Forschungszentrum Jülich

in der Helmholtz-Gemeinschaft

BERGISCHE UNIVERSITÄT WUPPERTAL

Optimizing LQCD on IBM Blue Gene

The Blue Gene family tree

- **QCDSP** (600GF based on Texas Instruments DSP C31)
  - Gordon Bell Prize for Most Cost Effective Supercomputer in '98
  - Columbia University Designed and Built
  - Optimized for Quantum Chromodynamics (QCD)
  - 12,000 50MF Processors, Commodity 2MB DRAM

- **QCDOC** (20TF based on IBM System-on-a-Chip)
  - Collaboration between Columbia University and IBM Research
  - Optimized for QCD
    IBM 7SF Technology (ASIC Foundry Technology)
  - 20,000 1GF processors (nominal)
  - 4MB Embedded DRAM + External Commodity DDR/SDR SDRAM

- **Blue Gene/L** (180/360 TF based on IBM System-on-a-Chip)
  - Designed by IBM Research in IBM CU-11 Technology
  - 64,000 2.8GF dual processors (nominal)
  - 4MB Embedded DRAM + External Commodity DDR SDRAM

Outline

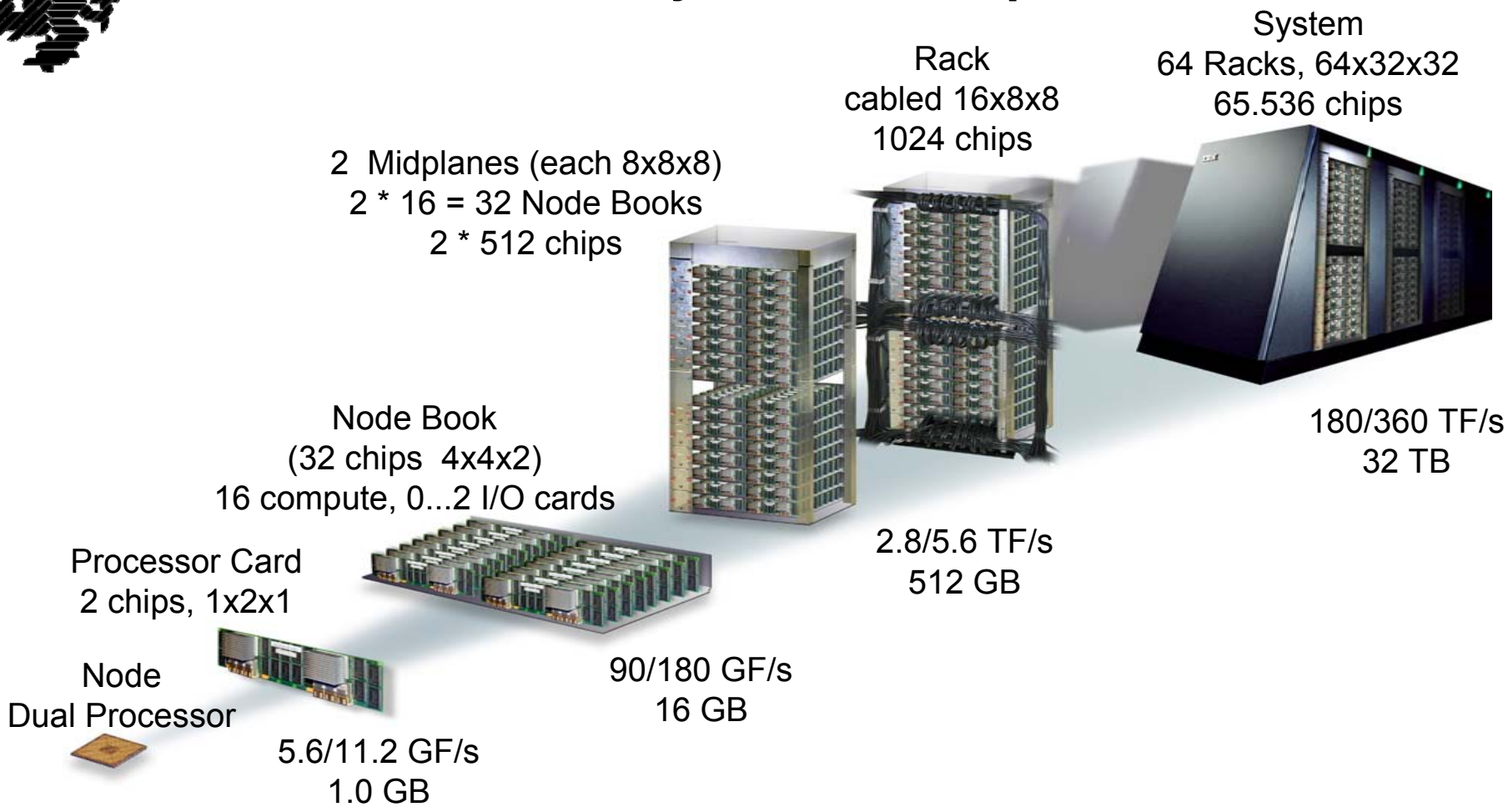
("JUBL") The 8 rack Blue Gene/L in Jülich

- Blue Gene/L hardware overview
  - The ppc440d chip
  - The network
  - Installation at NIC Jülich
- Coding for performance
  - Using the IBM XLC compiler
    - Strategies to accelerate C-code
    - Using the compiler intrinsics
  - Using GCC inline assembly
  - Comms: IBM QCD API & MPI
  - Useful RTS calls
- Outlook: LQCD on Blue Gene/P

Hardware Overview

The naked 8 rack Blue Gene/L in Jülich, during installation early 2006

Blue Gene/L System Buildup



ppc440d compute node

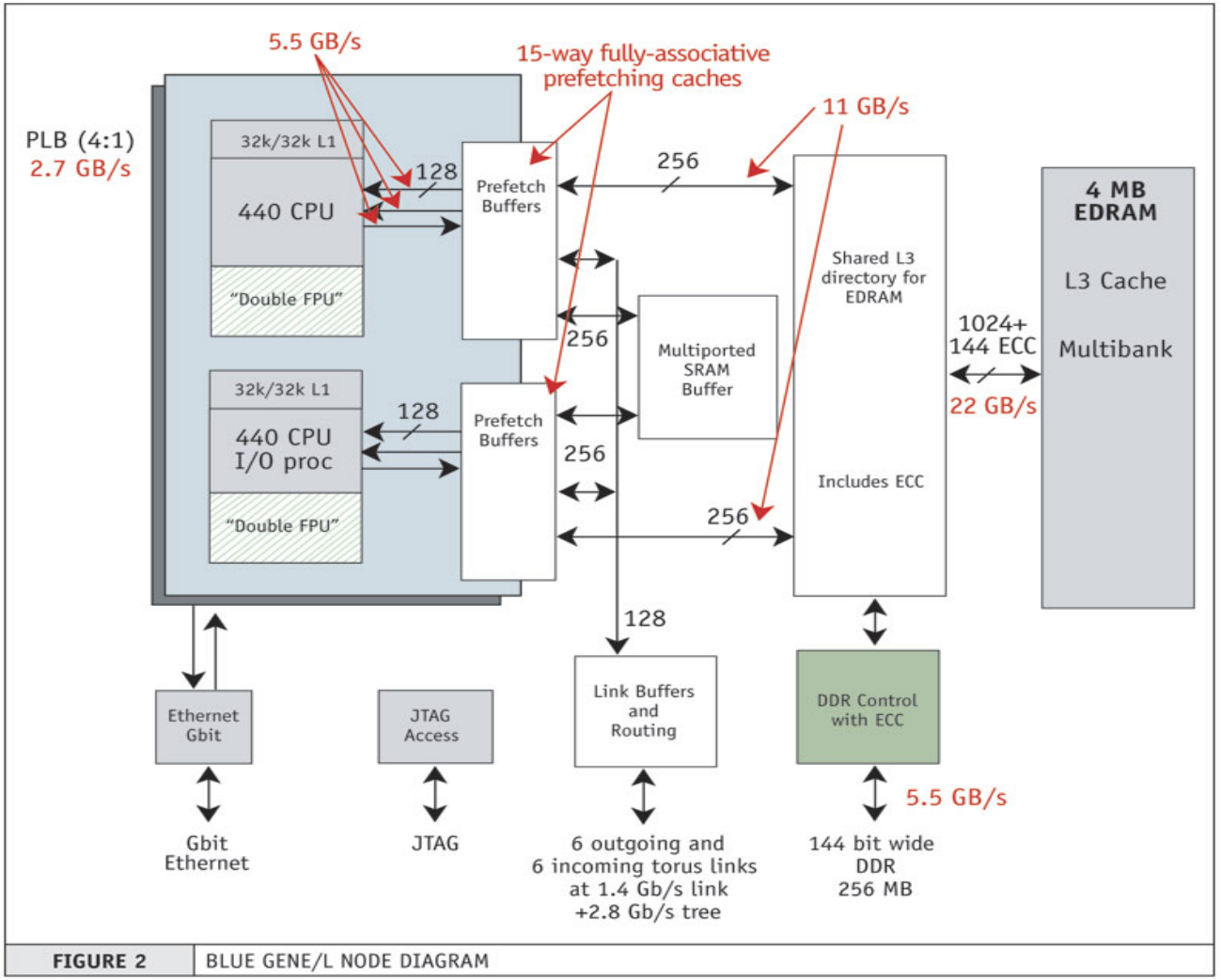
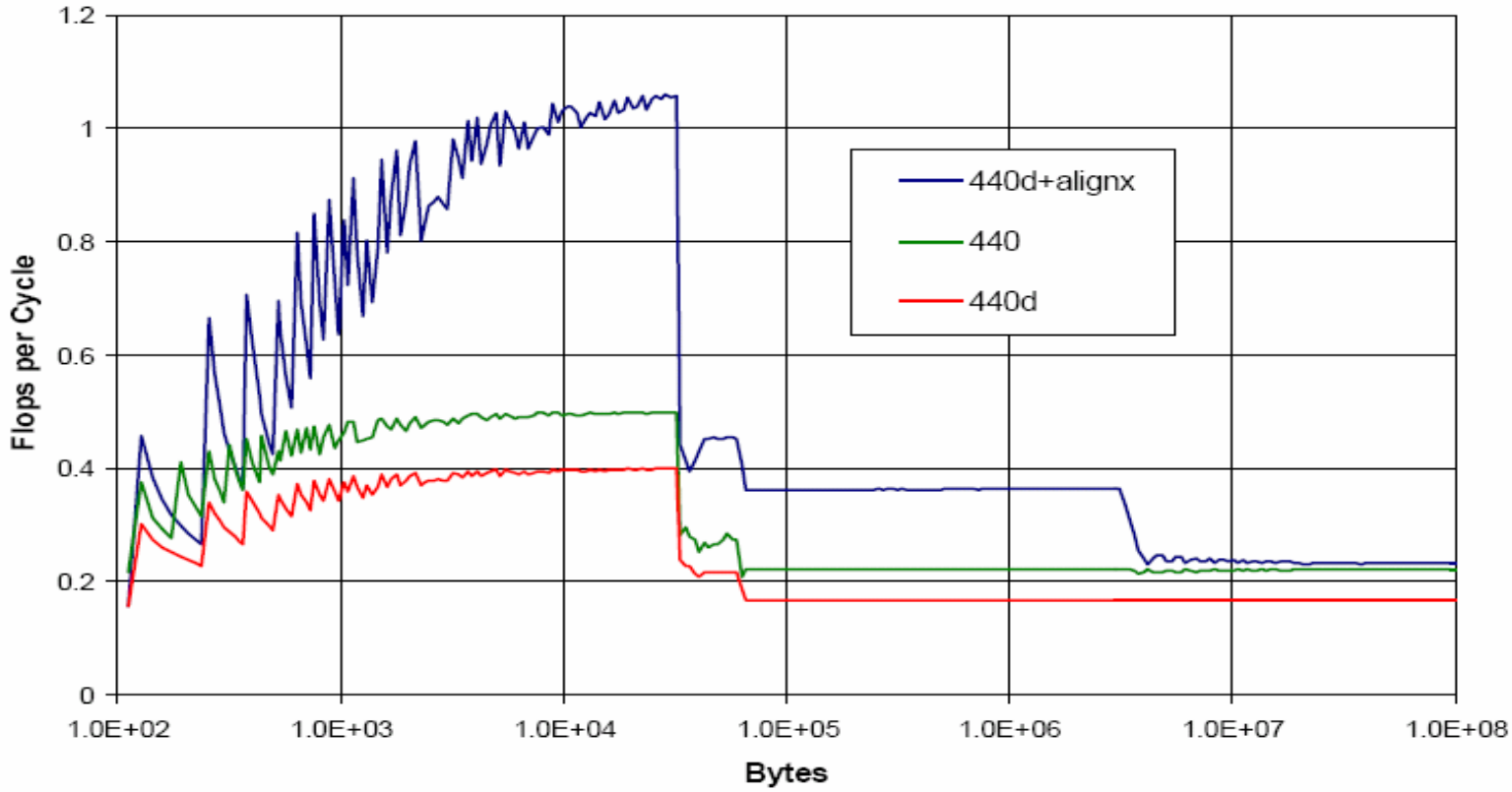


FIGURE 2 BLUE GENE/L NODE DIAGRAM

ppc440d performance data

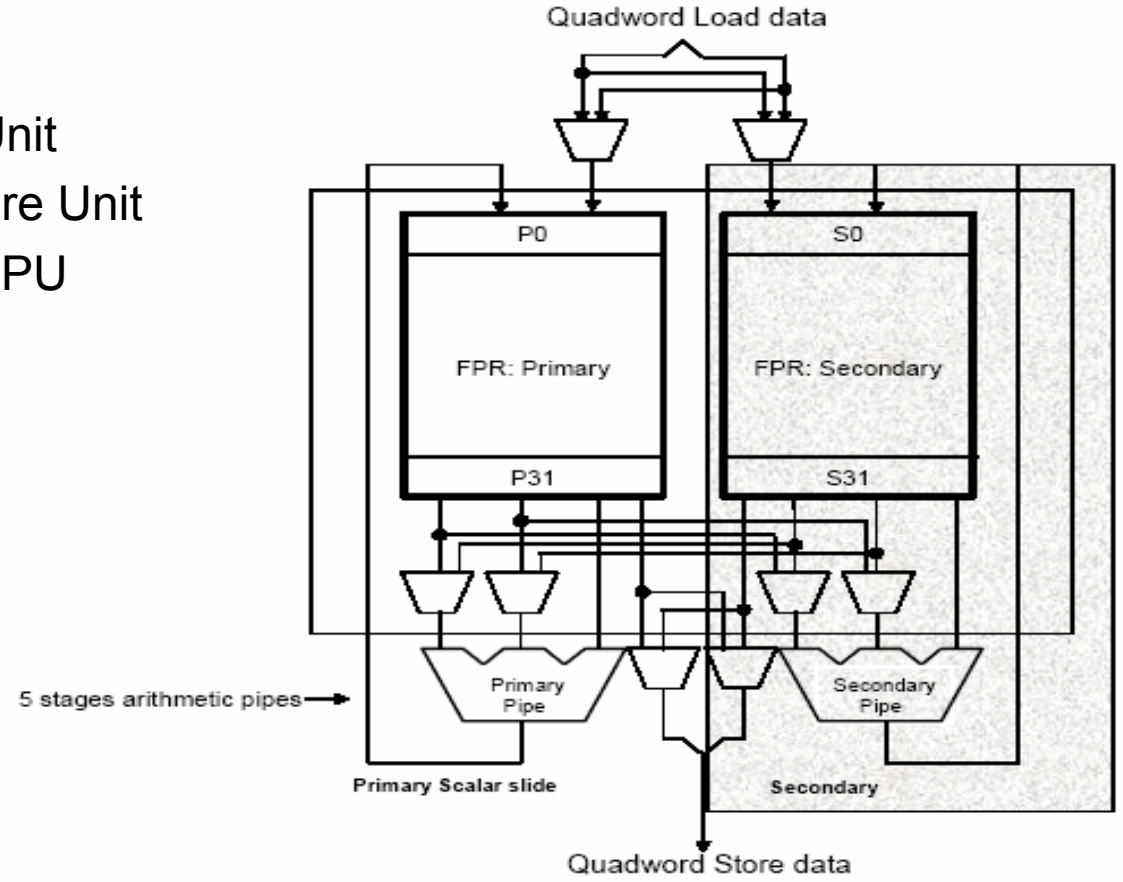
(VN MODE)	L1	L2	L3	Memory
Size/proc.	32KB data 32KB instr.	2KB	2MB	256MB
Latency (Clocks)	3	11	28/36/40 (h/m/m,b)	86
Bandwidth, random(B/clk)	NA	NA	1.8/1.2 (h/m)	0.5
Bandwidth, seq. (B/clk)	16.0	5.3	5.3	3.4
Line width	32	128	128	

Memory system Performance (daxpy)

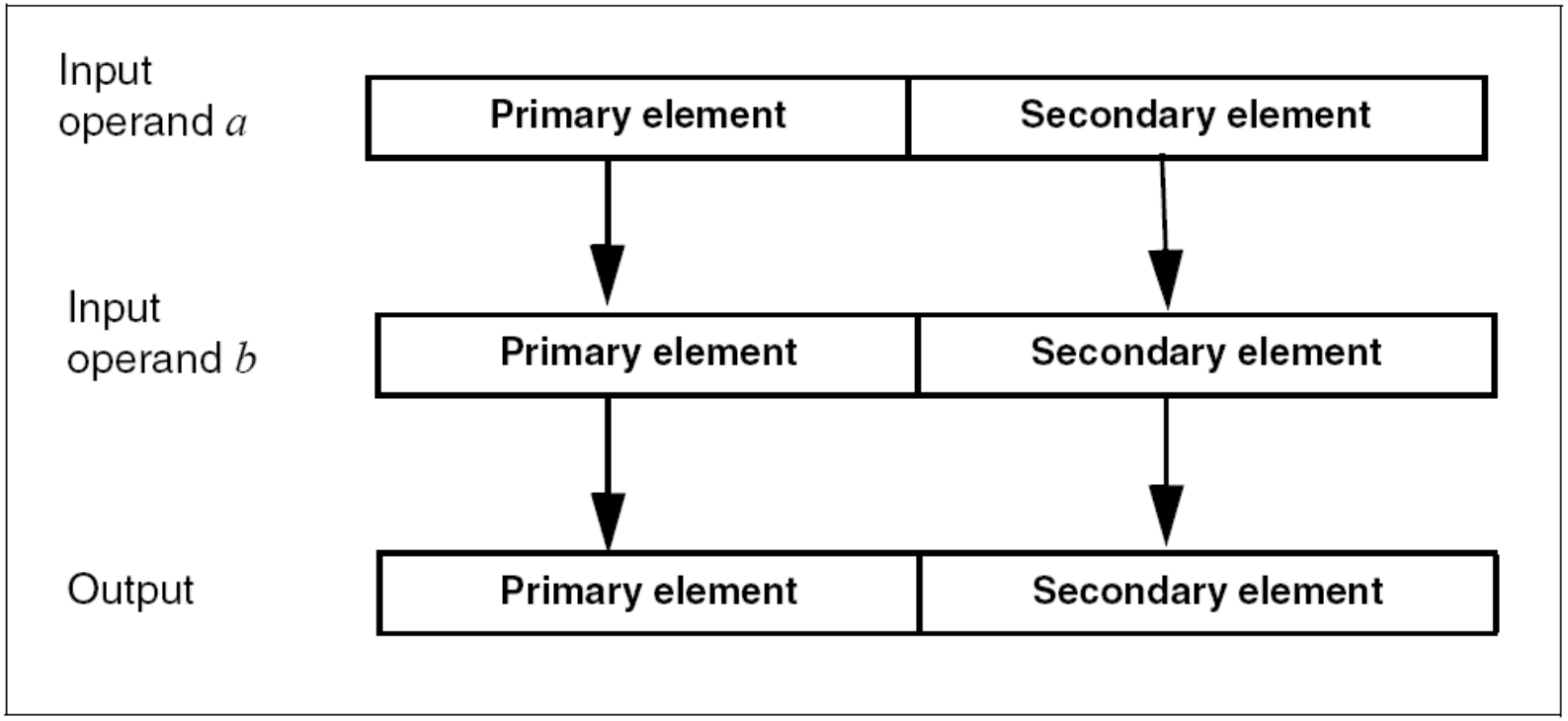


ppc440d “Double Hummer” FPU

- CPU contains
- One Integer Unit
- One Load/Store Unit
- One special FPU

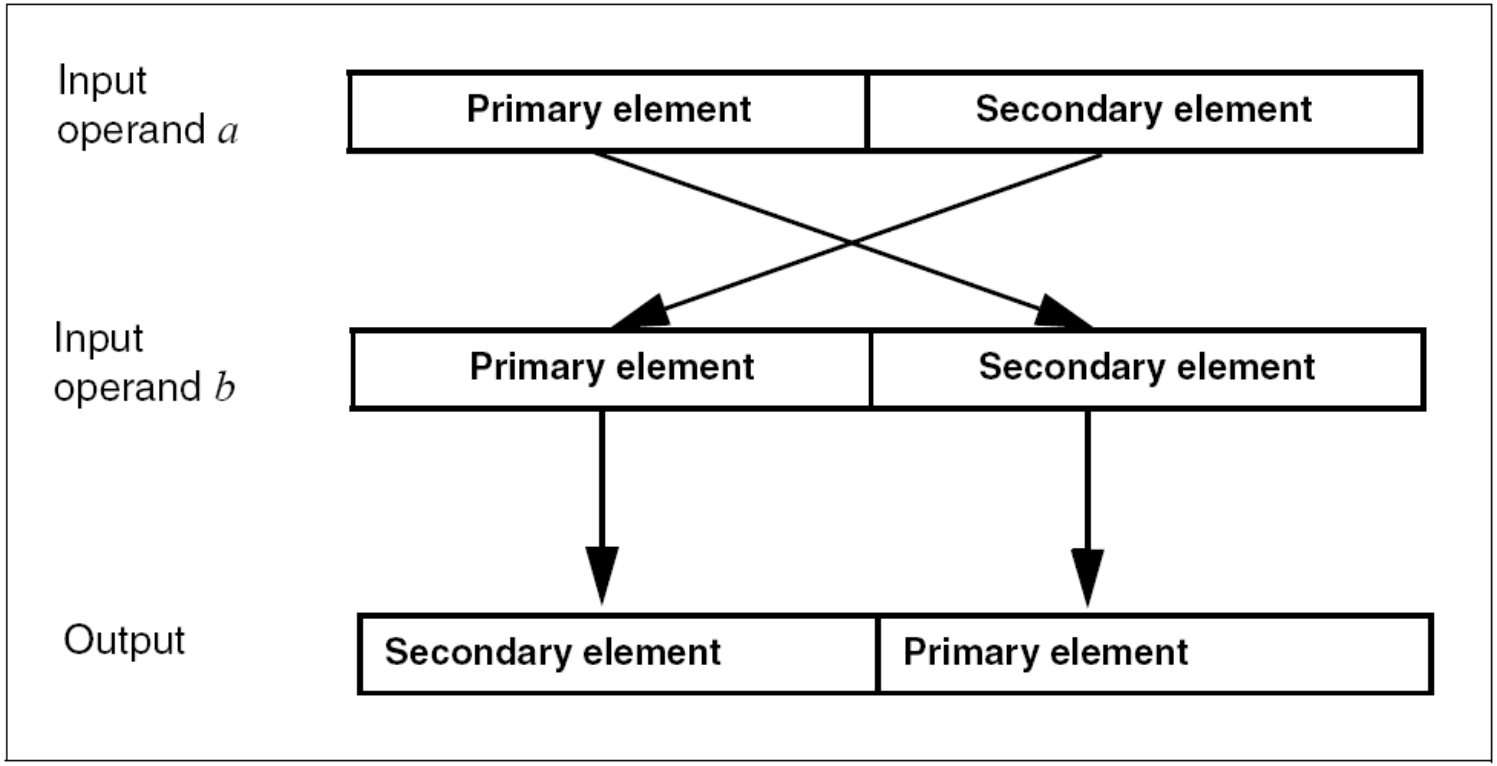


ppc440d “Double Hummer” FPU



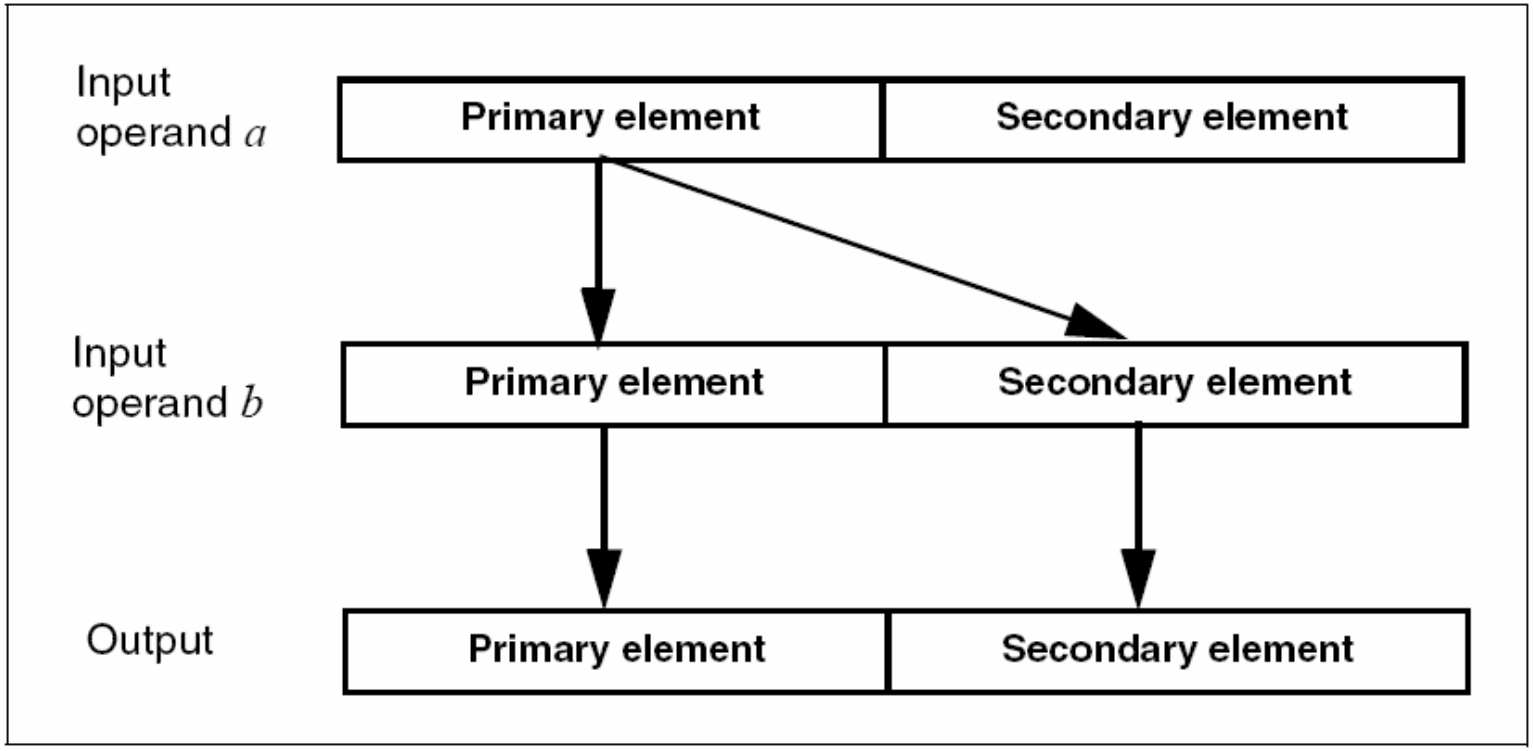
Parallel (vector proc. like) operations: fpadd, fpmul,… → fpmadd, …

ppc440d “Double Hummer” FPU



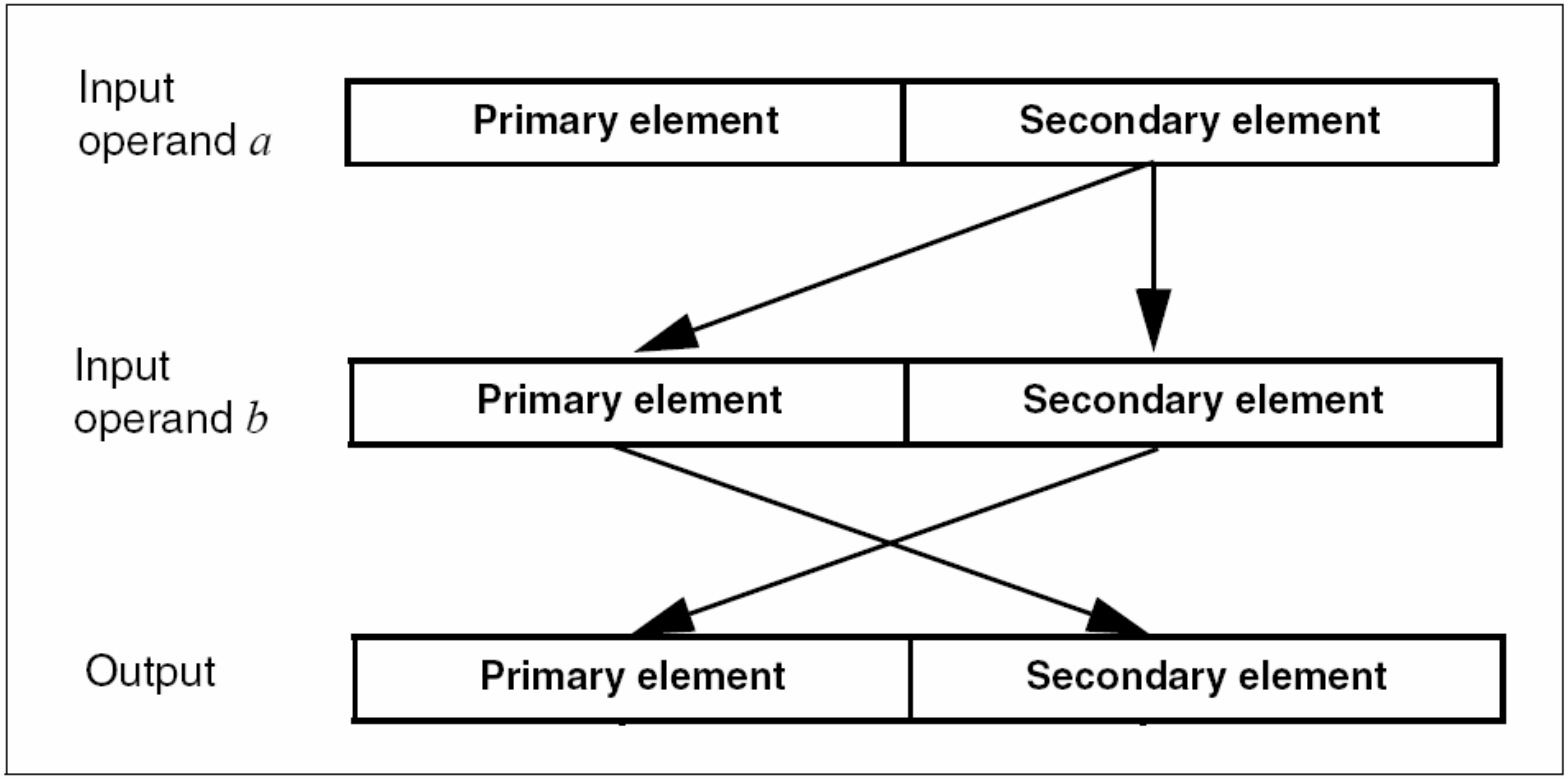
Cross operations: fxmul,… → fxmadd,…

ppc440d “Double Hummer” FPU



Cross copy (primary) operations: fxpmadd,… → Cross copy secondary ops

ppc440d “Double Hummer” FPU



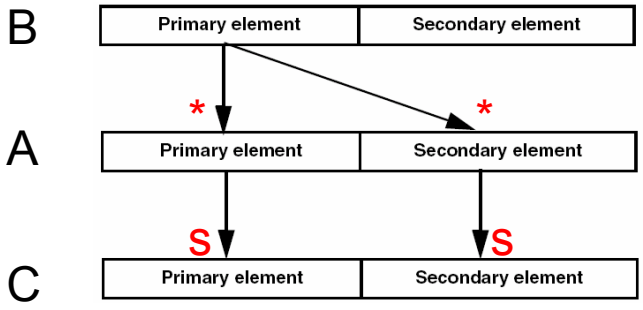
Cross mixed (secondary) operations: fxcxma,… → sub primary: fxcxnpma

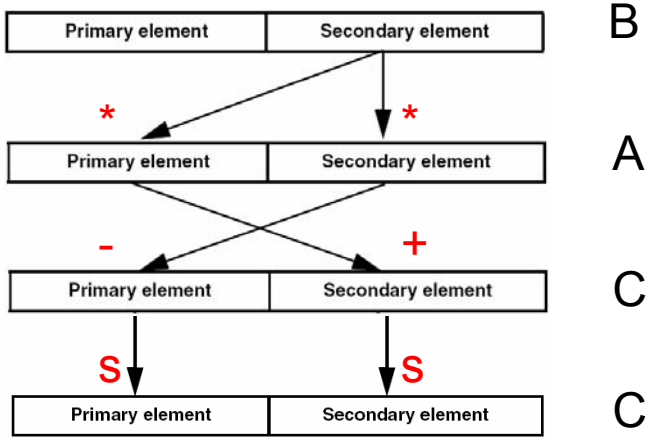
ppc440d “Double Hummer” FPU

- Blue Gene Compute Chip contains 2 FPUs with 32 registers each, second set only available indirectly via Double Hummer (“oedipus”) instructions
- X-Loads: Registers can be accessed from both FPUs
- Instruction set contains special instructions to implement complex algebra

Mnemonic	Primary Ap=	Secondary As=
fpadd A,B,C	Bp+Cp	Bs+Cs
fxpmul A,B,C	Bp*Cp	Bp*Cs
fxcxnpma A,B,C,D	-(Bs*Cs-Dp)	Bs*Cp+Ds
fxcpmadd A,B,C,D	Bp*Cp+Dp	Bp*Cs+Ds

ppc440d “Double Hummer” FPU





Complex multiply (A*B=C):

Re(C) =(Re(A)*Re(B) – Im(A)*Im(B))
Im(C) =(Im(A)*Re(B) + Re(A)*Im(B))

Required:

- a cross copy primary multiplication (2+1 register operands)
- a cross mixed negative secondary multiply – subtraction (3+1 register operands)

ppc440d “Double Hummer” FPU

- Complex number multiply requires 2 instructions
  (1 cycle each, 5 cycle pipeline latency)

  fxpmul A, B, C
  …
  fxcxnpma E, B, C, A

- SU(3) matrix vector multiply requires 2*9 instructions

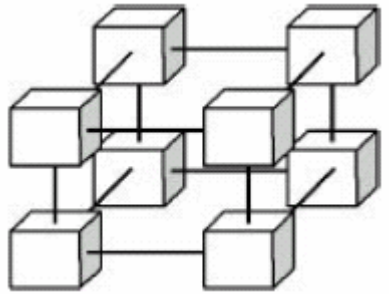
  … fxpmul … fxcxnpma … (fxcpmadd … fxcxnpma …)

  Matrix multiply has theoretical performance of 91.7% peak.

Blue Gene/L network overview

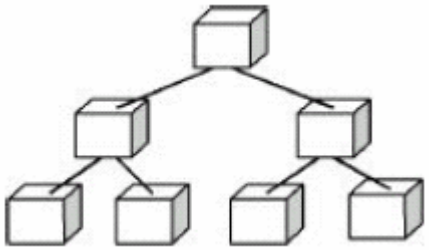
Blue Gene/L has 5 independent networks:

- Torus Network:
  12 links at 1.4 Gb/s
- Tree Network at 2.4 Gb/s



And:

- Tree Network for barriers and interrupts
- Gigabit Ethernet for file i/o
- Control Network



Torus network

- A Blue Gene/L node has 2*(3+1)+2*7 torus fifos
- (send 0/1: A, B, C, P; recv 0/1: ±X, ±Y, ±Z, P)
- Fifos are memory mapped: Writing to a fifo is storing to a *fixed* address
- All fifos are serviced simultaneously by the hardware
- Status registers can be polled to indicate send/receive status:
  - Check for enough space in a send fifo
  - Check for arrived data in a recv fifo
- Packet destination is defined in a 16Byte header
- All nodes in a partition can be addressed directly
- (RTS an) hardware routes packages

Torus network paket

Csum				
Hint Bits(6)			…	Pid
Size(3)	%	%	Dy	VC(2)
X target(8)				
Y target(8)				
Z target(8)				
…				
Payload				
…				

- {X,Y,Z} target: Coordinates of target node within job partition
- Hint bits: indicate the directions a paket wants to go to (+x, -x, +y, -y, +z, -z).
- Pid = Processor ID: Defines the reception-fifo group where the paket is to be received (group 0 or 1).
- Size: There are 8 packet sizes (+ four link protocol bytes at the end of each packet)
- %: two free bits that are left unchanged by the hardware (used by software).
- Dy = dynamical bit: Indicates adaptive or deterministic routing.
- VC: indicates one of 4 virtual channels.

Torus network & memory comms

- 8 hardware packet sizes are available. Useful for LQCD:
- message ≤ 16Byte → 32Byte hardware packet (complex number)
- message ≤ 112Byte → 128Byte hardware packet (2spinor)
- message ≤ 240Byte → 256Byte hardware packet (max size)
- +…
- You have to send a full package, that is you have to fill up with junk should your payload be too small

- Inter node communication can be done via scratchpad (uncached area in L3) or better: shared low latency SRAM → see V1R3 update
- “Lockbox” registers can (as barrier, …) to synchronize comms

Blue Gene/L runtime modes

Co-processor Mode:

- CPU0 does all computations
- CPU1 does all communications
- Communication may overlap with communication
- Peak performance is 5.6/2 = 2.8 GF
- #MPI-tasks = #Chips

Virtual Node Mode:

- CPU0, CPU1 run independent “virtual tasks”
- Each does own computation and communication
- CPUs communicate via cache/memory
- Communication and computation cannot overlap
- Peak performance is 5.6 GF
- #MPI-tasks = #CPUs

NIC Jülich BlueGene/L Installation Spec's

8 Racks à (2x16)x32 compute nodes:

- Two PPC440 770 MHz CPUs per compute node
- Main Memory: 512MB per Node (aggregate 4.1 TB)
- I/O Nodes: 288 (32 Nodes/pset)
- Networks:
  - 3D Torus nearest neighb.
  - Global tree/Collective
  - Gigabit Ethernet (I/O nodes)
- Peak Performance: 44.8 TFlops
- Linpack: 36.49

Service Node IBM p720:

- 8 Power5 1.6 GHz CPUs
- Total Memory: 16 GB
- Running SUSE Linux Enterprise

Login Node IBM p720:

- 8 Power5 1.6 GHz CPUs
- Total Memory: 8 GB
- Running SUSE Linux Enterprise

- jubl.zam.kfa-juelich.de

NIC Jülich BlueGene/L Installation Spec's

I/O Nodes / Login & Service Node

NFS, 4 GBit
→ 1Q06: GPFS, 16 GBit

IBM p690 (JUMP)

I/O Partitions on Login Node

Fiberchannel
ca. 12 GBit per FS

8 FastT 900 racks

- ca. 112 Fiberchannel HDD per Rack à 80 GB
- Total Diskspace ca. 75 TB
- 4 hot-spare disks per rack
- Access: GPFS

Fiberchannel
ca. 4 GBit
aggregate
bandwidth

16 STK Tape Robots

- 14,000 Tapes à 200 GB
- Total capacity: 2.8 PB
- Access: TSM/HSM

NIC Jülich Blue Gene/L Installation

Coding for performance

```
QuadLoadU(a, 10, o);
    asm volatile ( "fpmadd    6, 20, 0, 10");
QuadStoreU(g, 6, o);
    asm volatile ( "fpmadd    7, 21, 1, 11");
QuadLoadU(b, 11, o);
    asm volatile ( "fpmadd    8, 22, 2, 12");
QuadStoreU(g, 7, o);
    asm volatile ( "fpmadd    9, 23, 3, 13");
QuadLoadU(c, 12, o);
    asm volatile ( "fxcxnsma  5, 24, 4, 14");
QuadStoreU(g, 8, o);
    asm volatile ( "fxcxnsma 15, 16, 17, 18");
QuadLoadU(d, 13, o);
    asm volatile ( "fxcxnsma 25, 26, 27, 28");
QuadStoreU(g, 9, o);
    asm volatile ( "fxcxnsma 19, 29, 30, 31");
QuadLoadU(e, 14, o);
    asm volatile ( "fpmadd    6, 20, 0, 10");
QuadStoreU(g, 9, o);
    asm volatile ( "fpmadd    7, 21, 1, 11");
QuadLoadU(e, 14, o);
```

Using the IBM XLC compiler

- The XLC attempts to simdize code if –qarch=440d is used
- However: Simdization of generic C-code is complicated due to alignment issues
- The Double Hummer FPU can only operate on quad-aligned data (address on 16 byte boundary)
- Inside a function the alignment of a pointer may be unknown
- XLC will try to use primary FPU for padding
- Resulting performance is often poor and below –qarch=440, that is when only one of the FPUs is used
- Possible gains by helping the compiler understand the code:
  → use #pragma alignx , #pragma unroll to help avoiding pipeline stalls

Using the IBM XLC compiler

- Alternative: Write code with built-in floating point instructions (“intrinsics”), compile with –qarch=440d flag
- Intrinsics are peudo-assembly instructions
- User has to select the Double Hammer assembly instruction (see IBM Redbook BGL Application Development, chapter IV)
- Compiler will do scheduling of instructions
- #pragma’s are still required!
- Performance of code written using the intrinsics is usually much higher than performance of c
- The ppc440 cache manipulation instructions like DCBT are also available and their usage improves performance significantly

Example: Spinor algebra with intrinsics

```
void spinor_caxpy(double a[2], double *x,
      double *y){
#pragma disjoint (*x, *y)
   __alignx(16, x);
   __alignx(16, y);
   int i;

#pragma unroll(12)
   for(i=0; i<vol*12; i++){
      y[2*i   ] += a[0] * x[2*i] – a[1] * x[2*i+1];
      y[2*i+1] += a[1] * x[2*i] + a[0] * x[2*i+1];
   }
}
```

ESSL is now available on BGL, however spin. alg. routines written in above fashion seem to outperform the included BLAS

```
void spinor_caxpy(double a[2], double *x,
      double *y){
#pragma disjoint (*x, *y)
   __alignx(16, x);
   __alignx(16, y);
   int i;
  double _Complex x00,…,y00,…,t00,…;
  double _Complex alpha;
  alpha=__cmplx(a[0], a[1]);

#pragma unroll(12)
   for(i=0; i<vol; i++){
      x00=__lfpd(&x[12*i]);…
      y00=__lfpd(&y[12*i]);…
      t00=__fxcpmadd(y00,x00,__creal(alpha));…
      t13=__fxcxnpma(t00,x00, __cimag(alpha));…
      __stfpd(&y[12*i]);…
   }
}
```

Example: SU(3) matrix-vector multiplication

```
tm0  = __fxpmul(gaugex00, __creal(tmp_px_2s000));
tm1  = __fxpmul(gaugex11, __creal(tmp_px_2s001));
tm2  = __fxpmul(gaugex22, __creal(tmp_px_2s002));

tm3  = __fxpmul(gaugex01, __creal(tmp_px_2s011));
tm4  = __fxpmul(gaugex12, __creal(tmp_px_2s012));
tm5  = __fxpmul(gaugex20, __creal(tmp_px_2s010));

tm6  = __fxcxnpma(tm0, gaugex00, __cimag(tmp_px_2s000));
tm7  = __fxcxnpma(tm1, gaugex11, __cimag(tmp_px_2s001));
tm8  = __fxcxnpma(tm2, gaugex22, __cimag(tmp_px_2s002));

tm9  = __fxcxnpma(tm3, gaugex01, __cimag(tmp_px_2s011));
tm10 = __fxcxnpma(tm4, gaugex12, __cimag(tmp_px_2s012));
tm11 = __fxcxnpma(tm5, gaugex20, __cimag(tmp_px_2s010));

tm0  = __fxcpmadd(tm7, gaugex10, __creal(tmp_px_2s000));
tm1  = __fxcpmadd(tm8, gaugex21, __creal(tmp_px_2s001));
tm2  = __fxcpmadd(tm6, gaugex02, __creal(tmp_px_2s002));

tm3  = __fxcpmadd(tm10,gaugex11, __creal(tmp_px_2s011));
tm4  = __fxcpmadd(tm11,gaugex22, __creal(tmp_px_2s012));
tm5  = __fxcpmadd(tm9, gaugex00, __creal(tmp_px_2s010));
```

```
tm7  = __fxcxnpma(tm0, gaugex10, __cimag(tmp_px_2s000));
tm8  = __fxcxnpma(tm1, gaugex21, __cimag(tmp_px_2s001));
tm6  = __fxcxnpma(tm2, gaugex02, __cimag(tmp_px_2s002));

tm10 = __fxcxnpma(tm3, gaugex11, __cimag(tmp_px_2s011));
tm11 = __fxcxnpma(tm4, gaugex22, __cimag(tmp_px_2s012));
tm9  = __fxcxnpma(tm5, gaugex00, __cimag(tmp_px_2s010));

tm0  = __fxcpmadd(tm8, gaugex20, __creal(tmp_px_2s000));
tm1  = __fxcpmadd(tm6, gaugex01, __creal(tmp_px_2s001));
tm2  = __fxcpmadd(tm7, gaugex12, __creal(tmp_px_2s002));

tm3  = __fxcpmadd(tm11,gaugex21, __creal(tmp_px_2s011));
tm4  = __fxcpmadd(tm9 ,gaugex02, __creal(tmp_px_2s012));
tm5  = __fxcpmadd(tm10,gaugex10, __creal(tmp_px_2s010));

tm8  = __fxcxnpma(tm0, gaugex20, __cimag(tmp_px_2s000));
tm6  = __fxcxnpma(tm1, gaugex01, __cimag(tmp_px_2s001));
tm7  = __fxcxnpma(tm2, gaugex12, __cimag(tmp_px_2s002));

tm11 = __fxcxnpma(tm3, gaugex21, __cimag(tmp_px_2s011));
tm9  = __fxcxnpma(tm4, gaugex02, __cimag(tmp_px_2s012));
tm10 = __fxcxnpma(tm5, gaugex10, __cimag(tmp_px_2s010));
```

Using the GCC inline assembly

- A good way to generate Double Hummer code with good performance
- The programmer has to take care of everything:
  selecting the instruction, selecting the registers, scheduling the instructions
- The CPU does not necessarily obey the order of instructions
  → scheduling loads and prefetches can be more trail and error than …
- However this way one can use the QCD APIs provided by IBM
  (→ Pavlos Vranas IBM Watson, see Talk Jun Doi IBM Japan)

Communication with IBM QCD APIs

- The IBM QCD APIs are available in Juelich since the update to V1R3
- They allow using the full capabilities of the network:
  (see Talk of J. Doi, Boston conference:)
  - API function to prepare packet header
  - API macros to send/recv data from/to FPU register
    - Communication between node (XYZ) and between CPU (T) can be handled in same way
    - These macros are used with inline assembly to optimize instruction pipeline with computations
  - API function for internal barrier between 2 CPUs
  - API functions to send / recv through user buffer
    - These functions are used if we do not want to use inline assembly

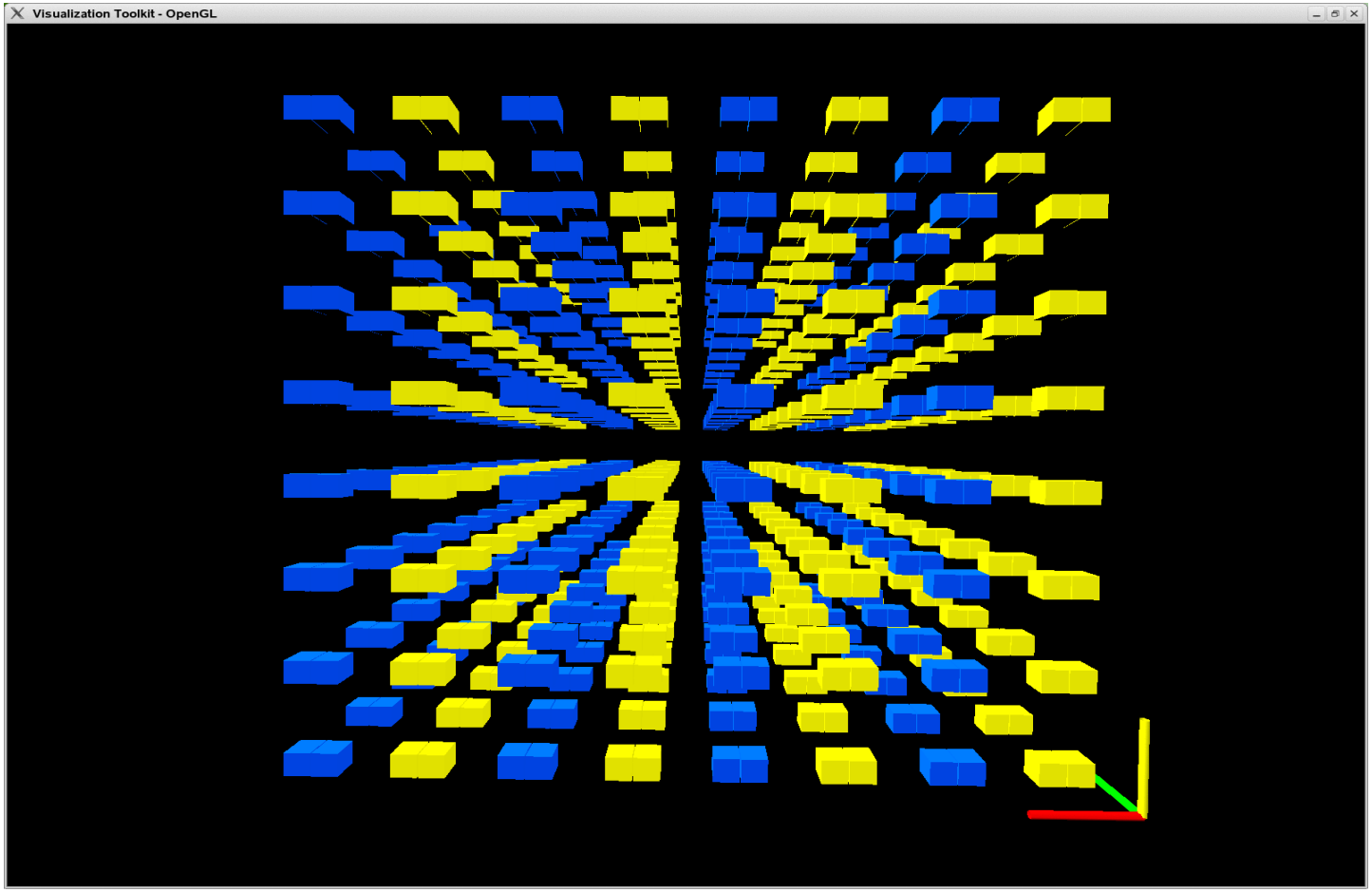
Communication with IBM QCD APIs

- Strategies for communication:
- Communicate buffers:
  - Collect data in buffer, communicate buffers while making sure to feed all torus injection fifos simultaneously
  - Use time while packets on the fly to communicate local directions
- Communicate directly from registers after spin projection (and color multiplication)
  - Each fifo has 1KB buffer space
  - Communication can be better hidden behind calculations
- In each case: To maximize bandwidth have one CPU communicating in +, the other in – direction, then swap their roles.

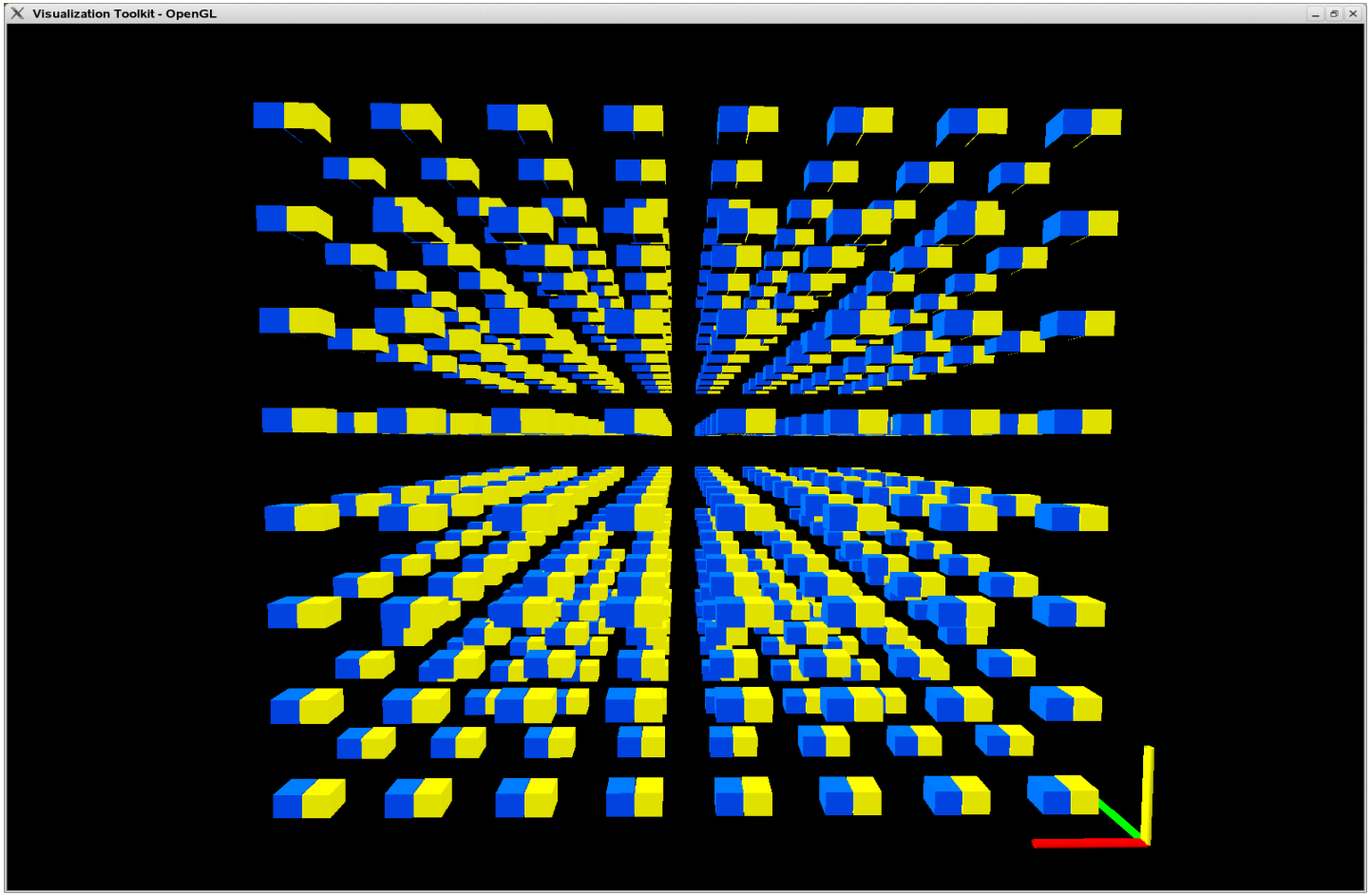
Communication with MPI

- The MPI performance is very good, however this is more true for bandwidth than for latency (surprise…)
- However they can obviously only be used to communicate buffers
- LQCD kernel: For small volumes the Kernel using APIs may prove to be as much as twice as fast as the MPI version
- For typical lattice sizes the difference is smaller.
- MPI Wilson Matrix multiplication peaks at >18%,
  (using persistent sends)
- Remarks: Make sure to use the right communicator
  (compare bgl_get_personality with MPI Cartesian coordinates)
  → Performance penalty may be severe.

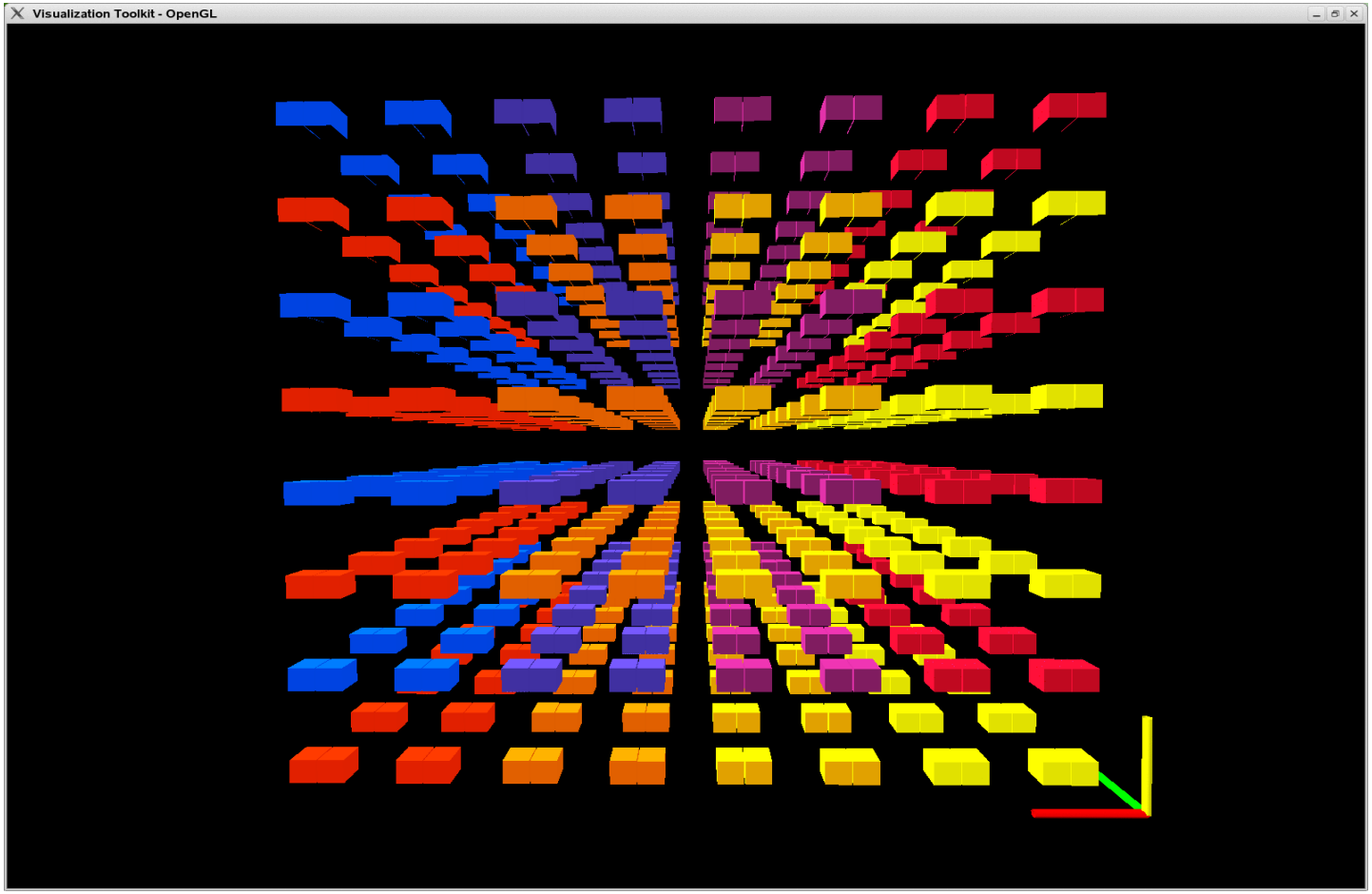
Dirac Matrix: Suboptimal Communicator T-Dir



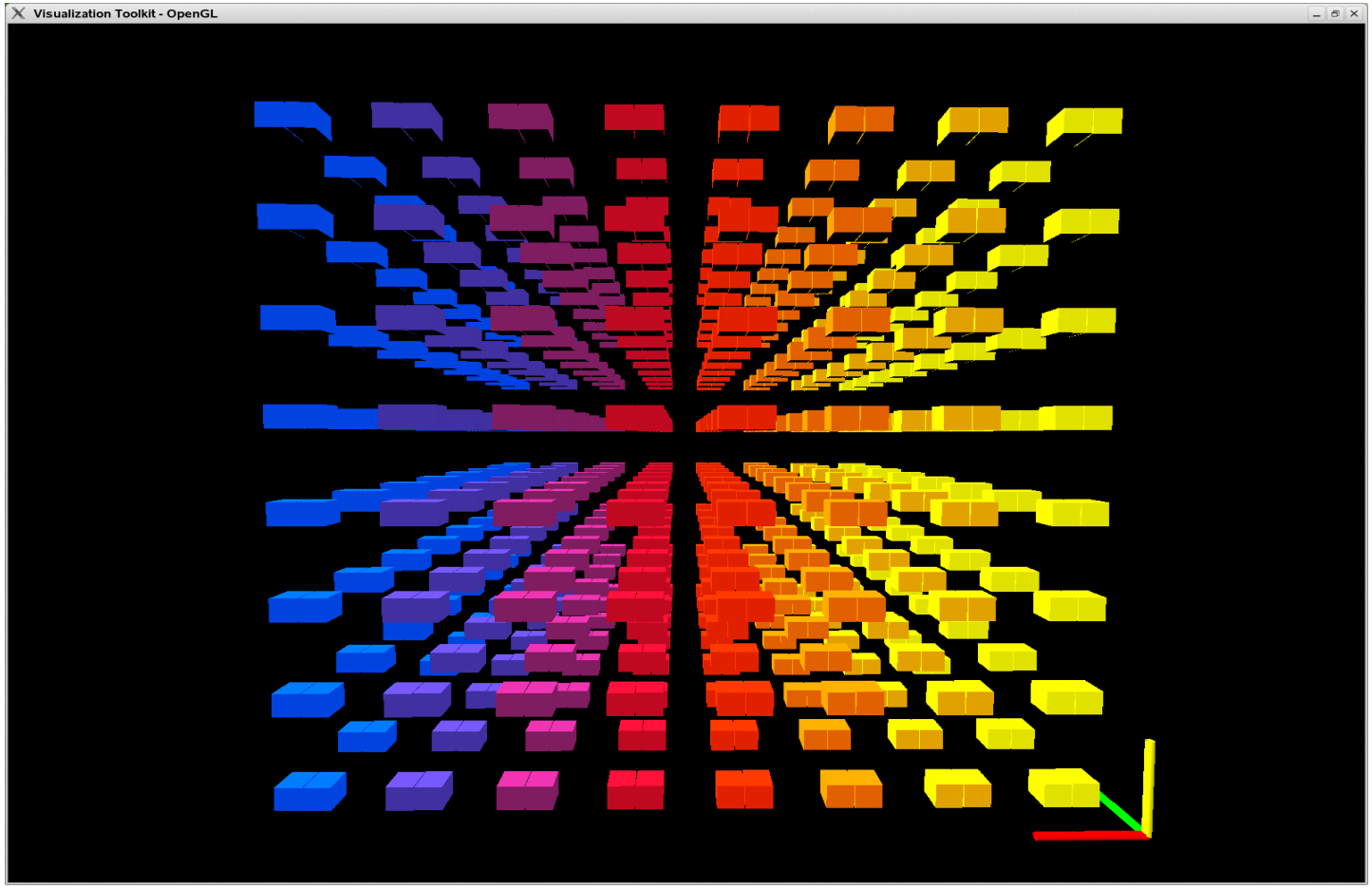
Dirac Matrix: Optimal Communicator T-Dir



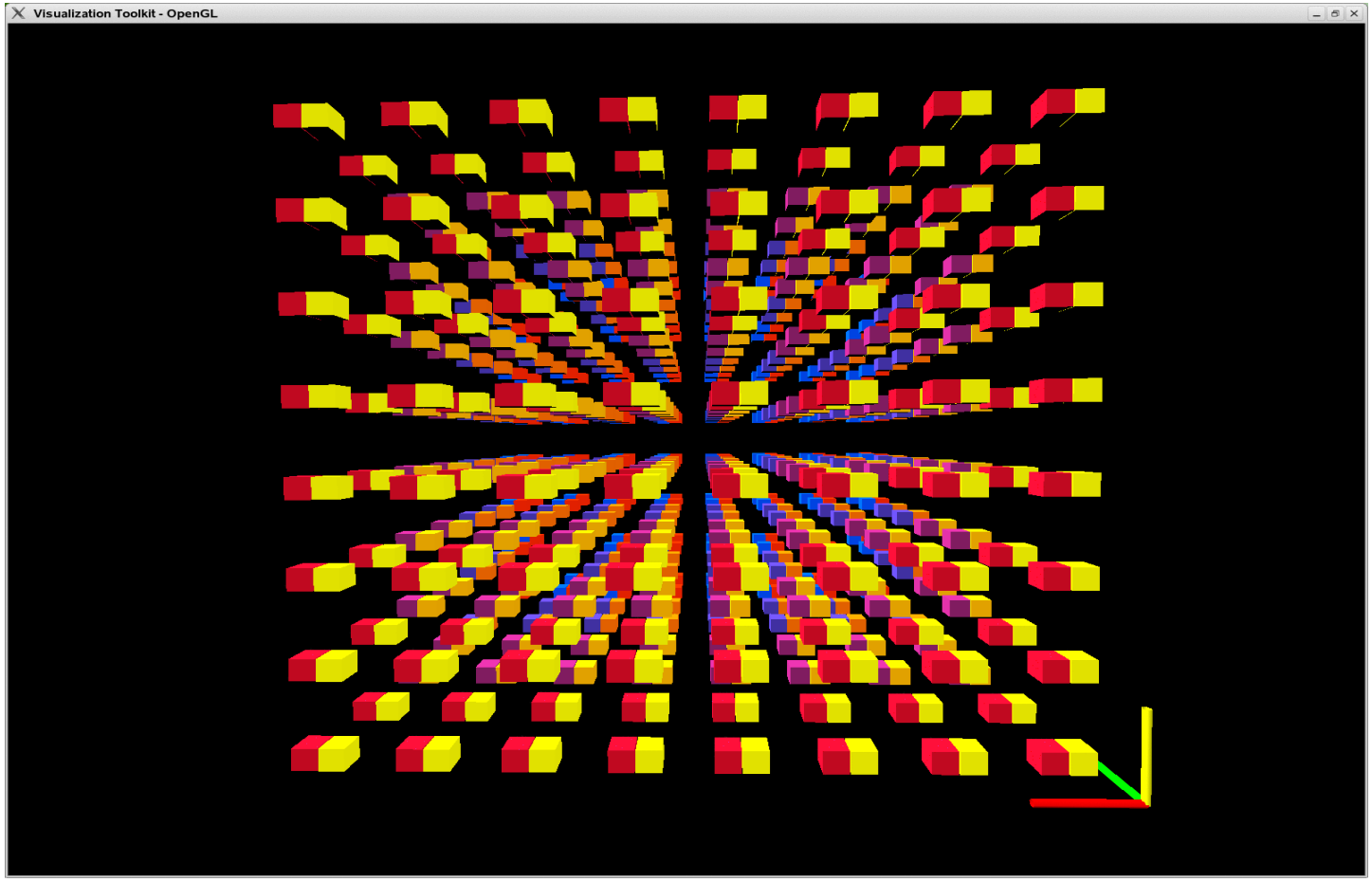
Dirac Matrix: Suboptimal Communicator Z-Dir



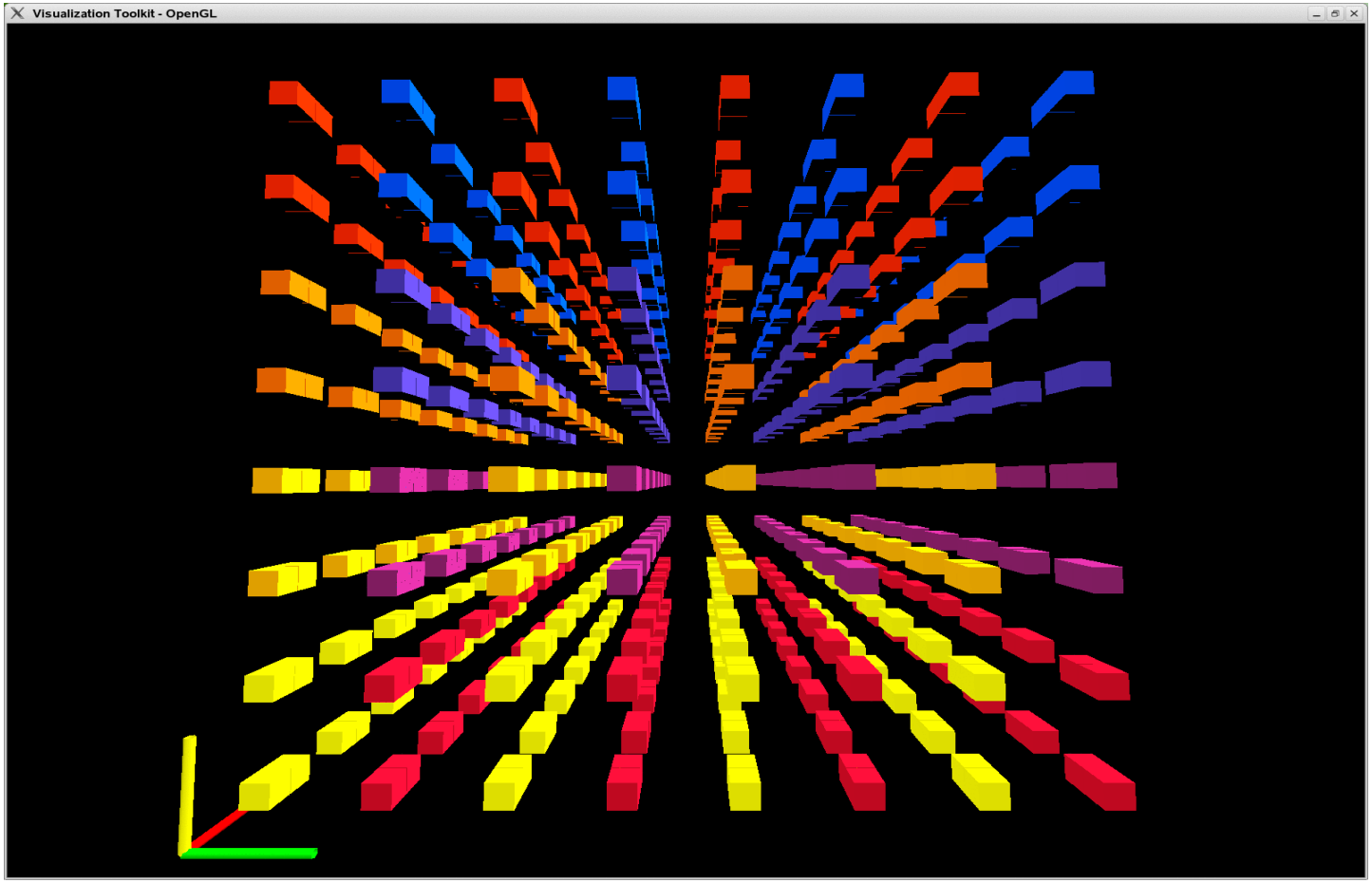
Dirac Matrix: Optimal Communicator Z-Dir



Dirac Matrix: Suboptimal Communicator X-Dir



Dirac Matrix: Suboptimal Communicator Y-Dir



Communication with MPI

There are a few of useful environment variables that can improve MPI performance:

- **BGLMPI_MAPPING=XYZT**
  - Influence the placing of the MPI processes to the nodes.
- **BGLMPI_PACING**=Y/N
  - Use/Don't use packet pacing (rendezvous protocol). Rendezvous protocol implies a node only sending when requested to do so by destination node.
- **BGLMPI_EAGER**=1000(default)
  - Set threshold message size from where MPI will switch from eager to rendezvous protocol.

Environment Variables

There are few more environment variables that influence the performance of the scalar part of the code

- **BGL_APP_L1_SWOA**=0/1
  - Caching strategy. Switch on/off “store without allocate” for all nodes.
- **BGL_APP_L1_WRITETHROUGH**=0/1
  - Performance on/off

Useful RTS calls

- **rts_get_personality(…)**:
  - Get node coordinates in partition (machine/physical coordinates)
  - Get partition size (X-Y-Z)
  - etc …

  rts_get_processor_id(…):
  - Get the id of the core on the node

  rts_get_timebase():
  - Get the value of the clock counter register. Usefull for timings.

Useful RTS calls

- **rts_get_scratchpad_window(…)**:
  - Can be used for comms (Possible conflicts with MPI!)
  - Get a window to a cache inhibited area in local memory

- **rts_rankForCoordinates(…)**:
  - Get the logical rank of the node.
  - Get the number of processors in the job

Useful RTS calls, V1R3 RTS version

(see the updated version of the IBM Redbook “Application Development”)

- New RTS release offers new system calls. Two of these are particularly interesting:
- **rts_get_dram_window(…)**: Allows to select the caching policy for a chunk of memory. However:
  - a total of 14 MB is available
  - Blocks can only be allocated in 1MB chunks
  - The application has to be linked differently (see Redbook for Details)

Useful RTS calls, V1R3 RTS version

- **rts_malloc_sram(…)**: Get a chunk of the shared SRAM. However:
  - Only a total of 8KB is available and MPI uses at least half of it
  - Can be allocated in chunks of 32Bytes only.

- Also:
  rts_get_virtual_process_window (…): See the other cores memory in virtual node mode

  →This might be a rather easy way to avoid the local comms…
  if one keeps in mind that the 1st level caches are not coherent

Outlook: QCD on Blue Gene/P

- Blue Gene/P features
  - a 4 way SMP node (with double FPUs)
  - higher clock frequency
  - Increased torus bandwidth to keep up with the increased compute performance
  - A fullblown DMA for communication (comms can be offloaded)
- Roughly speaking: Everything gets scaled by a factor of 2.4
- Kernel ported to BGP (with DMA comms) already reaches higher performance than BGL (measured in percent of peak)
- At this point, on BGP QCD gets **3130** MFlop/s per node compared to **1120** MFlop/s on BGL, **with room for improvement!**

Thank you for your attention!