# Performance Optimization I: Single Core/Node Vectorization, Memory - Overview and BG/Q


**Hal Finkel**

hfinkel@anl.gov

Leadership Computing Facility

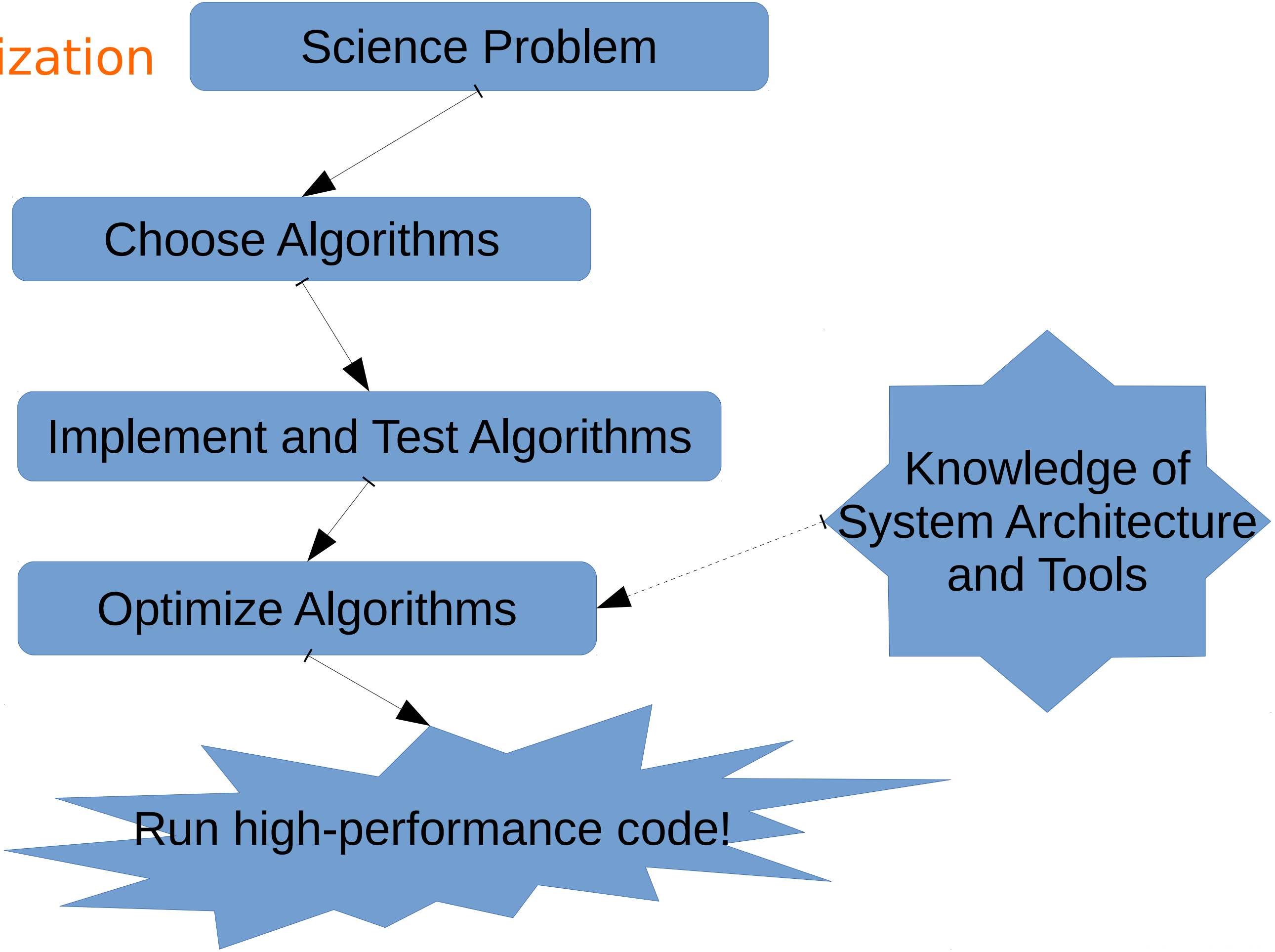
ALCF Computational Performance Workshop – May 2, 2017

# Optimizing for HPC

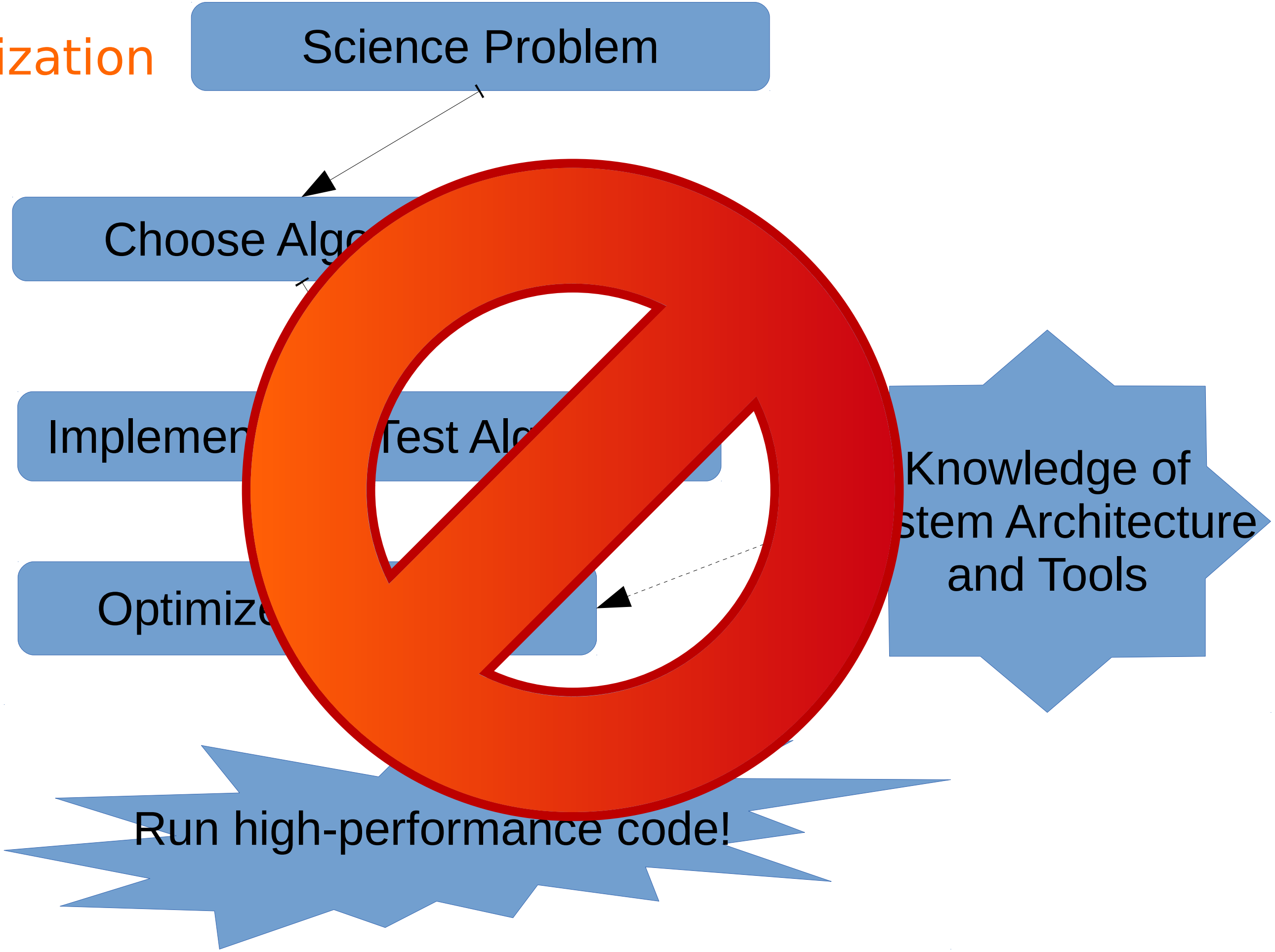
You want to know how
to make me compute quickly...


- Some trends in HPC architectures
- How you can optimize your code for these architectures (specifically the IBM BG/Q (Mira) and the Intel Xeon Phi (Theta)

# High-Level Optimization



# High-Level Optimization



# High-Level Optimization

Science Problem

Choose Algorithms
For the Target Architectures

Implement and Test Algorithms

Optimize Algorithms

Run high-performance code!

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!

Knowledge of System Architecture and Tools

# Compiling

When compiling your programs, please use our MPI wrappers (these are the softenv keys)...

(generally best performance)

- +mpiwrapper-xl.legacy
- +mpiwrapper-xl
- +mpiwrapper-bgclang.legacy
- +mpiwrapper-bgclang
- +mpiwrapper-gcc.legacy
- +mpiwrapper-gcc

(generally worst performance)

The “legacy” MPI gives the best performance unless you're using MPI_THREAD_MULTIPLE

bgclang has better C++ support than xl and gcc, but has no Fortran support (yet)

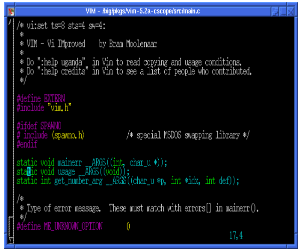
# Compiling

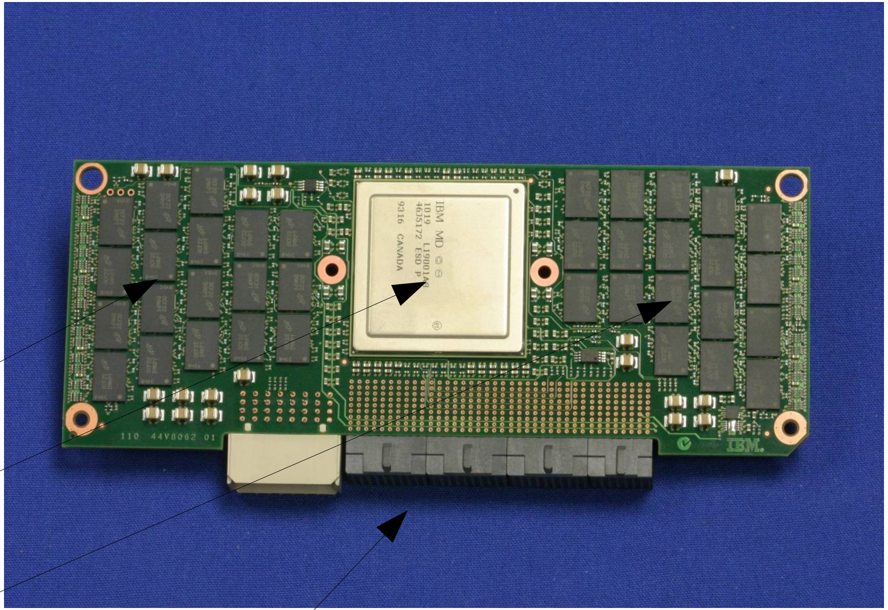
Basic optimization flags...

- -O3 – Generally aggressive optimizations (try this first: it is typically the best tested of all compiler optimization levels)
- -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)
- -qsmp=omp (xl) -fopenmp (bgclang and gcc) – Enable OpenMP (the pragmas will be ignored without this)
- -qnostrict (xl) -ffast-math (bgclang and gcc) – Enable “fast” math optimizations (most people don't need strict IEEE floating-point semantics). xl enables this by default at -O3 and above and you need to pass -qstrict to turn it off.

If you don't use -O<n> to turn on some optimizations, most of the previous material is irrelevant!

# What programs do...



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

# How fast can you go...

The speed at which you can compute is bounded by:

(the clock rate of the cores) x (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...

# There is only one sock

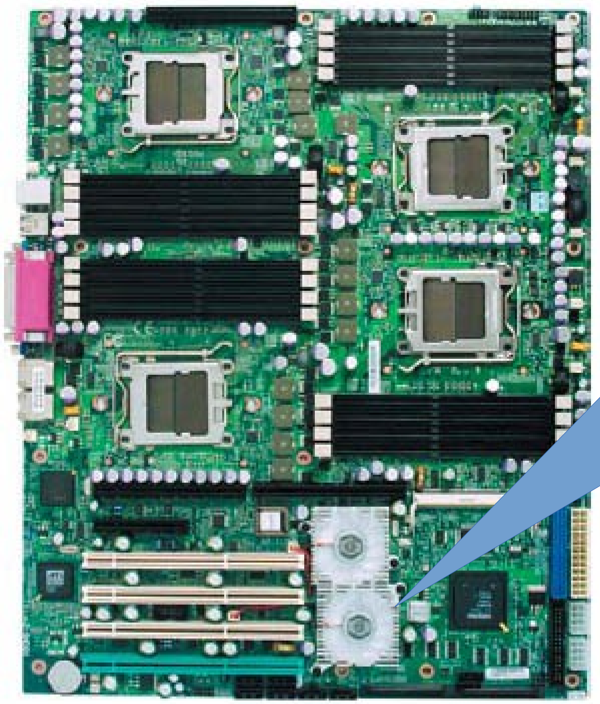
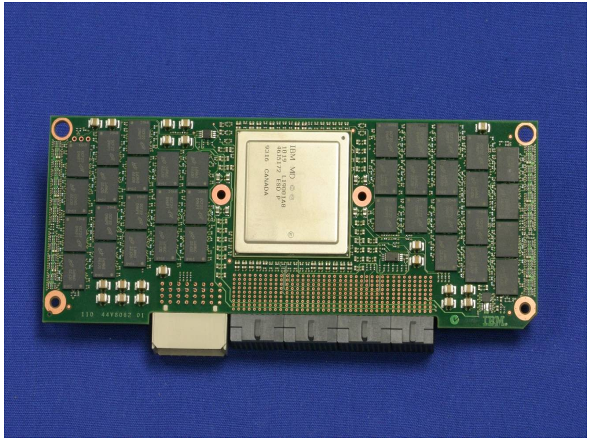


Image source: https://computing.llnl.gov/tutorials/linux_clusters/

Commodity HPC node with four sockets

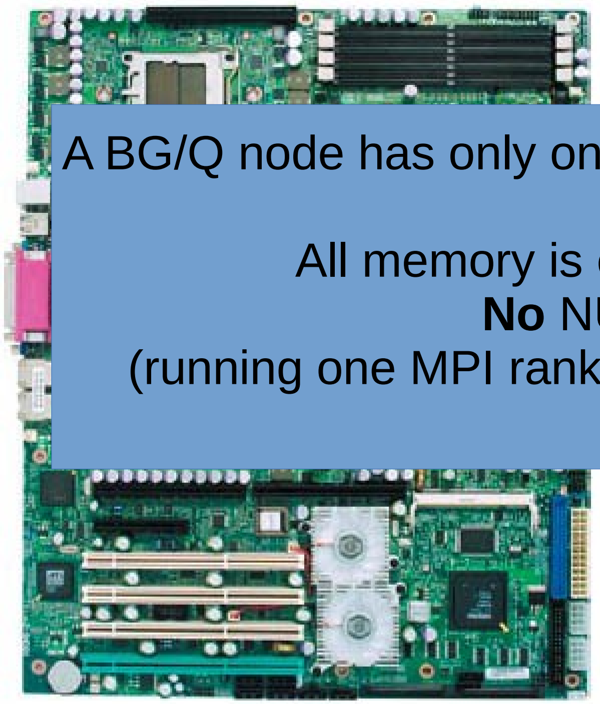
**Has** nonuniform memory access (NUMA):
each core has DRAM to which it is closer
(running multiple MPI ranks per node, one per socket, is probably best)

**Not** a BG/Q node



Argonne NATIONAL LABORATORY

# There is only one socket

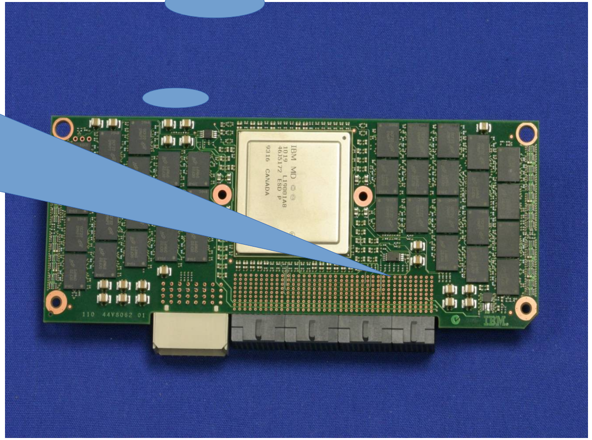


A BG/Q node has only one “socket” with one CPU

All memory is equally close:
**No** NUMA
(running one MPI rank per node works well)

Image source: https://computing.llnl.gov/tutorials/linux_clusters/

A BG/Q node



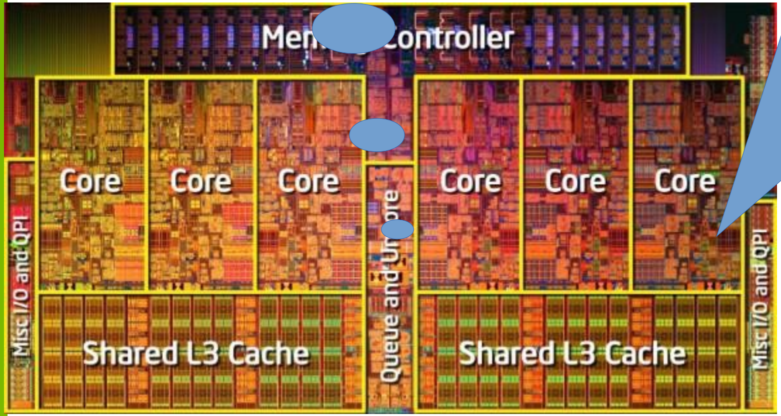
A BG/Q Node has:

- 1 PowerPC A2Q CPU
- 16 GB DDR3 DRAM

# There are 16 cores per node

**Not** a BG/Q core

Commodity HPC CPUs typically have only 4 - 12 cores (and the operating system does not have a dedicated core)



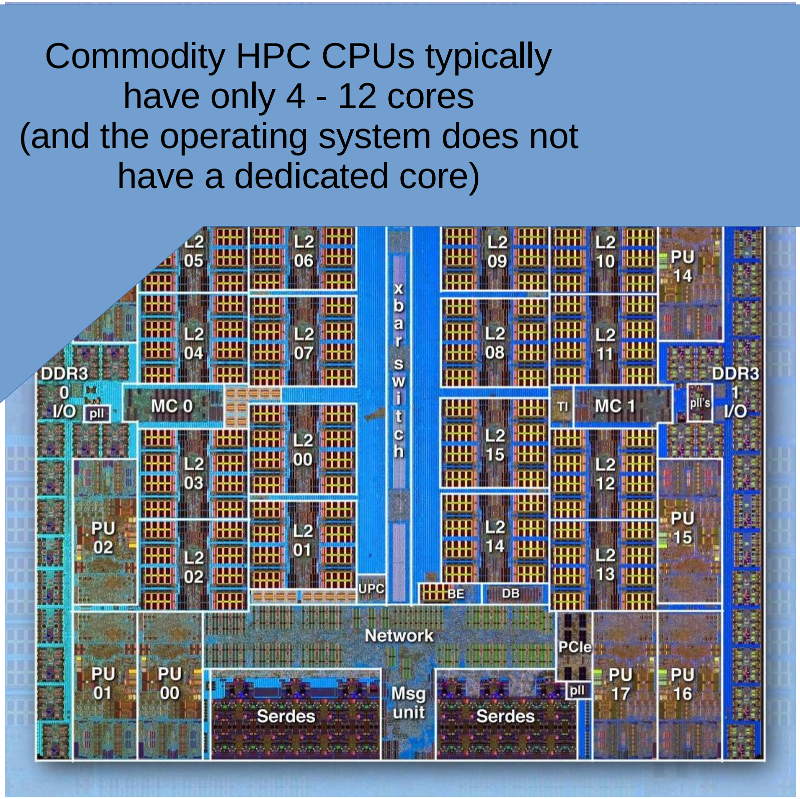
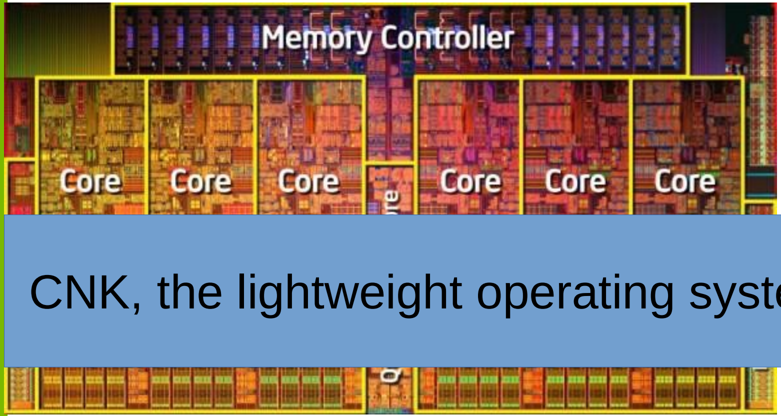



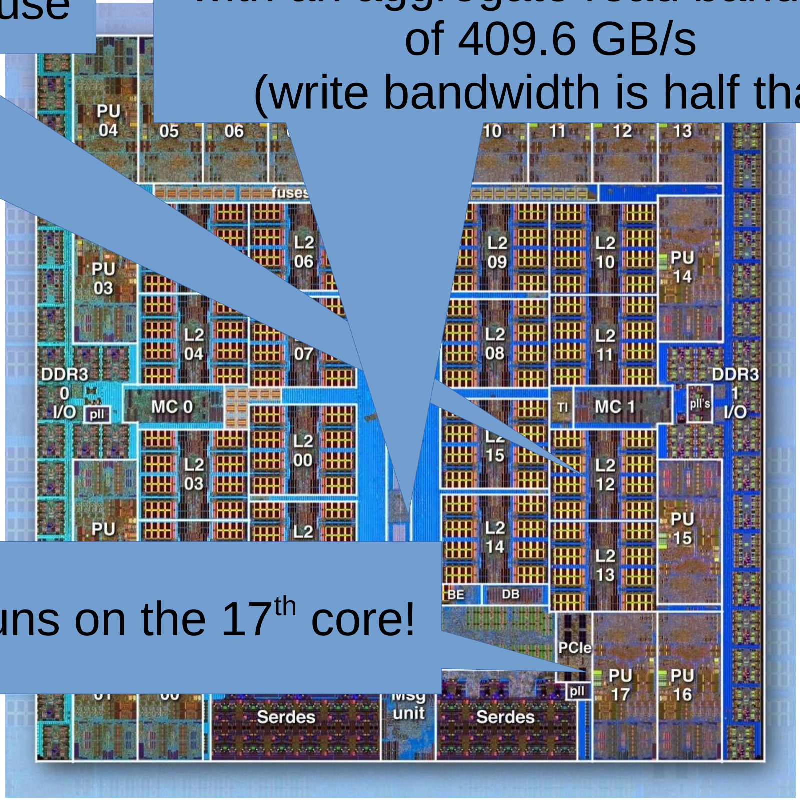

Image source: https://computing.llnl.gov/tutorials/linux_clusters/

# There are 16 cores per node

Each BG/Q CPU has 16 cores you can use

The cores are connected by a cross-bar interconnect with an aggregate read bandwidth of 409.6 GB/s (write bandwidth is half that)



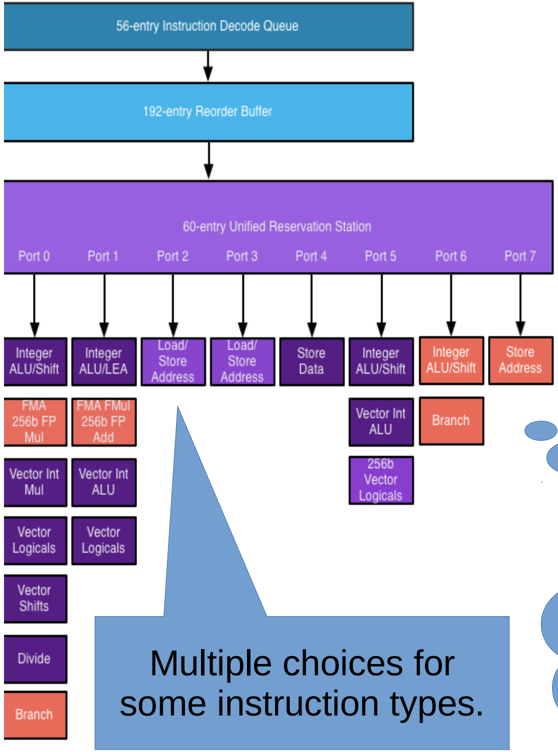




CNK, the lightweight operating system, runs on the 17th core!

Image source: https://computing.llnl.gov/tutorials/linux_clusters/

# There are two pipeline

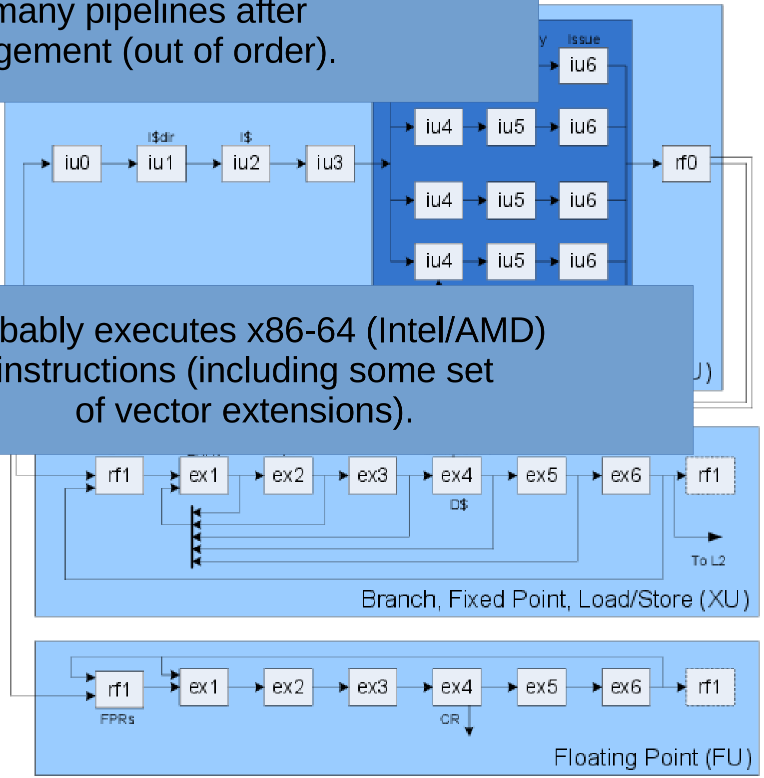
In commodity HPC cores, instructions are dispatched to many pipelines after dynamic rearrangement (out of order).




Multiple choices for some instruction types.

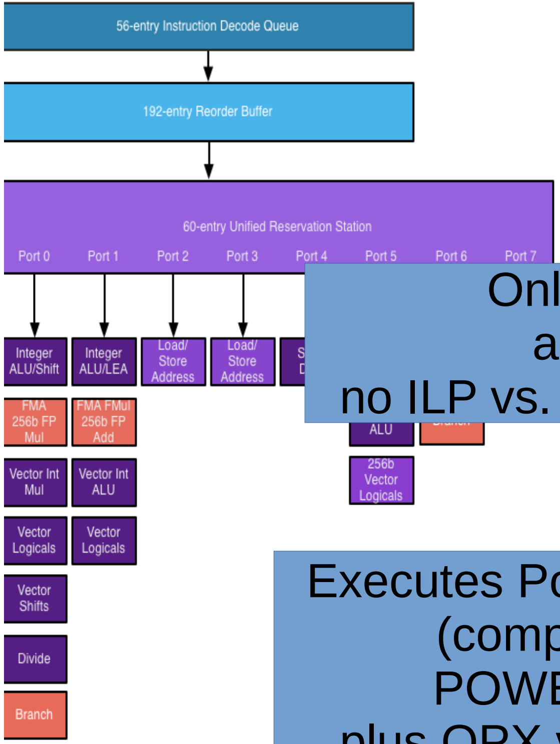
Probably executes x86-64 (Intel/AMD) instructions (including some set of vector extensions).

**Not** a BG/Q core

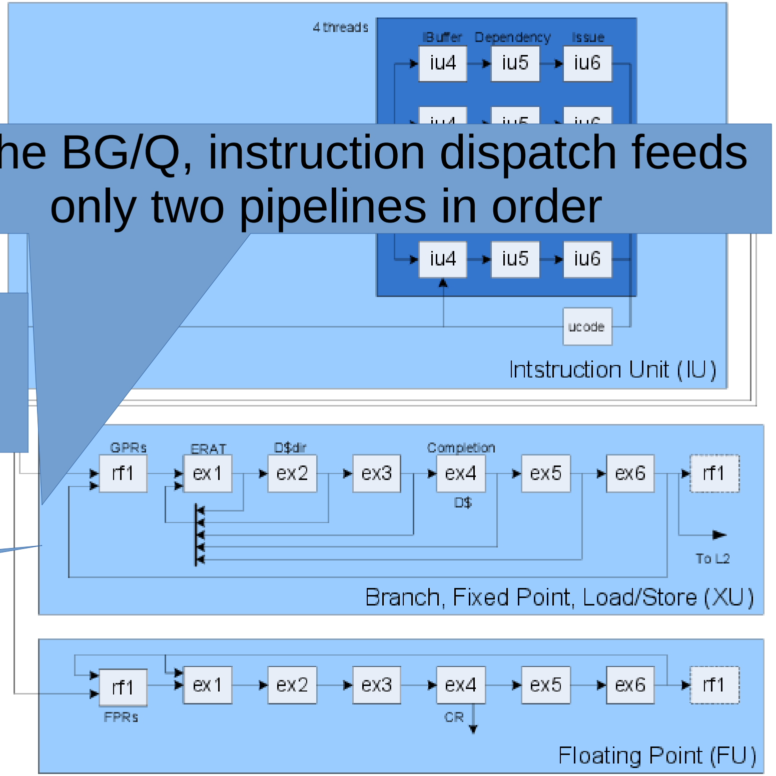



Argonne NATIONAL LABORATORY

# There are two pipelines per core




## PowerPC A2 Core:




On the BG/Q, instruction dispatch feeds only two pipelines in order

Only one choice for any instruction:
no ILP vs. vectorization tradeoffs!

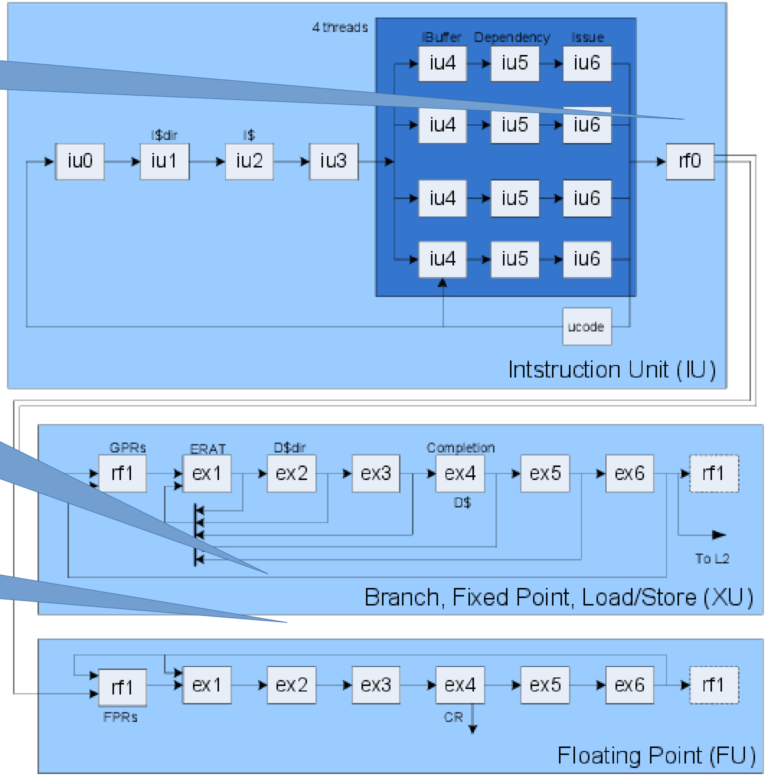
Executes PowerPC instructions (complying with the POWER ISA v2.06)
plus QPX vector instructions

# There are four hardware threads per core

Instructions from the four hardware threads are dispatched round-robin

The four threads share essentially all resources (except the register file)

The two pipelines can simultaneously start two instructions, but they must come from two different threads



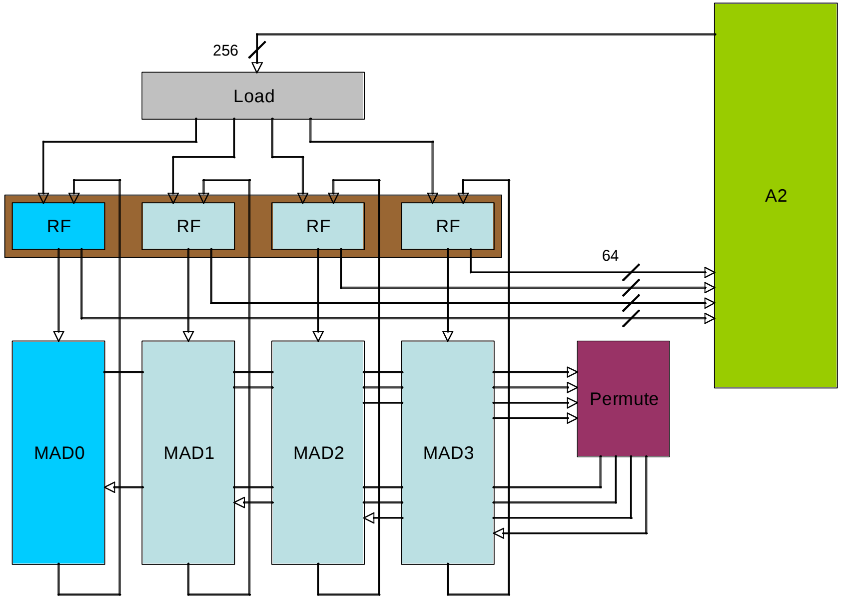

**You must have at least two threads (or processes) per core to efficiently use the BG/Q!**

Argonne NATIONAL LABORATORY

# Vectorization: The Quad-Processing eXtension (QPX)

- On the BG/Q, only QPX vector instructions are supported!
- Only <4 x double>, <4 x float> and <4 x bool> operations are provided.
- The only advantage of single precision over double precision is decreased memory bandwidth/footprint.

On commodity HPC hardware, integer operations can also be vectorized, but not on the BG/Q.



# Fused Multiply Add Instructions (FMA)

There are some FP (vector) instructions that combine both a multiply and an add/subtract into one instruction!

Many variants like these:

**qvfmadd:**

QRT0 ← [(QRA0)×(QRC0)] + (QRB0)
QRT1 ← [(QRA1)×(QRC1)] + (QRB1)
QRT2 ← [(QRA2)×(QRC2)] + (QRB2)
QRT3 ← [(QRA3)×(QRC3)] + (QRB3)

**qvfmsub:**

QRT0 ← [(QRA0)×(QRC0)] - (QRB0)
QRT1 ← [(QRA1)×(QRC1)] - (QRB1)
QRT2 ← [(QRA2)×(QRC2)] - (QRB2)
QRT3 ← [(QRA3)×(QRC3)] - (QRB3)

And a few like these with built-in permutations:

**qvfxmadd:**

QRT0 ← [(QRA0)×(QRC0)] + (QRB0)
QRT1 ← [(QRA0)×(QRC1)] + (QRB1)
QRT2 ← [(QRA2)×(QRC2)] + (QRB2)
QRT3 ← [(QRA2)×(QRC3)] + (QRB3)

**qvfxxnpmadd:**

QRT0 ← - ([(QRA1)×(QRC1)] - (QRB0) )
QRT1 ← [(QRA0)×(QRC1)] + (QRB1)
QRT2 ← - ([(QRA3)×(QRC3)] - (QRB2) )
QRT3 ← [(QRA2)×(QRC3)] + (QRB3)

Argonne NATIONAL LABORATORY

# Putting it all together...

You must vectorize to achieve
The peak FLOP rate
(on future machines, this factor
will be even larger)

You can only achieve the peak FLOP
rate using FMAs
(usually true on commodity hardware too)

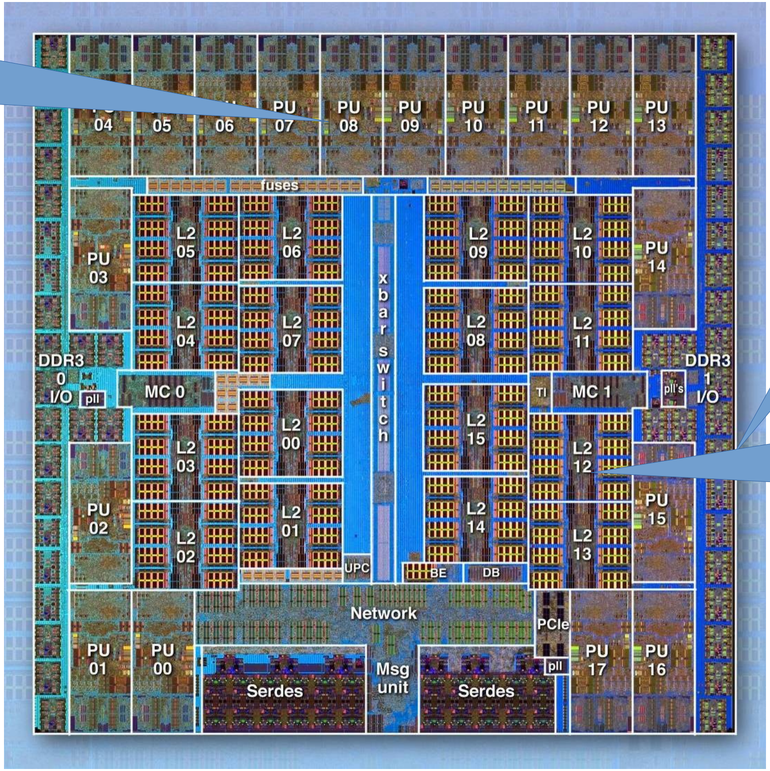
Peak FLOPS: (1.66 GHz) x (16 cores) x (4 vector lanes) x (2 operations per FMA) = 212.48 GFLOPS/node.

Remember you must use at least two
hardware threads (or processes)
or else you won't be able to
saturate the floating-point pipeline
in practice

Note: this is an order of magnitude
(on future machines, it will be nearly
two orders of magnitude)

# Memory

L1 cache and L1P internal buffer (per core)




DDR3 DRAM (2 controllers)

Commodity HPC cores often also have an L3 cache; we don't. However, they have an L2 cache that is only hundreds of KB.

L2 cache (16 slices) 16 MB in total

# 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!

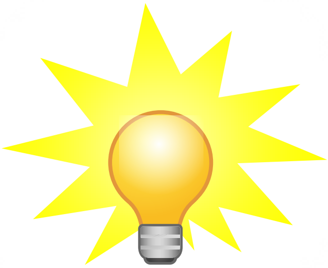
Argonne NATIONAL LABORATORY

# Computer Architecture

Traditional computers are built to:

- Move data
- Make decisions
- Compute polynomials (of relatively-low order)

$$f(x) = a_0 + a_1 x + a_2 x^2 + a_3 x^3 + a_4 x^4$$

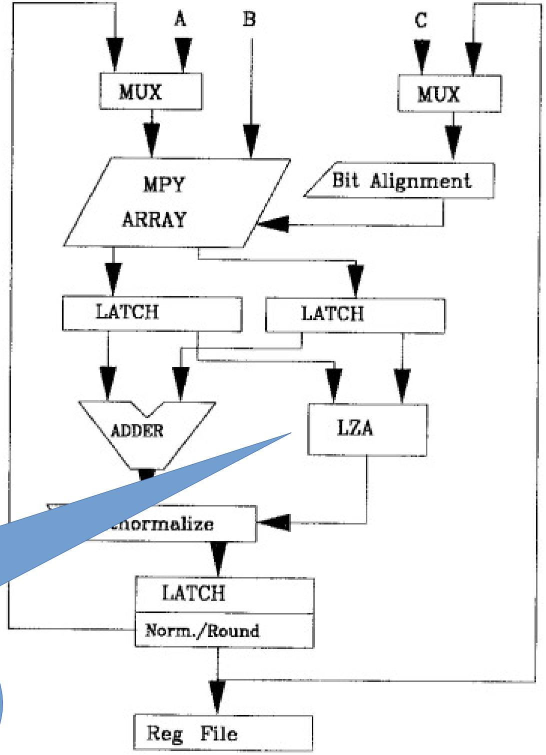


# Computer Architecture

```
$ 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
```

IEEE JOURNAL OF SOLID-STATE CIRCUITS, VOL. 25, NO. 5, OCTOBER 1990




But floating-point is complicated, so each operation cannot be completed in one clock cycle. ~6 clock cycles are needed.

Argonne NATIONAL LABORATORY

# Computer Architecture

But this is not good…

t0 = fma(a4, x, a3)
Waiting…
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?

Argonne NATIONAL LABORATORY

# 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?

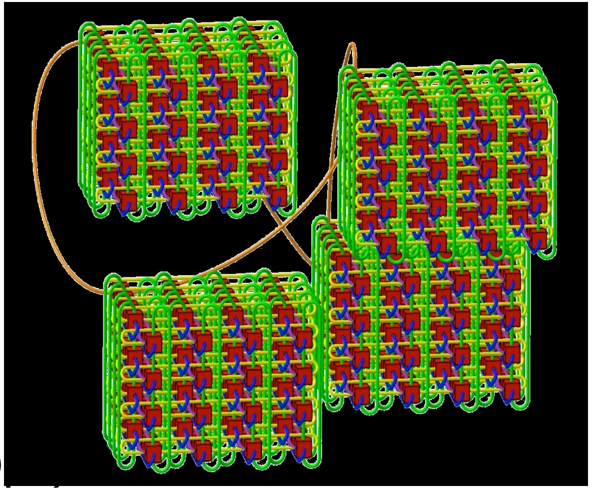
# Time Scales in Computing

```
Latency Comparison Numbers
--------------------------
L1 cache reference                           0.5 ns
Branch mispredict                            5   ns
L2 cache reference                           7   ns
Mutex lock/unlock                           25   ns
Main memory reference                      100   ns
Compress 1K bytes with Zippy             3,000   ns        3 us
Send 1K bytes over 1 Gbps network       10,000   ns       10 us
Read 4K randomly from SSD*             150,000   ns      150 us          ~1GB/sec SSD
Read 1 MB sequentially from memory     250,000   ns      250 us
Round trip within same datacenter      500,000   ns      500 us
Read 1 MB sequentially from SSD*     1,000,000   ns    1,000 us    1 ms  ~1GB/sec SSD
Disk seek                           10,000,000   ns   10,000 us   10 ms
Read 1 MB sequentially from disk    20,000,000   ns   20,000 us   20 ms   80x memory
Send packet CA->Netherlands->CA    150,000,000   ns  150,000 us  150 ms
```

Latency Numbers Every Programmer Should Know: https://gist.github.com/jboner/2841832

Argonne NATIONAL LABORATORY

# 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 ho
- 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

# Loop Unrolling

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

```
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
}
```

unroll by 2

```
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
}
```

I hope these are in cache

Each pair is independent,
so no waiting in between
dispatches00

Showing unroll by 2 so it fits on the slide,
you need to unroll by more to fully
hide FP or L1 latency

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!

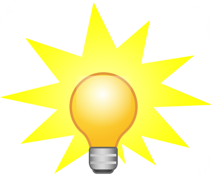
- x86_64 has 16 general-purpose registers (GPRs) – for scalar integer data, pointers, etc. – and 16 floating-point/vector registers
- With AVX-512 (e.g. with Knights Landing) there are 32 floating-point/vector registers
- AVX-512 also adds 8 operation mask registers
- PowerPC has 32 GPRs, 32 scalar floating-point 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

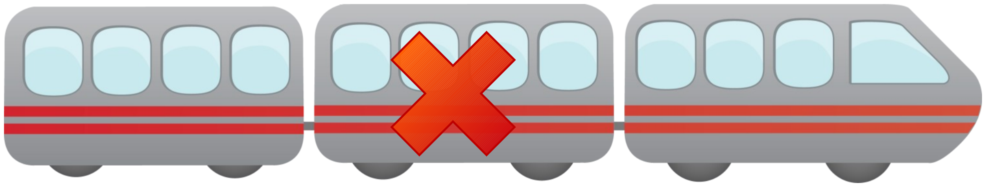
```
for (int i = 0; i < n; ++i) {
  x = Input[i]
  t0 = fma(a4, x, a3)
  t1 = fma(t0, x, a2)
  t2 = fma(t1, x, a1)
  t3 = fma(t2, x, a0)
  Output[i] = t3
}
```

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

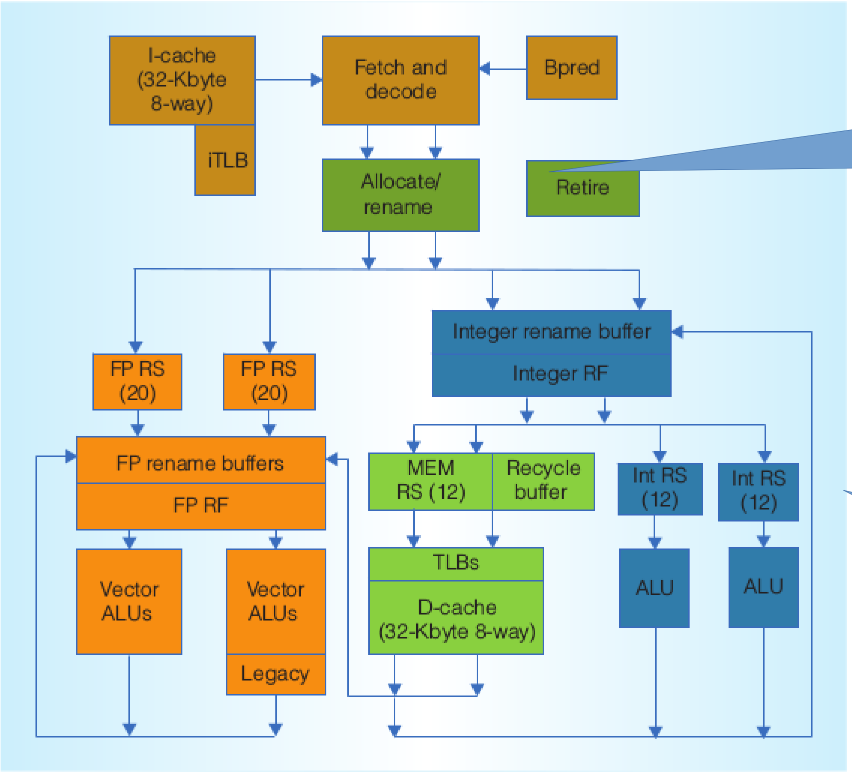


# OOO Execution

- 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!



# KNL Pipeline




Fetch/decode 16 bytes per cycle
(i.e. two instructions per cycle)
Careful: AVX-512 instructions can be up to 12 bytes each if they have non-compressed displacements!

2 FP/vector operations,
2 memory operations,
and 2 scalar integer operations per cycle!

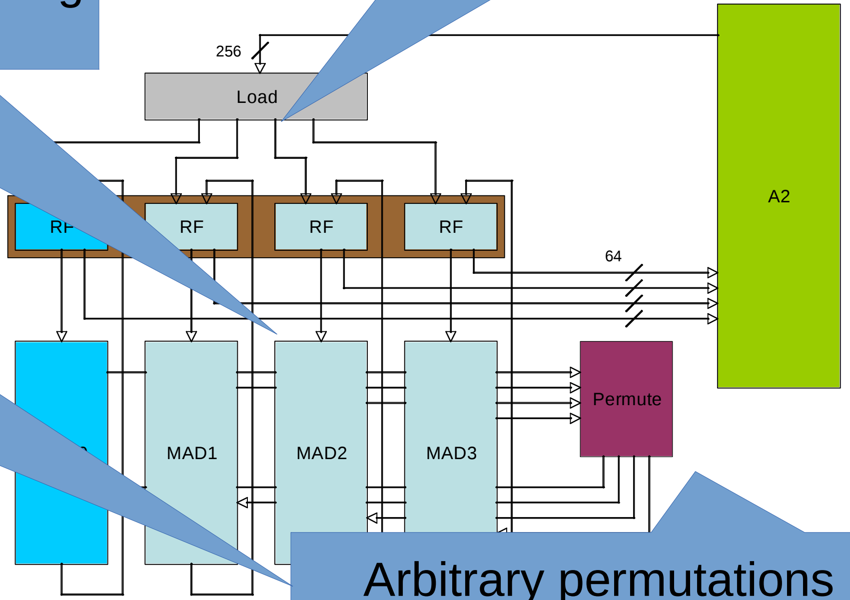
http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=7453080

# Vectorization: The Quad-Processing eXtension

32 QPX registers (and 32 general purpose registers) per thread

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

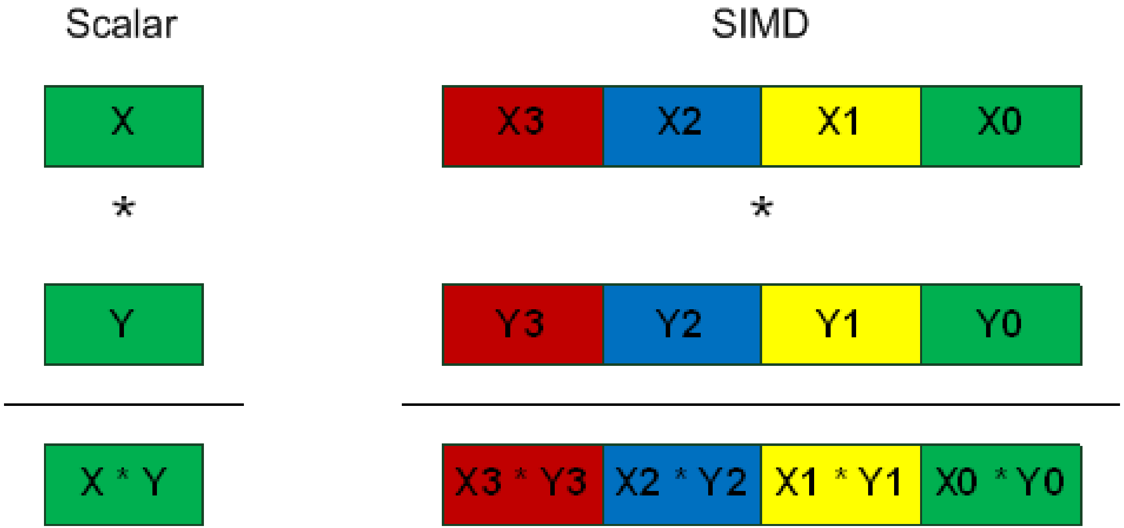
FP arithmetic completes in six cycles (and is fully pipelined).
Loads/stores execute in the XU pipeline (same as all other load/stores).




Arbitrary permutations complete in only two cycles.

(This is for the IBM BG/Q, but the picture is fairly generic)

# SIMD: What does it mean?



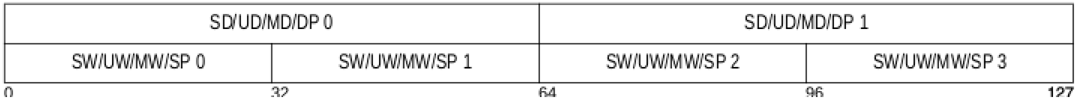

https://software.intel.com/en-us/articles/ticker-tape-part-2

Autovectorization (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

The same vector register can be divided in different ways




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

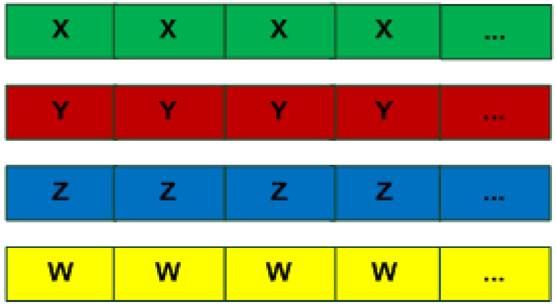
| | | Vector Length | | |
|---|---|---|---|---|
| | | 128 | 256 | 512 |
| element size | Byte | 16 | 32 | 64 |
| | Word | 8 | 16 | 32 |
| | Dword/SP | 4 | 8 | 16 |
| | Qword/DP | 2 | 4 | 8 |

Argonne NATIONAL LABORATORY

# AOS vs. SOA

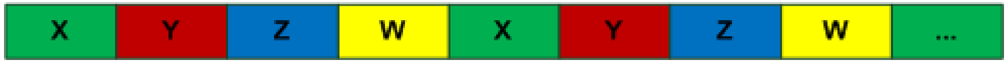
Easy to vectorize; uses lots of prefetching streams!

Structure of Arrays




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

Array of Structures




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

struct Particle *Particles;
```

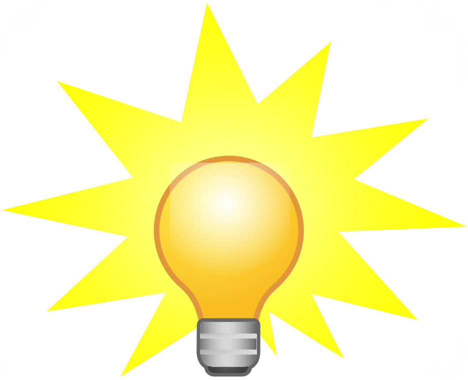
https://software.intel.com/en-us/articles/ticker-tape-part-2

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

# 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.



# ESSL

IBM provides ESSL: A library of optimized math functions (BLAS for linear algebra, FFTs, and more). For FFTs, there is an optional FFTW-compatible interface.

- ESSL is installed in /soft/libraries/essl/current
- You can choose either -lesslbg or -lesslsmpbg (the 'smp' version uses OpenMP internally to take advantage of multiple threads)

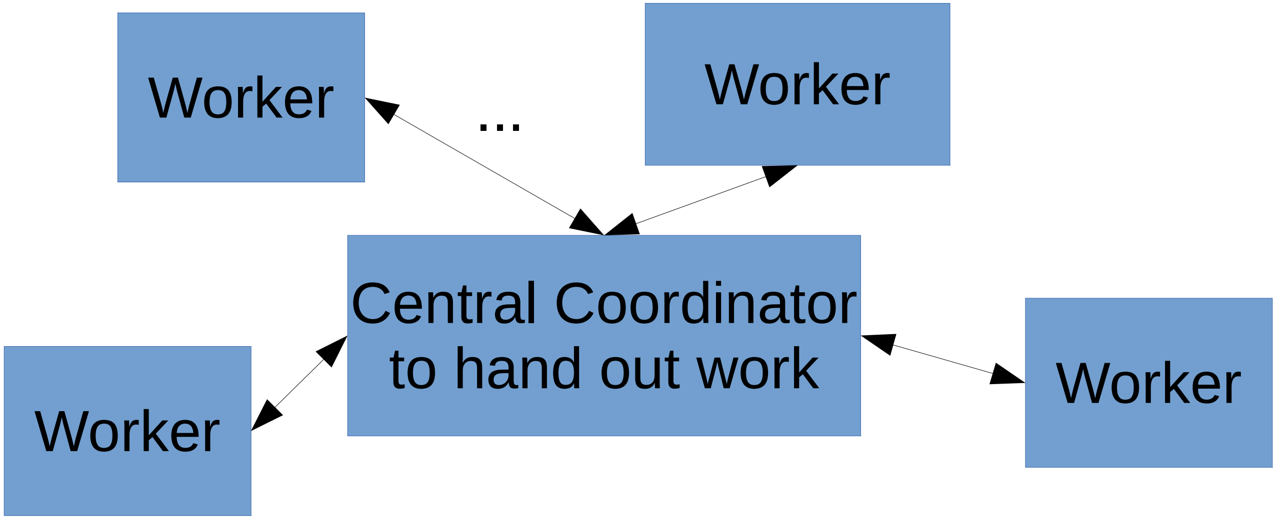
ESSL is on IBM PowerPC systems what MKL is on Intel systems.

# Memory partitioning

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

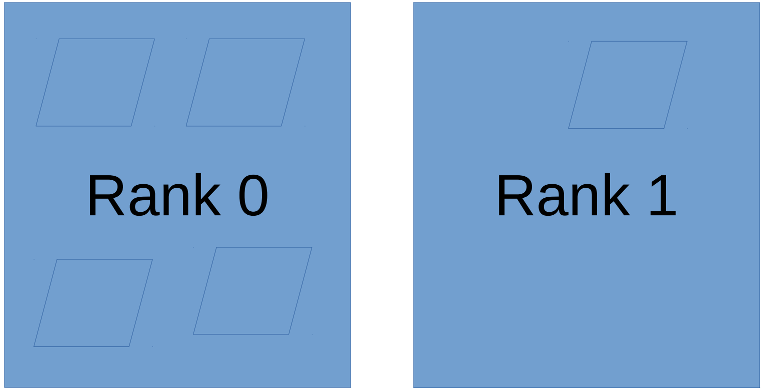
# Avoid central coordinators




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.

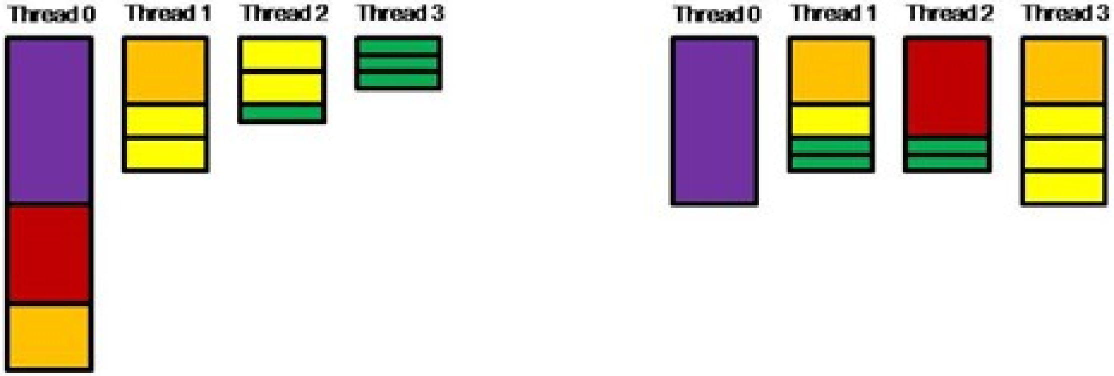



This is not good; rank 0 has much more work.

# schedule(dynamic) can be your friend...

```
#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.




(a) Unbalanced assignment of tasks to threads

(b) Balanced assignment of tasks to threads

https://software.intel.com/en-us/articles/load-balance-and-parallel-performance

# #pragma omp simd

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

```
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...

```
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;
}
```

https://software.intel.com/en-us/articles/enabling-simd-in-program-using-openmp40

# Some final advice...

Don't guess! Profile! (We'll have several talks about how to do that.) Your performance bottlenecks on the BG/Q might be very different from those on other systems.

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

?

Any questions?