CAE workloads and machine architecture

Stephen Behling
sbehling@cray.com
May, 2015

Outline

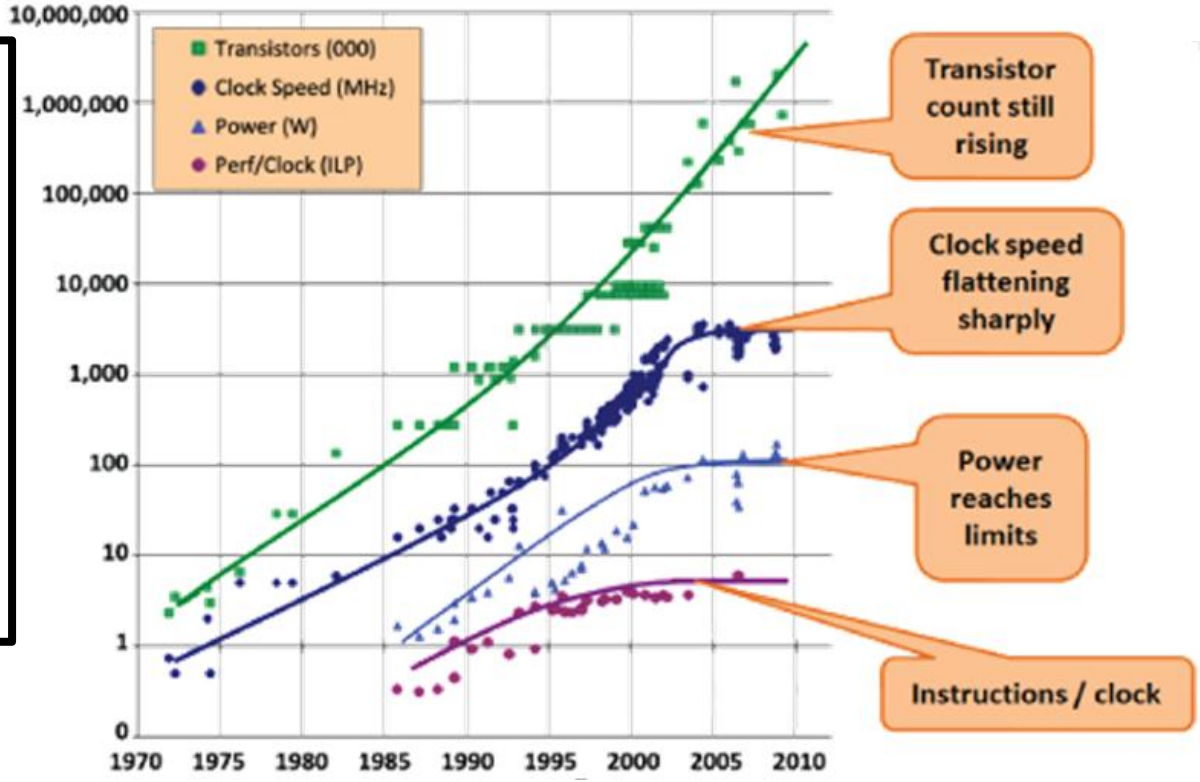
- **HPC hardware trends**
- **Examples of large analyses**
- **Aspects of writing codes or modifying codes for future machines**
  - Profile applications
  - Vectors
  - Threads
  - MPI
  - I/O

Processors frequencies are not increasing

http://www.extremetech.com/computing/116561-the-death-of-cpu-scaling-from-one-core-to-many-and-why-were-still-stuck

Faster processor frequencies no longer provide the required HPC performance increase

Increased parallelism/scaling is required to meet growing computing requirements



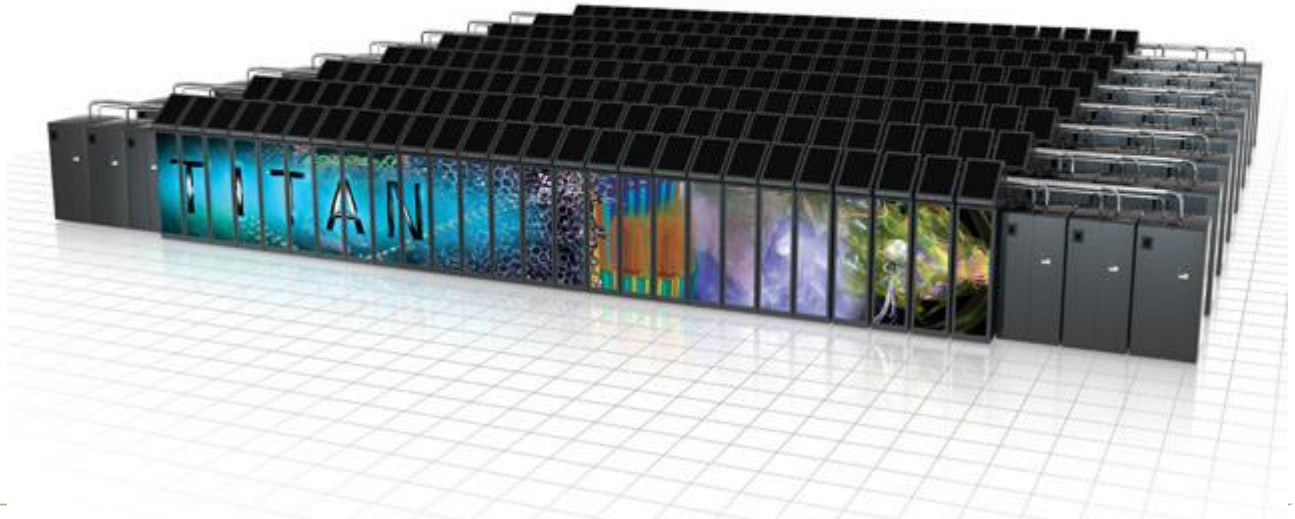
The future is more parallel

- Moore's Law is continuing for transistors per chip
- Processor power and energy limits frequency
- Hard to get more instructions per clock

Result:

- More cores per socket
- More sockets per system
- More threads/cores per job

CAE on these?

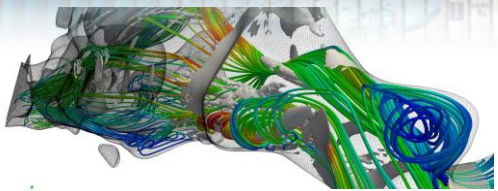


CFD Results from NCSA "Blue Waters" system
ALYA code: 3 Real-World Cases

Human Respiratory System

Transient incompressible turbulent flow

360M elements, scaled to 25,000 cores

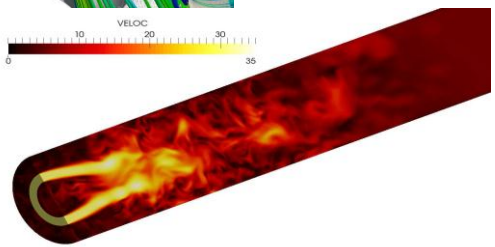


Kiln Furnace

Transient incompressible turbulent flow

Coupled with energy and combustion

4.22 billion elements, scaled to 100,000 cores

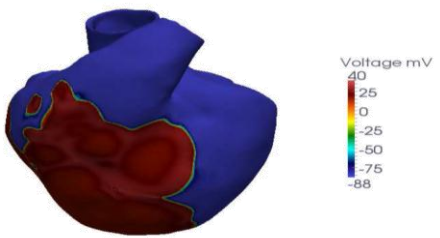


Human Heart

Non-linear solid mechanics

Coupled with electrical propagation

3.4 billion elements, scaled to 100,000 cores



Ref: "Growth of HPC Industrial Partnership", Merle Giles NCSA, Oct. 2014

U.S. DoD Helios program

- **“We hope to try 1 million cores to do CFD for helicopter designs” Roger Strawn, DoD**
  - Coupled codes:
    - 3D Cartesian grid for far field
    - Unstructured grid for near field
    - Overlap region between grids
    - Rotating meshes
  - Adaptive meshing
  - Parallel in space and time (periodic flow)
  - Compute graphics output “on the fly” rather than save data and post-process
  - See at: https://youtu.be/1pcsNIRKoEg

What about third party ISV applications?

- **Each ISV is responsible for porting and optimizing their own code**
  - ISVs need to support all their customers and maintaining multiple versions of a code is very expensive
    - Some ISVs still only have SSE builds; some have AVX builds
    - PowerFLOW from Exa Corporation → 2-10% faster with AVX
- **Cray will continue to work with key partners to ensure best performance on Cray systems**
  - Cray has its own environment and optimized MPI libraries
- **Cray performance team influences:**
  - Cray software design – better libraries, OS, I/O, etc.
  - ISV partner optimization

How to make your application efficient and scalable

1. **Make sure important work is vectorized**
   - Need good compiler
   - Need good profiling tools
2. **Implement on-node threading**
   - OpenMP 4.0
   - 16 cores per socket on current Intel® “Haswell”
   - On KNC and future KNL, >60 cores per socket
3. **Implement MPI for Internode parallelization**
   - MPI standard
4. **Use accelerator if problem is suited for it**

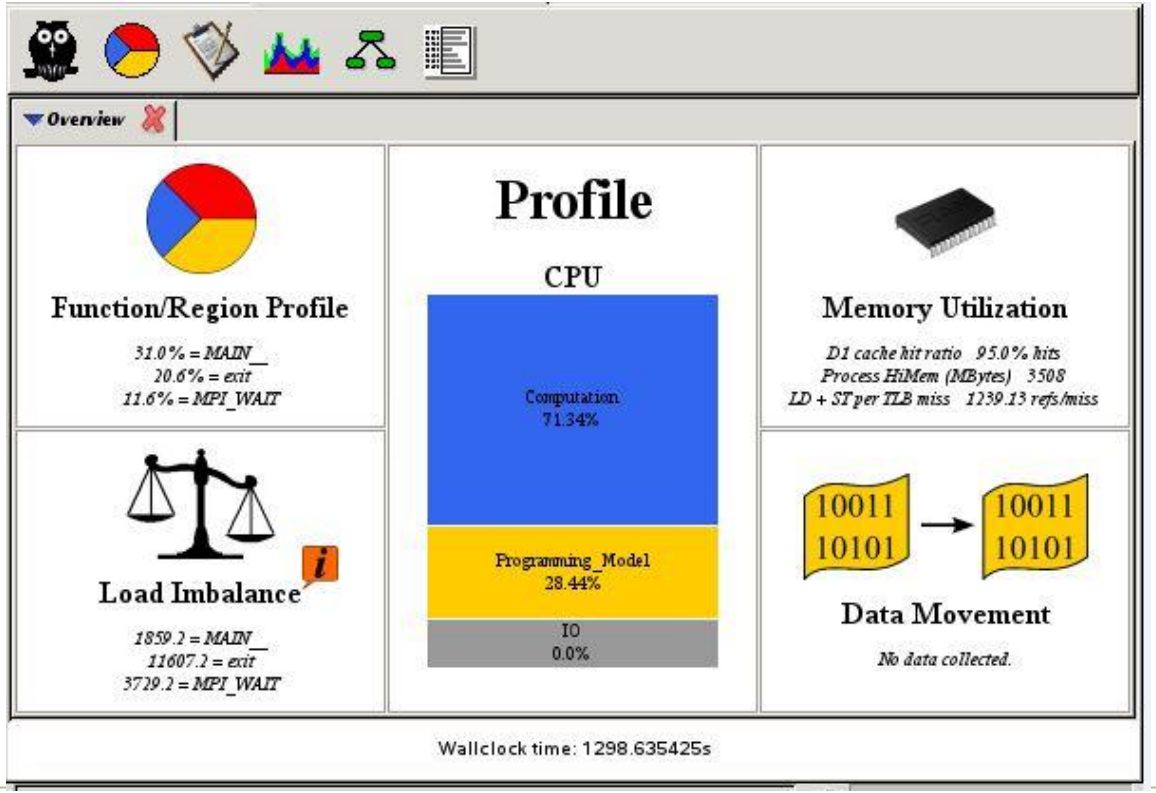
- **MUCH EASIER IF CODE IS DESIGNED FOR EFFICIENCY AND SCALABILITY FROM START THAN MODIFYING EXISTING CODE**

Profiling an application

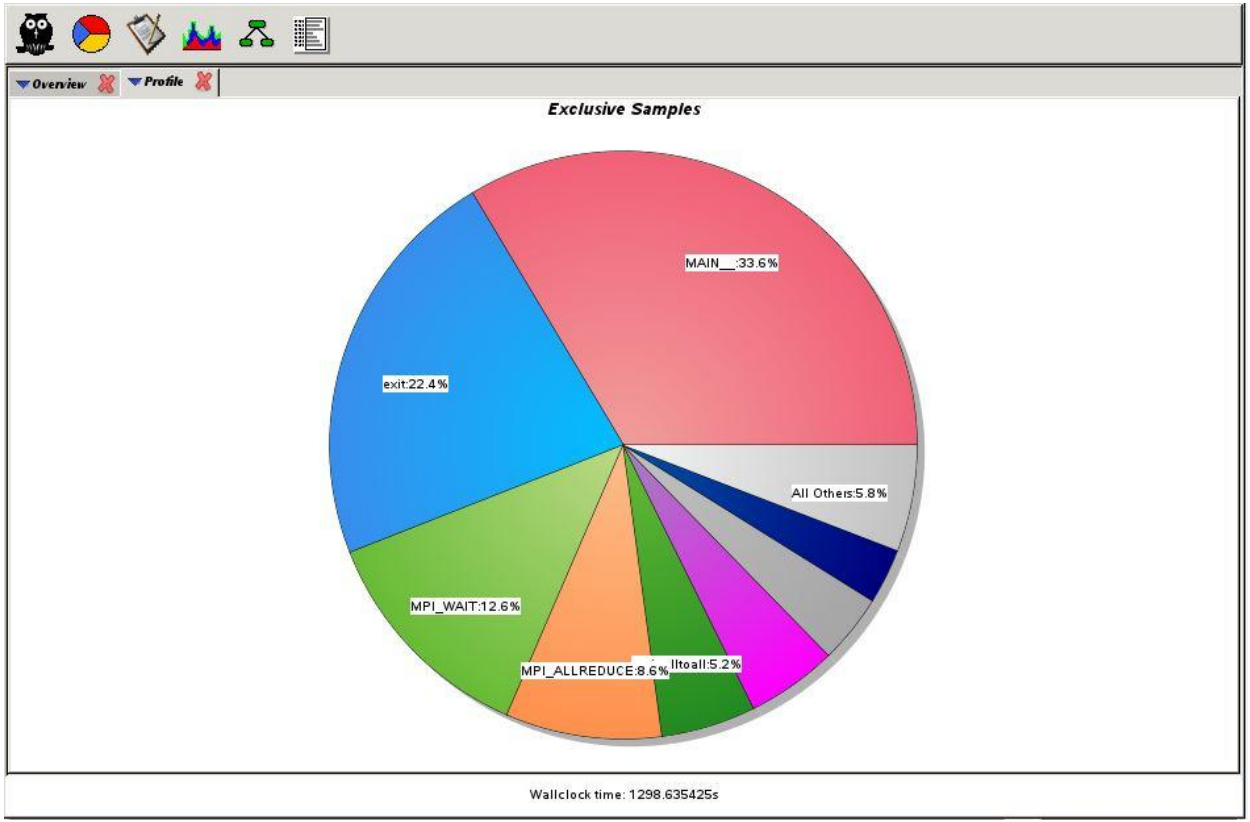
Find important work with a profiling tool

Example: CrayPat (perftools)

512 nodes, 1024 MPI ranks, 16 OpenMP threads per rank



More CrayPat



Profiling tools

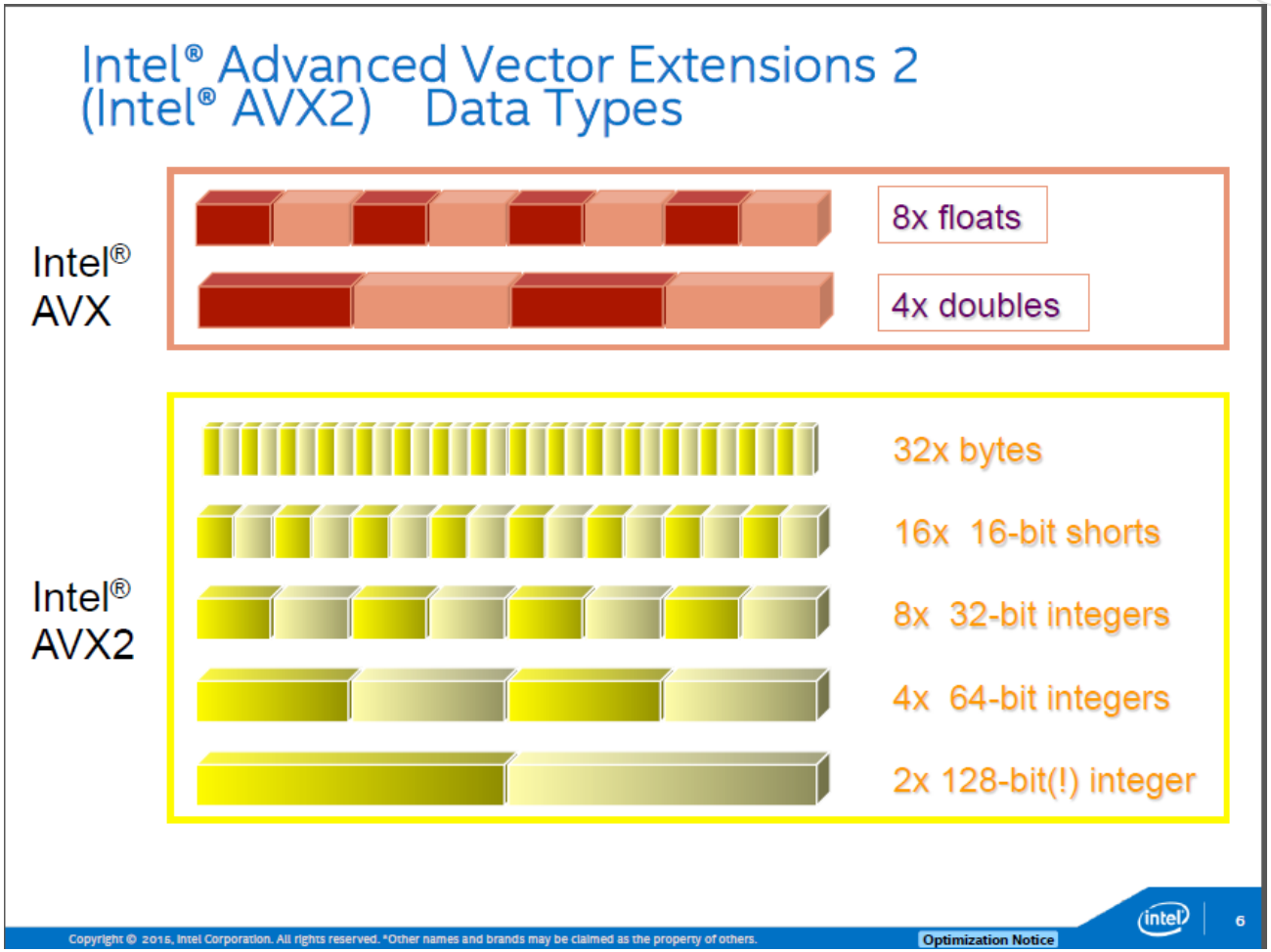
- **CrayPat**
  - Multiple compilers, accelerators
  - Sampling and tracing
- **Intel®**
  - Vtune
    - For use with Intel® compilers; single node
  - Intel® Parallel Studio
- **TAU**
  - Open source
  - Requires separate build for each compiler version/MPI library version/optimization choices/sample or trace choice
- **Vampir from ParaTools**
  - Works with TAU
- **High resolution timers and print statements**

The future is parallel

- **Vectors are parallel**
  - At Instruction level: SSE, AVX, AVX2, etc.
  - Computer hardware is very good doing this
- **Threads (OpenMP) are parallel**
  - Typically at loop level
  - Nested threading is possible, but complicated
- **MPI is parallel**
  - Typically at high level
  - PGAS is alternative to standard MPI

Vectors

Parallel at machine instruction level



Slide from:

Further Vectorization Features of the Intel® Compiler

Martyn Corden
Technical Consulting Engineer
Intel® Corporation

Look for vectorization with compiler

Intel® compiler: `ftn -qopt-report-phase=loop,vec -qopt-report=3 -c abc.f90`
Cray compiler (shown below): `ftn -rm -c abc.f90`

```
72.  + 1 b---------<     DO K = 1, NZ
73.  + 1 b b-------<         DO J = 1,NY
74.    1 b b Vb----<             DO I = 1, NX
75.    1 b b Vb                    SLICE_BACK =  GRID(I-1,J-1,K-1,IVAR) + GRID(I-1,J,K-1,IVAR) + GRID(I-1,J+1,K-1,IVAR) + &
76.    1 b b Vb                                  GRID(I  ,J-1,K-1,IVAR) + GRID(I  ,J,K-1,IVAR) + GRID(I  ,J+1,K-1,IVAR) + &
77.    1 b b Vb                                  GRID(I+1,J-1,K-1,IVAR) + GRID(I+1,J,K-1,IVAR) + GRID(I+1,J+1,K-1,IVAR)
78.    1 b b Vb                    SLICE_MINE =  GRID(I-1,J-1,K,IVAR)   + GRID(I-1,J,K,IVAR)   + GRID(I-1,J+1,K,IVAR) + &
79.    1 b b Vb                                  GRID(I  ,J-1,K,IVAR)   + GRID(I  ,J,K,IVAR)   + GRID(I  ,J+1,K,IVAR) + &
80.    1 b b Vb                                  GRID(I+1,J-1,K,IVAR)   + GRID(I+1,J,K,IVAR)   + GRID(I+1,J+1,K,IVAR)
81.    1 b b Vb                    SLICE_FRONT = GRID(I-1,J-1,K+1,IVAR) + GRID(I-1,J,K+1,IVAR) + GRID(I-1,J+1,K+1,IVAR) + &
82.    1 b b Vb                                  GRID(I  ,J-1,K+1,IVAR) + GRID(I  ,J,K+1,IVAR) + GRID(I  ,J+1,K+1,IVAR) + &
83.    1 b b Vb                                  GRID(I+1,J-1,K+1,IVAR) + GRID(I+1,J,K+1,IVAR) + GRID(I+1,J+1,K+1,IVAR)
84.  + 1 b b Vb                    WORK(I,J,K) = ( SLICE_BACK + SLICE_MINE + SLICE_FRONT ) / 27.0
85.    1 b b Vb---->             END DO
86.    1 b b------->         END DO
87.    1 b--------->     END DO
```

Threads

Parallelization within a shared memory node (SMP)

OpenMP threads can help with many core processors

- OpenMP 4.0 spec is now being implemented in many compilers
- Use #pragma statements in C; !$omp directives in Fortran
- Reveal from Cray is a good scoping tool for threading

- Works closely with Cray compilers.
- Automatically inserts pragmas/directives
- Resulting source can be used with any compiler.

OpenMP example

```
75.  + 1            !$omp parallel do  private(slice_back,slice_mine,slice_front)
76.    1            !dir$ blockable(k,j,i)
77.    1            !dir$ blockingsize(4)
78.  + 1 mb-------<     DO K = 1, NZ
79.    1 mb         !dir$ blockingsize(4)
80.  + 1 mb b-----<        DO J = 1,NY
81.    1 mb b       !dir$ blockingsize(128)
82.    1 mb b Vb--<            DO I = 1, NX
83.    1 mb b Vb                 SLICE_BACK =  GRID(I-1,J-1,K-1,IVAR) + GRID(I-1,J,K-1,IVAR) + GRID(I-1,J+1,K-1,IVAR) + &
84.    1 mb b Vb                               GRID(I  ,J-1,K-1,IVAR) + GRID(I  ,J,K-1,IVAR) + GRID(I  ,J+1,K-1,IVAR) + &
85.    1 mb b Vb                               GRID(I+1,J-1,K-1,IVAR) + GRID(I+1,J,K-1,IVAR) + GRID(I+1,J+1,K-1,IVAR)
86.    1 mb b Vb                 SLICE_MINE =  GRID(I-1,J-1,K,IVAR)   + GRID(I-1,J,K,IVAR)   + GRID(I-1,J+1,K,IVAR) + &
87.    1 mb b Vb                               GRID(I  ,J-1,K,IVAR)   + GRID(I  ,J,K,IVAR)   + GRID(I  ,J+1,K,IVAR) + &
88.    1 mb b Vb                               GRID(I+1,J-1,K,IVAR)   + GRID(I+1,J,K,IVAR)   + GRID(I+1,J+1,K,IVAR)
89.    1 mb b Vb                 SLICE_FRONT = GRID(I-1,J-1,K+1,IVAR) + GRID(I-1,J,K+1,IVAR) + GRID(I-1,J+1,K+1,IVAR) + &
90.    1 mb b Vb                               GRID(I  ,J-1,K+1,IVAR) + GRID(I  ,J,K+1,IVAR) + GRID(I  ,J+1,K+1,IVAR) + &
91.    1 mb b Vb                               GRID(I+1,J-1,K+1,IVAR) + GRID(I+1,J,K+1,IVAR) + GRID(I+1,J+1,K+1,IVAR)
92.  + 1 mb b Vb                 WORK(I,J,K) = ( SLICE_BACK + SLICE_MINE + SLICE_FRONT ) / 27.0
93.    1 mb b Vb-->            END DO
94.    1 mb b----->         END DO
95.    1 mb------->      END DO
96.    1            !$omp end parallel do
```

OpenMP example

```
   75.  + 1            !$omp parallel do  private(slice_back,slice_mine,slice_front)
   76.    1            !dir$ blockable(k,j,i)
   77.    1            !dir$ blockingsize(4)
   78.  + 1 mb-------<     DO K = 1, NZ
   79.    1 mb         !dir$ blockingsize(4)
   80.  + 1 mb b-----<         DO J = 1,NY
   81.    1 mb b       !dir$ blockingsize(128)
   82.    1 mb b Vb--<             DO I = 1, NX
   83.    1 mb b Vb                  SLICE_BACK =  GRID(I-1,J-1,K-1,IVAR) + GRID(I-1,J,K-1,IVAR) + GRID(I-1,J+1,K-
1,IVAR) + &
   …
   …
   90.    1 mb b Vb                                GRID(I   ,J-1,K+1,IVAR) + GRID(I   ,J,K+1,IVAR) + GRID(I   ,J+1,K+1,
IVAR) + &
   91.    1 mb b Vb                                GRID(I+1,J-1,K+1,IVAR) + GRID(I+1,J,K+1,IVAR) + GRID(I+1,J+1,K+1,
IVAR)
   92.  + 1 mb b Vb                  WORK(I,J,K) = ( SLICE_BACK + SLICE_MINE + SLICE_FRONT ) / 27.0
   93.    1 mb b Vb-->             END DO
   94.    1 mb b----->         END DO
   95.    1 mb------->     END DO
   96.    1            !$omp end parallel do
```

MPI

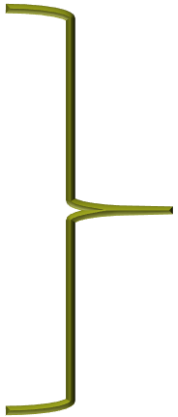
Parallelization between nodes
Distributed memory parallel (DMP)

MPI (Message passing interface)

- **Many MPI flavors**
  - Cray MPT: Only runs on Cray hardware
    - Based on Argonne MPICH for MPI 3.0 standard
    - Highly optimized for Cray networks
  - Intel® MPI
    - Version 5.0 has common ABI with Cray MPT 7
  - IBM Platform MPI
    - Formerly Platform Computing MPI
      - Formerly HP-MPI
  - MPICH
    - From Argonne National Laboratory
  - MVAPICH2
    - Open source; from Ohio State University
  - OpenMPI
    - Open source; hosted by Indiana University

MPI (Message passing interface)

- **Many MPI flavors**
  - Cray MPT: Only runs on Cray hardware
    - Based on Argonne MPICH
  - Intel® MPI
    - Version 5.0 has common ABI with Cray MPT 7
  - IBM Platform MPI
    - Formerly Platform Computing MPI
      - Formerly HP-MPI
  - MPICH
    - From Argonne National Laboratory
  - MVAPICH2
    - Open source; from Ohio State University
  - OpenMPI
    - Open source; hosted by Indiana University



Use Cray CCM with these MPI libraries:

```
module load ccm
ccmrun mpirun …
```

Parallel Limitation: Amdahl's Law

$$S = \frac{1}{\frac{f_P}{N} + f_s}$$

$S = \text{Maximum parallel speedup}$
$N = \text{Number of processors}$
$f_P = \text{fraction of program that is parallel}$
$f_s = 1 - f_P = \text{fraction of program that is serial}$

If parallel fraction = 0.9, speed up for infinite number of processors = 10
If parallel fraction = 0.95, speed up for infinite number of processors = 20
If parallel fraction = 0.99, speed up for infinite number of processors = 100

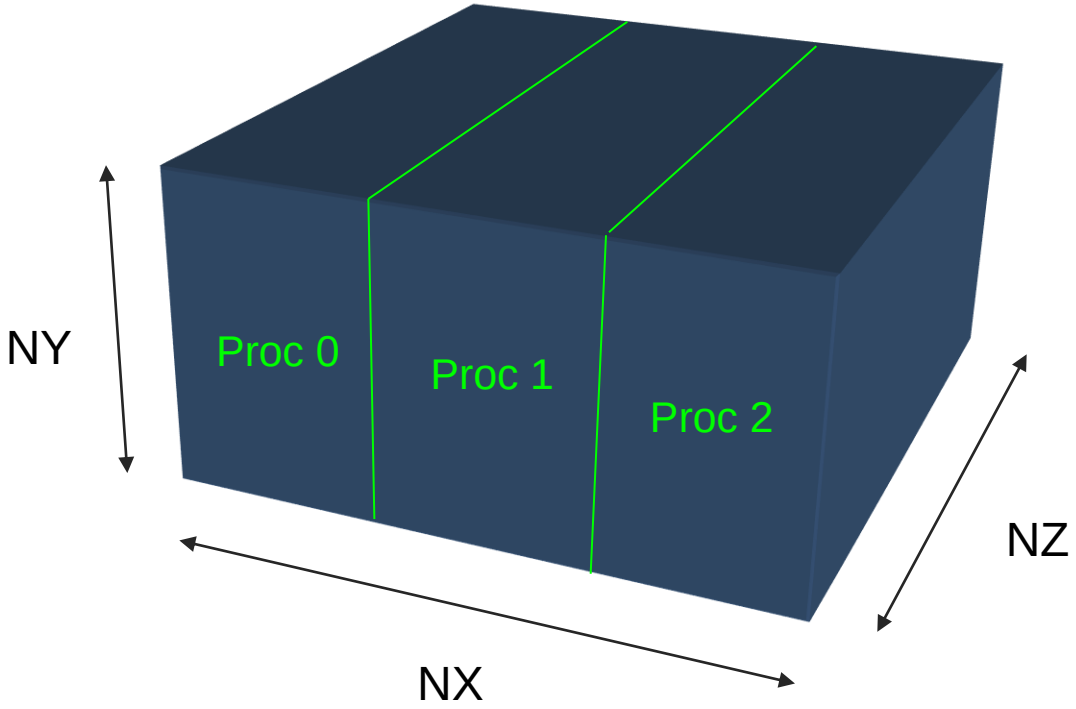
MPI performance

- **Key to MPI performance is minimal serial work**
- **Key to MPI performance is a good network**
  - High bandwidth
    - Many GB/s
  - Low latency
    - 1-2 microseconds for short messages
- **Key to MPI performance is a network that scales**
  - For thousands of nodes need to maintain high bandwidth and low latency across the system
  - Needs to be balanced
  - Minimal interference from other jobs
- **Key to MPI performance is good MPI software**
  - More important for larger systems

1-D MPI implementation for Cartesian mesh is simple

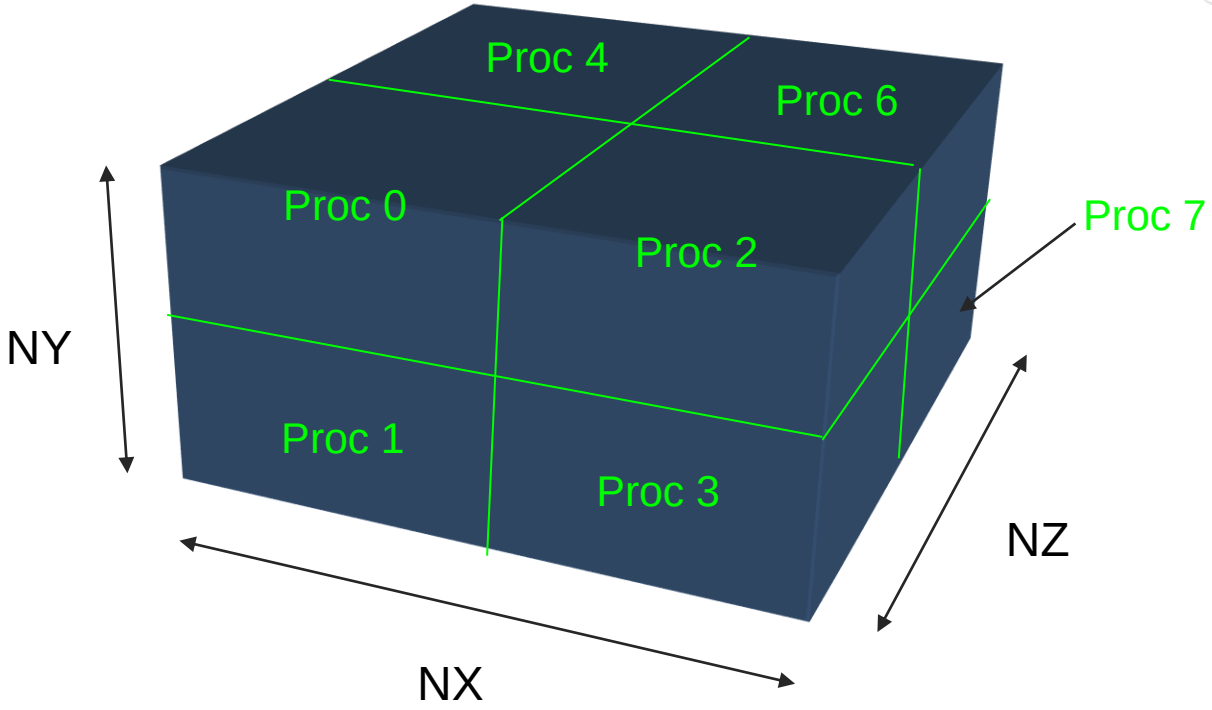
Experienced MPI programmer could implement in 3-4 weeks.

However, scaling is quite limited even for large data sets



2-D or 3-D MPI implementation is more complicated

- Handling the cells on the corners is difficult.
- Unstructured grids are more complicated.
- Scaling can be very good to very large processor counts.
- Key: Only move the data you need.



I/O can be parallel, too

I/O parallelization

- **Lustre file system is inherently parallel**
  - Many connections between compute nodes and file system
  - Caches hide disk latency
- **Lustre file striping for additional parallel performance**
  - lfs setstripe –c 4 my_directory my_file
  - lfs getstripe my_file
- **MPI-IO**
  - Single file written by many separate MPI ranks
  - Cray has MPI-IO enhancements
- **HDF5**
  - Supports MPI-IO (if enabled)

Putting it all together

- **Well vectorized code approaches maximum performance per thread**
  - Elapsed time to solution is more important than excellent scaling
- **MPI and Threads on 24 core nodes (two 12-core processors):**
  - Can have **24 MPI ranks per node** with **no additional threading**
    - How far can MPI scale for problem being solved?
  - Can have **1 MPI rank per node** with **24 OpenMP threads per rank**
    - Can OpenMP sections scale to 24 threads?
  - Can have **6 MPI ranks per node** with **4 OpenMP threads per rank**
    - Watch out for thread affinity; export MPICH_CPUMASK_DISPLAY=1
  - Also **12**:**2**, **8**:**3**, **4**:**6**, ~~**3**:**8**~~, **2**:**12**
- **On KNL processor with 60+ cores**
  - Fast memory and/or DDR memory usage is another choice
  - Similar decisions for MPI ranks or threads per socket

What processor should I use?

- **Intel® Xeon® CPU E5-2698 v3 (“Haswell”)**
  - Well-established HPC processor
  - Compilers know how to produce fast code
- **Intel® MIC Architecture**
  - Potential for unprecedented performance
  - Compatible with known Xeon® architecture
  - Threading more important
- **GPU**
  - If movement of data between CPU memory and GPU memory can be minimized and data is reused many times on the GPU, potential speed is very good

Questions