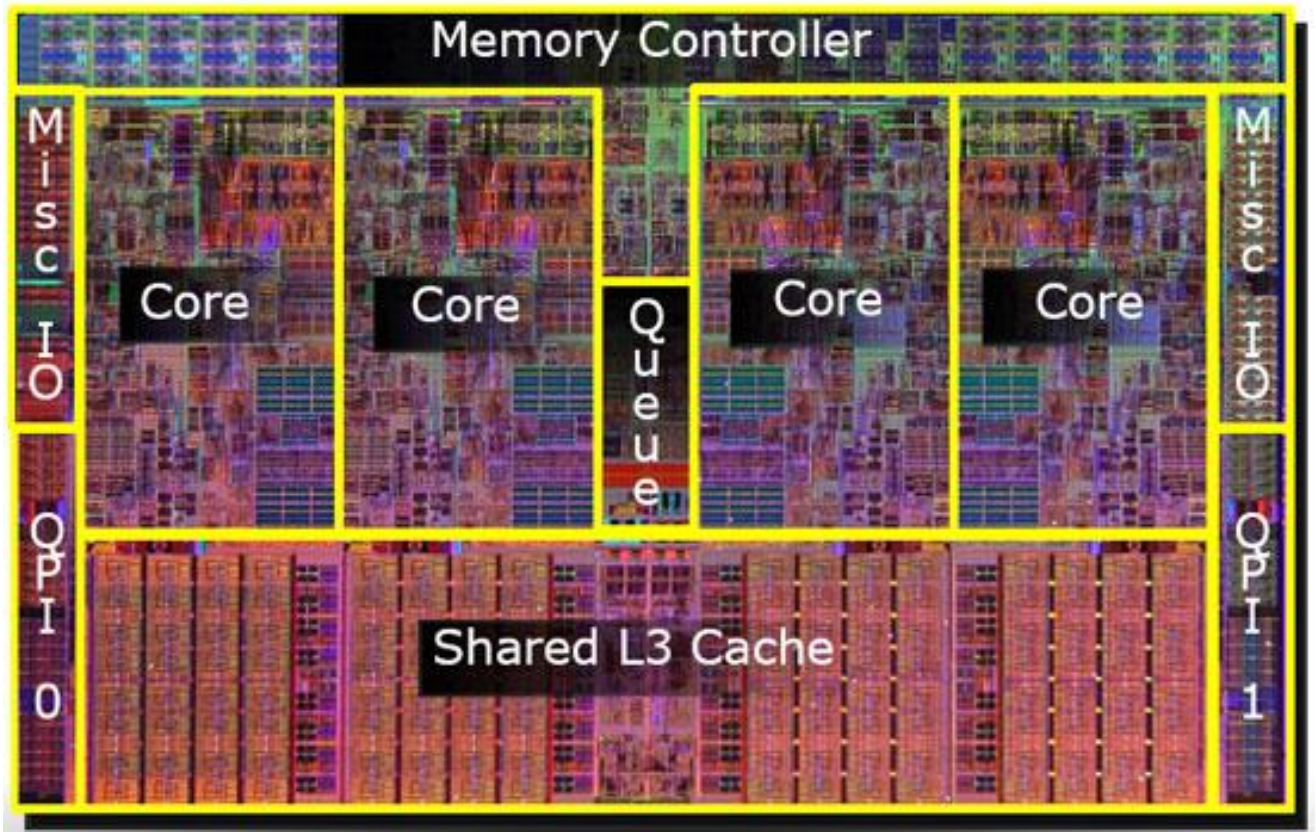
JÜLICH
FORSCHUNGSZENTRUM

Blue Gene/Q optimization

12. Oktober 2012 Stefan Krieg

Everyday parallelism



© intel

Intel Core i7

4 Cores

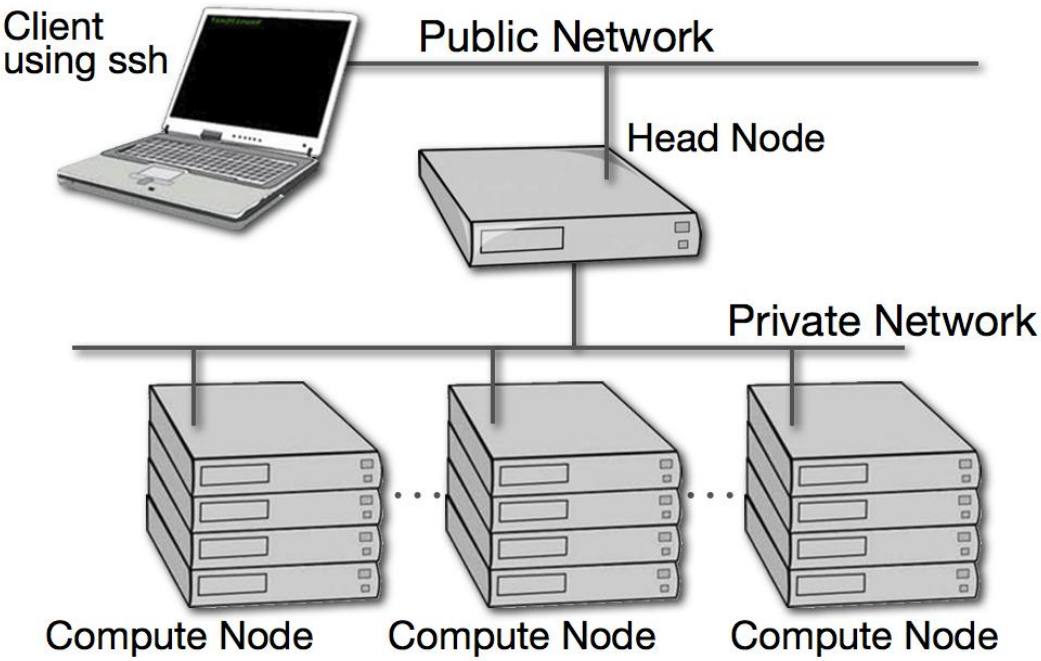
2 way HT

→ 8 HW threads

BG/Q is no cluster

© BUW

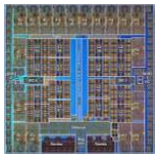
BG/Q is no cluster



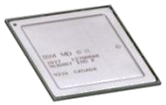
© udel.edu

- Cheap:
  commodity components
- Fast cores with high clock freq *and* high power consumption
- Commonly used in LQCD for valence calculations (naïvely parallel)
- Nodes run Linux
- Limited scaling:
  components are not designed for MPP

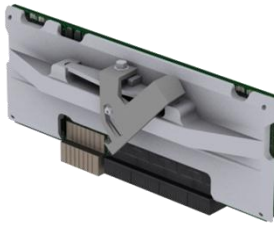
BG/Q system



2. Module
Single Chip



3. Compute Card
One single chip module,
16 GB DDR3 Memory

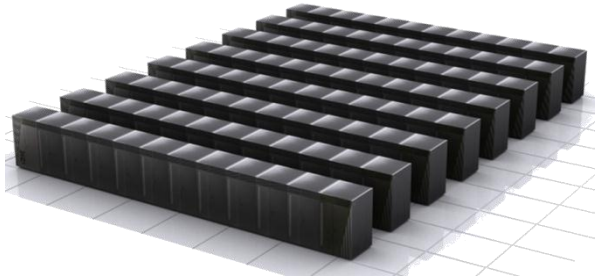


5b. I/O Drawer
8 I/O Cards
8 PCIe Gen2 slots

5a. Midplane
16 Node Cards

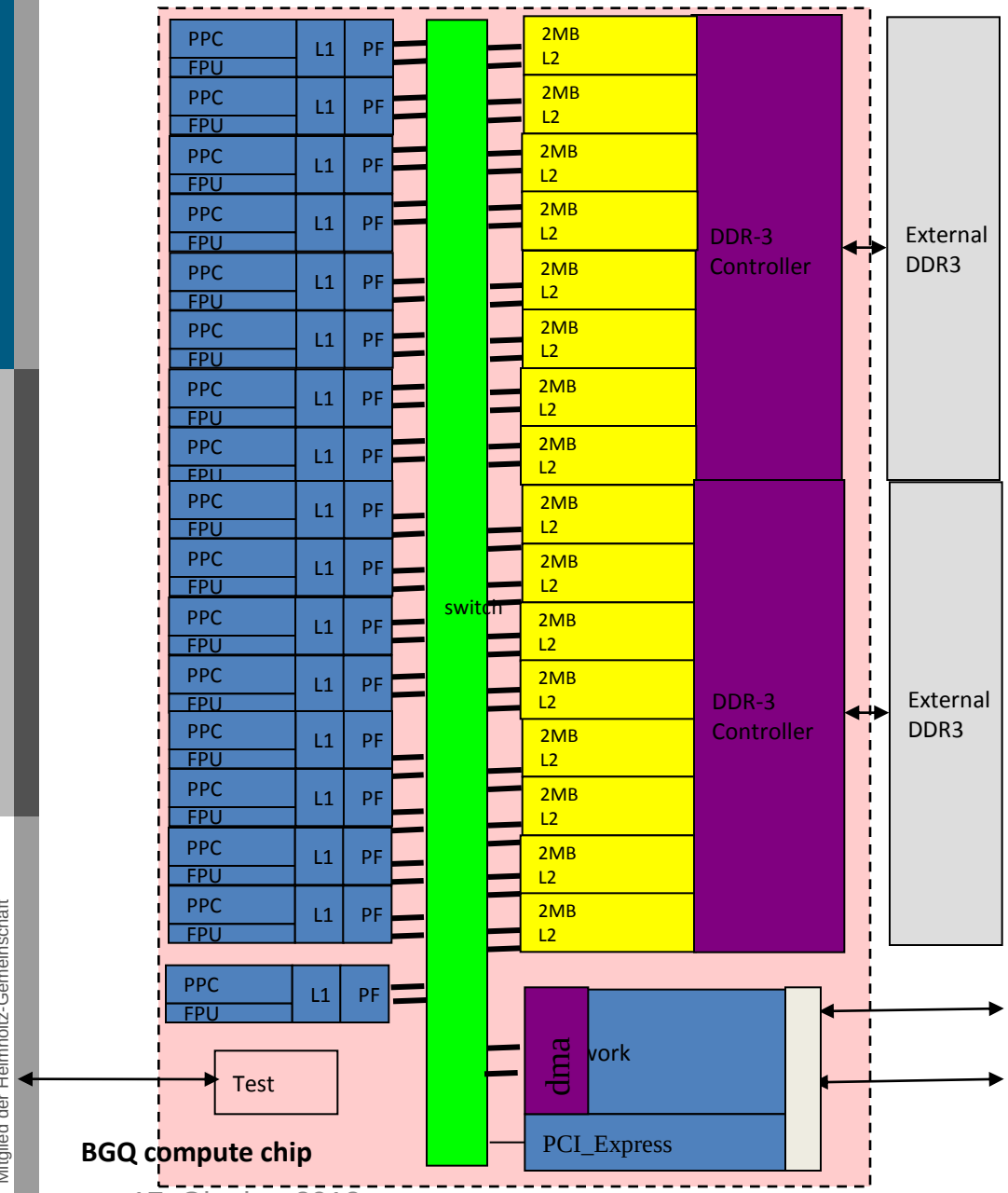
6. Rack
2 Midplanes
1, 2 or 4 I/O Drawers

7. System
20PF/s



© IBM

BGQ Chip architecture



- 16+1 core SMP
  Each core 4 way hardware threaded
- Transactional memory and thread level speculation
- Quad float point unit on each core
  204.8 GF peak node
- Frequency target of (1.6) GHz
- 563 GB/s bisection bandwidth to shared L2
  (BGL at LLNL has 700 GB/s system bisection)
- 32 MB shared L2 cache
- 42.6 GB/s DDR3 bandwidth
  (2 channels each with chip kill protection)
- 10 intrarack interprocessor links each at 2.0GB/s
- 1 I/O link at 2.0 GB/s
- 4-8 GB memory/node
- ~30 Watts chip power

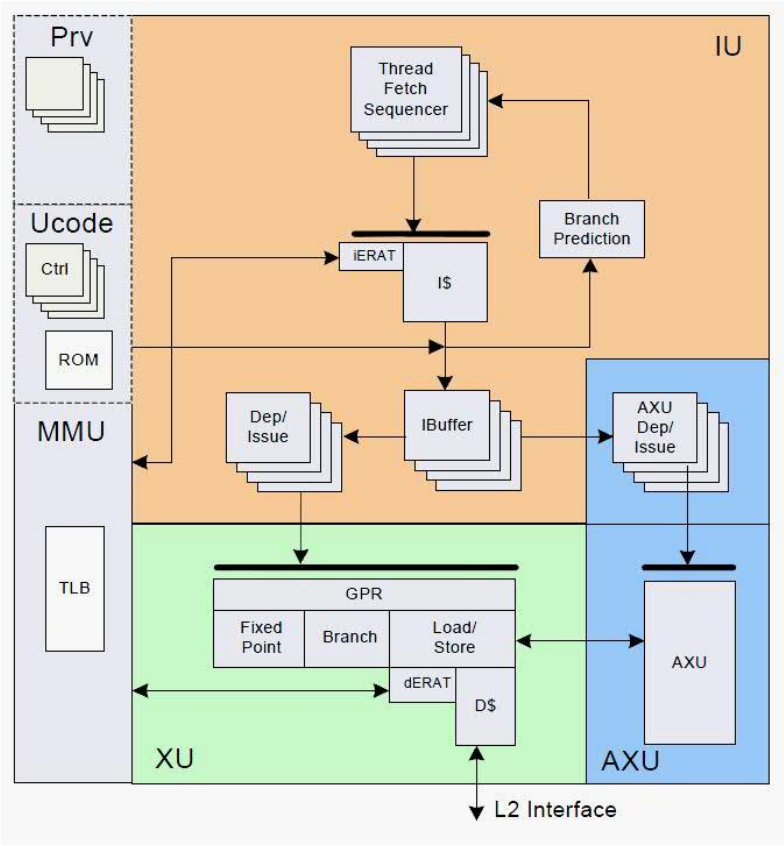
2 GB/s I/O link (to I/O subsystem)

10*2GB/s Intrarack (5-D torus)

** chip I/O shares function with PCI_Express

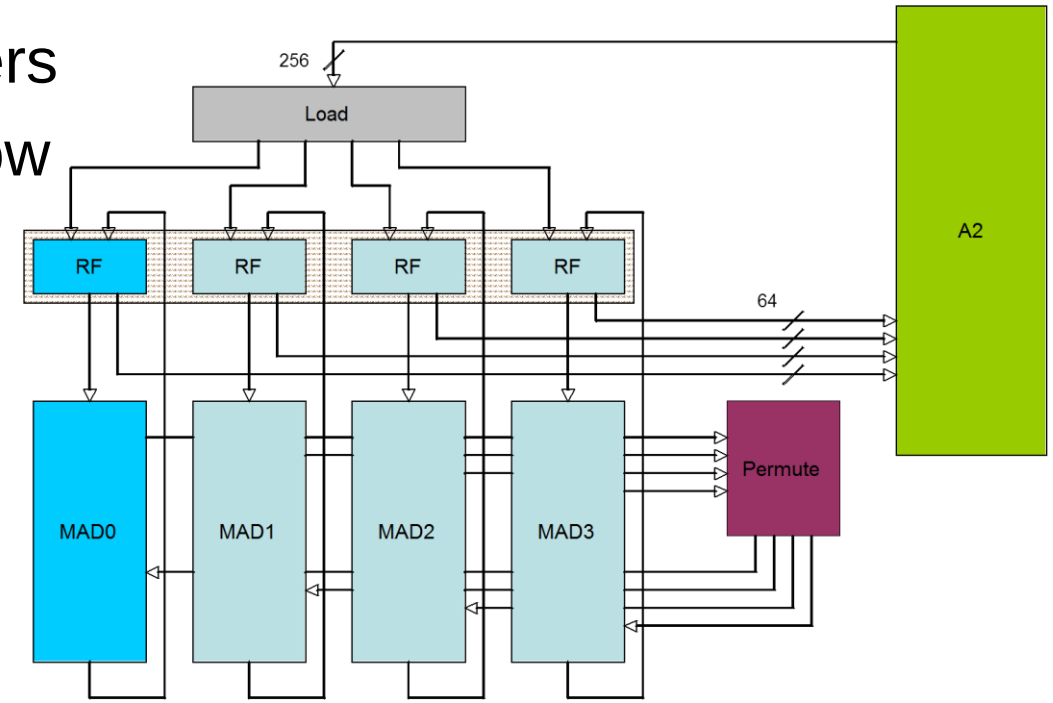
A2 core

- Embedded processor core
- 64-bit Power ISA
- 4-way simultaneously multi-threaded (SMT)
- In-order dispatch, execution, completion
- Execution unit (XU)
  - *32x4x64-bit general purpose registers*
  - *Dynamic branch prediction*
- Auxiliary execution unit (AXU)
  - *BG/Q: Quad-FPU*



Quad Floating-Point Processing Unit

- ISA extension: Quad Processing eXtension (QPX)
- 4 64-bit pipelines, SIMD processing
  - *Vector operands*
- 32x4x256 bit registers
- Multiply-Add Dataflow
  - *Can process in each cycle, e.g., ± [(A * B) ± C]*
- Peak performance
  4 FMA / cycle
  → 12.8 GFlops
  at 1.6 GHz



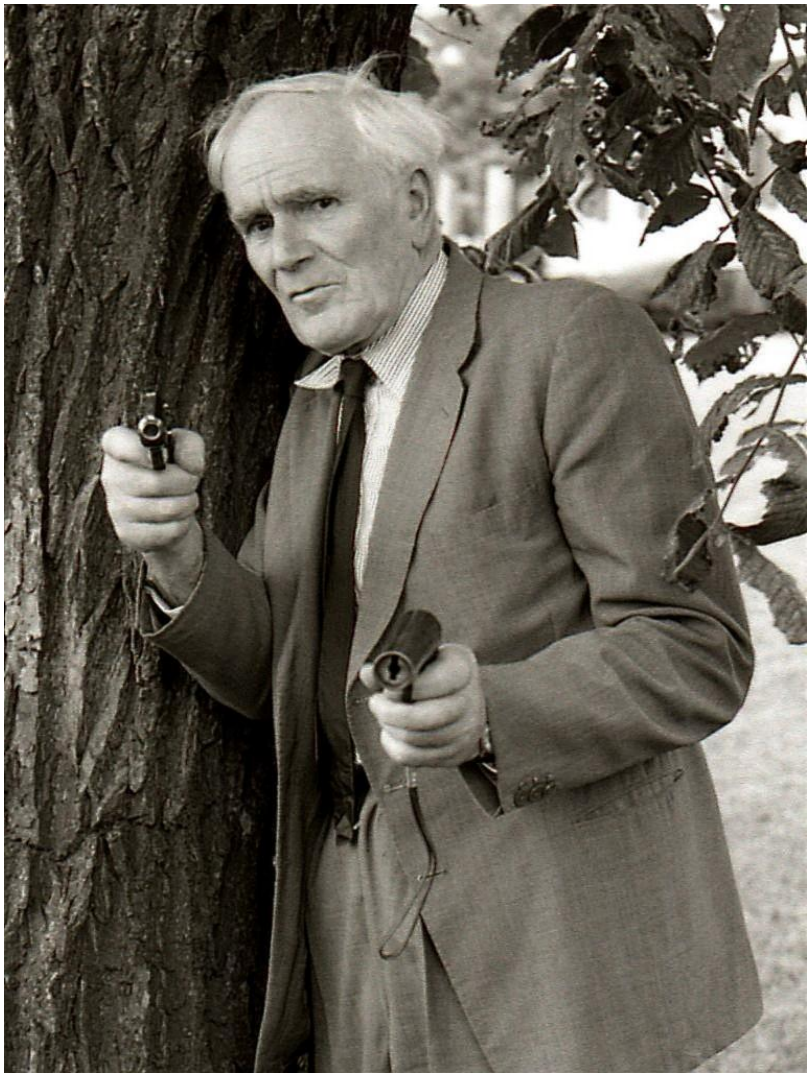
Summary: the Q

Node:

- 16 cores/node (A2)
- 4 way HT
- 1.6 GHz
- 4 way SIMD (double)
- In order execution

Interconnect:

- 5d torus network
- Barrier/collective using torus ≠ BG/{P,L}



Towpilot@Wikipedia, GFDL

Blue Gene/Q vs. Blue Gene/P

Blue Gene/Q	Blue Gene/P
• Midplane = 105 Tflop/s • 32768 HW threads • = 3.2 Gflop/s per thread • 5d interconnect: • 512 = 2x4x4x4x4	• 16 MP = 111 Tflop/s • 32768 HW threads • = 3.4 Gflop/s per thread • 3d interconnect: • 8192 = 16x16x32

→ **USE master threads if possible!**

Comparison of relevant specs

Blue Gene/P	Blue Gene/Q	Q vs. P
300.000 threads	6.291.456 threads	21
1 Pflop/s	20 Pflop/s	20
Cache: 2.0 MB/core	Cache: 2.0 MB/core	1
Cache: 109 GB/s BW	Cache: 563 GB/s BW	6
Mem: 0.5 GB/thread	Mem: 0.5 GB/thread	1
Mem: 13.6 GB/s BW	Mem: 42.6 GB/s BW	3
FPU: 4 Flop/c/core	FPU: 8 Flop/c/core	2
FPU: 13.6 Gflop/s/node	FPU: 204.8 Gflop/s/node	15

P vs. Q: things to know

JÜLICH FORSCHUNGSZENTRUM

Blue Gene/P:	Blue Gene/Q:

• 32 FP registers/core
• FP pipline latency is 6 cycles
• Align your loads!
  • 16 Bytes Load/Store
  • 32 Bytes for DMA
• 4 pending loads
• Manual prefetching
	• 32 FP registers/HW thread
	• FP pipeline latency is 6 cycles
	• Align and stripmine your loads!
	• 8 pending loads
	• Prefetch engine

BG/Q P2P Communication

Blue Gene/P

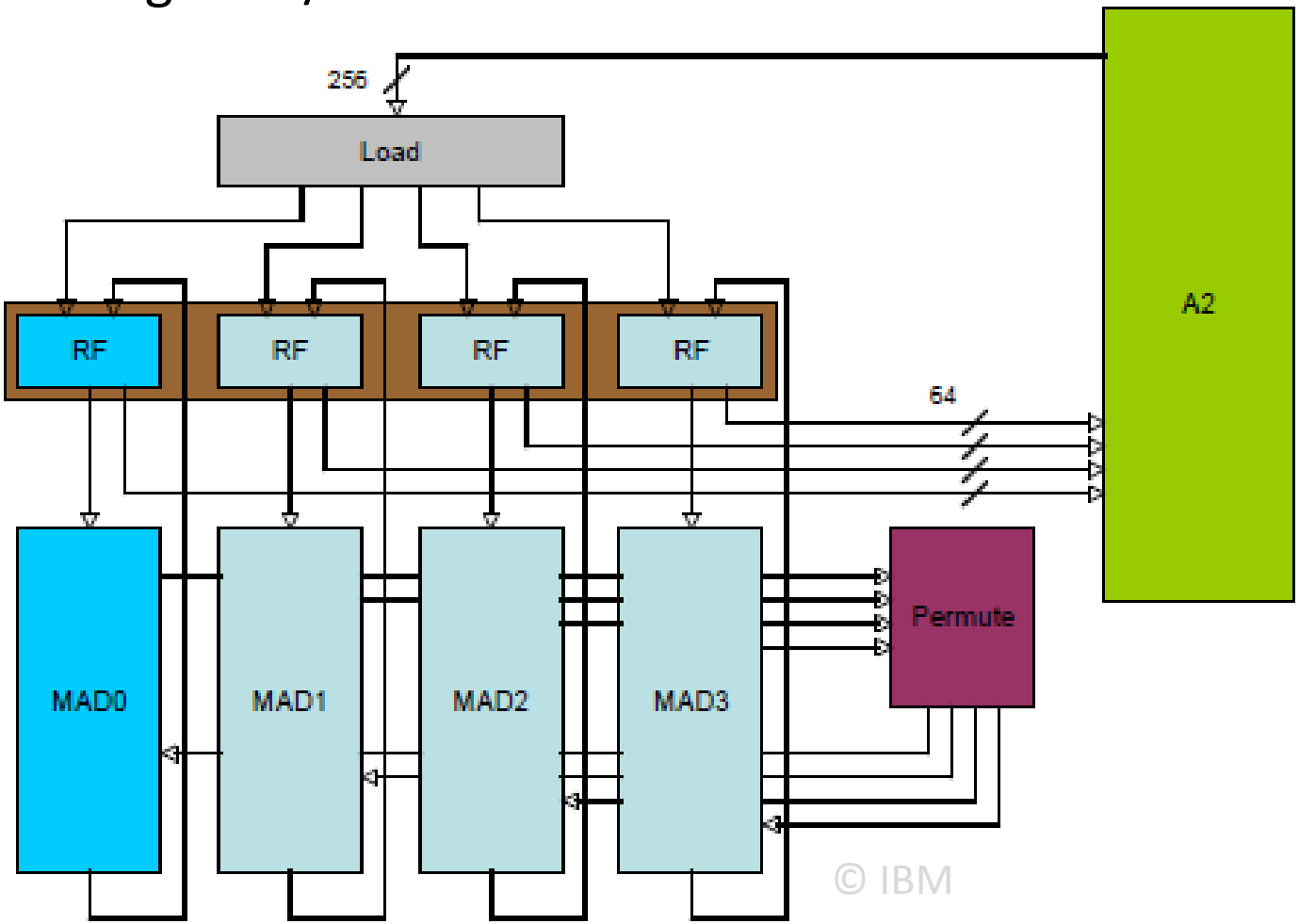
- Persistent communication:
  - Inject descriptors once
  - Reuse as often as possible
  - Communication startup slow (as slow as non-persistent generic communication)
  - Communication restart fast: manipulate (memory) Fifo head & rec. and inj. counters
- HW:
  - 4 Fifo groups, 32 Fifos/group
  - 4 Cntr. groups, 64 couters/group

Blue Gene/Q

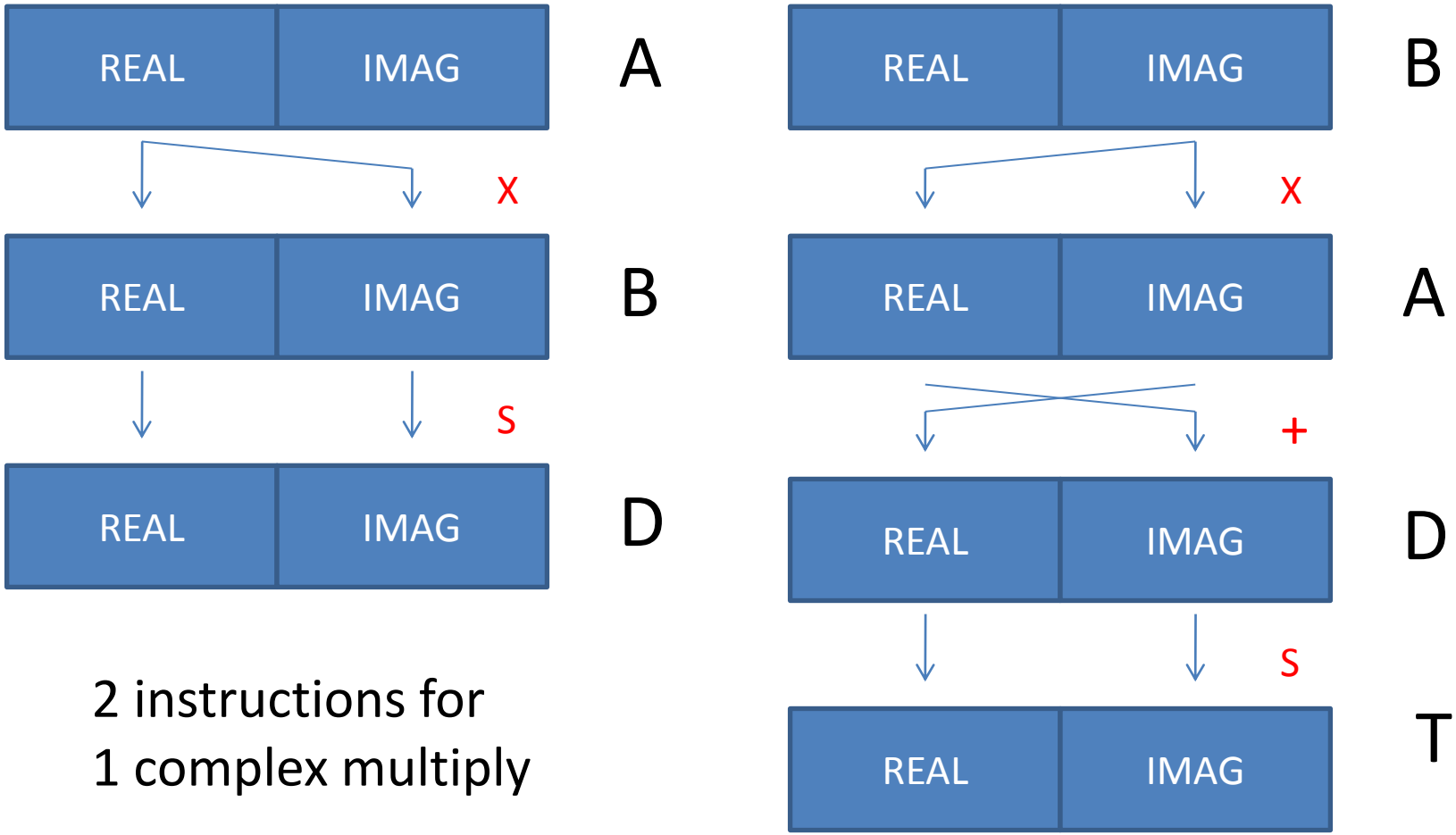
- Persistent communication
  - Same setup as for BGP
  - However: no address associated with reception counter
- HW:
  - 512 Fifos (= 32 fifos/core)
  - 512 BAT (base address table) entries (= 32 entries/core)
  - Number of counters limited by number of BAT entries
  - BAT entry required also for direct put operation.

QPX

32 FP registers/HW thread

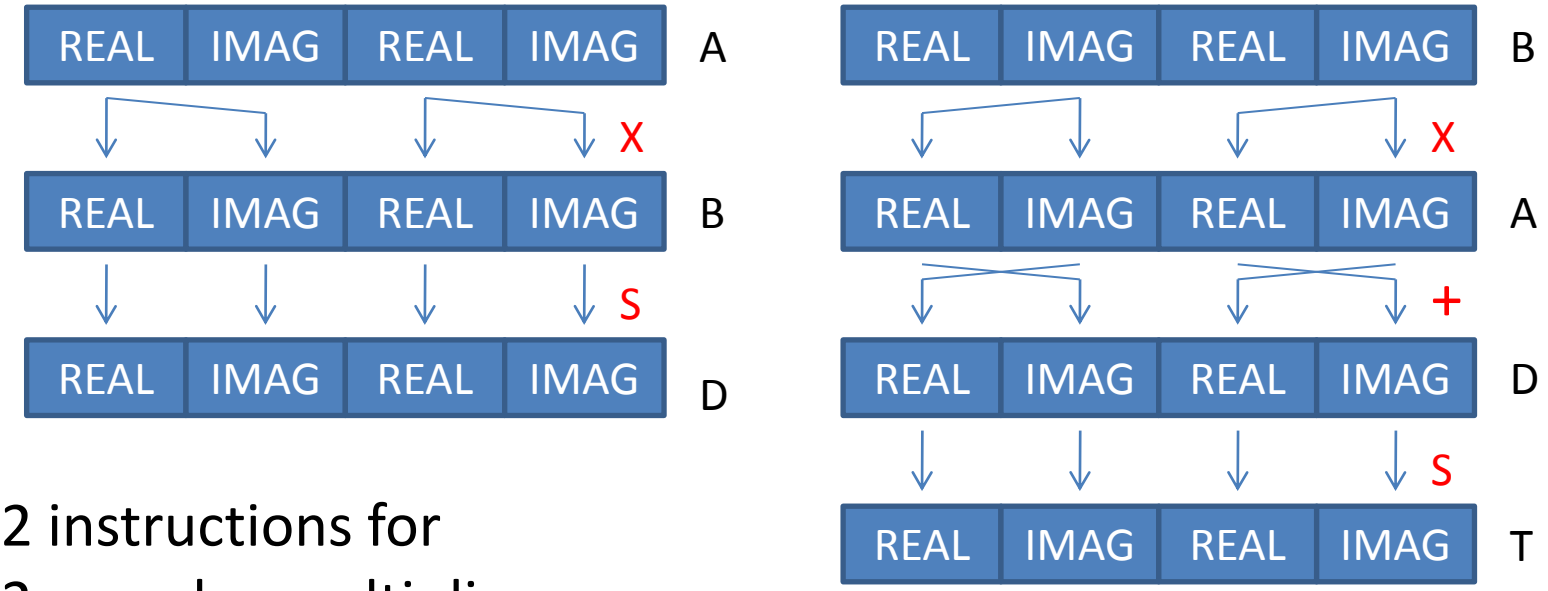


BG/P FPU complex multiply



2 instructions for
1 complex multiply

BG/Q (twin) complex multiply



2 instructions for
2 complex multiplies

SU(3) vs. SIMD(4)?

No instructions to use 2,3 with 0,1 → 2nd site or stream
Or.. Forget about 2,3 ??

XL C intrinsics

- New data type: vector4double
- Example:

```
double r[4] = {1.0, 2.0, 3.0, 4.0};
vector4double v1 = (vector4double) {1.0, 2.0, 3.0, 4.0};
vector4double v2,v3;
v2 = vec_ld(0L, &r[0]);    /* load from array */
v3 = vec_add(v1,v2);/* add two vectors */
vec_st(v3, 0L, &r[0]);     /* Store result */
```

- No overloaded operators. However:

```
vector4double *v1ptr = &v1;
vector4double v4 = *v1ptr;
vector4double v5 = v4;
```

XL C intrinsics

- Also:

```
vector4double v1 = (vector4double) {1., 2., 3., 4.};
double d1, d2, d3, d4;
d1 = v1[0];         // d1=1.
d2 = v1[1];         // d2=2.
d3 = v1[2];         // d3=3.
d4 = v1[3];         // d4=4.
```

- And:

```
sizeof(v1); // vector type size: 32
sizeof(&v1); // address of vector size: 8
__alignof__(v1); // vector type alignment: 32
__alignof__(&v1); // address of vector alignment: 8
```

XL C intrinsics

- Basic operations:

```
vector4double v1,v2,v3,v4;
v1 = vec_ld(0,&ptr);   //ptr = *{_Complex} {double, float}
v1 = vec_lds(0,&ptr);  //ptr = *{_Complex} {double, float}
v1 = vec_ld2(0,&ptr);  //ptr = *{double, float}

v1 = vec_add(vec_madd(v2,v3,v4), v1);

vector4double pattern = vec_gpci(01230);
v = vec_perm(v,v,pattern);
```

- (Twin) complex multiply:

```
v1 = vec_xxnpmadd(v2, v3, vec_xmul(v3, v2));
```

- Reference:

http://www-01.ibm.com/support/docview.wss?uid=swg27027065&aid=1

Threading

- Use OpenMP or pthreads
- Different potential layouts:

  1, 2, 4, 8, 16, 32, or 64 task per node

  64,32,16,8,4,2, or 1 thread per task
- MPI likes to have 2 threads for own use
- Here: pthreads in 16/4 approach
- 1/64 mode implemented, 10-15% slower

A note on OpenMP

- Simple, pragma based parallelization
- Hint: use `_Pragma("omp parallel for")` in macros

Example:

```
int i, a[N];

#pragma omp parallel for
for (i = 0; i < N; i++)
{
   a[i]= 2 * i;
}
```

Local barrier

- Memory subsystem (L2) supports "atomics"
- Multiple tasks/node: use shared memory window
- Procedure:
  1. Kernel_L2AtomicsAllocate
  2. L2_AtomicLoadIncrement
  3. wait(bvar<status)

  (last node sets bvar=status)

Node-wide barriers – L2 atomics

```
#include <spi/include/l2/barrier.h>
#include <spi/include/l2/atomic.h>
#include <spi/kernel/process.h>

L2_Barrier_t barrier;

uint64_t rc = Kernel_L2AtomicsAllocate(barrier, sizeof(L2_Barrier_t));
if (rc) exit(1);

 L2_Barrier(barrier, Kernel_ProcessCount());
```

Node-wide barriers – L2 atomics

```
__INLINE__ void L2_Barrier(L2_Barrier_t *b, int numthreads)
{
    uint64_t target = b->start + numthreads;
    uint64_t current = L2_AtomicLoadIncrement(&b->count) + 1;

    if (current == target) {
        b->start = current;  // advance to next round
    } else {
        while (b->start < current);  // wait for advance to next round
    }
}
```

Inter node barrier - SPI

```
MUSPI_GIBarrier_t barrier_ref;
uint32_t classrt = 0;

uint32_t rc = Kernel_GetGlobalBarrierUserClassRouteId(&classrt );
if (rc) exit(1);

rc  = MUSPI_GIBarrierInit( &barrier_ref, classrt );
if (rc) exit(1);

uint32_t rc = MUSPI_GIBarrierEnterAndWait( &barrier_ref );
If (rc) exit(1);
```

BG (compute) network

- 5d torus network

	A	B	C	D	E
– Nodeboard:	2x	2x	2x	2x	2
– Midplane:	4x	4x	4x	4x	2
– Sequoia:	16x	16x	16x	12x	2

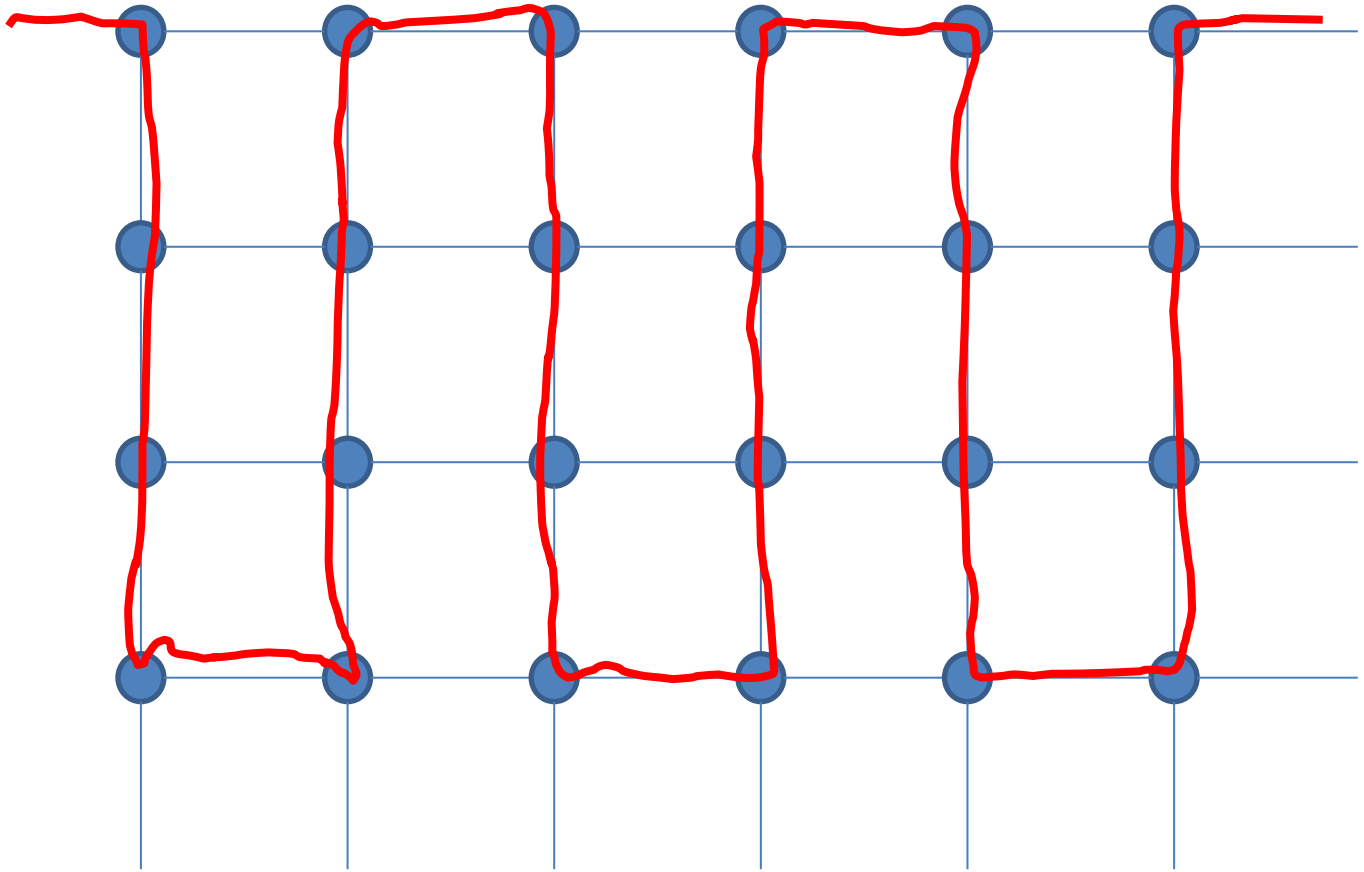
"Compactifiation"

Lattice QCD is 4d (space-time)

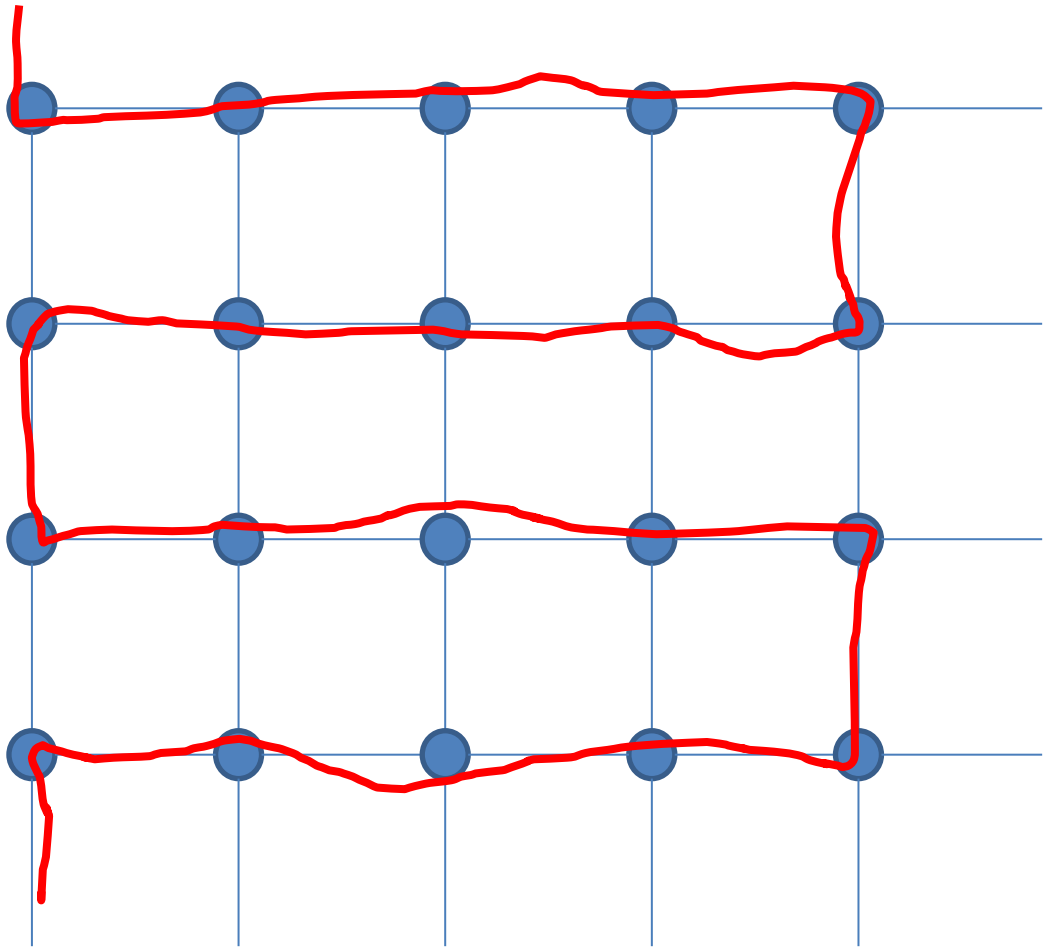
Torus is 5d

In physics, where there are (way) too many dimensions, we do a little "compactification" …

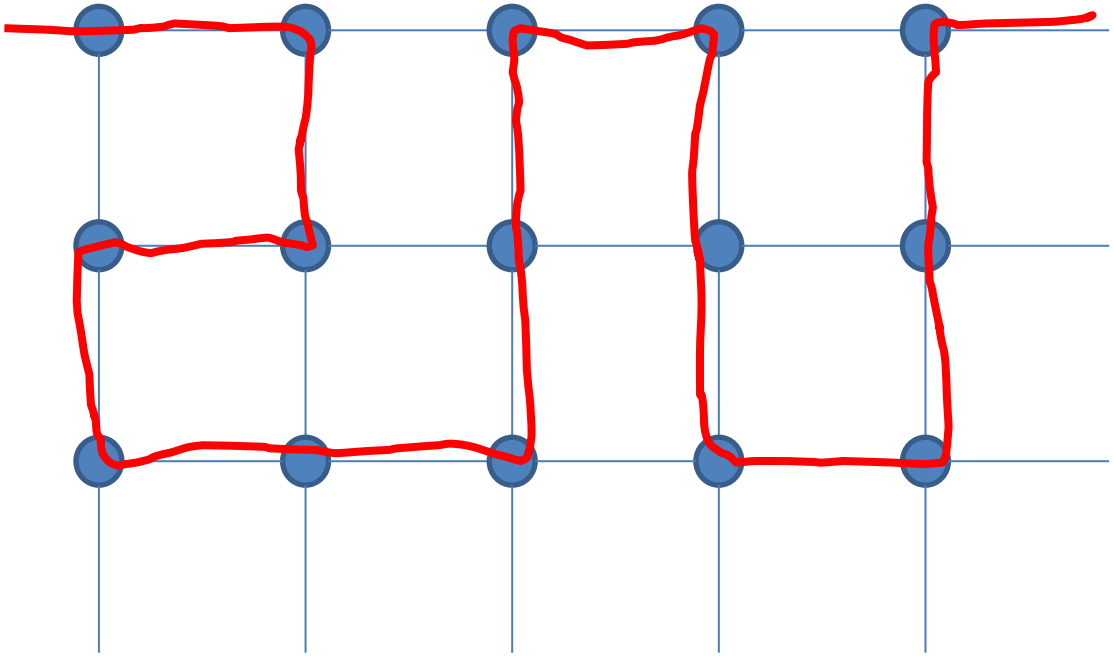
"Compactifiation"



"Compactifiation"



"Compactifiation"



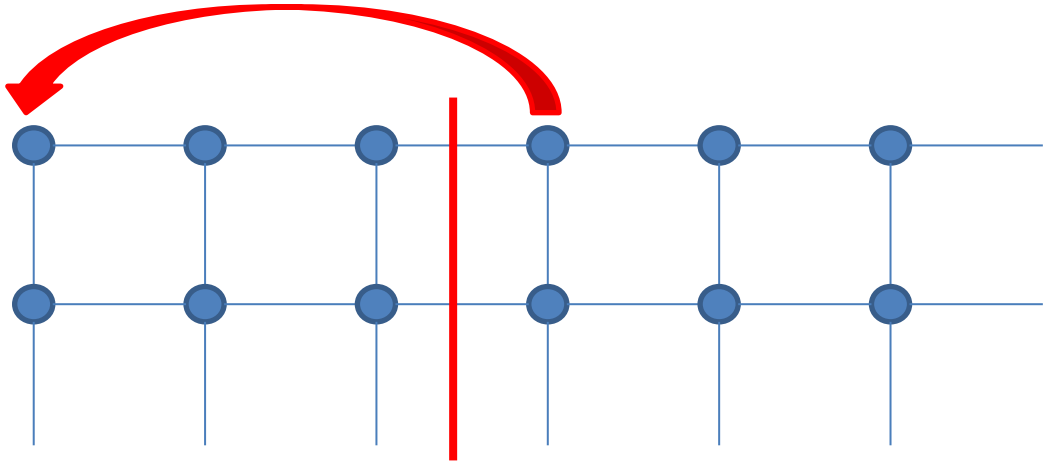
Ansatz

- Iterate mapping, 5d->3d

  2x4x4x4x4 -> 8x4x16

  – Use 16 cores as 4th dimension

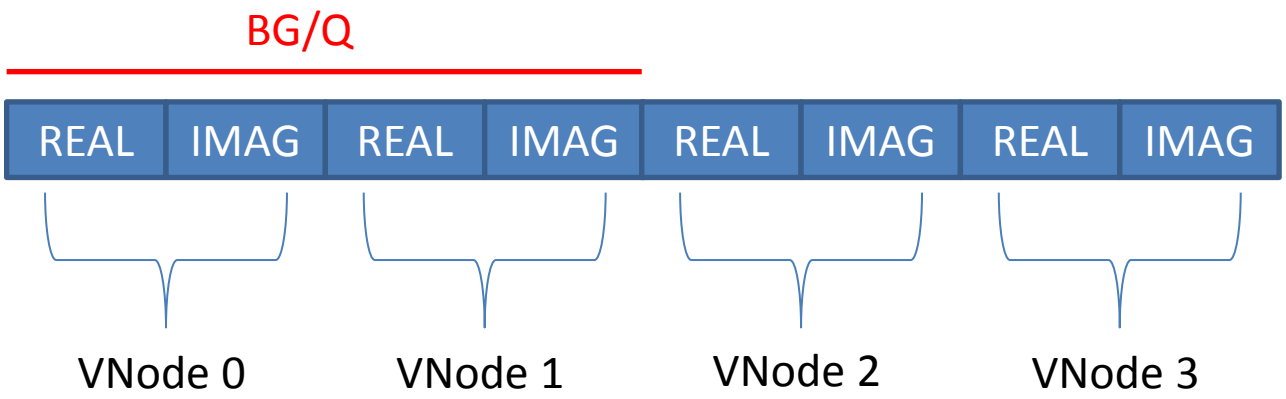
  – Local 4th dimension -> SIMD “virtual node”



“Virtual node” approach

Don’t think VN mode, were talking SIMD here!

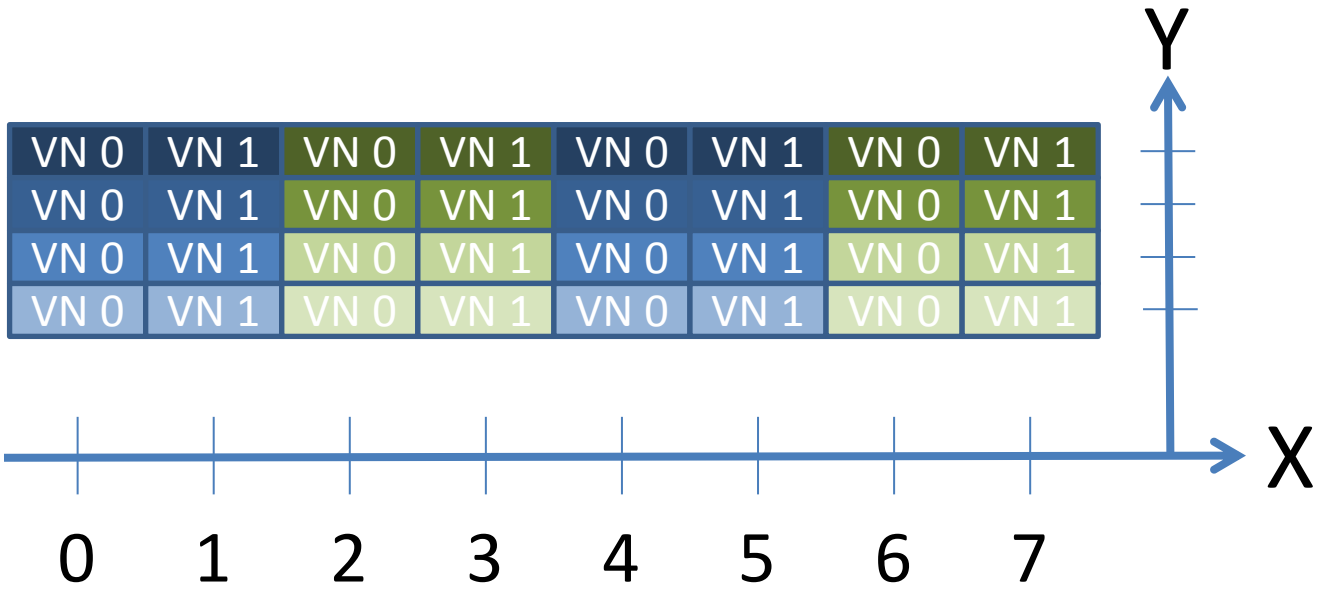
- Map a logical node to every subset



- Works for any kind SIMD vector length
- Parallelize (in NUMA spirit)

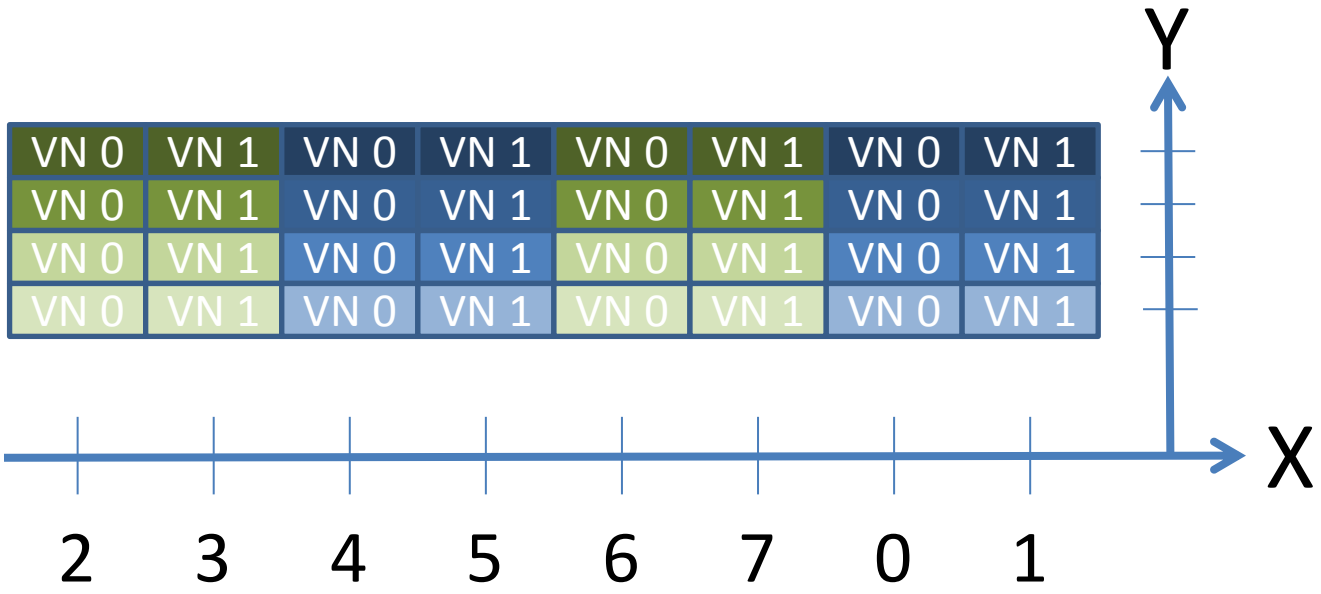
“Virtual node” approach

- Straightforward:



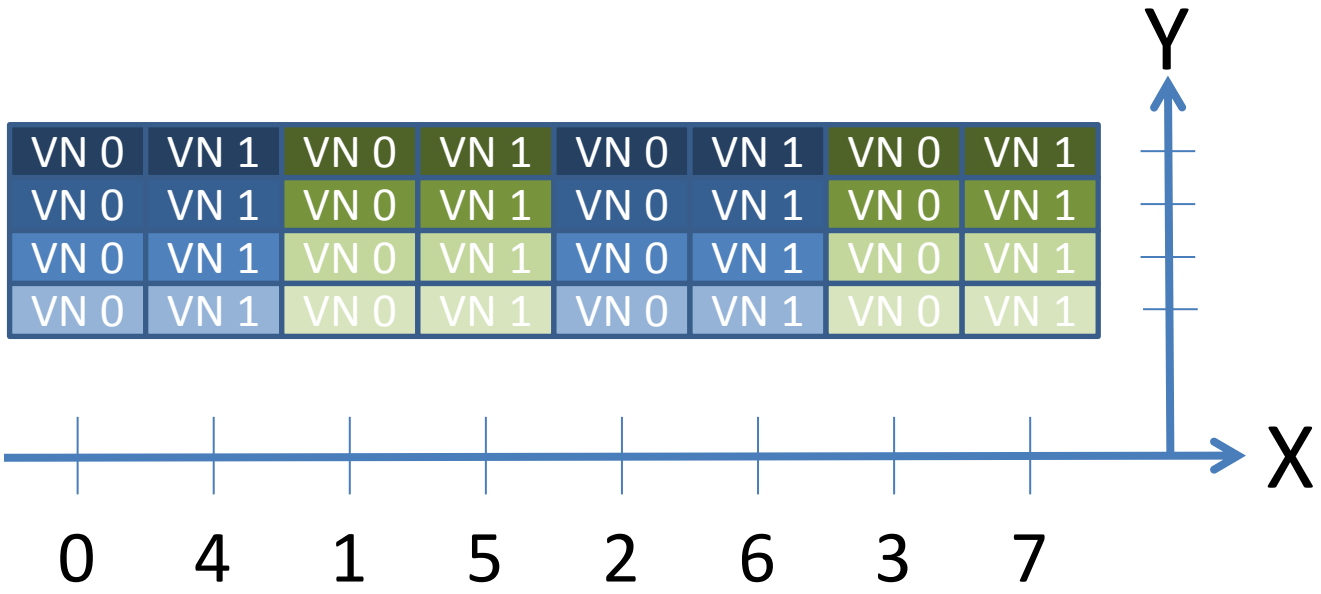
“Virtual node” approach

- Shift



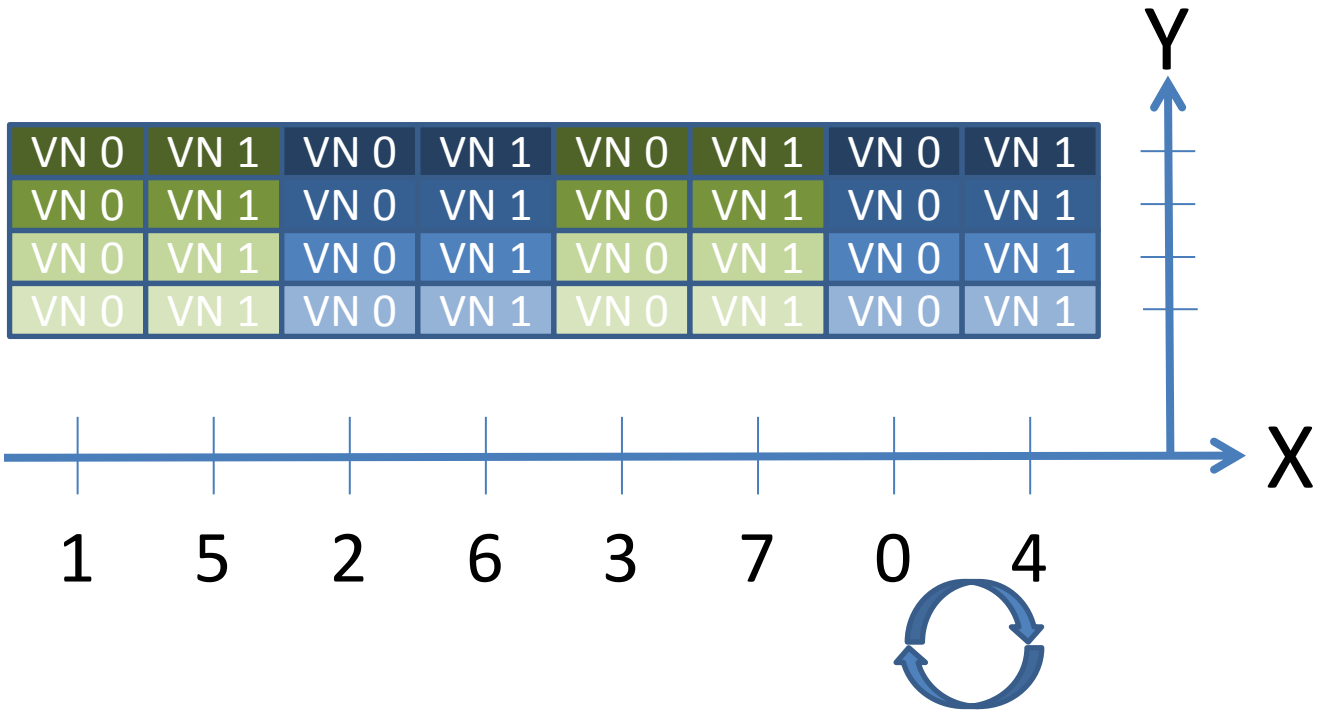
“Virtual node” approach

- Alternate



“Virtual node” approach

- Shift



- We only have to swap the surface

BG/P P2P Communication

JÜLICH FORSCHUNGSZENTRUM

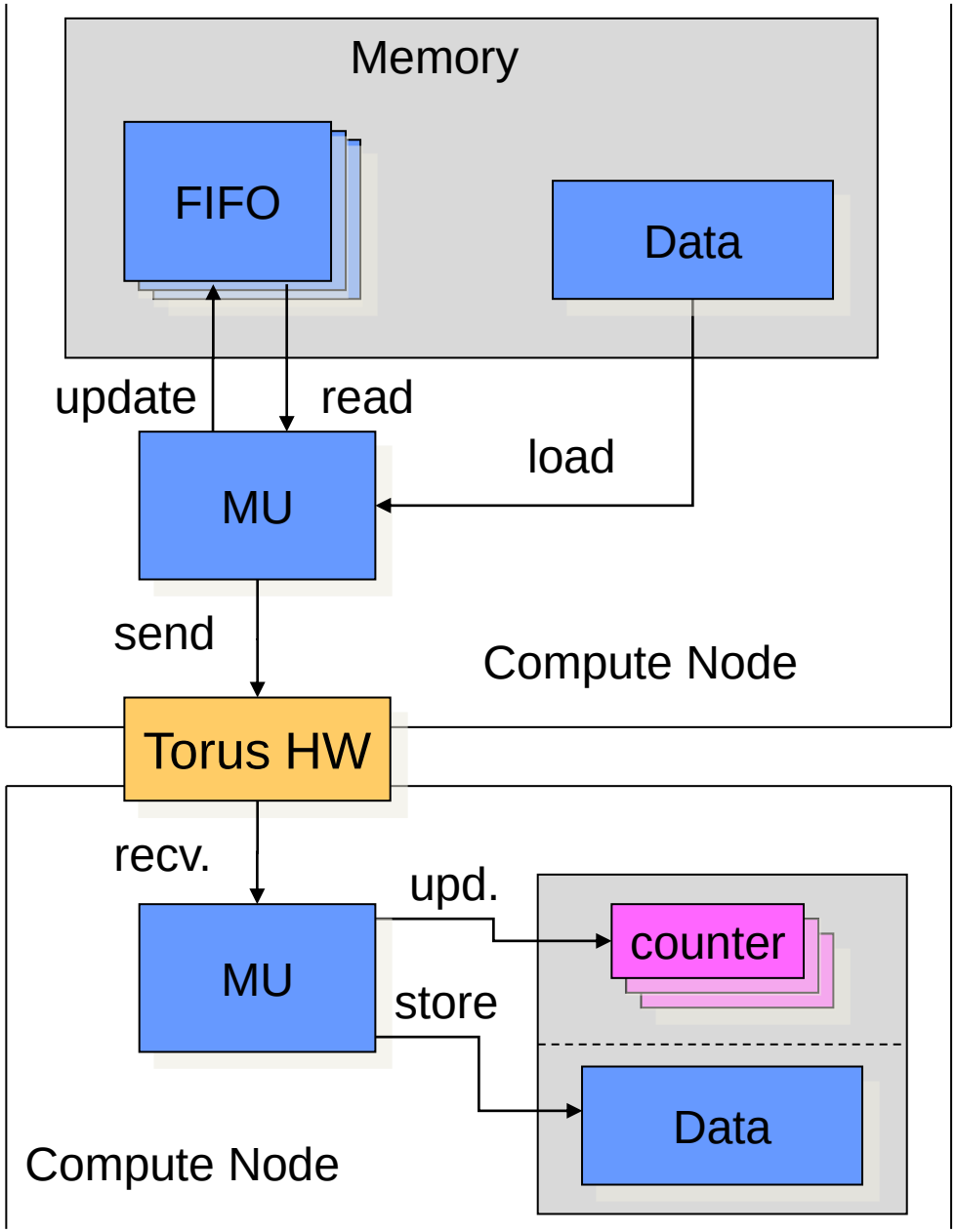
For a “direct-put“ using SPI one could proceed as follows:

- Allocate 1 inj. FIFO, 1 rec. and 1 inj. counter
- Set inj. counter base address
- Set the injection counter
- (destination node) set the reception counter base
- (destination node) Set the reception counter max address
- (destination node) Set the reception counter value

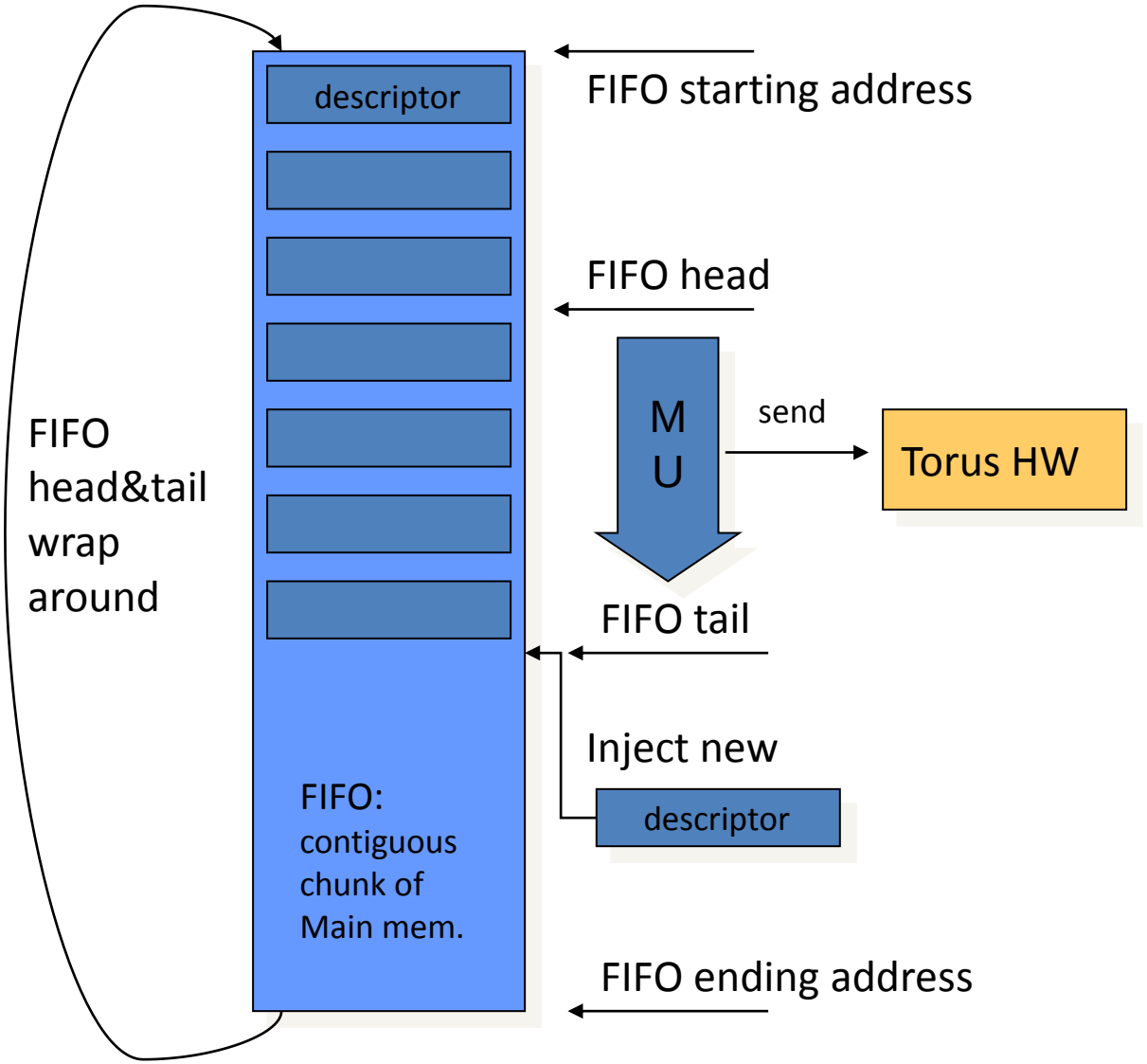
Synchronize

For each continuous chunk of data, inject 1 descriptor

- Calculate the address offset relative to inj. counter base
- Calculate the address offset relative to rec. counter base
- Give the message size



"direct put"


descriptor
FIFO starting address
FIFO head
M
U
send
Torus HW
FIFO
head&tail
wrap
around
FIFO tail
Inject new
descriptor
FIFO:
contiguous
chunk of
Main mem.
FIFO ending address

Collectives

- Blue Gene/P
  - Used tree network
  - Only UInt operations available in HW
  - Work on mantissa and exponent separately for FP
- Blue Gene/Q
  - Use the torus network
  - FP operations supported in HW
  - High flexibility to define "sub-communicators"

Collectives

HW support

- And, Or, XOR (UInt only)
- Add
- Min
- Max

for

- Signed, unsigned Integers
- Floating point numbers

→ To be implemented as "direct put" etc.

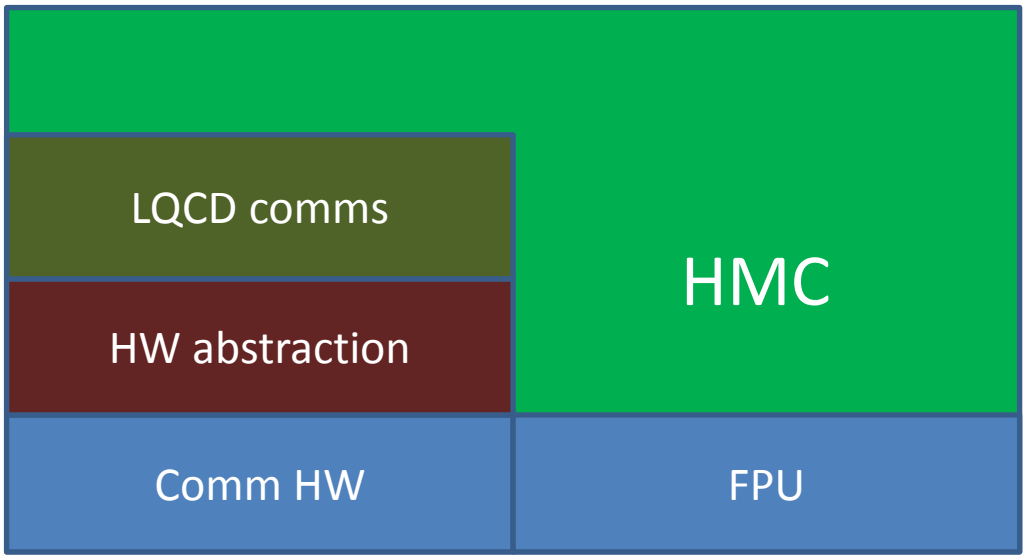
Software

- Use low level interfaces to hardware

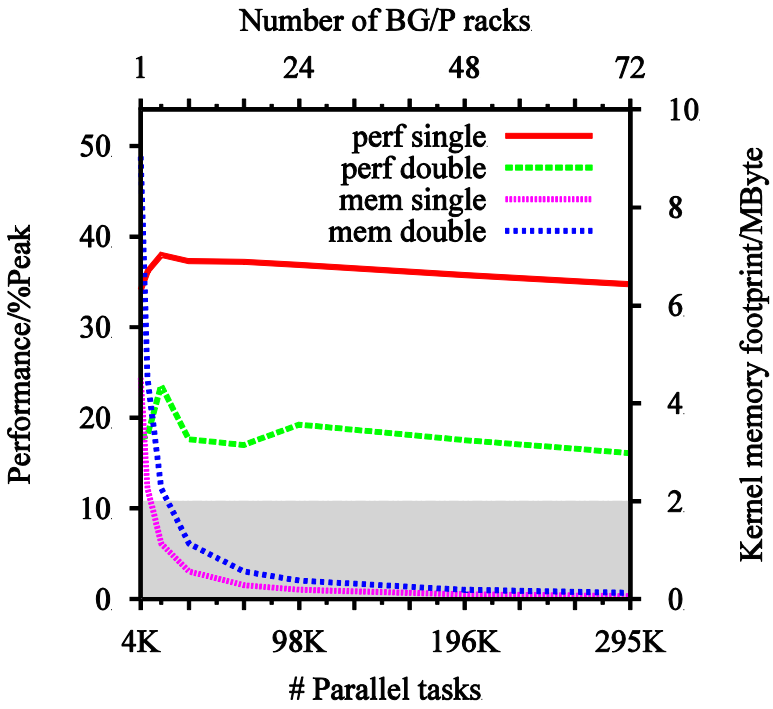
→Avoid MPI et al

→Implement persistent comms, collectives, barriers

- Use 4 threads per core
- Use compiler intrinsics

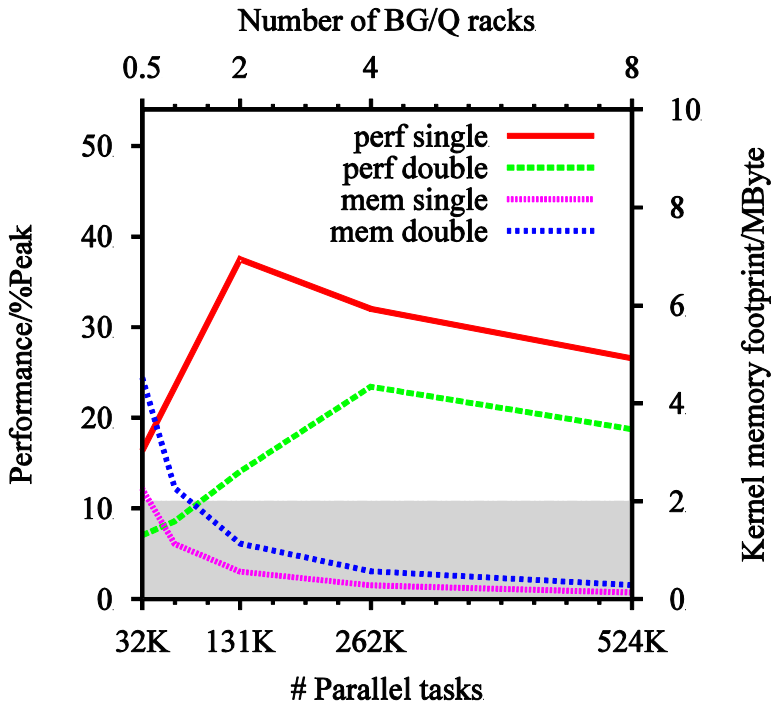


Kernel



Blue Gene/P

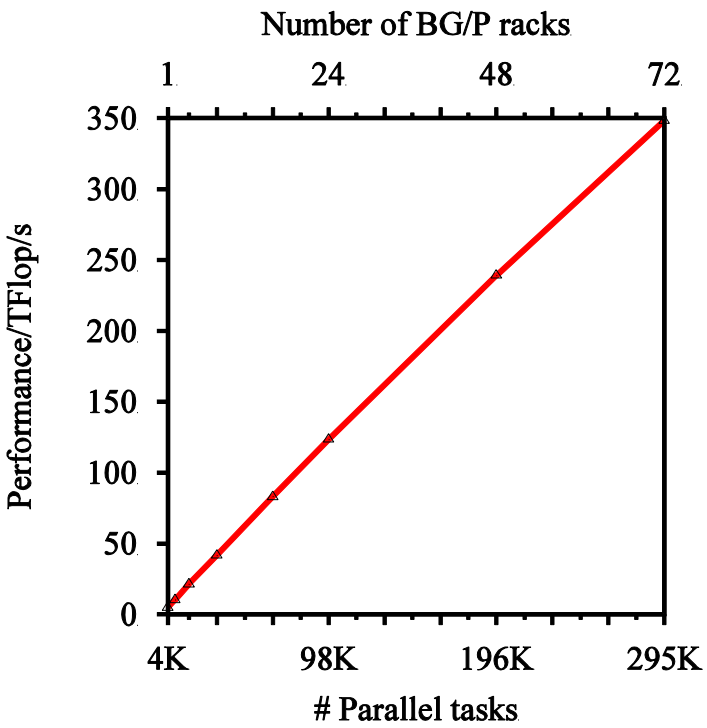
Lattice size: $64^3 \times 144$



Blue Gene/Q

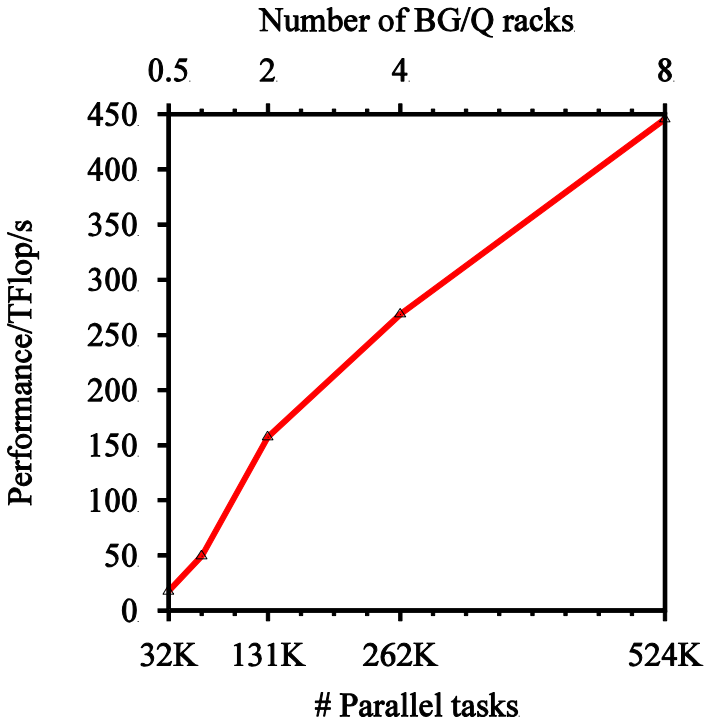
Lattice size: $64^3 \times 128$

Kernel



Blue Gene/P

Lattice size: $64^3 \times 144$



Blue Gene/Q

Lattice size: $64^3 \times 128$

Kernel

- Kernel peaks at about 38% (38% on BG/P)
- Intrinsics
- No explicit prefetching
- SPI (persistent) communication
- Presently: no site fusion

  → 16/4 layout instead of 1/64
- Threading based on pthreads + SPI