Optimizing for HPC

- Some trends in HPC architectures
- How you can optimize your code for these architectures
- Q&A
High-Level Optimization

1. Science Problem
2. Choose Algorithms
3. Implement and Test Algorithms
4. Optimize Algorithms
5. Knowledge of System Architecture and Tools
6. Run high-performance code!
High-Level Optimization

1. Science Problem
2. Choose Algorithms
3. Optimize Algorithms
4. Implement and Test Algorithms
5. Knowledge of System Architecture and Tools
6. Run high-performance code!
High-Level Optimization

Science Problem

Choose Algorithms For the Target Architectures

Implement and Test Algorithms

Optimize Algorithms

Knowledge of System Architecture and Tools

Trade-offs between:
- Basis functions
- Resolution
- Lagrangian vs. Eulerian representations
- Renormalization and regularization schemes
- Solver techniques
- Evolved vs computed degrees of freedom
- And more…

Cannot be made by a compiler!

Run high-performance code!
Traditional computers are built to:
- Move data
- Make decisions
- Compute polynomials (of relatively-low order)

\[ f(x) = a_0 + a_1 x + a_2 x^2 + a_3 x^3 + a_4 x^4 \]
```c
#include <math.h>

double foo(double a0, double a1, double a2, double a3, double a4, double x) {
    return a0 + a1*x + a2*pow(x, 2) + a3*pow(x, 3) + a4*pow(x, 4);
}

$ gcc -O3 -S -o /tmp/f0.c
...  movsd  %xmm0, 8(%rsp)
movapd %xmm5, %xmm0
movsd  %xmm1, 56(%rsp)
movsd  .LC0(%rip), %xmm1
movsd  %xmm2, 48(%rsp)
movsd  %xmm3, 40(%rsp)
movsd  %xmm4, 32(%rsp)
movsd  %xmm5, 24(%rsp)
call  pow
...
call  pow
```

These calls are expensive!

Not useful work.

Yes, -ffast-math will fix this...
$ cat /tmp/f.c
double foo(double a0, double a1, double a2, double a3, double a4, double x) {
    return a0 + a1*x + a2*x*x + a3*x*x*x + a4*x*x*x*x;
}

$ gcc -O3 -S -o /tmp/f.c ...
mulsd %xmm5, %xmm1
mulsd %xmm5, %xmm2
mulsd %xmm5, %xmm4
mulsd %xmm5, %xmm3
addsd %xmm1, %xmm0
mulsd %xmm5, %xmm2
mulsd %xmm5, %xmm4
mulsd %xmm5, %xmm3
addsd %xmm2, %xmm0
mulsd %xmm5, %xmm2
movapd %xmm0, %xmm2
movapd %xmm4, %xmm0
addsd %xmm3, %xmm0
mulsd %xmm5, %xmm0
mulsd %xmm0, %xmm5
addsd %xmm5, %xmm2
movapd %xmm2, %xmm0
ret

This is better, but...
$ cat /tmp/f1.c
double foo(double a0, double a1, double a2, double a3, double a4, double x) {
    return a0 + x*(a1 + x*(a2 + x*(a3 + a4*x)));
}

$ gcc -O3 -S -o /tmp/f1.c
...
mulsd  %xmm5, %xmm4
addsd  %xmm4, %xmm3
mulsd  %xmm5, %xmm3
addsd  %xmm3, %xmm2
mulsd  %xmm5, %xmm2
addsd  %xmm2, %xmm1
mulsd  %xmm5, %xmm1
addsd  %xmm1, %xmm0
ret

And this is better, but...
Computer Architecture

```
$ cat /tmp/f1.c
double foo(double a0, double a1, double a2, double a3, double a4, double x) {
    return a0 + x*(a1 + x*(a2 + x*(a3 + a4*x)));
}
```

And remember the correct target flags...

```
$ gcc -O3 -S -o - /tmp/f1.c
...
mulsd  %xmm5, %xmm4
addsd  %xmm4, %xmm3
mulsd  %xmm5, %xmm3
addsd  %xmm3, %xmm2
mulsd  %xmm5, %xmm2
addsd  %xmm2, %xmm1
mulsd  %xmm5, %xmm1
addsd  %xmm1, %xmm0
ret
```

```
$ gcc -O3 -S -o - /tmp/f1.c -march=native
...
vfmadd231sd %xmm5, %xmm4, %xmm3
vfmadd231sd %xmm5, %xmm5, %xmm2
vfmadd231sd %xmm5, %xmm5, %xmm1
vfmadd231sd %xmm5, %xmm5, %xmm0
ret
```

PowerPC, etc. uses 
-mcpu= instead of 
-march=

A fused multiply-add
Computer Architecture

```c
$ cat /tmp/f1.c
double foo(double a0, ..., double x) {
    return a0 + x*(a1 + x*(a2 + x*(a3 + a4*x)));
}
t0 = fma(a4, x, a3)
t1 = fma(t0, x, a2)
t2 = fma(t1, x, a1)
t3 = fma(t2, x, a0)
return t3
```

But floating-point is complicated, so each operation cannot be completed in one clock cycle. ~6 clock cycles are needed.
t0 = fma(a4, x, a3)
Waiting…
Waiting…
Waiting…
Waiting…
t1 = fma(t0, x, a2)
...
t2 = fma(t1, x, a1)
...
t3 = fma(t2, x, a0)
...
return t3

A lot of computer architecture revolves around this question:
How do we put useful work here?
Hardware Threads

One way is to use hardware threads...

t0 = fma(a4, x, a3) [thread 0]
t0 = fma(a4, x, a3) [thread 1]
t0 = fma(a4, x, a3) [thread 2]
t0 = fma(a4, x, a3) [thread 3]
t0 = fma(a4, x, a3) [thread 4]
t0 = fma(a4, x, a3) [thread 5]
t1 = fma(t0, x, a2)
...
t2 = fma(t1, x, a1)
...
t3 = fma(t2, x, a0)
...
return t3

These can be OpenMP threads, pthreads, or, on a CPU, different processes.

How many threads do we need?
How much latency do we need to hide?
# Time Scales in Computing

## Latency Comparison Numbers

<table>
<thead>
<tr>
<th>Operation</th>
<th>Time</th>
<th>Latency</th>
</tr>
</thead>
<tbody>
<tr>
<td>L1 cache reference</td>
<td>0.5 ns</td>
<td></td>
</tr>
<tr>
<td>Branch mispredict</td>
<td>5 ns</td>
<td></td>
</tr>
<tr>
<td>L2 cache reference</td>
<td>7 ns</td>
<td></td>
</tr>
<tr>
<td>Mutex lock/unlock</td>
<td>25 ns</td>
<td></td>
</tr>
<tr>
<td>Main memory reference</td>
<td>100 ns</td>
<td></td>
</tr>
<tr>
<td>Compress 1K bytes with Zippy</td>
<td>3,000 ns</td>
<td>3 us</td>
</tr>
<tr>
<td>Send 1K bytes over 1 Gbps network</td>
<td>10,000 ns</td>
<td>10 us</td>
</tr>
<tr>
<td>Read 4K randomly from SSD*</td>
<td>150,000 ns</td>
<td>150 us</td>
</tr>
<tr>
<td>Read 1 MB sequentially from memory</td>
<td>250,000 ns</td>
<td>250 us</td>
</tr>
<tr>
<td>Round trip within same datacenter</td>
<td>500,000 ns</td>
<td>500 us</td>
</tr>
<tr>
<td>Read 1 MB sequentially from SSD*</td>
<td>1,000,000 ns</td>
<td>1,000 us</td>
</tr>
<tr>
<td>Disk seek</td>
<td>10,000,000 ns</td>
<td>10,000 us</td>
</tr>
<tr>
<td>Read 1 MB sequentially from disk</td>
<td>20,000,000 ns</td>
<td>20,000 us</td>
</tr>
<tr>
<td>Send packet CA-&gt;Netherlands-&gt;CA</td>
<td>150,000,000 ns</td>
<td>150,000 us</td>
</tr>
</tbody>
</table>
The IBM BG/Q network is fast...

- Each A/B/C/D/E link bandwidth: 4 GB/s
- Bisection bandwidth (32 racks): 13.1 TB/s
- HW latency
  - Best: 80 ns (nearest neighbor)
  - Worst: 3 µs (96-rack 20 PF system, 31 hops)
- MPI latency (zero-length, nearest-neighbor): 2.2 µs

MPI does add overhead which is generally minimal. If you’re sensitive to it, you can use PAMI (or the SPI interface) directly.
Supercomputing “Swim Lanes”

“Many Core” CPUs

GPUs


https://forum.beyond3d.com/threads/nvidia-pascal-speculation-thread.55552/page-4
Supercomputing “Swim Lanes”

“Many Core” CPUs

• 4 hardware threads per core
• To make up the rest, relies on:
  • OOO processing with branch prediction
  • Loop unrolling
  • SIMD (vectorization)

GPUs

• Lots of hardware threads
• Many hardware threads share the instruction stream (SIMT)

Many threads, but SIMT minimizes the per-thread control state/logic.
Some CUDA Terminology

- Dispatched threads are organized into a grid; all threads in a grid execute the same kernel function.
- A grid is decomposed into a 2D array of blocks (gridDim.x by gridDim.y).
- Each block is decomposed into a 3D array of threads (blockDim.x by blockDim.y by blockDim.z).
- The size of each block is limited to 1024 threads.
- Threads in different blocks cannot synchronize (using __syncthreads() - they might execute in any order).

```c
__global__ void add(int *a, int *b, int *c) {
    int index = threadIdx.x + blockIdx.x * blockDim.x;
    c[index] = a[index] + b[index];
}
```

Each thread has access to its coordinates and grid/block dimensions so it can figure out what to do.
• Threads are divided into groups of 32, called “warps” in NVIDIA's terminology (AMD calls these “wavefronts” - with a size of 64)
• All threads in each warp share many instruction stream resources (i.e. they have the same instruction pointer)
• The size of a warp is akin to the number of vector lanes on a CPU's SIMD unit
• Beware of branch divergence...

GPU Layout

GPUs have “cores”, but NVIDIA calls them “streaming multiprocessors” or SMs:
- Kepler: 15 SMs
- Maxwell: 24 SMs
- Pascal: 56 SMs

https://devblogs.nvidia.com/parallelforkall/inside-pascal/
• Single precision / SM
  • Pascal: 64
  • Kepler: 192
• Double precision / SM
  • Pascal: 32
  • Kepler: 64
• Max 64 warps / SM
• Max blocks / SM
  • Pascal: 32
  • Kepler: 16

https://devblogs.nvidia.com/parallelforall/inside-pascal/
Register Pressure!

- Each SM has a 256 KB register file
- Each thread can use up to 255 32-bit registers
- An SM running its maximum 2048 threads, however, could support only ~32 registers / thread!

```
t0 = fma(a4, x, a3)
t1 = fma(t0, x, a2)
t2 = fma(t1, x, a1)
t3 = fma(t2, x, a0)
return t3
```

This calculation needs ~3 registers: one for x, one for a\textless n\textgreater, one for t\textless n\textgreater.
But the compiler might use more by default! (see docs on \texttt{__launch_bounds__})

Unlike on a CPU, after the first 32, there is a significant cost to the incremental use of each register!
Loop Unrolling

CPUs have a fixed register file per thread, and the compiler can use that to hide latency...

```c
for (int i = 0; i < n; ++i) {
    x = Input[i]
    t0 = fma(a4, x, a3)
    t1 = fma(t0, x, a2)
    t2 = fma(t1, x, a1)
    t3 = fma(t2, x, a0)
    Output[i] = t3
}
```

```c
for (int i = 0; i < n; i += 2) {
    x = Input[i]
    y = Input[i+1]
    t0 = fma(a4, x, a3)
    u0 = fma(a4, y, a3)
    t1 = fma(t0, x, a2)
    u1 = fma(u0, y, a2)
    t2 = fma(t1, x, a1)
    u2 = fma(u1, y, a1)
    t3 = fma(t2, x, a0)
    u3 = fma(u2, y, a0)
    Output[i] = t3
    Output[i+1] = u3
}
```

If you need to tune this yourself, most compilers have a '#pragma unroll' feature.
CPU Registers

You can't unroll enough to completely hide anything but “on core” latencies (e.g. L1 cache hits and from FP pipeline) – you just don't have enough registers!

- x86_64 has 16 general-purpose registers (GPRs) – for scalar integer data, pointers, etc. – and 16 floating-point/vector registers
- With AVX-512 (e.g. with Knights Landing) there are 32 floating-point/vector registers
- AVX-512 also adds 8 operation mask registers
- PowerPC has 32 GPRs, 32 scalar floating-point registers and 32 vector registers (modern cores with VSX effectively combine these into 64 floating-point/vector registers)
OOO Execution and Loops

- CPUs, including Intel's Knights Landing, use out-of-order (OOO) execution to hide latency
- So to say that there are only 16 GPRs, for example, isn't the whole story: there are just 16 GPRs that the compiler can name

```c
for (int i = 0; i < n; ++i) {
    x = Input[i]
    t0 = fma(a4, x, a3)
    t1 = fma(t0, x, a2)
    t2 = fma(t1, x, a1)
    t3 = fma(t2, x, a0)
    Output[i] = t3
}
```

Processor can predict this will be true, and can start issuing instructions for multiple iterations at a time!
OOO Execution

- Importing to exploiting instruction-level parallelism (ILP) – each core's multiple pipelines
- Combined with branch prediction, can effectively provide a kind of dynamic loop unrolling
- Limited by the number of “rename buffer entries” (72 on Knights Landing)
- Limited by the number of “reorder buffer entries” (72 on Knights Landing)
- Mispredicted branches can lead to wasted work!
Vectorization: The Quad-Processing eXtension (QPX)

The first vector element in each vector register is the corresponding scalar FP register.

FP arithmetic completes in six cycles (and is fully pipelined). Loads/stores execute in the XU pipeline (same as all other load/stores).

(This is for the IBM BG/Q, but the picture is fairly generic)
SIMD: What does it mean?

Scalar

\[
\begin{align*}
X &\quad * \\
Y &
\end{align*}
\]

\[
X \times Y
\]

SIMD

\[
\begin{align*}
X3 &\quad X2 \quad X1 \quad X0 \\
Y3 &\quad Y2 \quad Y1 \quad Y0
\end{align*}
\]

\[
X3 \times Y3 \quad X2 \times Y2 \quad X1 \times Y1 \quad X0 \times Y0
\]


Auto-vectorization (or manual vectorization)
Vectors Have Many Types

- A 512-bit vector can hold 8 double-precision numbers, 16 single-precision numbers, etc.
- Different assembly instructions have different assumptions about the data types
- Except on the IBM BG/Q (where only FP is supported), both integer and FP types are supported

(This diagram is from the IBM POWER ISA manual, showing the 128-bit VSX registers)
KNL ISA

KNL implements all legacy instructions
- Legacy binary runs w/o recompilation
- KNC binary requires recompilation

KNL introduces AVX-512 Extensions
- 512-bit FP/Integer Vectors
- 32 registers, & 8 mask registers
- Gather/Scatter

Conflict Detection: Improves Vectorization
Prefetch: Gather and Scatter Prefetch
Exponential and Reciprocal Instructions

1. Previous Code name Intel® Xeon® processors
2. Xeon Phi = Intel® Xeon Phi™ processor

Avinash Dodani CGO PPoPP HPCA Keynote 2016
What's in AVX-512?

### AVX
- 256-bit basic FP
- 16 registers
- NDS (and AVX128)
- Improved blend
- MASKMOV
- Implicit unaligned

### AVX2
- Float16 (IVB 2012)
- 256-bit FP FMA
- 256-bit integer
- PERMD
- Gather

### AVX-512
- 512-bit FP/Integer
- 32 registers
- 8 mask registers
- Embedded rounding
- Embedded broadcast
- Scalar/SSE/AVX “promotions”
- HPC additions
- Transcendental support
- Gather/Scatter

### Future Processors (KNL & SKX)
- In planning, subject to change

---

for(i=0; i<16; i++) { A[B[i]]++;

index = vload &B[i] // Load 16 B[i]
old_val = vgather A, index // Grab A[B[i]]
new_val = vadd old_val, +1.0 // Compute new values
vscatter A, index, new_val // Update A[B[i]]
}

index = vload &B[i]
pending_elem = 0xFFFF;
do {
    currElem = get_conflict_free_subset(index, pending_elem)
    old_val = vgather {curr_elem} A, index // Grab A[B[i]]
    new_val = vadd old_val, +1.0 // Compute new values
    vscatter A {curr_elem}, index, new_val // Update A[B[i]]
    pending_elem = pending_elem ^ curr_elem // remove done idx
} while (pending_elem)

AVX-512 Conflict Detection
VPCONFLICT(D,Q) zmm1(k1), zmm2/mem
VPBROADCASTM(W2Q,D2Q) zmm1, k2
VPTESTNM(D,Q) zmm2, zmm3/mem
VPLZCNT(D,Q) zmm1 {k1}, zmm2/mem

The compiler, not you, should do this!
AVX-512 has 8 mask registers (64-bits each)
Why Masking is Special...

VADDPS ZMM0 {k1}, ZMM3, [mem]

- Mask bits used to:
  1. Suppress individual elements read from memory
     - hence not signaling any memory fault
  2. Avoid actual independent operations within an instruction happening
     - and hence not signaling any FP fault
  3. Avoid the individual destination elements being updated,
     - or alternatively, force them to zero (zeroing)

```
for (I in vector length)
{
    if (no_masking or mask[I]) {
        dest[I] = OP(src2, src3)
    } else {
        if (zeroing_masking)
            dest[I] = 0
        else
            // dest[I] is preserved
    
```

Caveat: vector shuffles do not suppress memory fault
Exceptions as mask refers to “output” not to “input”

AVX-512 Embedded Broadcasts

Embedded Broadcasts

VFMADD231PS zmm1, zmm2, C {1to16}

- Scalars from memory are first class citizens
  - Broadcast one scalar from memory into all vector elements before operation
- Memory fault suppression avoids fetching the scalar if no mask bit is set to 1

Other “tuples” supported

- Memory only touched if at least one consumer lane needs the data
- For instance, when broadcast a tuple of 4 elements, the semantics check for every element being really used
  - E.g.: element 1 checks for mask bits 1, 5, 9, 13, ...

\[
\text{float32 } A[N], B[N], C;
\]
\[
\text{for}(i=0; i<8; i++)
\]
\[
\{ \\
\quad \text{if}(A[i]! = 0.0) \\
\quad \quad A[i] = A[i] + C \times B[i]; \\
\}
\]

VBROADCASTSS zmm1 {k1}, [rax]
VBROADCASTF64X2 zmm2 {k1}, [rax]
VBROADCASTF32X4 zmm3 {k1}, [rax]
VBROADCASTF32X8 zmm4, {k1}, [rax]
...

Why Masking Matters?

void foo(float * restrict x, float * restrict y, float * restrict z, float * restrict v, float * restrict out, int n) {
    for (int i = 0; i < n; ++i) {
        float r2 = x[i]*x[i] + y[i]*y[i] + z[i]*z[i];
        if (r2 > eps) {
            out[i] = f(v[i], r2);
        } else {
            out[i] = 0;
        }
    }
}

Traditionally, a compiler could not autovectorize this! (not a pointer aliasing problem)

To vectorize, we essentially convert this into (m == # vector lanes):

\[
\begin{align*}
    r2[i:i+m] &= <r2_i, r2_{i+1}, \ldots > \\
    \text{out}[i:i+m] &= r2[i:i+m] > <\text{eps}, \text{eps}, \ldots > \ ? f(\text{v}[i:m], r2[i:i+m]) : <0, 0, \ldots >
\end{align*}
\]

Why? The compiler needs to deal with this (hypothetical) situation:

What if it were the case that the array “v” was not as long as x, y, and z (i.e. < n), but the programmer has arranged that (r2 > eps) will be false for all indices i invalid for the array v?

With AVX-512 masking, this is not a problem (we can mask off the access we don't need).

Note: Fortran (potentially) does not have this problem, even without masking (it knows the length of the arrays)!
void foo(float * restrict x, float * restrict y, float * restrict z, float * restrict v, float * restrict out, int n) {
    ...
}

“restrict” only a keyword in C. Use __restrict in C++

restrict means: Within the scope of the restrict-qualified variable, memory accessed through that pointer, or any pointer based on it, is not accessed through any pointer not based on it.
What programs do...

- Read data from memory
- Compute using that data
- Write results back to memory
- Communicate with other nodes and the outside world
Caches

- KNL cores are paired into a “tile”, which share an 1 MB L2 cache
- L2 cache can deliver 1 read cache line and 0.5 write cache lines per cycle
- Each core has its own 32 KB L1 I-cache and 32 KB L1 data cache
- The cache is “writeback” - the processor reads a cache line to write to it
- Each cache line is 64 bytes (the size of one 512-bit vector)

http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=7453080

NVIDIA Pascal has 64 KB shared memory per SM (see docs on __shared__)
Memory Requests

- CPU Load Pipeline
- L1 Data Cache
- Cache line containing x
- L2 Cache
- Cache line containing x
- Memory Controller (request coalescing buffer)
- (MC)DRAM

Always fetch whole cache lines (arrange your data accordingly)

GPUs have coalescing buffers here too, and they can afford to wait longer!
Tiles are arranged on a mesh
L2 caches are coherent, so we need tag directories to keep track of which tile owns which cache lines
How the cache lines are mapped to tag directories has three modes (selected at boot time): all-to-all, quadrant, and sub-NUMA clustering
KNL All-to-all mode

Address uniformly hashed across all distributed directories

No affinity between Tile, Directory and Memory

Most general mode. Lower performance than other modes.

Typical Read L2 miss
1. L2 miss encountered
2. Send request to the distributed directory
3. Miss in the directory. Forward to memory
4. Memory sends the data to the requestor
KNL Quadrant Mode

Chip divided into four virtual Quadrants

Address hashed to a Directory in the same quadrant as the Memory

Affinity between the Directory and Memory

Lower latency and higher BW than all-to-all. SW Transparent.

1) L2 miss, 2) Directory access, 3) Memory access, 4) Data return

Avinash Sodani CGO PPoPP HPCA Keynote 2016
KNL Sub-NUMA Clustering (SNC) Mode

Note, however, that quadrants are not symmetric!

Each Quadrant (Cluster) exposed as a separate NUMA domain to OS.

Affinity between Tile, Directory and Memory

Local communication. Lowest latency of all modes.

SW needs to NUMA optimize to get benefit.

Run one MPI rank per quadrant?

1) L2 miss, 2) Directory access, 3) Memory access, 4) Data return
HBM Modes

Three Modes. Selected at boot

**Cache Mode**
- 16GB MCDRAM
- DDR
- SW-Transparent, Mem-side cache
- Direct mapped. 64B lines.
- Tags part of line
- Covers whole DDR range

**Flat Mode**
- 16GB MCDRAM
- DDR
- MCDRAM as regular memory
- SW-Managed
- Same address space

**Hybrid Mode**
- 8 or 12GB MCDRAM
- DDR
- Part cache, Part memory
- 25% or 50% cache
- Benefits of both
How Flat Mode Looks

MCDRAM exposed as a separate NUMA node

Memory allocated in DDR by default \(\rightarrow\) Keeps non-critical data out of MCDRAM. Apps explicitly allocate critical data in MCDRAM. Using two methods:

- “Fast Malloc” functions in High BW library ([https://github.com/memkind/memkind](https://github.com/memkind/memkind))
- Built on top to existing `libnuma` API
- “FASTMEM” Compiler Annotation for Intel Fortran

Flat MCDRAM with existing NUMA support in Legacy OS

Avinash Sodani CGO PPoPP HPCA Keynote 2016
Flat Mode Memory Management

C/C++ *(https://github.com/memkind)*

Allocate into DDR

```c
float *fv;
fv = (float *)malloc(sizeof(float)*100);
```

Allocate into MCDRAM

```c
float *fv;
fv = (float *)hbw_malloc(sizeof(float) * 100);
```

Intel Fortran

Allocate into MCDRAM

```fortran
!DEC$ ATTRIBUTES FASTMEM :: A
NSIZE=1024
allocate array 'A' from MCDRAM
allocate (A(1:NSIZE))
```
Unified memory enables “lazy” transfer on demand – will mitigate/eliminate the “deep copy” problem!
CUDA UM (The Old Way)

**CPU Code**

```c
void sortfile(FILE *fp, int N) {
    char *data;
    data = (char *)malloc(N);
    fread(data, 1, N, fp);
    qsort(data, N, 1, compare);
    use_data(data);
    free(data);
}
```

**CUDA 6 Code with Unified Memory**

```c
void sortfile(FILE *fp, int N) {
    char *data;
    cudaMallocManaged(&data, N);
    fread(data, 1, N, fp);
    qsort<<<...>>>(data, N, 1, compare);
    cudaDeviceSynchronize();
    use_data(data);
    cudaFree(data);
}
```
Pointers are “the same” everywhere!
Types of parallelism

- Parallelism across nodes (using MPI, etc.)
- Parallelism across sockets within a node [Not applicable to the BG/Q, KNL, etc.]
- Parallelism across cores within each socket
- Parallelism across pipelines within each core (i.e. instruction-level parallelism)
- Parallelism across vector lanes within each pipeline (i.e. SIMD)
- Using instructions that perform multiple operations simultaneously (e.g. FMA)

Hardware threads tie in here too!
How fast can you go...

The speed at which you can compute is bounded by:

\[
\text{(the clock rate of the cores)} \times \text{(the amount of parallelism you can exploit)}
\]

- BG/Q: Fixed 1.66 GHz
- KNL: 1.30 GHz (dynamically scaled)
- Kepler: 0.8 GHZ
- Pascal: 1.30 GHz

Your hard work goes here...
Hardware Prefetcher

- L1 hardware prefetcher monitors access patterns and generates requests to the L2 in advance of anticipated need.
- L2 hardware prefetcher does the same, issuing requests to main memory.
- The KNL L2 prefetcher supports 48 independent streams (that's shared among all running threads). Running 4 hardware threads per core, two cores per tile: 6 streams per thread!
AOS vs. SOA

Structure of Arrays

```
struct Particles {
  float *x;
  float *y;
  float *z;
  float *w;
};
```

```
struct Particle {
  float x;
  float y;
  float z;
  float w;
};
```

Easy to vectorize; uses lots of prefetching streams!

Array of Structures

```
struct Particle *Particles;
```

Better cache locality; fewer prefetcher streams with scatter/gather support, maybe vectorization is not so bad!
Compiling

Basic optimization flags...

- **-O3** – Generally aggressive optimizations (try this first)
- **-g** – Always include debugging symbols (**really, always**! - when your run crashes at scale after running for hours, you want the core file to be useful)
- **-fopenmp** – Enable OpenMP (the pragmas will be ignored without this)
- **-ffast-math** (clang, gcc, etc.) – Enable “fast” math optimizations (most people don't need strict IEEE floating-point semantics).

If you don't use -O<n> to turn on some optimizations, most of the previous material is irrelevant!
An example... (what tuning might look like)

```c
void foo(double * restrict a, double * restrict b, etc.) {
    #pragma omp parallel for
    for (i = 0; i < n; ++i) {
        a[i] = e[i]*(b[i]*c[i] + d[i]) + f[i];
        m[i] = q[i]*(n[i]*o[i] + p[i]) + r[i];
    }
}
```

We use restrict here to tell the compiler that the arrays are disjoint in memory.

We likely want at least 2 threads per core, probably 4.

Split the loop

```c
void foo(double * restrict a, double * restrict b, etc.) {
    #pragma omp parallel for
    for (i = 0; i < n; ++i) {
        a[i] = e[i]*(b[i]*c[i] + d[i]) + f[i];
    }
    #pragma omp parallel for
    for (i = 0; i < n; ++i) {
        m[i] = q[i]*(n[i]*o[i] + p[i]) + r[i];
    }
}
```

Each statement requires 5 prefetcher streams, on some systems this is too many...

We could also change the data structures being used so that we have arrays of structures (although that might inhibit vectorization).
void foo(double * restrict a, double * restrict b, etc.) {
    #pragma omp parallel for
    for (i = 0; i < n; ++i) {
        a[i] = e[i]*(b[i]*c[i] + d[i]) + f[i];
    }
    #pragma omp parallel for
    for (i = 0; i < n; ++i) {
        m[i] = q[i]*(n[i]*o[i] + p[i]) + r[i];
    }
}

(don't actually split the parallel region)

We did a bit too much splitting here (starting each of these parallel regions can be expensive).

void foo(double * restrict a, double * restrict b, etc.) {
    #pragma omp parallel
    {
        #pragma omp for
        for (i = 0; i < n; ++i) {
            a[i] = e[i]*(b[i]*c[i] + d[i]) + f[i];
        }
        #pragma omp for
        for (i = 0; i < n; ++i) {
            m[i] = q[i]*(n[i]*o[i] + p[i]) + r[i];
        }
    }
}
void foo(double * restrict a, double * restrict b, etc.) {
    #pragma omp parallel
    {
        #pragma omp for
        for (i = 0; i < n; ++i) {
            a[i] = e[i]*(b[i]*c[i] + d[i]) + f[i];
        }
    }
    ...
schedule(dynamic) can be your friend...

```c
#pragma omp parallel for schedule(dynamic)
for (i = 0; i < n; i++) {
    unknown_amount_of_work(i);
}
```

You can use `schedule(dynamic, <n>)` to distribute in chunks of size n.

#pragma omp simd

Starting with OpenMP 4.0, OpenMP also supports explicit vectorization...

```c
char foo(char *A, int n) {
    int i;
    char x = 0;
    #pragma omp simd reduction(+:x)
    for (i=0; i<n; i++){
        x = x + A[i];
    }
    return x;
}
```

Can combine with threading...

```c
char foo(char *A, int n) {
    int i;
    char x = 0;
    #pragma omp parallel for simd reduction(+:x)
    for (i=0; i<n; i++){
        x = x + A[i];
    }
    return x;
}
```

Coarse Grained vs. Fine Grained Parallelism

Amdahl's law says the speedup is limited to: 
\[ \frac{1}{1 - p} \]. So if 5% of the program remains serial, then the speedup from parallelization is limited to 20x.

https://en.wikipedia.org/wiki/Amdahl%27s_law

This is expensive (many thousands of cycles)

This is expensive too!
C++17 Parallel Algorithms

- Parallel versions, and parallel+vectorized versions, of almost all standard algorithms (plus a few new ones)

<table>
<thead>
<tr>
<th>adjacent_difference</th>
<th>adjacent_find</th>
<th>all_of</th>
<th>any_of</th>
</tr>
</thead>
<tbody>
<tr>
<td>copy</td>
<td>copy_if</td>
<td>count</td>
<td></td>
</tr>
<tr>
<td>count_if</td>
<td>equal</td>
<td>exclusive_scan</td>
<td>fill</td>
</tr>
<tr>
<td>fill_n</td>
<td>find</td>
<td>find_end</td>
<td>find_first_of</td>
</tr>
<tr>
<td>find_if</td>
<td>find_if_not</td>
<td>for_each</td>
<td>for_each_n</td>
</tr>
<tr>
<td>generate</td>
<td>generate_n</td>
<td>includes</td>
<td>inclusive_scan</td>
</tr>
<tr>
<td>inner_product</td>
<td>inplace_merge</td>
<td>is_heap</td>
<td>is_heap_until</td>
</tr>
<tr>
<td>is_partitioned</td>
<td>is_sorted</td>
<td>is_sorted_until</td>
<td>lexicographical_compare</td>
</tr>
<tr>
<td>max_element</td>
<td>merge</td>
<td>min_element</td>
<td>minmax_element</td>
</tr>
<tr>
<td>mismatch</td>
<td>move</td>
<td>none_of</td>
<td>nth_element</td>
</tr>
<tr>
<td>partial_sort</td>
<td>partial_sort_copy</td>
<td>partition</td>
<td>partition_copy</td>
</tr>
<tr>
<td>reduce</td>
<td>remove</td>
<td>remove_copy</td>
<td>remove_copy_if</td>
</tr>
<tr>
<td>remove_if</td>
<td>replace</td>
<td>replace_copy</td>
<td>replace_copy_if</td>
</tr>
<tr>
<td>replace_if</td>
<td>reverse</td>
<td>reverse_copy</td>
<td>rotate</td>
</tr>
<tr>
<td>rotate_copy</td>
<td>search</td>
<td>search_n</td>
<td>set_difference</td>
</tr>
<tr>
<td>set_intersection</td>
<td>set_symmetric_difference</td>
<td>set_union</td>
<td>sort</td>
</tr>
<tr>
<td>stable_partition</td>
<td>stable_sort</td>
<td>swap_ranges</td>
<td>transform</td>
</tr>
<tr>
<td>transform_exclusive_scan</td>
<td>transform_inclusive_scan</td>
<td>transform_reduce</td>
<td>uninitialized_copy</td>
</tr>
<tr>
<td>uninitialized_copy_n</td>
<td>uninitialized_fill</td>
<td>uninitialized_fill_n</td>
<td>unique</td>
</tr>
</tbody>
</table>

Table 2 — Table of parallel algorithms

[Note: Not all algorithms in the Standard Library have counterparts in Table 2. — end note]
C++17 Parallel Algorithms

vector<float> a;

... for_each(par_seq, a.begin(), a.end(), [&](float &f) {
    f += 2.0;
});

Coming soon to a compiler near you!
OpenMP Evolving Toward Accelerators

Shared memory

Processor X
Cache
A

Memory
A

Processor Y
Cache
A

Distributed memory

Processor X
Cache
A

Memory X
A

Accelerator Y

Memory Y
A

Threads have access to a *shared* memory


New in OpenMP 4
int main(int argc, const char* argv[]) {
    float *x = (float*) malloc(n * sizeof(float));
    float *y = (float*) malloc(n * sizeof(float));
    // Define scalars n, a, b & initialize x, y

    for (int i = 0; i < n; ++i){
        y[i] = a*x[i] + y[i];
    }

    free(x); free(y); return 0;
}
OpenMP Accelerator Support – An Example (SAXPY)

int main(int argc, const char* argv[]) {
    float *x = (float*) malloc(n * sizeof(float));
    float *y = (float*) malloc(n * sizeof(float));
    // Define scalars n, a, b & initialize x, y

#pragma omp target data map(to:x[0:n])
{
#pragma omp target map(tofrom:y)
#pragma omp teams num teams(num blocks) num_threads(bsize)

#pragma omp distribute
for (int i = 0; i < n; i += num blocks)
{
    workshare (w/o barrier)

#pragma omp parallel for
for (int j = i; j < i + num blocks; j++)
{
    workshare (w/ barrier)
    y[j] = a*x[j] + y[j];
}
}
free(x); free(y); return 0;
}

Memory transfer if necessary.

Traditional CPU-targeted OpenMP might only need this directive!
OpenMP and UVM?

How does OpenMP accelerator support interact with unified memory? We don't yet know!
MKL, cuBLAS, ESSL, etc.

Vendors provide optimized math libraries for each system (BLAS for linear algebra, FFTs, and more).

- MKL on Intel systems, ESSL on IBM systems, cuBLAS (and others) for NVIDIA GPUs
- For FFTs, there is often an optional FFTW-compatible interface.
Using threads vs. multiple MPI ranks per node: it’s about...

- Memory
  - Sending data between ranks on the same node often involves “unnecessary” copying (unless using MPI-3 shared memory windows)
  - Similarly, your application may need to manage “unnecessary” ghost regions
  - MPI (and underlying components) have data structures that grow linearly (at best) with the total number of ranks

- And Memory
  - When threads can work together they can share resources instead of competing (cache, memory bandwidth, etc.)
  - Each process only gets a modest amount of memory per core

- And parallelism
  - You’ll likely see the best overall results from the scheme that exposes the most parallelism
And finally, be kind to the file system...

- Use MPI I/O - use collective I/O if the amounts being written are small
- Give each rank its own place within the file to store its data (avoid lock contention)
- Make sure you can validate your data (use CRCs, etc.), and then actually validate it when you read it
  (We've open-sourced a library for computing CRCs: http://trac.alcf.anl.gov/projects/hpcrc64/)

You probably want to design your files to be write optimized, not read optimized! Why?
You generally write more checkpoints than you read (and time reading from smaller jobs is “free”).
And writing is slower than reading.

And use load + broadcast instead of reading the same thing from every rank...

- Static linking is the default for all IBM BG/Q compilers for good reason... loading shared libraries from
tens of thousands of ranks may not be fast
- The same is true for programs using embedded scripting languages... loading lots of small script files
  from tens of thousands of ranks is even worse
## ALCF Systems

### How They Compare

<table>
<thead>
<tr>
<th></th>
<th>Mira</th>
<th>Theta</th>
<th>Aurora</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Peak Performance</strong></td>
<td>10 PF</td>
<td>&gt;8.5 PF</td>
<td>180 PF</td>
</tr>
<tr>
<td><strong>Compute Nodes</strong></td>
<td>49,152</td>
<td>&gt;2,500</td>
<td>&gt;50,000</td>
</tr>
<tr>
<td><strong>Processor</strong></td>
<td>PowerPC A2 1600 MHz</td>
<td>2nd Generation Intel Xeon Phi</td>
<td>3rd Generation Intel Xeon Phi</td>
</tr>
<tr>
<td><strong>System Memory</strong></td>
<td>768 TB</td>
<td>&gt;480 TB</td>
<td>&gt;7 PB</td>
</tr>
<tr>
<td><strong>File System Capacity</strong></td>
<td>26 PB</td>
<td>10 PB</td>
<td>&gt;150 PB</td>
</tr>
<tr>
<td><strong>File System Throughput</strong></td>
<td>300 GB/s</td>
<td>200 GB/s</td>
<td>&gt;1 TB/s</td>
</tr>
<tr>
<td><strong>Intel Architecture (x86-64) Compatibility</strong></td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td><strong>Peak Power Consumption</strong></td>
<td>4.8 MW</td>
<td>1.7 MW</td>
<td>&gt;13 MW</td>
</tr>
<tr>
<td><strong>GFLOPS/watt</strong></td>
<td>2.1</td>
<td>&gt;5</td>
<td>&gt;13</td>
</tr>
</tbody>
</table>

## Common Algorithm Classes in HPC

<table>
<thead>
<tr>
<th>Algorithm Science areas</th>
<th>Dense linear algebra</th>
<th>Sparse linear algebra</th>
<th>Spectral Methods (FFTs)</th>
<th>Particle Methods</th>
<th>Structured Grids</th>
<th>Unstructured or AMR Grids</th>
<th>Data Intensive</th>
</tr>
</thead>
<tbody>
<tr>
<td>Accelerator Science</td>
<td></td>
<td></td>
<td></td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>Astrophysics</td>
<td>X</td>
<td>X</td>
<td></td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>Chemistry</td>
<td>X</td>
<td>X</td>
<td></td>
<td>X</td>
<td></td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>Climate</td>
<td></td>
<td></td>
<td></td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>Combustion</td>
<td></td>
<td></td>
<td></td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>Fusion</td>
<td>X</td>
<td>X</td>
<td></td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>Lattice Gauge</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Material Science</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

http://crd.lbl.gov/assets/pubs_presos/CDS/ATG/WassermanSOTON.pdf
# Common Algorithm Classes in HPC - What do they need?

<table>
<thead>
<tr>
<th>Science areas</th>
<th>Algorithm</th>
<th>Dense linear algebra</th>
<th>Sparse linear algebra</th>
<th>Spectral Methods (FFT)s</th>
<th>Particle Methods</th>
<th>Structured Grids</th>
<th>Unstructured or AMR Grids</th>
<th>Data Intensive</th>
</tr>
</thead>
<tbody>
<tr>
<td>Accelerator Science</td>
<td></td>
<td>High</td>
<td></td>
<td>High</td>
<td>High performance memory system</td>
<td></td>
<td>Low latency, efficient gather /scatter</td>
<td>Storage, Network Infrastructure</td>
</tr>
<tr>
<td>Astrophysics</td>
<td></td>
<td>High performance memory system</td>
<td></td>
<td>High performance memory system</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Chemistry</td>
<td>High Flop/s rate</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Storage, Network Infrastructure</td>
<td></td>
</tr>
<tr>
<td>Climate</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>High flop/s rate</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Combustion</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Fusion</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Lattice Gauge</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Material Science</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Performance Limited By...

- Memory-Latency Bound (Pipeline better)
- Memory-Bandwidth Bound (Use a more-compressed representation)
- Compute Bound (Use a better algorithm)

- Using more registers
- Using more cache (increase cache locality)
These top four all use vector intrinsics!

HPC is dominated by C, C++ and Fortran for good reason!
How do we express parallelism?

![Bar chart showing programming models used at NERSC 2015](http://llvm-hpc2-workshop.github.io/slides/Tian.pdf)

- **MPI**: 90% - 100%
- **OpenMP**: 40% - 50%
- **Posix Threads**: 10% - 20%
- **PGAS**: Less than 10%

*Courtesy of Yun (Helen) He, Alice Koniges, et. al., (NERSC) at OpenMPCon'2015*
How do we express parallelism - MPI+X?

✓ OpenMP is about 50%, out of all choices of X

Courtesy of Yun (Helen) He, Alice Koniges, et. al., (NERSC) at OpenMPCon’2015

The Challenge of the Future: Power

The DOE wants 1 exaflop at < 20 MW

About Using HPC Systems...

- Why is HPC Hard?
- What can you do about it?
You need to share

Your job won't run right away...

These big jobs have been waiting for days

These were on hold for a while... I hope
You have a limited amount of time

- A large allocation on Mira (ALCF's production resource) is a few hundred million core hours
- \[100 \text{ M core hours} / (16 \text{ (cores per node)} \times 1024 \text{ (nodes per rack)} \times 48 \text{ (racks)}) = 127 \text{ hours}\]
- 127 hours is 5.3 days
- Running on the whole machine (48 racks) for 24 hours is the largest possible job
- Thus, with 5 jobs (plus some test runs), a 100-M-core-hour allocation could be gone
Supercomputers are not commodity machines

- Even when built from commodity parts, the configuration and scale are different
- The probability that you'll try to do something in your application that has never been tested before is high
- The system software will have bugs, and the hardware might too.
- The libraries on which your code depends might not be available.
Your jobs will fail

- The probability that a node will die, its DRAM will silently corrupt your data (including those in the storage subsystem), etc. is very low.
- However, if you spend a large fraction of your life running on large machines, you'll see these kinds of problems.
Your jobs will fail (cont.)

From the introduction of Fiala, et al. 2012:

- Servers tend to crash twice per year (2-4% failure rate) (Schroeder, et al. 2009). **HPC node hardware is designed to be somewhat more reliable, but...**

- 1-5% of disk drives die per year (Pinheiro, et al. 2007). **HPC storage systems use RAID, but...** *(also, many hardware RAID controllers don't do proper error checking, see Krioukov, et al. 2009 – older paper, but personal experience says this is still true today)*

- DRAM errors occur in 2% of all DIMMs per year (Schroeder, et al. 2009)

- ECC alone fails to detect a significant number of failures (Hwang, et al. 2012)
Most storage subsystem failures are not from disk failures (although many are).

Jiang, et al. 2008
Your jobs will fail (sometimes worse)

### Table 2. Estimated rates of UDEs in \( \frac{\text{UDEs}}{\text{I/O}} \).

<table>
<thead>
<tr>
<th>UDE Type</th>
<th>Estimated Rate</th>
<th>Nearline</th>
<th>Enterprise</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dropped I/O</td>
<td>( 9 \cdot 10^{-13} )</td>
<td>( 9 \cdot 10^{-14} )</td>
<td></td>
</tr>
<tr>
<td>Near-off Track I/O</td>
<td>( 10^{-13} )</td>
<td>( 10^{-14} )</td>
<td></td>
</tr>
<tr>
<td>Far-off Track I/O</td>
<td>( 10^{-12} )</td>
<td>( 10^{-13} )</td>
<td></td>
</tr>
</tbody>
</table>


So you should expect to see silent data corruption once in every (even with RAID): \( \frac{1}{(2 \times 10^{-13}) \times 4 \times 1024} = 2 \times 10^{16} \text{ bytes} \) (20 PB)

Mira's file system is \( \sim 28 \text{ PB} \), so this is not an unthinkable number (and, from personal experience, the rate is somewhat higher than that).
Debugging is hard

Your favorite debugger is great, but probably won’t run at scale...

Can you type in 10,000 terminals at once?
Debugging is hard (cont.)

Tools for debugging, profiling, etc. at scale are available, but they won't be what you're used to, and they might fail as well.
Everything is fast, but too slow...

- The network is fast, but likely slower than you'd like
- The same is true for the memory subsystem
- The same is true for the storage subsystem

Mira's file system can provide 240 GB/s – but you need to use the whole machine to get that rate. In addition, writing is slower than reading.

If you had the whole machine, and were the only one using the file system, then reading in enough to fill all 768 TB of DRAM would take:

\[(768 \times 1024 / 240) / 60 = 55 \text{ minutes}\]
Everything is fast, but too slow... (cont.)

Also remember that:

- There is a big difference between latency and bandwidth
- I/O performance tends to be quirky (memory performance does too, to a lesser extent)
- Load balancing can be tricky
So, now what do you do?
Experiment in parallel

This guy has the right idea

Some of these (time,size) combinations are better than others
Don't wait until the last minute...

Your allocation probably ends along with many others, and many users procrastinate, don't be one of them!

Jobs can be in the queue for more than a week!
Make your code exit!

“Begin at the beginning,” the King said, very gravely, “and go on till you come to the end: then stop.”
- Lewis Carroll, Alice in Wonderland

On Mira, 1 minute on the whole machine is 13,107 core hours!

- Take the time to figure out how long your code takes to run, and make it exit (don't always run your jobs until the system kills them).
- Exiting cleanly (using an exit code of 0), is often necessary for dependency chaining to work.
Save your work

- Save all of your configuration files for any experiments you do.
- Save your log files.
- Document how to run your code and process the results, what you've actually run, and where the data lives.

Store the run configuration in every output file (as comments, metadata, etc.)
Serious, just do it...
Optimize your code

- Read some documentation on the system, compiler, etc.
- Do you need strict IEEE floating point semantics? If not, turn them off.
- Profile your code
- Then optimize (using better algorithms first)

Always compile with -g (debugging symbols)
Essential for debugging if your code crashes, but also useful for profiling, etc.
Contact support

If something is wrong, or you need help for any other reason, contact the facility's support service:

(system reservations for debugging are often possible, just ask!)

These people are not scary!
On using libraries

Using libraries written by experts is really important, but remember that if you're using something obscure, you'll “own” that dependency.

Some popular libraries:
- Trilinos
- PETSc
- HDF5
- FFTW
- BLAS/LAPACK

Taking the road less traveled is often not a good idea

Programming-Language Features

- Be careful when using the latest-and-greatest programming-language features
- We're just getting C++11 and Fortran 2008 support in compilers now (not counting co-arrays)

```cpp
// C++14: new expressive power
auto size = [](const auto& m) { return m.size(); };
```

Yeah, not so fast...

(the facilities try their best to support these things using open-source compilers, etc. but, as a user, you'll likely want the option of using the vendor's compilers)
Validation

- Have test problems, hopefully both small and large ones, and run them on the system, with the same binary you plan to use for production, before starting your production science.
- Make sure all of the data files have checksums (CRCs) so that you can validate that the data you wrote is the same as the data you read in.
- Build physical diagnostics into your simulations (conservation of energy, power spectrum calculations, etc.) and actually check them.
Save your build settings

➔ Make your code print out or save its configuration when it starts, and also:
  ➔ The compilers and build flags used
  ➔ The version control revision information for the source being built

If you don't know what version control is, learn about git.

There is no general automatic way to do these things:
You'll need to hack your build system. It will be worth it.
Some thoughts on testing

- Make as much of your code testable at small scale as possible.
- Unit testing is trendy for a good reason.
- Learn how to use Valgrind, and run your code at small scale with it.
- Add print statements in your code for anomalous situations: lots of them.
- Make sure you actually check for error return codes on routines that have them (for MPI, communication failures will kill your application by default, file I/O errors won't).
Avoid the network

- Network bandwidth, relative to FLOPS, is decreasing
- Choose, to the extent possible, communication-avoiding algorithms
- If your problem has multiple physical time/length scales, try to separate out the shorter/faster ones and keep them rank-local (local sub-cycling).
- More generally, learn about split-operator methods.
Avoid central coordinators

A scheme like this is highly unlikely to scale!
Load Balancing

- Keep "work units" being distributed between ranks as large as possible, but try hard to keep everything load balanced.
- Think about load balancing early in your application design: it is the largest impediment to scaling on large systems.

This is not good; rank 0 has much more work.
Memory Bandwidth

- Memory bandwidth will be low compared to the compute capability of the machine.
- To get the most out of the machine, you'll need to use FMA instructions (these machines were built to evaluate polynomials, many of them in parallel, so try to cast what you're doing in those terms).
- Try to do as much as possible with every data value you load, and remember that gathering data from all over memory is expensive.

\[ \text{FMA} = \text{Fused Multiply Add} = (a \times b) + c \] [with no intermediate rounding]
Expensive Algorithms

- Don't dismiss seemingly-expensive algorithms without benchmarking them (higher-order solvers, forward uncertainty propagation, etc. all might have high data reuse so the extra computational expense might be “free”).
- If you're using an iterative solver, the number of iterations you use will often dominate over the expense of each iteration.
More on I/O

- Make your files “write optimized”.
- Don’t use one file per rank, but don’t have all ranks necessarily write to the same file either: make the number of files configurable.
- The optimal mapping between ranks and files will be system specific (ask the system experts what this is).
- There is often lock contention on blocks, files, directories, etc.
- Pre-allocate your file extents when possible.
- Use collective MPI I/O when the amount of data per rank is small (a few MB or less per rank).

This is a pain, I know.
Some final advice...

Don't guess! Profile! Your performance bottlenecks might be very different on different systems.

And don't be afraid to ask questions...

Any questions?

Come to the focus session tonight!