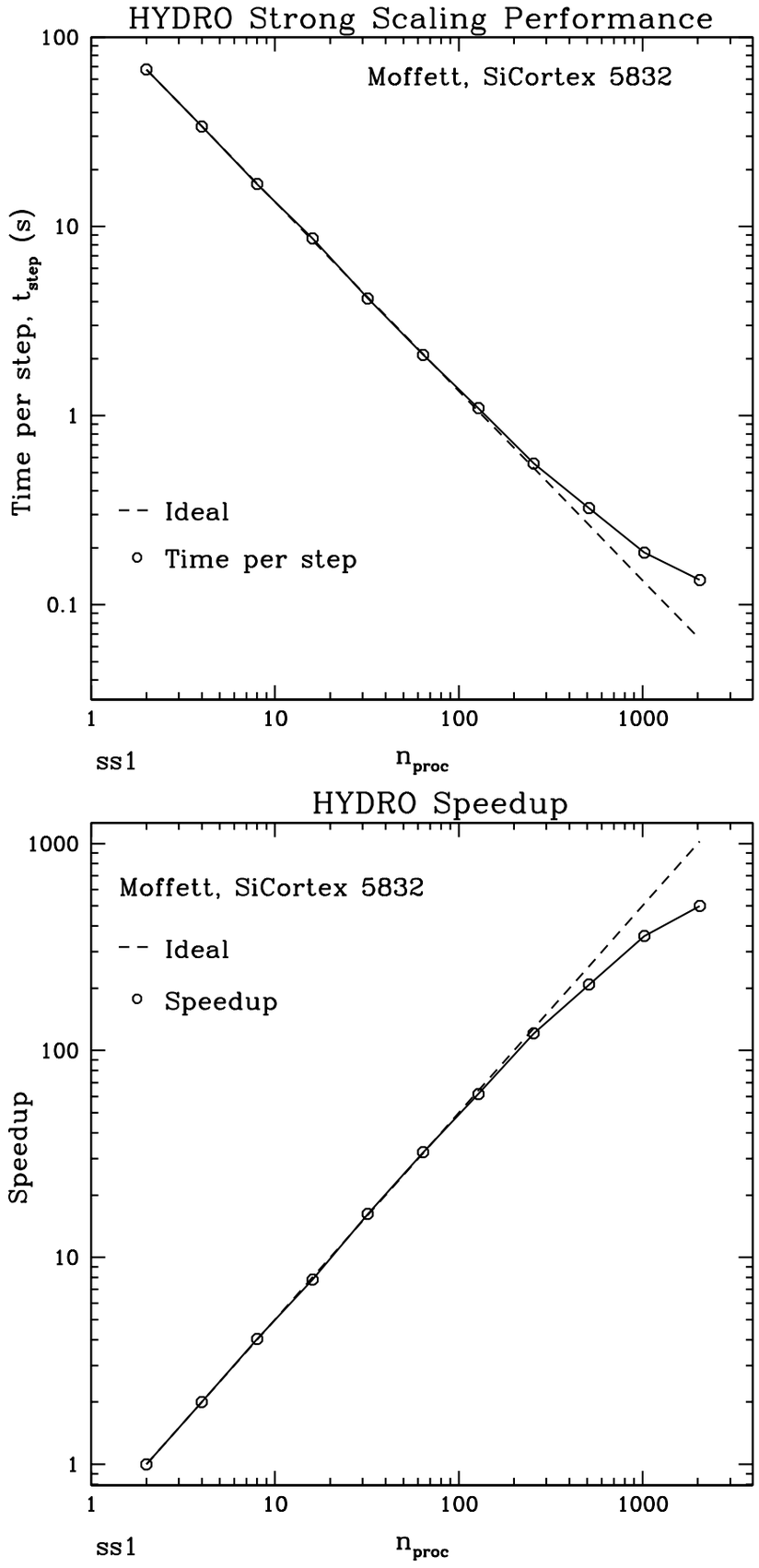


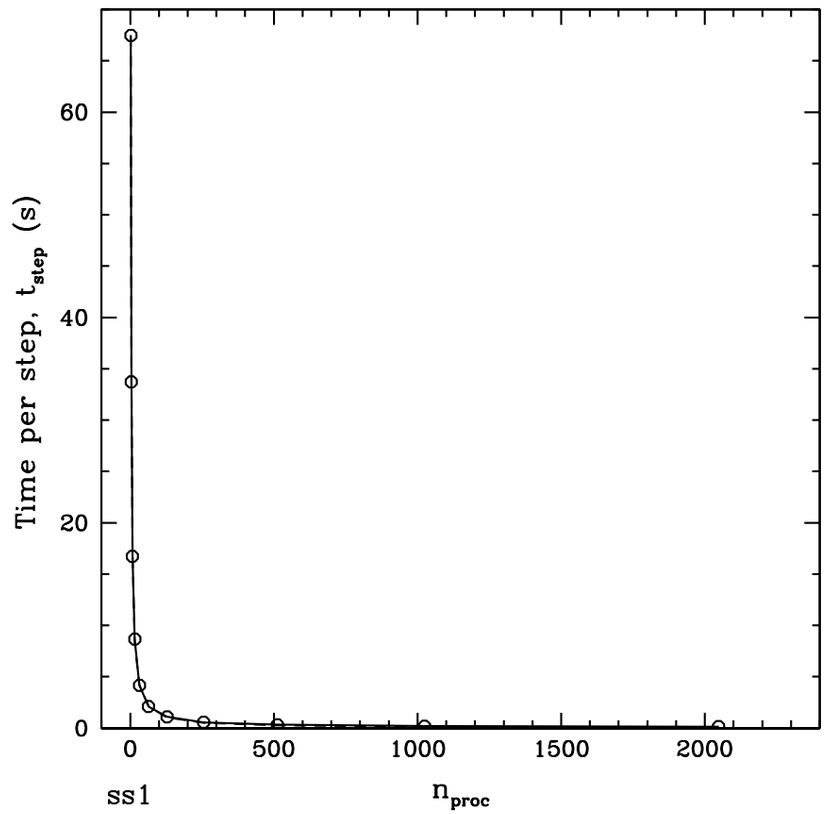
Speedup

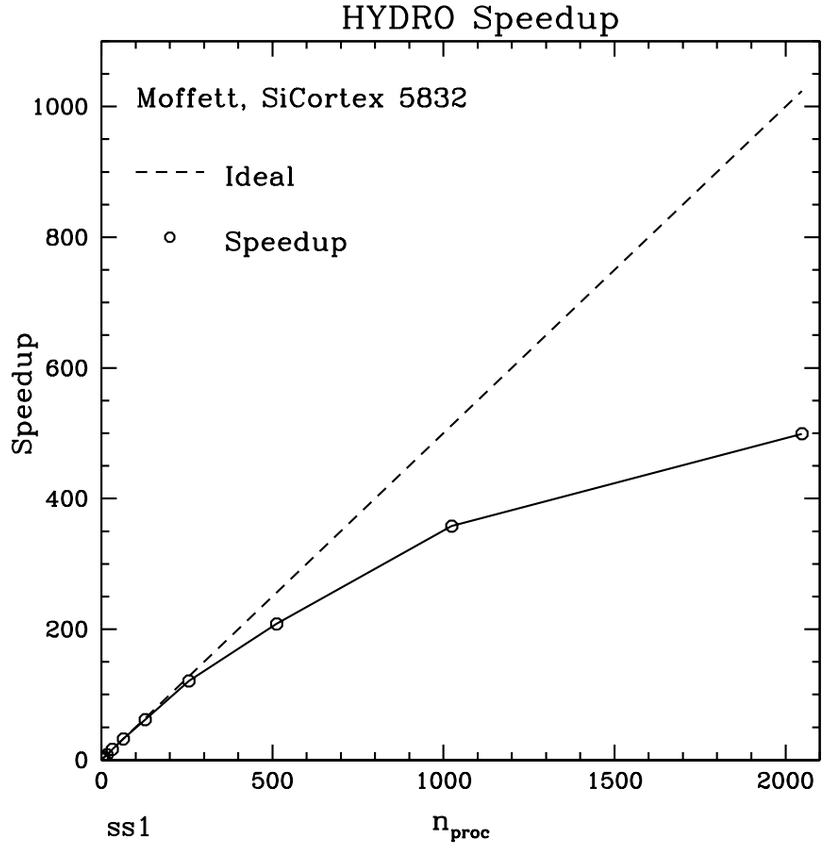
- Another measure is the speedup, $S = \dfrac{\text{Time on 1 processor}}{\text{Time on } N \text{ processors}}$



Linear vs. Logarithmic Scaling







Additional Notes

THE UNIVERSITY OF IOWA

- You need to choose what time you will use for the scaling tests:
  -Do you want to include or exclude initialization time?

- Be sure that your code does not write to the disk or the screen at any time during the scaling test (turn off output if possible), as this will lead to degraded performance.

Outline

- General Comments on Optimization
- Measures of Parallel Performance
- Debugging Parallel Programs
- Profiling and Optimization

Parallel Debuggers

THE UNIVERSITY OF IOWA

• One can always debug by hand (inserting lines to write out output as the code progresses).

-Generally the easiest approach

-Time consuming

-Difficult to debug problems particular to parallel codes, for example race conditions.

• In addition to serial debuggers to find errors in your source, such as gdb, a valuable tool for parallel programming is the use of parallel debuggers.

• Common parallel debuggers are TotalView and DDT (Distributed Debugging Tool)

• Parallel debuggers treat all tasks in an MPI job simultaneously, giving the user a view of the synchronization of different MPI tasks

- This enables the identification of race conditions and other problems that are otherwise difficult to identify.

• Running out of memory is common problem in parallel applications

Outline

- General Comments on Optimization
- Measures of Parallel Performance
- Debugging Parallel Programs
- Optimization and Profiling

Code Optimization

General approaches to code optimization:

- Automatic Optimization at compile time (`mpif90 -O3`)
  -Level 3 optimization may produce incorrect results, so be careful

- Use libraries of optimized routines for common mathematical operations
  - BLAS and LAPACK for matrix computations
  - FFTW for Fast Fourier Transforms
  -Intel's Math Kernel Library (MKL) has routines optimized for particular architectures

- By hand, ensure innermost loops do no unnecessary computation

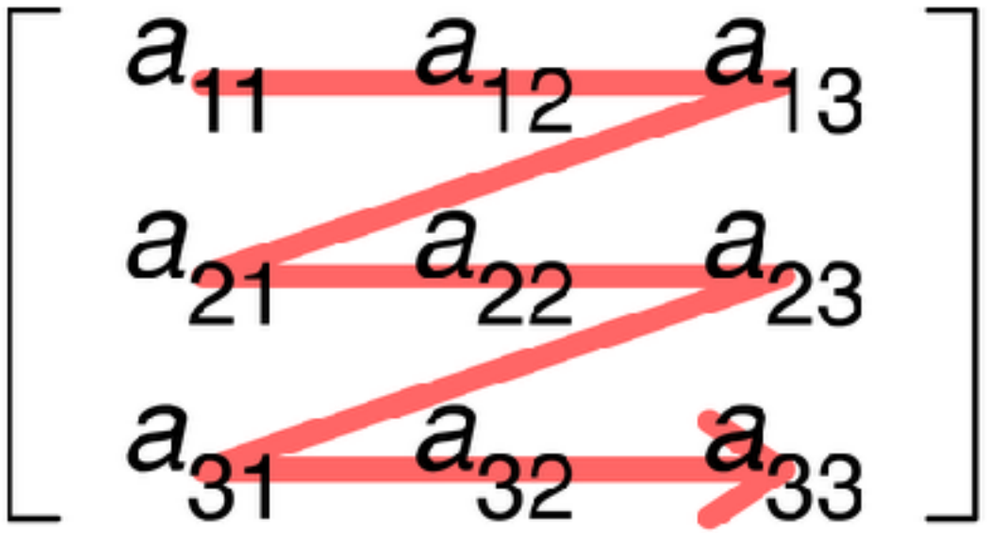
- Profiling:
  -Measuring the performance of the running code to generate a profile or trace file
  -Although it does introduce some overhead, it is a good way to measure code performance until typical running conditions

Array Element Ordering

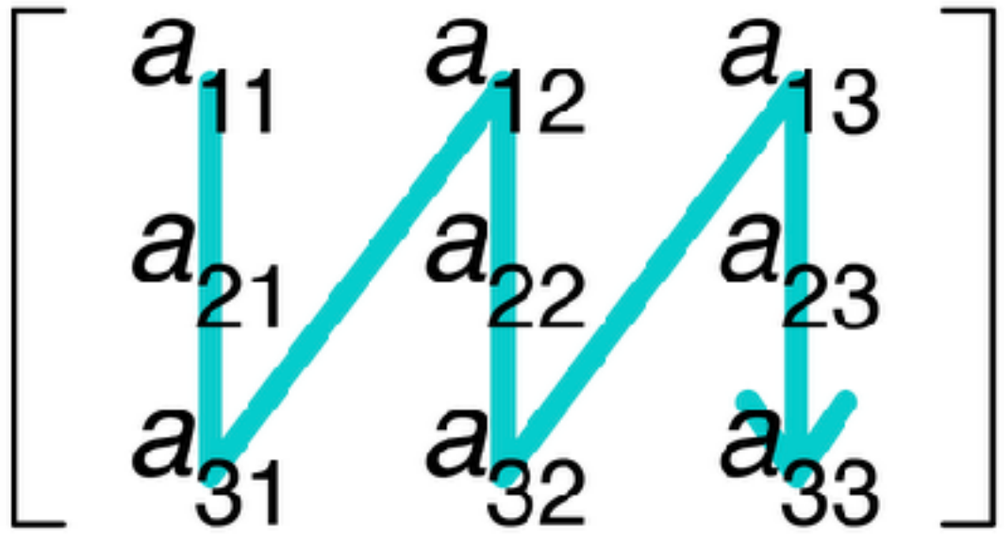
THE UNIVERSITY OF IOWA

• Array Elements are stored linearly in memory, and designing your code optimally requires taking this into account

Row-major order



Column-major order



- Row Major Languages: C/C++, NumPy in Python
- Column Major Languages: Fortran, MATLAB
- Neither Row nor Column Major: Java (ordered by age), Python (lists of lists)

Array Element Ordering

- Linearly memory storage obviously extends to arrays of dimension >2
- In Fortran, leftmost subscript varies most rapidly

Two-Dimensional Array BAN (3,4)

1	BAN(1,1)	4	BAN(1,2)	7	BAN(1,3)	10	BAN(1,4)
2	BAN(2,1)	5	BAN(2,2)	8	BAN(2,3)	11	BAN(2,4)
3	BAN(3,1)	6	BAN(3,2)	9	BAN(3,3)	12	BAN(3,4)

Memory Positions

Three-Dimensional Array BOS (3,3,3)

19	BOS(1,1,3)	22	BOS(1,2,3)	25	BOS(1,3,3)		
20	BOS(2,1,3)	23	BOS(2,2,3)	26	BOS(2,3,3)		
10	BOS(1,1,2)	13	BOS(1,2,2)	16	BOS(1,3,2)	27	BOS(3,3,3)
11	BOS(2,1,2)	14	BOS(2,2,2)	17	BOS(2,3,2)		
1	BOS(1,1,1)	4	BOS(1,2,1)	7	BOS(1,3,1)	18	BOS(3,3,2)
2	BOS(2,1,1)	5	BOS(2,2,1)	8	BOS(2,3,1)		
3	BOS(3,1,1)	6	BOS(3,2,1)	9	BOS(3,3,1)		

Memory Positions

Array Element Ordering

THE UNIVERSITY OF IOWA

- Why does this matter?
  - The CPU draws blocks of data needed for a computation into the cache
  - If the data needed is not within the cache, the data needs to be swapped
    - This is called a cache miss, and can dramatically slow down a code
    - This problem particularly arises for large problem size

Array Element Ordering

- Example (Fortran90):

```
n=1024
allocate(data(1:n,1:n,1:n))

do i=1,n
   do j=1,n
      do k=1,n
         data(i,j,k)=(sin(real(i*j*k))* &
            exp(real(k)/(real(i)*real(j))) +1.0 )**(1./3.)
      enddo
   enddo
enddo
```

In this ordering, rightmost index changes most rapidly (not optimal)

- Rightmost first: `(i,j,k)`

```
time hw12_left.e
 BEGIN: HW12 Simulation: Array Element Ordering: Lefttmost first
 END: HW12 Simulation
       92.27 real        88.21 user         3.38 sys
time hw12_right.e
 BEGIN: HW12 Simulation: Array Element Ordering: Righttmost first
 END: HW12 Simulation
      165.06 real       154.81 user         7.86 sys
```

- Leftmost first: `(k,j,i)`

Array Element Ordering

THE UNIVERSITY OF IOWA

- Optimizing Compilers will sometimes re-order calculation to make it faster
  - Previous example turned off optimization

    `ifort -O0`

  - Re-compiling the example with

    `ifort -O4`

```
time hw12_left.e
 BEGIN: HW12 Simulation: Array Element Ordering: Lefttmost first
 END: HW12 Simulation
       31.07 real         25.49 user          3.62 sys
time hw12_right.e
 BEGIN: HW12 Simulation: Array Element Ordering: Righttmost first
 END: HW12 Simulation
       30.85 real         27.90 user          2.74 sys
```

Profiling

THE UNIVERSITY OF IOWA

Profiling: Measure performance by collecting statistics of a running code

- Two methods for triggering when to collect data:

-Sampling
  -Triggered by timer interrupt or hardware counter overflow

-Instrumentation
  -Based on events in the code (function calls, etc.)
  -Instrumentation code can be automatically generated or inserted by hand

- Two types of performance data:
  -Profile: Summation of events over time
  -Trace file: Sequence of events over time

Tools for Profiling

THE UNIVERSITY OF IOWA

- You'll want to look at the system you are running on to see what profiling tools are installed

- Example Software tools for profiling on Moffett
  - Valgrind: Useful for serial or multi-threaded (not MPI parallel) codes
  - Intel VTune: Profiling tool for Intel Compilers
  - PAPI: Measures general application performance
  - MpiP: Measure's MPI Performance
  - HPCToolKit: Event based sampling and profiling related to source code
  - TAU (Tuning and Analysis Utilities)
  - Vampir from ParaTools
  - GPTL (General Purpose Timing Library): Timers and counters
  - IOex: Measure I/O statistics
  - Pfmon: Performance monitor
  - Oprofile: Single-node statistical profiler

- Many of the tools above are available on many different platforms

Profiling Using PAPI and MpiP

THE UNIVERSITY OF IOWA

- For the remainder of this talk, we do some profiling using the tools PAPI and MpiP

- On the following pages, we will look at metrics derived by PAPI and the load balance and MPI statistics from MpiP.

PAPI Results on HYDRO

THE UNIVERSITY OF IOWA

Total Computational Speed:

```
MFLOPS Aggregate (wallclock) ..................  332.93
```

This is the total floating-point computational speed of your parallel computation

```
MFLOPS ....................................... 24.16
```

Average floating-point computational speed per processor.

Floating-point vs. non-floating-point Instructions:

```
Non-FP Instructions % ........................  76.87
FP Instructions % ............................  23.13
```

Floating-point instructions to compute your answer:

```
FP Arith.  Instructions % ....................  7.79
FMA Instructions % ...........................  1.87
```

Computational Intensity and Cache Misses:

```
Flops per Load/Store .........................  0.23
Flops per L1 D-cache Miss ....................  3.70
```

PAPI Results on HYDRO

THE UNIVERSITY OF IOWA

Memory Stall:

```
Total Est.  Memory Stall % ....................  36.32
```

Measured and Estimated Stall:

```
Total Measured Stall % ....................  12.64
Total Underestimated Stall % ..............  40.05
Total Overestimated Stall % ...............  48.96
```

Ideal MFLOPS:

```
Ideal MFLOPS (max.  dual) ....................  62.14
Ideal MFLOPS (cur.  dual) ....................  64.67
```

Parallel Communication Overhead:

```
MPI cycles % ..............................  2.08
```

PAPI Results on HYDRO

THE UNIVERSITY OF IOWA

Memory Usage per Processor:

```
task_0.txt:Mem.  resident peak KB ..........................  62464
task_1.txt:Mem.  resident peak KB ..........................  61696
task_10.txt:Mem.  resident peak KB ..........................  61888
task_11.txt:Mem.  resident peak KB ..........................  61824
task_12.txt:Mem.  resident peak KB ..........................  61824
task_13.txt:Mem.  resident peak KB ..........................  59136
task_2.txt:Mem.  resident peak KB ..........................  61696
task_3.txt:Mem.  resident peak KB ..........................  61888
task_4.txt:Mem.  resident peak KB ..........................  61696
task_5.txt:Mem.  resident peak KB ..........................  61824
task_6.txt:Mem.  resident peak KB ..........................  61824
task_7.txt:Mem.  resident peak KB ..........................  61824
task_8.txt:Mem.  resident peak KB ..........................  61824
task_9.txt:Mem.  resident peak KB ..........................  61824
```

MpiP Results on HYDRO

THE UNIVERSITY OF IOWA

MPI Time and Load Balance:

```
@--- MPI Time (seconds) ------------------------------
--------------------------------------------------------
 Task   AppTime   MPITime   MPI%
   0      96.9     0.411   0.42
   1      96.8      3.09   3.19
   2      96.8     0.513   0.53
   3      96.8     0.287   0.30
   4      96.8     0.707   0.73
   5      96.8     0.378   0.39
   6      96.8     0.442   0.46
   7      96.8      3.09   3.19
   8      96.8     0.454   0.47
   9      96.8      2.93   3.03
  10      96.8      1.69   1.74
  11      96.8      1.58   1.63
  12      96.8      1.67   1.72
  13      96.8      16.4  16.90
```