

Ohio Supercomputer Center

An **OH·TECH** Consortium Member

Performance Tuning Workshop

Dr. Judy Gardiner

July 27, 2017

Workshop Philosophy

- Aim for “reasonably good” performance
- Discuss performance tuning techniques common to most HPC architectures
 - Compiler options
 - Code modification
- Focus on serial performance
 - ***Most important: Unit stride memory access
- Touch on parallel processing
 - Multithreading
 - MPI



More Important than Performance!

- Correctness of results
- Code readability / maintainability
- Portability – future systems
- Time to solution vs. execution time



Factors Affecting Performance

- Effective use of processor features
 - High degree of internal concurrency in a single core
- Data locality
 - Memory access is slow compared to computation
- File I/O
 - Use an appropriate file system
- Scalable algorithms
- Compiler optimizations
 - Modern compilers are amazing!
- Explicit parallelism



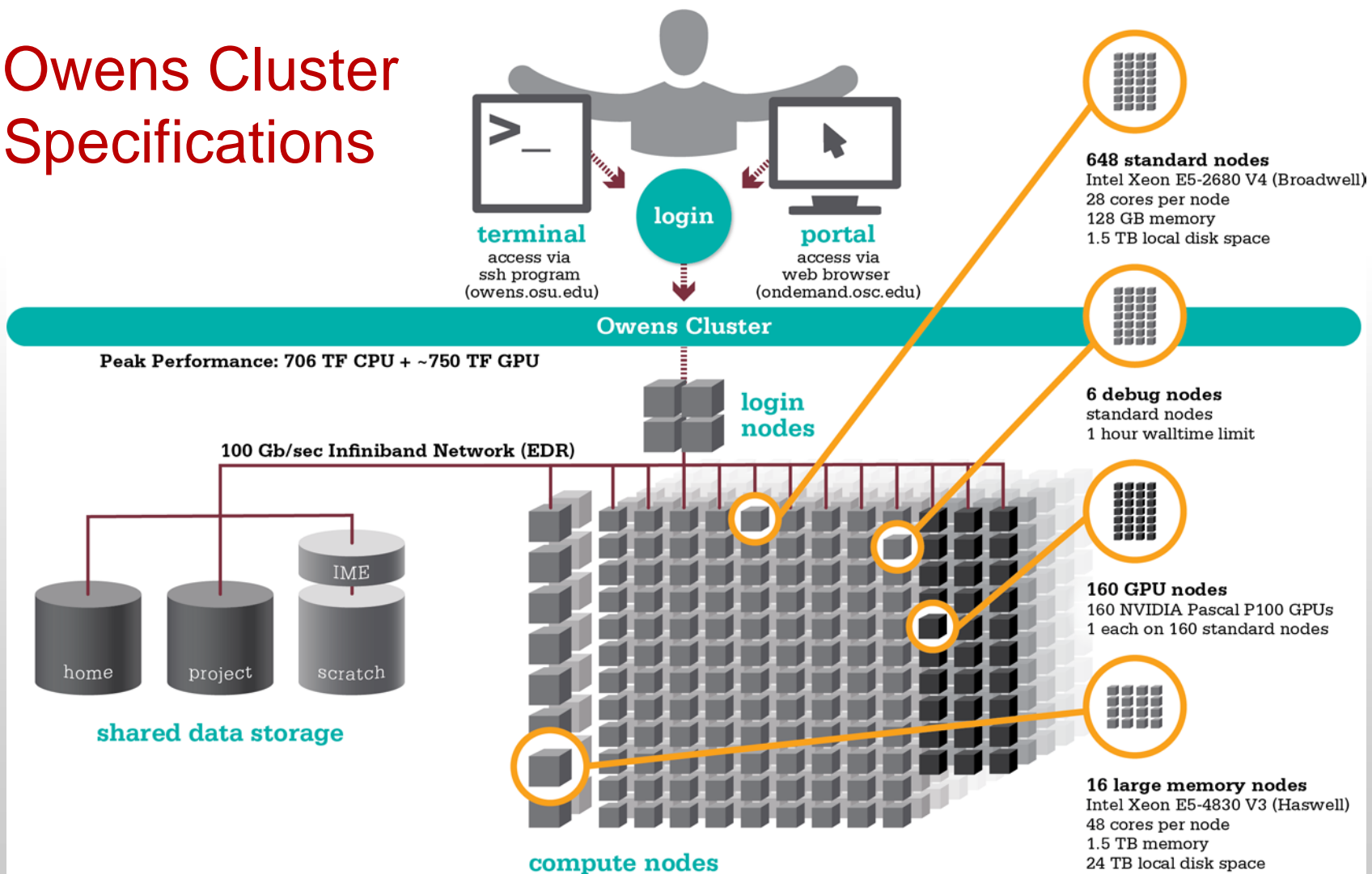


Outline

- Hardware overview
- Performance measurement and analysis
- Help from the compiler
- Code tuning / optimization
- Parallel computing
- Optional hands-on exercises

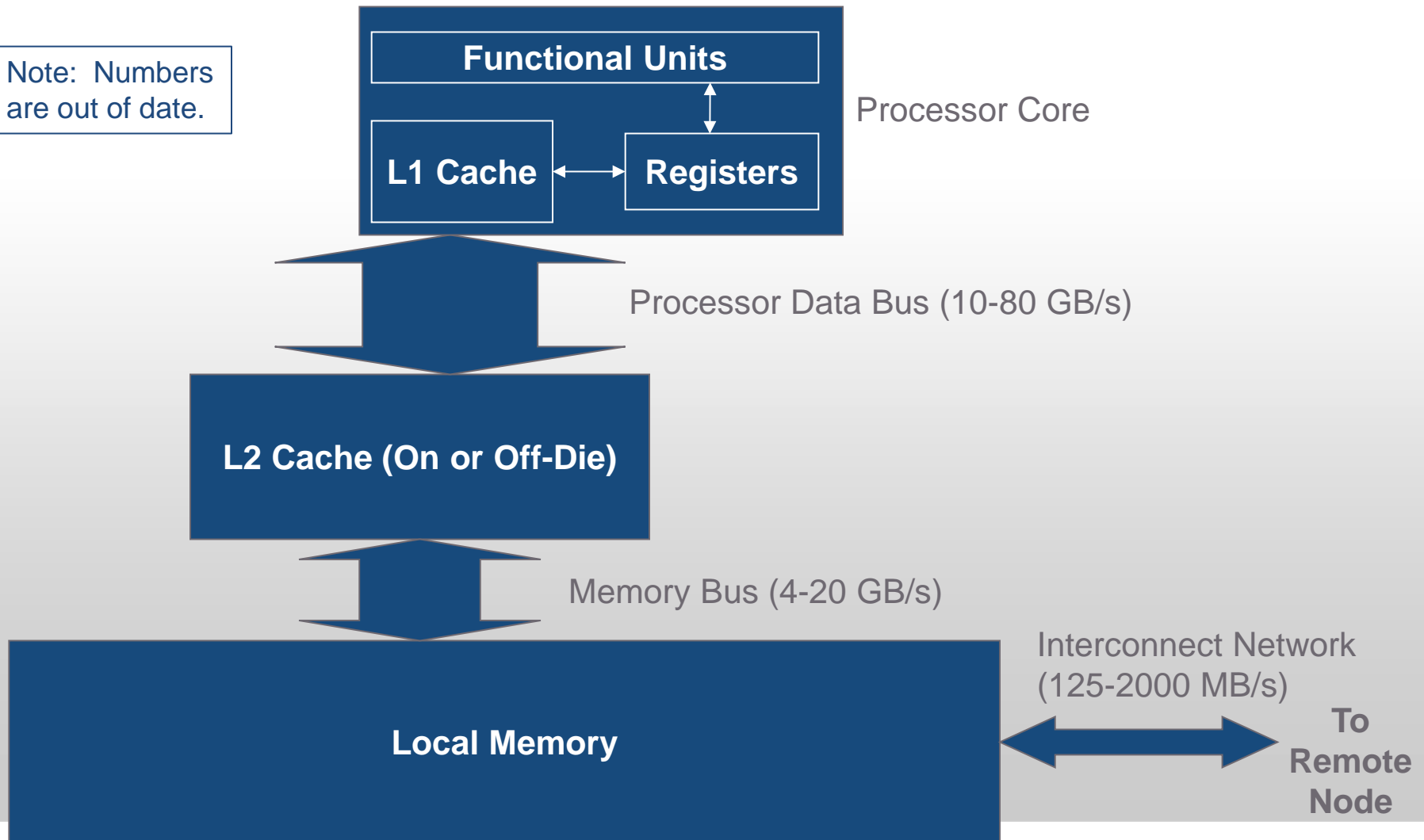


Owens Cluster Specifications



Hierarchical Memory

Note: Numbers are out of date.



Some Processor Features

- 28 cores per node
 - 14 cores per socket * 2 sockets per node
- Vector unit
 - Supports AVX 2
 - Vector length 4 double or 8 single precision values
 - Fused multiply-add
- Hyperthreading
 - Hardware support for 2 threads per core
 - Not currently enabled on OSC systems



Keep data close to the processor – file systems

- **NEVER DO HEAVY I/O IN YOUR HOME DIRECTORY!**
 - Home directories are for long-term storage, not scratch files
 - One user's heavy I/O load can affect all users
- For I/O-intensive jobs
 - Local disk on compute node (not shared)
 - Stage files to and from home directory into \$TMPDIR
 - Execute program in \$TMPDIR
 - Scratch file system
 - /fs/scratch/username or \$PFSDIR
 - Faster than other file systems
 - Good for parallel jobs
 - May be faster than local disk



What is good performance?

- FLOPS
 - Floating Point OPerations per Second
- Peak performance
 - Theoretical maximum (all cores fully utilized)
 - Owens – 859 trillion FLOPS (859 teraflops)
- Sustained performance
 - LINPACK benchmark
 - Owens – 706 teraflops
- Application performance is typically much less



Performance Measurement and Analysis

- Wallclock time
 - How long the program takes to run
- Performance reports
 - Easy, brief summary
- Profiling
 - Detailed information, more involved



Timing – command line

- Time a program
 - `/usr/bin/time command`

```
/usr/bin/time j3
5415.03user 13.75system 1:30:29elapsed 99%CPU \
(0avgtext+0avgdata 0maxresident)k \
0inputs+0outputs (255major+509333minor)pagefaults 0swaps
```

- Note: Hardcode the path – less information otherwise
- `/usr/bin/time` gives results for
 - user time (CPU time spent running your program)
 - system time (CPU time spent by your program in system calls)
 - elapsed time (wallclock)
 - % CPU -- (user+system)/elapsed
 - memory, pagefault, and swap statistics
 - I/O statistics



Timing routines embedded in code

- Time portions of your code
 - C/C++
 - Wallclock: `time(2)`, `difftime(3)`, `getrusage(2)`
 - CPU: `times(2)`
 - Fortran 77/90
 - Wallclock: `SYSTEM_CLOCK(3)`
 - CPU: `DTIME(3)`, `ETIME(3)`
 - MPI (C/C++/Fortran)
 - Wallclock: `MPI_Wtime(3)`



Profiling Tools Available at OSC

- Profiling tools
 - Allinea Performance Reports
 - Allinea MAP
 - Intel VTune
 - Intel Trace Analyzer and Collector (ITAC)
 - Intel Advisor



What can a profiler show you?

- Whether code is
 - compute-bound
 - memory-bound
 - communication-bound
- How well the code uses available resources
 - Multiple cores
 - Vectorization
- How much time is spent in different parts of the code
- Profilers use hardware performance counters



Compilation flags for profiling

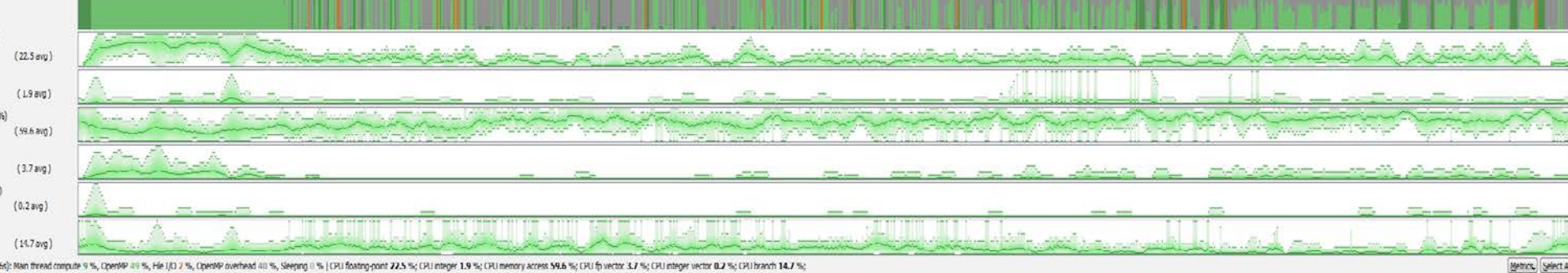
- For profiling
 - Use -g flag
 - Explicitly specify optimization level -O*n*
 - Example: `icc -g -O3 -o mycode mycode.c`
- Use the same level of optimization you normally do
 - Bad example: `icc -g -o mycode mycode.c`
 - Equivalent to -O0



Allinea MAP

- Interpretation of profile requires some expertise
- Gives details about your code's performance
- For a non-MPI program:
 - `module load allinea`
 - `map --profile --no-mpi ./mycode [args]`
- For an MPI program:
 - `map --profile -np num_procs ./mycode [args]`
- View and explore resulting profile using Allinea client





```
1100
1101
1102 //printf("plog %d\n",lower_level_match);
1103
1104 }
1105 else
1106 {
1107     count blunder = DecisionMPs(true,count MPs,subBoundary,GridPT3,level,grid resolution,iteration,Size Grid2D,filename mps_pre,filename mps,proinfo.save_filepath,
1108                               Hinterval,glower level match,sprc Sigma,sprc mean,count results,smink mps,sminkH mps,minmaxHeight,
1109                               SubMapImages_L,SubMapImages_R,grid_resolution, Image_res[0],LRFCs,BRRCs,
1110                               Imagesize,data_size_l[level],SubImages_L,Imagesize,data_size_r[level],SubImages_R,Template_size,
1111                               param,Grid wgs,GridPT,
1112                               NumOfIAparam, t_Rimseparam, Lstartpos, Rstartpos,
1113                               proinfo.save_filepath,row,col,proinfo.pre_RenTif);
1114
1115     count MPs = count results[0];
1116     //printf("plog %d\n",lower_level_match);
1117     printf("row = %d,col = %d,level = %d,iteration = %d,End blunder noisain" row,col,level,iteration);
```

OpenMP State	OpenMP Regions	Functions	
Total	Child	Overhead	Function
16.3%	16.3%	16.3%	__kmp_launch_worker(void*)
16.3%	16.3%	16.3%	__kmp_launch_thread
16.3%	16.3%	16.3%	wait (OpenMP Overhead)
16.3%	16.3%	16.3%	__kmp_fork_barrier(int, int)
16.3%	16.3%	16.3%	__kmp_hyper_barrier_release(barrier_type, kmp_int*, int, int, int, void*) (OpenMP Overhead)
15.0%	0.1%	0.1%	VerticalLineLocs_blunder (OpenMP region 1)
14.3%	14.3%	14.3%	__kmp_wait_yield_1
9.2%	9.2%	9.2%	__kmp_wait_template (OpenMP Overhead)
7.7%	7.7%	7.7%	suspend [inlined] (OpenMP Overhead)
7.7%	7.7%	7.7%	__kmp_suspend_64 (OpenMP Overhead)
7.7%	7.7%	7.7%	__kmp_suspend_template [inlined] (OpenMP Overhead)
7.7%	7.7%	7.7%	pthread_cond_wait (OpenMP Overhead)
7.1%	7.1%	7.1%	__kmp_wait_template [inlined] (OpenMP Overhead)
6.2%	6.2%	6.2%	__kmp_dispatch_next
6.2%	6.2%	6.2%	test_then_inc_arg [inlined]
5.4%	5.4%	5.4%	exp.L
10.0%	5.8%	10.0%	Orientation [inlined]
7.9%	4.4%	7.9%	GetObjectToImageRPC_single
3.6%	0.7%	3.6%	VerticalLineLocs (OpenMP region 1)
2.4%	2.4%	2.4%	__kmp_x86_pause [inlined]
3.3%	0.6%	3.3%	malloc
4.5%	2.3%	4.5%	_JO_vfsync_internal
2.0%	2.0%	2.0%	__ll_lock_wait_private
1.7%	1.7%	1.7%	__kmp_wait_yield_4
1.7%	<0.1%	<0.1%	SetHeightRange_blunder (OpenMP region 0)
1.6%	1.6%	1.6%	__close_nochannel
1.5%	1.5%	1.5%	sched_yield
1.5%	1.5%	1.5%	sn.N
5.0%	3.6%	5.0%	ps2ops_single [inlined]
1.4%	1.4%	1.4%	__kmp_x86_pause [inlined] (OpenMP Overhead)
1.2%	1.2%	1.2%	__kmp_A0_4
1.0%	1.0%	1.0%	_int_free
1.0%	1.0%	1.0%	isCreateImageYramid(void) (OpenMP region 1)
1.7%	0.6%	1.7%	__strtof_l_internal
0.8%	0.8%	0.8%	pow.L
0.8%	0.8%	0.8%	__kmp_launch_monitor(void*)
17.4%	16.7%	5.0%	VerticalLineLocs_blunder (OpenMP)
0.7%	0.7%	0.7%	LZWDecode
0.7%	0.7%	0.7%	atan2
0.6%	0.6%	0.6%	__mun_dirren
0.6%	0.6%	0.6%	_int_malloc
0.6%	0.6%	0.6%	sched_yield (OpenMP Overhead)

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 – not allowed at OSC (job gets killed)
- CPU time vs. wall time (% CPU)
 - Low CPU utilization may mean excessive I/O delays






Allinea Performance Reports

- Easy to use
 - “-g” flag not needed - works on precompiled binaries
- Gives a summary of your code's performance
 - view report with browser
- For a non-MPI program:
 - `module load allinea`
 - `perf-report --no-mpi ./mycode [args]`
- For an MPI program:
 - `perf-report -np num_procs ./mycode [args]`



Processes: 1 node (20 physical, 20 logical cores per node)
Memory: 63 GB per node
Tasks: 1 process, OMP_NUM_THREADS was 0
Machine: r0019.ten.osc.edu
Start time: Fri Aug 28 11:03:22 2015
Total time: 667 seconds (11 minutes)
Full path: /nfs/14/judithg/howat/SETSM_c_0224
Input file:
Notes:

Summary: setsm is **Compute-bound** in this configuration

Compute	96.0%		Time spent running application code. High values are usually good. This is very high ; check the CPU performance section for advice.
MPI	0.0%		Time spent in MPI calls. High values are usually bad. This is very low ; this code may benefit from a higher process count.
I/O	4.0%		Time spent in filesystem I/O. High values are usually bad. This is very low ; however single-process I/O may cause MPI wait times.

This application run was **Compute-bound**. A breakdown of this time and advice for investigating further is in the **CPU** section below.

As very little time is spent in **MPI** calls, this code may also benefit from running at larger scales.

CPU

A breakdown of the **96.0%** CPU time:

Single-core code	24.5%	
OpenMP regions	75.5%	
Scalar numeric ops	24.0%	
Vector numeric ops	3.8%	
Memory accesses	63.9%	

The per-core performance is **memory-bound**. Use a profiler to identify time-consuming loops and check their cache performance.

MPI

A breakdown of the **0.0%** MPI time:

Time in collective calls	0.0%
Time in point-to-point calls	0.0%
Effective process collective rate	0.00 bytes/s
Effective process point-to-point rate	0.00 bytes/s

No time is spent in **MPI** operations. There's nothing to optimize here!



More Information about Allinea Tools

- www.osc.edu/documentation/software_list/allinea
- www.allinea.com



Compiler and Language Choice

- HPC software traditionally written in Fortran or C
- OSC supports several compiler families
 - Intel (icc, icpc, ifort)
 - Usually gives fastest code on Intel architecture
 - Portland Group (PGI - pgcc, pgc++, pgf90)
 - Good for GPU programming, OpenACC
 - GNU (gcc, g++, gfortran)
 - Open source, universally available



Compiler Options for Performance Tuning

- Why use compiler options?
 - Processors have a high degree of internal concurrency
 - Compilers do an amazing job at optimization
 - Easy to use – Let the compiler do the work!
 - Reasonably portable performance
- Optimization options
 - Let you control aspects of the optimization
- Warning:
 - Different compilers have different default values for options



Compiler Optimizations

- Function inlining
 - Eliminate function calls
- Interprocedural optimization/analysis (ipo/ipa)
 - Same file or multiple files
- Loop transformations
 - Unrolling, interchange, splitting, tiling
- Vectorization
 - Operate on arrays of operands
- Automatic parallelization of loops
 - Very conservative multithreading



What compiler flags do I try first?

- General optimization flags (-O2, -O3, -fast)
- Fast math
- Interprocedural optimization / analysis
- Profile again, look for changes
- Look for new problems / opportunities



Floating Point Speed vs. Accuracy

- Faster operations are sometimes less accurate
- Some algorithms are okay, some quite sensitive
- Intel compilers
 - Fast math by default with -O2 and -O3
 - Use `-fp-model precise` if you have a problem (slower)
- GNU compilers
 - Precise math by default with -O2 and -O3 (slower)
 - Use `-ffast-math` for faster performance



Interprocedural Optimization / Inlining

- Inlining
 - Replace a subroutine or function call with the actual body of the subprogram
- Advantages
 - Overhead of calling the subprogram is eliminated
 - More loop optimizations are possible if calls are eliminated
- One source file
 - Typically automatic with -O2 and -O3
- Multiple source files compiled separately
 - Use compiler option for compile and link phases



Optimization Compiler Options – Intel compilers

-fast	Common optimizations
-On	Set optimization level (0, 1, 2, 3)
-ipo	Interprocedural optimization, multiple files
-O3	Loop transforms
-xHost	Use highest instruction set available
-parallel	Loop auto-parallelization

- Don't use **-fast** for MPI programs with Intel compilers
- Use same compiler command to link for **-ipo** with separate compilation
- Many other optimization options are available
- See **man** pages for details
- Recommended options:
-O3 -xHost
- Example:
ifort -O3 program.f90



Optimization Compiler Options – PGI compilers

-fast	Common optimizations
-On	Set optimization level (0, 1, 2, 3, 4)
-Mipa	Interprocedural analysis
-Mconcur	Loop auto-parallelization

- Many other optimization options are available
- Use same compiler command to link for **-Mipa** with separate compilation
- See `man` pages for details
- Recommended options:
 - fast**
- Example:

```
pgf90 -fast program.f90
```



Optimization Compiler Options – GNU compilers

<code>-On</code>	Set optimization level (0, 1, 2, 3)
N/A for separate compilation	Interprocedural optimization
<code>-O3</code>	Loop transforms
<code>-ffast-math</code>	Potentially unsafe float pt optimizations
<code>-march=native</code>	Use highest instruction set available

- Many other optimization options are available
- See `man` pages for details
- Recommended options:
 - `-O3 -ffast-math`
 - `-march=native`
- Example:

```
gfortran -O3
program.f90
```



Compiler Optimization Reports

- Let you understand
 - how well the compiler is doing at optimizing your code
 - what parts of code need work
- Generated at compile time
 - Describe what optimizations were applied at various points in the source code
 - May tell you why optimizations could not be performed



Options to Generate Optimization Reports

- Intel compilers
 - `-opt-report`
 - Output to a file
- Portland Group compilers
 - `-Minfo`
 - Output to stderr
- GNU compilers
 - `-fopt-info`
 - Output to stderr by default



Sample from an Optimization Report

```
LOOP BEGIN at laplace-good.f(10,7)
  remark #15542: loop was not vectorized: inner loop was already vectorized

  LOOP BEGIN at laplace-good.f(11,10)
    <Peeled loop for vectorization>
  LOOP END

  LOOP BEGIN at laplace-good.f(11,10)
    remark #15300: LOOP WAS VECTORIZED
  LOOP END

  LOOP BEGIN at laplace-good.f(11,10)
    <Remainder loop for vectorization>
    remark #15301: REMAINDER LOOP WAS VECTORIZED
  LOOP END

  LOOP BEGIN at laplace-good.f(11,10)
    <Remainder loop for vectorization>
  LOOP END
LOOP END
```



A word about algorithms

- Problem-dependent – can't generalize
- Scalability is important
 - How computational time increases with problem size
- 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 2D convolutions with 2D FFTs
 - $O(n^4) \rightarrow O(n^2 \log(n))$
- Replace home-grown algorithm with call to optimized library



Code Modifications for Optimization

- Memory optimizations
 - Unit stride memory access
 - Efficient cache usage
- Vectorization
 - Vectorizable loops
 - Vectorization inhibitors



Unit Stride 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



Data Layout: Object-Oriented Languages

- Arrays of objects may give poor performance on HPC systems if used naively
 - C structs
 - C++ classes
 - Fortran 90 user-defined types
- Inefficient use of cache – not unit stride
 - Can often get factor of 3 or 4 speedup just by fixing it
- You can use them efficiently! Be aware of data layout.
- Data layout may be the only thing modern compilers can't optimize



Efficient Cache Usage

- Cache lines
 - 8 words (64 bytes) of consecutive memory
 - Entire cache line is loaded when a piece of data is fetched
- Good example – Entire cache line used
 - 2 cache lines used for every 8 loop iterations
 - Unit stride

```
real*8 a(N), b(N)
do i=1,N
  a(i)=a(i)+b(i)
end do
```

```
2 cache lines:
a(1),a(2),a(3), ... a(8)
b(1),b(2),b(3), ... b(8)
```



Efficient Cache Usage – Cache Lines (cont.)

- Bad example – Unneeded data loaded
 - 1 cache line loaded for *each* loop iteration
 - 8 words loaded, only 2 words used
 - Not unit stride

```
TYPE :: node
  real*8 a, b, c, d, w, x, y, z
END TYPE node
TYPE(node) :: s(N)
do i=1,N
  s(i)%a = s(i)%a + s(i)%b
end do
```

```
cache line:
a(1),b(1),c(1),d(1),w(1),x(1),y(1),z(1)
```



Vectorization / Streaming

- Code is structured to operate on arrays of operands
 - Single Instruction, Multiple Data (SIMD)
- Vector instructions built into processor (AVX, SSE, etc.)
 - Vector length 8 single or 4 double precision on Owens
- Requires unit stride
- Fortran 90, MATLAB have this idea built in
- A vectorizable loop:

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



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
- Too many conditionals
 - “if” statements



Optimized Mathematical Libraries

- MKL (Intel Math Kernel Library)
 - BLAS
 - LAPACK
 - FFT
 - Vectorized transcendental functions (sin, cos, exp)
- FFTW
- ScaLAPACK
- SuperLU
- ... and many others



Parallel Computing

- Multithreading
 - Shared-memory model (single node)
 - OpenMP support in compilers
- Message Passing Interface (MPI)
 - Distributed-memory model (single or multiple nodes)
 - Several available libraries
- Accelerators / Coprocessors
 - GPUs
 - Intel Xeon Phi (not currently available at OSC)



What is OpenMP?

- Shared-memory, threaded parallel programming model
- Portable standard
- A set of compiler directives
- A library of support functions
- Supported by vendors' compilers
 - Intel
 - Portland Group
 - GNU
 - Cray



Parallel loop execution – Fortran

- Inner loop vectorizes
- Outer loop executes on multiple threads

```
PROGRAM omploop
INTEGER, PARAMETER :: N = 1000
INTEGER i, j
REAL, DIMENSION(N,N) :: a, b, c, x
... ! Initialize arrays
!$OMP PARALLEL DO
do j=1,N
  do i=1,N
    a(i,j)=b(i,j)+x(i,j)*c(i,j)
  end do
end do
!$OMP END PARALLEL DO
END PROGRAM omploop
```



Parallel loop execution – C

- Inner loop vectorizes
- Outer loop executes on multiple threads

```
[owens-login01]$ cat omploop.c
int main()
{
    int N = 1000;
    float *a, *b, *c, *x;
    ... // Allocate and initialize arrays
    #pragma omp parallel for
        for (int i=0; i<N; i++) {
            for (int j=0; j<N; j++) {
                a[i*N+j]=b[i*N+j]+x[i*N+j]*c[i*N+j]
            }
        }
}
```



Compiling a program with OpenMP

- Intel compilers
 - Add the `-qopenmp` option

```
[owens-login01]$ ifort -qopenmp ompex.f90 -o ompex
```

- gnu compilers
 - Add the `-fopenmp` option

```
[owens-login01]$ gcc -fopenmp ompex.c -o ompex
```

- Portland Group compilers
 - Add the `-mp` option

```
[owens-login01]$ pgf90 -mp ompex.f90 -o ompex
```



Running an OpenMP program

- Request multiple processors through PBS
 - Example: `nodes=1:ppn=28`
- Set the `OMP_NUM_THREADS` environment variable
 - Default: Use all available cores
- For best performance run at most one thread per core
 - Otherwise too much overhead
 - Applies to typical HPC workload, exceptions exist



Running an OpenMP program – Example

```
[owens-login01]$ cat omploop.pbs
#PBS -N omploop
#PBS -j oe
#PBS -l nodes=1:ppn=28
#PBS -l walltime=1:00

cd $PBS_O_WORKDIR
export OMP_NUM_THREADS=28
/usr/bin/time ./omplloop
```



More Information about OpenMP

- www.openmp.org
- OpenMP Application Program Interface
 - Version 3.1, July 2011
 - <http://www.openmp.org/wp-content/uploads/OpenMP3.1.pdf>



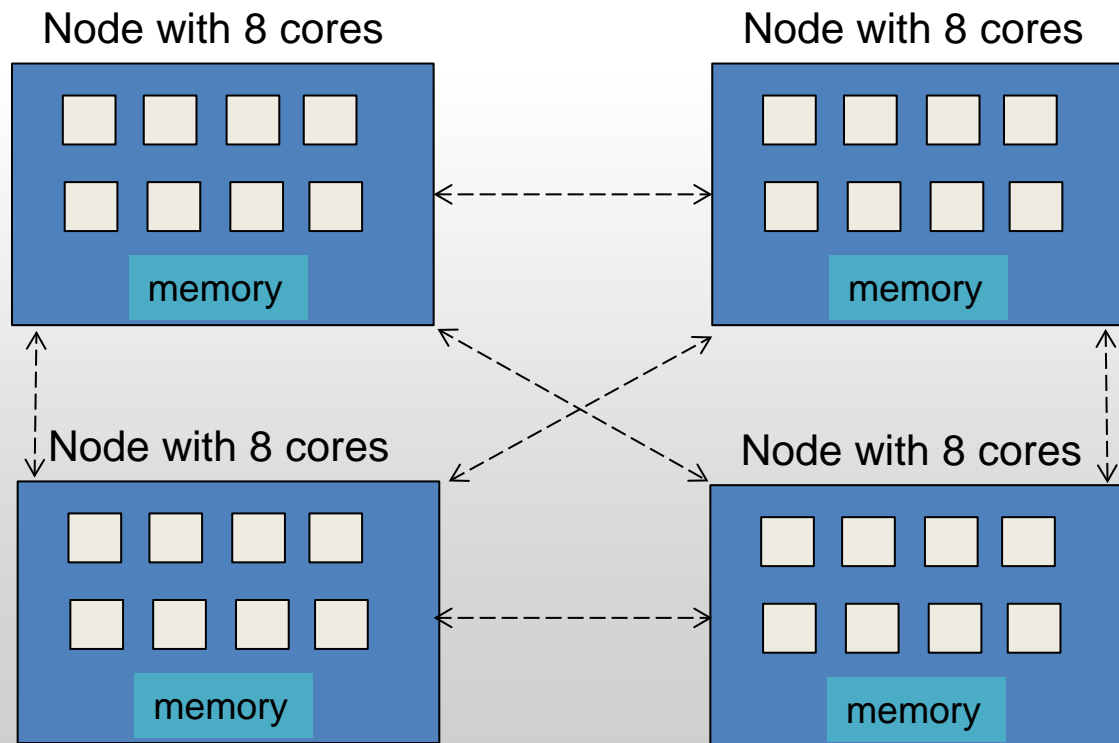
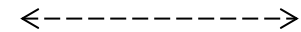
What is MPI?

- Message Passing Interface
 - Multiple processes run on one or more nodes
 - Distributed-memory model
- A message passing library
- A run-time environment
 - *mpiexec*
- Compiler wrappers
- Supported by all major parallel machine manufacturers



OpenMP vs. MPI

Infiniband network



A simple MPI program

```
[owens-login01]$ cat hello.c
#include <mpi.h>
#include <stdio.h>
int main(int argc, char *argv[])
{
    int rank, size;

    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);
    printf("Hello from node %d of %d\n", rank, size);
    MPI_Finalize();
    return(0);
}
```



MPI Implementations Available at OSC

- mvapich2
 - default
- IntelMPI
 - available only with Intel compilers
- OpenMPI



Compiling MPI programs

- Compile with the MPI compiler wrappers
 - `mpicc`, `mpicxx` and `mpif90`
 - Accept the same arguments as the compilers they wrap

```
[owens-login01]$ mpicc -o hello hello.c
```

- Compiler and MPI implementation depend on modules loaded



Running MPI programs

- MPI programs must run in batch only
 - Debugging runs may be done with interactive batch jobs
- *mpiexec*
 - Automatically determines execution nodes from PBS
 - Starts the program running, $2 \times 28 = 56$ copies

```
[owens-login01]$ cat hello.pbs
#PBS -N mpi_hello
#PBS -j oe
#PBS -l nodes=2:ppn=28
#PBS -l walltime=1:00

cd $PBS_O_WORKDIR
mpiexec ./hello
```



More Information about MPI

- www.mpi-forum.org
- MPI: A Message-Passing Interface Standard
 - Version 3.1, June 4, 2015
 - <http://mpi-forum.org/docs/mpi-3.1/mpi31-report.pdf>



GPU-Accelerated Computing

- GPU = Graphics Processing Unit
 - Can be used to accelerate computation
- OSC clusters have some nodes with NVIDIA GPUs
- Manycore processors
 - more cores than multicore
- Can be programmed with CUDA
 - low level
- PGI and gnu compilers support OpenACC
 - easier than CUDA
 - similar to OpenMP



Summary: 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
 - Including vectorization



Other Sources of Information

- Online manuals
 - `man ifort`
 - `man pgc++`
 - `man gcc`
- Related workshop courses
 - www.osc.edu/supercomputing/training
- Online tutorials from Cornell
 - <https://cvw.cac.cornell.edu/>
- oschelp@osc.edu



Questions

Judy Gardiner

Scientific Applications Engineer

Ohio Supercomputer Center

judithg@osc.edu

1224 Kinnear Road

Columbus, OH 43212

Phone: (614) 292-9623



ohiosupercomputercenter



ohiosupercomputerctr

