Unfolding Blue Gene

Forschungszentrum Jülich

Michael Hennecke
HPC Systems Architect
hennecke >at< de.ibm.com
The Blue Gene/L project – Executive summary

- A research partnership with Lawrence Livermore National Laboratory – delivery of a 65,536-node machine in 2004/2005
  - 360 Tflops peak, 32 TBytes memory capacity
  - 1024 I/O nodes to cluster-wide parallel filesystem
- Support for our Blue Gene science program in investigating the mechanisms of biologically important processes, particularly protein folding
  - 20,480-node system being set up in IBM Watson
- IBM working with collaborators to better understand application behavior on Blue Gene/L, and broaden the scope of the machine
Blue Gene/L Project History

- **December 1999:** IBM Research announced a 5 year, 100M USD, effort to build a petaflop supercomputer to attack science problems such as protein folding. Goals:
  - Advance the state of the art in computer design and software for extremely large scale systems.
  - Advance the state of the art of biomolecular simulation.
- **November 2001:** Announced research partnership with Lawrence Livermore National Laboratory (LLNL)
- **November 2002:** Announced planned acquisition of a Blue Gene/L machine by LLNL as part of the ASCI Purple contract
- **June 2003:** First BLC chips (DD1) completed
- **November 2003 Top500:** ½ rack DD1 (512 nodes) ranked #73
- **February 2004:** Second pass BG/L chips (DD2) delivered to IBM Research
- **June 2004 Top500:** 4 racks DD1 (500MHz) IBM Rochester at #4, 2 racks DD2 (700MHz) IBM Yorktown at #8
Blue Gene/L Project History (continued)

- November 2004 Top500: first 16 racks at LLNL become #1 at 70.72 TFLOP/s (RCH,YKT now #8,#15)
- December 2004: First Customer acceptance received
- January 2005: First Customer Administration Class
- February/March 2005: ITSO Residency "Commercializing BG/L"
- March 2005: Deep Computing Capacity on Demand Center with BG/L
- June 2005: 16 systems on Top500 (32 racks at LLNL again #1, 20 racks IBM YKT #2, 6 racks Astron #6, 4 racks AIST, EPFL at #8, #9)
- Nov. 2005: 19 systems on Top500, 64 racks LLNL again #1, 20 racks IBM YKT #2, 6 racks Astron #9
- June 2006: 24 systems on Top500, FZ Jülich (8) #8
  In Top10: LLNL (64) #1, IBM YKT (20) #2, FZ Jülich (8) #8
- Nov. 2006: 28 systems on Top500, FZ Jülich (8) #13
  In Top10: LLNL (64) #1, IBM YKT (20) #3
- June 2007: 34 systems on Top500, FZ Jülich (8) #18
  In Top10: LLNL (64) #1, IBM YKT (20) #4, Stony Brook (18) #5, Rensselaer (16) #7
Blue Gene System Buildup

2 Midplanes (each 8x8x8)
2 * 16 = 32 Node Books
2 * 512 chips

Node Book
(32 chips 4x4x2)
16 compute, 0...2 I/O cards

Processor Card
2 chips, 1x2x1

Dual Processor

Rack
Cabled 16x8x8
1024 chips

2 * 16 = 32 Node Books
2 * 512 chips

16 compute, 0...2 I/O cards

2.8/5.6 TF/s
512 GB

90/180 GF/s
16 GB

5.6/11.2 GF/s
1.0 GB

System
64 Racks, 64x32x32
65,536 chips

180/360 TF/s
32 TB

5.6/11.2 GF/s
16 GB

1024 chips
Supercomputer Power Efficiencies

![Graph showing supercomputer power efficiencies over time with points marking different systems and years.](image)
Hardware Overview
Blue Gene System-on-a-chip ASIC

- Fused Multiply/Add ➔ 2 FLOPs/cycle
- Double FP pipeline ➔ 4 FLOPs/(cycle * FPU)
- 2 FPUs ➔ 8 FLOPs/cycle
- 700 MHz ➔ 5.6 GFLOPS/chip
- On-chip memory bandwidth matches this rate
- Huge eDRAM cache ~43 cycles latency
- External DRAMs only ~90 cycles latency
Blue Gene Hardware – Dual FPU Architecture

- Two 64 bit floating point units
- Designed with input from compiler and library developers
- SIMD instructions over both register files
  - FMA operations on double precision data
  - More general operations available with cross and replicated operands
    - Useful for complex arithmetic, matrix multiply, FFT
- Parallel (quadword) loads/stores
  - Fastest way to transfer data between processors and memory
  - Data needs to be 16-byte aligned
  - Load/store with swap order available
    - Useful for matrix transpose
Blue Gene Chip Design and Power Consumption

- IBM CMOS 0.13μ Cu-11 ASIC process technology
- 123mm² chip, 95M transistors

<table>
<thead>
<tr>
<th>Unit</th>
<th>Active power (W)</th>
<th>Size (cells)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Clock tree + access</td>
<td>1.15</td>
<td>264k</td>
</tr>
<tr>
<td>CPU/FPU/L1</td>
<td>7.54</td>
<td>14,700k</td>
</tr>
<tr>
<td>Torus network</td>
<td>0.67</td>
<td>4,963k</td>
</tr>
<tr>
<td>Collective network</td>
<td>0.25</td>
<td>2,350k</td>
</tr>
<tr>
<td>L2/L3/DDR control</td>
<td>2.6</td>
<td>18,310k</td>
</tr>
<tr>
<td>Others</td>
<td>0.49</td>
<td>10,720k</td>
</tr>
<tr>
<td>Leakage</td>
<td>0.2</td>
<td></td>
</tr>
</tbody>
</table>
Blue Gene Interconnection Networks

3-Dimensional Torus:
- Interconnects all *compute* nodes (max. 65,536)
- Virtual cut-through hardware routing
- 1.4Gb/s on all 12 node links (2.1 GB/s per node)
- Communications backbone for p2p computations
- 350 / 700 GB/s bisection bandwidth (on 64k system)

Global Tree / Collective Network:
- One-to-all broadcast; reduction operations
- 2.8 Gb/s of bandwidth per link
- Latency of tree traversal in the order of 2 µs
- Interconnects all compute *and* I/O nodes (max. 1024)

Gigabit Ethernet / Functional Network:
- Built into every BGC, active *only* in I/O nodes (min. 1:64)
- Used for external comm. (file I/O, control, user interaction)
Blue Gene Processor Card

- Mostly identical for Compute Node and I/O Node...
Blue Gene 32-way (4x4x2) Node Book

- **16 compute cards**
- **2 optional I/O cards**
- **Midplane connectors:** torus, tree, barrier, service Ethernet, clock, power
- **DC-DC converters**
- **Ethernet-JTAG FPGA**
- **Gigabit Ethernet connectors (Functional Network)**
- **Latching and retention**
Blue Gene Service Card
Blue Gene Link Card
Blue Gene 512-way (8x8x8) Midplane

- Link Card (without cables)
- Service Card
- 2-way I/O card
- 2-way compute card
- 32-way Node Book
Blue Gene Rack – without & with Torus Cabling
System Architecture
Blue Gene System Architecture

- **Service Node**
  - `mpirun`
  - `LoadL`

- **Front-End Nodes** (I/O Node 0)
  - `Linux`
  - `fs client`
  - `ciod`
  - **Control Ethernet**
  - **Functional Gigabit Ethernet**

- **File Servers** (I/O Node 1023)
  - `Linux`
  - `fs client`
  - `ciod`

- **Service Node** (C-Node 0)
  - `app`
  - `CNK`

- **Service Node** (C-Node 63)
  - `app`
  - `CNK`

- **MMCS**
  - `DB2`

- **System Console**
  - `mpirun`
  - `LoadL`

- **Control Ethernet**
  - `IDo chip`

- **I/O Node 1023**
  - `Linux`
  - `fs client`
  - `ciod`

- **C-Node 0**
  - `app`
  - `CNK`

- **C-Node 63**
  - `app`
  - `CNK`

- **Tree**
  - `Pset 0`

- **Torus**
  - `Pset 1023`
Blue Gene Hierarchical Software Organization

- **Compute nodes (CNs)** dedicated to running user application, and almost nothing else - simple compute node kernel (CNK)
- **I/O nodes (IONs)** run Linux and provide a more complete range of OS services – files, sockets, process management, debugging
- **Service node (SN)** performs system management services (e.g., heart beating, monitoring errors) – transparent to application
  - the SN is a pSeries server running SLES9, *not a BG node*
Blue Gene System Management Components

- **Master** creates, monitors, and restarts the other processes
- **Discoverer** finds and initializes new hardware
- **Proxy** virtualizes the IDo hardware, providing reliable and atomic connection
- **Monitor** monitors environmentals, such as temperature and voltages
- **MMCS** configures and IPLs partitions of the machine, controls jobs
- **DB2** stores all hardware information, environmental, RAS, *partition/job data*
Blue Gene Programming Environment

- **Linux (SLES9) cross-compilation environment**
  - User logs in to PPC frontend nodes for compilation, job submission, debugging

- **Space sharing**
  - Exactly one parallel job per BG/L partition
  - CO mode: one MPI process per node, comm. offloading
  - VN mode: two MPI processes per node, each 256 MB

- **XL Fortran, XL C/C++, and MPI message passing**
  - Virtual memory limited to physical memory
  - Libraries statically linked, some syscall limitations

- **SPMD model**
  - All compute nodes run the same executable (no scripts)
  - v1r3 driver has added limited fork()/exec() support for MPMD
Blue Gene Software Stack

Front-end Node
- GNU tools
- Debuggers
- mpirun
- LoadLeveler
- XL compilers
- Linux

Service Node
- Navigator
- LoadLeveler
- MMCS
- Proxy
- DB2 & Firmware
- Linux

I/O Node
- Debuggers
- CIOD
- File system
- Linux

Compute Node
- Application
- MPI
- Run-time
- Kernel (CNK)
How File I/O works in Blue Gene applications

- **Application**: fscanf
  - libc
    - read
  - CNK
    - Tree packets
  - BG/L ASIC

- **cioed**
  - read
  - read
  - data
  - Linux
    - *FS
    - IP
    - Tree packets
  - BG/L ASIC

- **File server**

- **Tree**

- **Ethernet**
Application development environment
Before porting to BG/L – port to pSeries Linux...

- **The first step towards Blue Gene porting**
  - Easy to do on a frontend node
  - No memory constraints
  - No system calls limitations (see below)
  - Can address all other p-Linux porting problems

- **Generate a gprof flat profile**
  - Shows where to concentrate tuning effort
  - may do all arch-independent tuning here
Some features not supported on BG/L

- **Replace timer calls:**
  - use rts_get_timebase() from librts.a, divide by 700MHz

- **No fork()** – compiles OK but generates run-time error...
  - and no associated syscalls: getppid(), wait(), waitpid()
  - limited fork/exec support in v1r3 SW release

- **No getrlimit(), getrusage(), ...**

- **No mmap() or shared memory**

- **No IP address**, e.g. gethostbyname()

- **Socket client calls works**, e.g. socket(), connect(),
  but server stuff doesn't, e.g. bind(), accept()

- **MPI / IO is now supported**
More BG/L porting tips

- **Blue Gene specific header files:**
  - `/bgl/BlueLight/ppcfloor/bglsys/include`

- **Use `#ifdef __blrts__`**
  - for special code for Blue Gene run time system

- **No virtual memory (and no ulimit for stack/data)**
  - `sbrk(0)` to check heap size
  - address of local variable to check stacksize

- **Good XL compiler flags to start with:**
  - `-qarch=440` (or 440d, see double-FPU later on)
  - `-qtune=440`  `-O3`  `-qhot`  `-g`
  don't use `auto` in a cross-compile environment (implied by `-O5`)
Sample Blue Gene makefile

BGLSYS = /bgl/BlueLight/ppcfloor/bglsys

CC       = /usr/bin/blrts_xlc
CPPC     = /usr/bin/blrts_xlc
FC       = /usr/bin/blrts_xlf90
OPTFLAGS= -qarch=440d -qtune=440 -O3 -qhot
CFLAGS= $(OPTFLAGS) -qlist -qsourc -g
        -I$(BGLSYS)/include -L$(BGLSYS)/lib
FFLAGS= ...
BGL_LIBS= -lmpich.rts -lmsglayer.rts -lrts.rts -ldevices.rts

# by convention, a.out on BG/L has extention ".rts" ...
myprog.rts:   myprog.o
        $(FC) $(FFLAGS) -o myprog.rts myprog.o $(BGL_LIBS)
Application Developer’s view of Blue Gene

- **Two CPU cores per chip, frequency is 700 MHz, each CPU can do 2 double precision multiply/adds**
  - peak performance is 2.8 or 5.6 GFlops/sec per node
  - 2.87 or 5.73 TFlop/s peak per rack (4.71 LINPACK)
- **3D torus network with virtual cut-through routing**
  - point to point: MPI_SEND, MPI_RECV
- **Global combine/broadcast tree network**
  - collectives: MPI_GATHER, MPI_SCATTER
- **Global interrupt network**
  - fast MPI_BARRIER
Two modes to use the hardware

**Communication coprocessor (CO) mode:**

- CPU0 does all the computations, has **512MB** memory
- CPU1 does (some) communication offload, so communication *may* overlap with computation
- Peak compute performance is 2.8 GFlop/s

**Virtual node (VN) mode:**

- CPU0, CPU1 run independent MPI task, each has **256MB** memory
  - Intra-CPU MPI via memory buffers
- Each does its own computation *and* communication, so computation and communication *cannot* overlap
- Peak compute performance is 5.6 GFlop/s
Starting parallel jobs – mpirun (see also FZJ’s llrun)

- **mpirun** has different ways to request CPUs:
  - `-np 32 [-mode CO]` # get 32 tasks
  - `-np 64 -mode VN` # get 32*2 tasks
  - `-shape 4x4x2` # get 32=4x4x2 tasks
  - `-shape 4x4x2 -np 16` # get 32, use only 16
  - `-partition=R00-M0-NC` # use that partition

- **some important BG/L specific options:**
  - `-connect {TORUS|MESH}` # only applies to >= 512
  - `-mapfile "mapfile-name"` # or symbolic, e.g. TXYZ
  - `BGLMPI_EAGER` environment variable

- **followed by your program name and other options:**
  - `-exe /full/path/to/your/a.rts`
  - `-cwd /full/path/to/your/workdir`
  - `-bgl_input "file" -args "..."
  - `-env "var=val" -exp_env "var-names"`
HPC Tools / Software for Blue Gene

- **XL Compilers**
  - Externals preserved
  - New options to optimize for specific Blue Gene functions
  - Specific Blue Gene XL runtime

- **LoadLeveler (RPQ)**
  - Same externals for job submission and system query functions
  - Backfill scheduling to achieve maximum system utilization

- **GPFS (RPQ)**
  - Provides high performance file access, as in pSeries and xSeries clusters
  - Runs on IO nodes and disk servers

- **ESSL (RPQ) / MASSV (comes /w XLF)**
  - Optimization library and intrinsic for better application performance
  - Callable from FORTRAN, C, and C++

- **MPI**
  - Message passing library, based on MPICH2, tuned for Blue Gene architecture

Other Software Support:

- **Etnus TotalView**
  - Parallel Debugger

- **GNU Compilers and gdb**
  - Ported to Blue Gene

- **FFT Library**
  - Tuned functions by TU-Vienna

- **Performance Tools**
  - HPC Toolkit
  - others...
Other application development tools

- **FEN is a standard SLES9@POWER environment**
  - full Linux environment with all available SW tools
  - but all these only run on FEN, not on BG/L itself...
  - ... and you cannot run ps or top on the CNKs

- **Debuggers**
  - gdb is shipped with BG/L (connects to gdbserver440)
    - through MMCS, but only to already running jobs
    - through mpirun, also to start job under debugger control
  - TotalView for BG/L is available
Other application development tools

- **Profiling**
  - prof/gprof on BG/L now fully supported
  - Xprofiler available on p-Linux as part of HPC toolkit

- **Hardware performance counters**
  - BG/L has Universal Performance Counter (UPC) unit
    - somewhat limited in usefulness, see details presentation
  - Low Level Interface: libbgl_perfctr.rts.a
  - Higher level interfaces: PAPI and HPM
The IBM High Performance Computing Toolkit

- **MP_Profiler**
  - for MPI performance measurements
- **Xprofiler and HPMCOUNTER**
  - for CPU performance
- **PeekPerf**
  - a common visualisation and analysis GUI
- **On FZJ „JUBL“, installed in /bgl/local/hpct_bgl/**

Not everything ready yet for BG/L, but work in progress to have the whole toolkit available eventually...
Xprofiler – call graph display
Xprofiler – flat profile display

![Xprofiler Flat Profile Display](image)
PeekPerf – display of MPI_Profiler data
Third party performance tools (work in progress)

- PARAVER (UPC @ U of Barcelona)

- KOJAK (ICL @ U of Tennessee and ZAM @ FZ Jülich)
  Kit for Objective Judgement and Knowledge-based Detection of Performance Bottlenecks

- PAPI (ICL @ U of Tennessee)
  Performance Application Programming Interface

- TAU (U of Oregon)
  [http://www.cs.uoregon.edu/research/paracomp/tau/tautools/](http://www.cs.uoregon.edu/research/paracomp/tau/tautools/)
  Tuning and Analysis Utilities

- mpiP (LLNL)
  Lightweight, Scalable MPI Profiling
BlueGene references


- Blue Gene system documentation (updates with new SW releases) (http://www.redbooks.ibm.com/):
  - Blue Gene/L: System Administration, SG24-7178-04
  - Blue Gene/L: Hardware Installation and Serviceability, SG24-6743-00
  - Blue Gene/L: Hardware Overview and Planning, SG24-6796-02
  - Blue Gene/L: Application Development, SG24-7179-04
  - Blue Gene/L: Performance Analysis Tools, SG24-7278-00
  - Blue Gene/L: Configuring and Maintaining Your Environment, SG24-7352-00
  - Blue Gene/L: Problem Determination Guide, SG24-7211-00
  - Unfolding the IBM eServer Blue Gene Solution, SG24-6686
  - Blue Gene/L: Safety Considerations, REDP-3983-01
  - GPFS Multicluster with the IBM System Blue Gene Solution and eHPS Clusters, REDP-4168
  - Evolution of the IBM System Blue Gene Solution, REDP-4247


BlueGene/P Update – System Buildup

- **System Card**
  - 32 Node Cards
  - Cabled 8x8x16
  - 72 Racks
  - 1 PF/s
  - 144 TB

- **Chip**
  - 4 processors
  - 13.6 GF/s
  - 8 MB EDRAM

- **Compute Card**
  - 1 chip, 20 DRAMs

- **Node Card**
  - (32 chips 4x4x2)
  - 32 compute, 0-2 IO cards

- **Rack**
  - 435 GF/s
  - 64 GB

- **System**
  - 14 TF/s
  - 2 TB
BlueGene/P Architectural Highlights

Scaled performance through density and frequency bump
- 2x performance through doubling the processors/node
- 1.2x from frequency bump due to technology (700 MHz → 850 MHz)

Enhanced function
- 4 way SMP – usable in three modes: SMP / DUAL / VNM
- L2, L3 changed to accommodate SMP
- DMA for torus, remote put-get, user programmable memory prefetch
- Memory chipkill implemented
- Greatly enhanced 64 bit performance counters (including ppc450 core)
- Double Hummer FPU and networks (Torus, Collective, Barrier, JTAG) architectures remain the same as in BG/L

Higher signaling rate
- 2.4x higher bandwidth, lower latency for Torus and Tree networks
- 10x higher bandwidth for Ethernet I/O

72K nodes in 72 racks for 1PF peak performance (target)
- Preserve compute cards, node cards, midplanes, power, and cooling dimensions as much as possible
- Low power through aggressive power management
## BlueGene/P Feature Comparison to BlueGene/L

<table>
<thead>
<tr>
<th>Property</th>
<th>BG/L</th>
<th>BG/P</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Node Properties</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Node Processors</td>
<td>2 * 440 PowerPC</td>
<td>4 * 450 PowerPC</td>
</tr>
<tr>
<td>Processor Frequency</td>
<td>0.7 GHz</td>
<td>0.85 GHz</td>
</tr>
<tr>
<td>Coherency</td>
<td>Software managed</td>
<td>SMP</td>
</tr>
<tr>
<td>L1 Cache (private)</td>
<td>32KB / core</td>
<td>32KB / core</td>
</tr>
<tr>
<td>L2 Cache (private)</td>
<td>14 stream prefetching</td>
<td>14 stream prefetching</td>
</tr>
<tr>
<td>L3 Cache size (shared)</td>
<td>4 MB</td>
<td>8 MB</td>
</tr>
<tr>
<td>Main Store/Node</td>
<td>512 MB, later 1 GB version</td>
<td>2 GB</td>
</tr>
<tr>
<td>Main Store Bandwidth</td>
<td>5.6 GB/s (16B wide)</td>
<td>13.6 GB/s (2*16B wide)</td>
</tr>
<tr>
<td>Peak Performance</td>
<td>5.6 GF/node</td>
<td>13.6 GF/node</td>
</tr>
<tr>
<td><strong>Torus Network</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Bandwidth</td>
<td>6<em>2</em>175MB/s = 2.1 GB/s</td>
<td>6<em>2</em>425MB/s = 5.1 GB/s</td>
</tr>
<tr>
<td>Hardware Latency (Nearest Neighbor)</td>
<td>200 ns (32B packet)</td>
<td>100 ns (32B packet)</td>
</tr>
<tr>
<td></td>
<td>1.6 us (256B packet)</td>
<td>800 ns (256B packet)</td>
</tr>
<tr>
<td>Hardware Latency (Worst Case)</td>
<td>6.4 us (64 hops)</td>
<td>3.0 us (64 hops)</td>
</tr>
<tr>
<td><strong>Collective Network</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Bandwidth</td>
<td>2*3350MB/s = 700 MB/s</td>
<td>2*0.85GB/s = 1.7 GB/s</td>
</tr>
<tr>
<td>Hardware Latency (Round Trip Worst Case)</td>
<td>5.0 us</td>
<td>3.0 us</td>
</tr>
<tr>
<td><strong>System Properties</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Peak Performance</td>
<td>360 TF (64k nodes)</td>
<td>1 PF (72k nodes)</td>
</tr>
<tr>
<td>Total Power</td>
<td>1,7 MW</td>
<td>TBD (about 2,3 MW)</td>
</tr>
</tbody>
</table>
BlueGene/P Compute Card
Questions?

( hennecke >at< de.ibm.com )
Single-Node Performance
**Characteristics of the memory subsystem**

<table>
<thead>
<tr>
<th>Attribute</th>
<th>L1</th>
<th>L2</th>
<th>L3 embedded DRAM</th>
<th>Scratch SRAM</th>
<th>Main memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>Size</td>
<td>32 KiB (I)</td>
<td>2 KiB per processor</td>
<td>2 banks of 2 MiB/bank = 4 MiB total shared by both processors</td>
<td>16 KiB shared by both processors</td>
<td>512 MiB shared by both processors</td>
</tr>
<tr>
<td>Latency (pclk)</td>
<td>3</td>
<td>11</td>
<td>28/36/40 (hit/miss precharged/missed busy)</td>
<td>15</td>
<td>86 (L3 cache enabled)</td>
</tr>
<tr>
<td>Sustained bandwidth: random quad load access (B/pclk)</td>
<td>NA</td>
<td>NA</td>
<td>1.8/1.2 (hit/miss)</td>
<td>2.0</td>
<td>0.8/0.5 (single/dual processor)</td>
</tr>
<tr>
<td>Sustained bandwidth: sequential access (B/pclk)</td>
<td>16.0</td>
<td>5.3</td>
<td>5.3/5.3 (hit/miss)</td>
<td>5.3</td>
<td>5.1/3.4 (single/dual processor)</td>
</tr>
<tr>
<td>Line width (B)</td>
<td>32</td>
<td>128</td>
<td>128</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Number of lines</td>
<td>1,024</td>
<td>16</td>
<td>32,768</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Coherent</td>
<td>No</td>
<td>Yes (weakly)</td>
<td>Yes</td>
<td>Yes (weakly)</td>
<td>Yes</td>
</tr>
<tr>
<td>Associativity</td>
<td>64 way</td>
<td>Fully associative</td>
<td>8 way/bank</td>
<td>NA</td>
<td>NA</td>
</tr>
</tbody>
</table>
Architecture Constraints of Dual-FPU Unit

- **Only stride-one vectors may be loaded or stored**
  - Stride-one means data that is stored consecutively in memory
  - No memory gather/scatter support
  - Bad for stride array accesses, e.g., \( a[2i+1] \), or random/indirect accesses

- **Only 16-byte aligned data may be efficiently loaded/stored**
  - \( a[i+1] = b[i+1] + c[i+1] \) vs. \( a[i+1] = b[i+2] + c[i+3] \)

- **Misaligned data can be loaded using cross-instructions**
  - Data realignment is encoded in the instruction opcodes, makes handling of runtime alignment difficult

- **Non-uniform instruction set:**
  - Supports only double precision floating point operation in SIMD mode
Experiment with the double-FPU / simdization

- To enable the "double hummer" SIMDization (in addition to –qarch=440d –qtune=440):
  - either –qhot, -O4, or –O5 will enable simdization

- To get compiler diagnostic information
  - Use –qdebug=diagnostic:
    - gives information on whether a loop is simdizable, and why a loop cannot be simdized
    - sometimes, compiler directives can help

- Good tutorial from Yorktown/Toronto available...
Simdizable loop diagnostic information

- Simdizable loops
  - Marked as "(simdizable)[...]
  - [...] further characterizes simdizable loops that may have performance impact
    - “misalign(....)”: simdizable loop with misaligned accesses
      - “store-misalign”: write to misaligned memory
    - “shift(....)”: simdizable loop with stream shift inserted
      - “<n> compile-time”: <n> stream shifts are compile-time
    - “priv”: simdizable loop has private variable
    - “unk-UB”: simdizable loop has unknown loop upper bound
Non-Simdizable loop diagnostic information

- **Alignment:**
  - “misalign(....)”: simdizable loop with misaligned accesses
    - “non-natural”: non naturally aligned accesses
    - “runtime”: runtime alignment

- **Structure of the loop:**
  - “loop not innermost”: not innermost loop
  - “not single block loop”: loops contain if-statements
  - “upper bound is not a loop-invariant”: while-loops
  - “small trip count”: very small trip count
  - “runtime upper bound not guarded”: unsafe runtime trip count
  - “bump not normalized”: no header for region
  - “contains function call“: contains function call
Non-Simdizable loop diagnostic information (contd.)

- **Dependence**
  - “dependence due to aliasing”
  - “dependence”: dependences not due to aliasing

- **Scalar references**
  - “non-simdizable scalar var / reductions”

- **Array references**
  - “access not stride one”: non stride-one accesses
  - “mem accesses with unsupported alignment”
  - “contains runtime shift”

- **Pointer references**
  - “non normalized pointer accesses”

- **Native Mapping and native data types**
  - “no intrinsic mapping for <op type>: ....”: no direct native instruction to support the simdized version of this operation
  - “non supported vector element types”: non char data types
Verifying the double-FPU / Simdization

- **Look at pseudo assembler to check SIMD**
  - Use –qlist to get assembly listing

- **Look for quad word loads and stores**
  - lfd, lfsdx, std, stfsdx (non SIMD – may indicate a problem)
  - lfpdx, stfpdx ("p" for parallel, d.p. quad loads and stores)
  - also s.p., secondary-d.p. and cross-d.p. versions available

- **Also, new arithmetic opcodes specific to double-FPU**
  - fpmadd, fxmadd, more cross- and complex-opcodes, fpsel

- **-O3 also uses reciprocal estimate and newton iteration for divide and square root**
  - fpre, fprsqt
How to load from misaligned memory?

- **Load-store unit of dual FPU**
  - Quad load/store from 16-byte aligned address: single cycle
  - Quad load/store from misaligned address: thousands of cycles

- **How to efficiently load from misaligned memory?**

![Diagram showing load/store operations](image)

1 misaligned quad load ➔ 2 aligned quad cross-loads + 1 select
How to access misaligned memory (cont’)?

- **Loading a chunk of the memory from a misaligned address**
  - reuse quad load-across
    
    ![Diagram showing misaligned memory access]

    On average: 1 misaligned quad load $\Rightarrow$ 1 aligned quad cross-loads + 1 select
Supplying the compiler with more information

- **Alignment**
  - Assert that pointers are 16-byte aligned or at least naturally aligned
    - __alignx(16, p); or __alignx(8, p);
  - Tell the compiler to map data to 16-byte aligned memory
    - double a[256] __attribute__((aligned(16));
  - Use static/global arrays (compiler will put them at 16-byte boundary)

- **Aliasing**
  - Tell the compiler that data accessed through pointers are disjoint:
    - #pragma disjoint (*a, *b)

- **Use constant loop bounds, and data flow instead of control flow**
  - e.g. #define
  - use select instead of if/then/else, macro instead of function calls
Use libraries

- **mass, massv, essl**
  - Need linker option `-Wl,--allow-multiple-definition`
  - Tuned functions for Blue Gene

- **-qhot can generate calls to massv**
  - mass is now part of XL compilers

- **Use special fft libraries (e.g. TU-Vienna)**
libmass.a and libmassv.a timings (versus libm.a)

<table>
<thead>
<tr>
<th>Function</th>
<th>libm.a</th>
<th>libmass.a</th>
<th>libmassv.a</th>
<th>range</th>
</tr>
</thead>
<tbody>
<tr>
<td>sqrt</td>
<td>102</td>
<td>40</td>
<td>7.9</td>
<td>(0,10**10)</td>
</tr>
<tr>
<td>rsqrt</td>
<td>134</td>
<td>35</td>
<td>5.5</td>
<td>(0,10**10)</td>
</tr>
<tr>
<td>exp</td>
<td>167</td>
<td>56</td>
<td>22.8</td>
<td>(-50,50)</td>
</tr>
<tr>
<td>log</td>
<td>316</td>
<td>68</td>
<td>23.6</td>
<td>(0,10**10)</td>
</tr>
<tr>
<td>sin</td>
<td>191</td>
<td>66</td>
<td>29</td>
<td>(0, 2pi)</td>
</tr>
<tr>
<td>cos</td>
<td>199</td>
<td>66</td>
<td>29</td>
<td>(0, 2pi)</td>
</tr>
<tr>
<td>tan</td>
<td>315</td>
<td>90</td>
<td>44</td>
<td>(0, 2pi)</td>
</tr>
<tr>
<td>atan</td>
<td>220</td>
<td>114</td>
<td>27</td>
<td>(-100,100)</td>
</tr>
<tr>
<td>sinh</td>
<td>266</td>
<td>81</td>
<td>32</td>
<td>(-50,50)</td>
</tr>
<tr>
<td>cosh</td>
<td>227</td>
<td>67</td>
<td>31</td>
<td>(-50,50)</td>
</tr>
<tr>
<td>atan2</td>
<td>396</td>
<td>127</td>
<td>-</td>
<td>(-50,50) both x and y</td>
</tr>
<tr>
<td>pow</td>
<td>522</td>
<td>167</td>
<td>74</td>
<td>(0,20) both x and y</td>
</tr>
</tbody>
</table>
Blue Gene Networks Details
Torus network

- Each node has 6 bi-directional links to its neighbors (x+, x-, y+, y-, z+, z-)
  - 2 bit / cycle / link / direction
    = 3B/cycle or 2.1GB/s (raw)
  - payload factor: 240/(256+14)
    = 2.64B/cycle or 1.85GB/s (payload)
  - 100ns HW latency per hop

- Nodes in 32-way node book are hard-wired as a 4x4x2 mesh

- Blue Gene link chips connect all 32 node books into 8x8x8 midplane mesh
  - 4 link cards @ 6 link chips @ 16 ports
  - all six 8x8 surfaces are wired to them
  - "mode 1" uses cables to form larger torus, "mode 2" wraps midplane into 8x8x8 torus
Blue Gene Link Chip – modes set through JTAG

- Port C: In, Regular cable
- Port D: In, Split cable
- Port E: Out, Split cable
- Port F: Out, Regular cable

Mode 2

Mode 3

JTAG or I2C block for setting modes

JTAG port

Port B: Out to midplane

Port A: In from midplane
Collective network – node book view
Collective network – midplane view
Global Interrupt network – node book view
Global Interrupt network – midplane view

Each line represents eight wires, four \( OR \) signals and four \( RETURN \) signals.

- Node card
- Node FPGA chip
- Upper interrupt bus
- Cable connections
- Four link cards
- Four link FPGA chips
- Lower interrupt bus D
- Lower interrupt bus B
- Lower interrupt bus A
- Quadrant head card
MPI from a user's point of view
Outline

- This is not a talk about how to invoke mpirun.
  - System software folks will have presented that to you

- This is a talk about what does, and what doesn't work in Blue Gene/L MPI.
  - Basic things you should avoid doing
  - Ideas about obtaining good performance
    - point-to-point messaging
      - scaling
      - mapping application into network
    - collective messaging
Summary I: will Blue Gene like me?

- Blue Gene/L MPI is like other implementations of MPI
  - looks like Argonne National Labs' MPICH2
  - because that's what it is.
- As an MPI developer you will have relatively few surprises
  - Blue Gene/L MPI is MPI standard 1.2 compliant
    - no one-sided communication
    - no spawning of processes
    - supports thread model MPI_THREAD_SINGLE
    - MPI I/O is still under development
  - Getting good performance out of BG/L MPI is inherently harder
    - large scale, funny network
Summary II: Kinds of annoyance you can cause

- **Crashing an application:** somewhat easier than on other platforms, because:
  - limited memory on nodes, no virtual memory on nodes
    - memory leaks are going to make their presence felt
  - communication network is in userspace
    - good: high performance
    - bad: user have opportunity to kill process with wild pointers
- **Invoking the Halting Problem:** deadlocking the machine
- **Violating MPI semantics:** laws you didn't know were on the books
- **Causing bad performance:** malice not required
  - “nicely” map the application into the network (hard)
  - avoid load imbalance (hard)
  - avoid network jams (very hard)
Blue Gene/L MPI Software Architecture

Message passing

Abstract Device Interface
- pt2pt
- datatype
- MPICH2
- topo
- collectives
- CH3
- MM
- bgl
- socket

Packet Layer
- Torus Device
- Tree Device
- GI Device

Process management

PMI
- torus
- tree
- GI
- simple
- uniprocessor
- mpd
- bgltorus

Message Layer

“glue”

CIO Protocol
Deadlocking the system

- Talk before you listen.
- Illegal MPI code
  - find it in most MPI books
  - tech support will be very annoyed
- Blue Gene/L MPI is designed not to deadlock easily.
  - It will likely survive this code.
- This code will cause MPI to allocate memory to deal with unexpected messages. If MPI runs out of memory, it will stop with an error message

```
CPU1 code:
MPI_Send(cpu2);
MPI_Recv(cpu2);

CPU2 code:
MPI_Send(cpu1);
MPI_Recv(cpu1);
```
Force MPI to allocate too much memory

- Post receives in one order, sends in the opposite order

- This is legal MPI code

- Blue Gene/L MPI will choke if the sum of buffers is greater than the amount of physical memory
  - this is an implementation defect that will be fixed in the future

```plaintext
CPU1 code:
MPI_Isend(cpu2, tag1);
MPI_Isend(cpu2, tag2);
...
MPI_Isend(cpu2, tag_n);

CPU2 code:
MPI_Recv(cpu1, tag_n);
MPI_Recv(cpu1, tag_{n-1});
...
MPI_Recv(cpu1, tag_1);
```
Sneaky: violate MPI buffer ownership rules

- **write send/receive buffers before completion**
  - results in data race on any machine

- **touch send buffers before message completion**
  - not legal by standard
  - BG/L MPI will survive it today
  - no guarantee about tomorrow

- **touch receive buffers before completion**
  - BG/L MPI will yield wrong results

```c
req = MPI_Isend (buffer);
buffer[0] = something;
MPI_Wait(req);
```

```c
req = MPI_Isend (buffer);
z = buffer[0];
MPI_Wait (req);
```

```c
req = MPI_Irecv (buffer);
z = buffer[0];
MPI_Wait (req);
```
Causing memory overruns: never wait for MPI_Test

- Have to wait for all requests
  - The standard requires waiting
  - or testing until MPI_Test returns true
- This code works on many other architectures
  - causes tiny memory leaks
- On BG/L this will run the system out of memory very fast
  - MPI_Request requires a lot of memory
  - It's a scaling issue

```c
req = MPI_Isend( ... );
MPI_Test (req);
... do something else; forget about req ...
```
Straddle collectives with point-to-point messages

- On the ragged edge of legality
- Blue Gene/L MPI works
- Multiple networks issue:
  - Isend handled by torus network
  - Barrier handled by GI network

CPU 1 code:
```c
req = MPI_Isend (cpu2);
MPI_Barrier();
MPI_Wait(req);
```

CPU 2 code:
```c
MPI_Recv (cpu1);
MPI_Barrier();
```
Send flood

- This is legal MPI code
  - also ... stupid MPI code
  - not scalable, even when it works
- Blue Gene/L MPI will run out of buffer space
  - This is a bug, and will be fixed
- We have seen this kind of code in the wild
  - Don't write code such as this
  - Even if you think it should work

**CPU 1 to n-1 code:**

```c
MPI_Send(cpu0);
```

**CPU 0 code:**

```c
for (i=1; i<n; i++)
  MPI_Recv(cpu[i]);
```
Virtual Node Mode vs. Coprocessor mode

- **Virtual Node Mode:**
  - twice the processing power!
  - but not twice the performance
    - half of memory per CPU
    - half of cache per CPU
    - half of network per CPU
    - CPU has to do both computation and communication

- **Coprocessor mode:**
  - only one CPU available to execute user code
  - but have all memory!
  - other CPU helps with communication
  - currently, only point-to-point communication benefits
    - that is about to change
Point-to-point performance (I)

- **Two kinds of network routing on Blue Gene/L:**
  - deterministic routing:
    - each packet goes along the same path
    - maintains packet order
    - creates network hotspots
  - adaptive routing
    - packets overtake
    - equalized network load
    - harder on CPUs
  - MPI matching semantics are always correct!

- **MPI Short protocol:**
  - very short (<=240 bytes) messages.
  - Deterministically routed

- **MPI Eager protocol:**
  - medium size messages
  - “send without asking”
  - deterministically routed
  - latency around 3.3 µs

- **MPI Rendezvous protocol:**
  - large (> 10KBytes) messages
  - adaptively routed
  - bandwidth optimized
Point to point performance (II)

- **The rendezvous threshold (10KBytes) can be changed**
  - environment variable: `BGLMPI_EAGER`

- **Lower the rendezvous threshold if**
  - running on a large message
  - many short messages are overloading the network
  - eager messages are creating artificial hotspots
  - program is not latency sensitive

- **Increase the rendezvous threshold if**
  - most communication is nearest-neighbor
    - or at least close in Manhattan distance
  - relatively longer messages
  - you need better latency on medium size messages
Bandwidth vs. message size

6-way send+recv

1 byte/cycle = 700MB/s
Point-to-point performance (III): Dos and Don'ts

- Overlapping communication and computation:
  - a bad idea on Blue Gene/L
  - keep programs in sync as much as you can
    - alternate computation and communication phases

- Avoid load imbalance
  - bad for scaling

- Shorten Manhattan distance messages have to traverse
  - send to nearest neighbors!

- Avoid synchronous sends
  - increases latency

- Avoid buffered sends
  - memory copies are bad for your health

- Avoid vector data, non-contiguous data types
  - BG/L MPI doesn't have a nice way to deal with them.

- Post receives in advance
  - unexpected messages damage performance
The all-important torus mapping

- NAS BT
  - 2D mesh communication pattern
  - Map on 3D mesh/torus?
  - Folding and inverting planes in the 3D mesh
- Example shows naive mapping (in 1D MPI rank order) versus optimized mapping
- Mapping can be optimized by
  - tools external to the application,
  - or by using MPI topologies (covered later on)
How to map an application to the torus?

- set up a mapping file:

```
0 0 0 0
1 0 0 0
2 0 0 0
0 1 0 0
...  
```

associates torus coordinates to MPI ranks 0 to n-1.

- Yeah, but why quadruplets?  --> VN mode

- Use mapping file as argument in mpirun invocation
MPI Collective performance

- **Rule 1: Use collectives whenever you can**
  - Point-to-point performance has huge overheads
  - “I can do a better job with point-to-point than you miserable jocks can do with collectives”
    - We don't think so.

- **Rule 2: Mapping is all-important for good collective performance**
  - Most collective implementations prefer certain communicator shapes

- **Rule 3: Don't do anything crazy, like**
  - Use different buffer sizes for a broadcast call (illegal)
  - Use heterogeneous data types for broadcast (legal, but crazy)
  - Use misaligned buffers (legal and not crazy, but we don't like it anyway)
  - Run point-to-point messages across the communicator at the same time that a collective is underway (legal, but not cheap)
## Summary of Optimized BG/L MPI Collectives

<table>
<thead>
<tr>
<th>Condition</th>
<th>Network</th>
<th>Performance</th>
</tr>
</thead>
<tbody>
<tr>
<td>Barrier COMM_WORLD</td>
<td>GI</td>
<td>1.5us</td>
</tr>
<tr>
<td></td>
<td>Tree</td>
<td>5 us</td>
</tr>
<tr>
<td></td>
<td>Torus</td>
<td>10-15 us</td>
</tr>
<tr>
<td>Broadcast COMM_WORLD</td>
<td>Tree</td>
<td>350 Mbytes/s</td>
</tr>
<tr>
<td></td>
<td>Torus</td>
<td>320 Mbytes/s (0.48 Bytes/cycle)</td>
</tr>
<tr>
<td>Broadcast RECT COMM</td>
<td>Tree</td>
<td>TBD: low latency</td>
</tr>
<tr>
<td>Allreduce COMM_WORLD, fixed point</td>
<td>Tree</td>
<td>350 Mbytes/s, low latency</td>
</tr>
<tr>
<td>Allreduce COMM_WORLD, floating pt</td>
<td>Tree</td>
<td>40 Mbytes/s (0.06Bytes/c)</td>
</tr>
<tr>
<td>Allreduce Hamilton Path</td>
<td>Torus</td>
<td>TBD: &gt;= 120MB/s, low latency</td>
</tr>
<tr>
<td></td>
<td>Torus</td>
<td>120 Mbytes/s</td>
</tr>
<tr>
<td></td>
<td>Torus</td>
<td>80 Mbytes/s</td>
</tr>
<tr>
<td></td>
<td>Torus</td>
<td>10-15 us latency</td>
</tr>
<tr>
<td></td>
<td>Torus</td>
<td>TBD: other shapes</td>
</tr>
<tr>
<td></td>
<td>Torus</td>
<td>TBD: high bandwidth FP</td>
</tr>
<tr>
<td>Alltoall[v] Any communicator</td>
<td>Torus</td>
<td>84-97% of peak</td>
</tr>
<tr>
<td>Allgatherv rectangular</td>
<td>Torus</td>
<td>Same as broadcast</td>
</tr>
</tbody>
</table>
Optimized collectives: Alltoall[v]

- Performance measured as percentage of peak
  - which is function of partition “shape”
- MPICH2 implementation not suitable for torus network
- Optimized implementation: 90% of peak
  - Impl. by Charles Archer
- measured on an 8x8x8 partition
MPI topologies

- Defined in ch06 of the MPI 1.1 standard
- Idea: attach knowledge of application's inherent topology (2D grid, etc.) to an MPI communicator
  - inside the program, not external mapping like --mapfile
- Create a new communicator based on
  - input communicator (e.g. MPI_COMM_WORLD)
  - description of the app's topology (shape, periodicity)
  - programmer may allow the runtime to reorder, or not
    - advice is to ALLOW reordering to optimize placement of tasks onto the torus network
- Then, use new communicator in your MPI calls, instead of the usual MPI_COMM_WORLD
Using cartesian communicators

- **Simplest case: just rely on the re-ordered rank**
  - if your program's coord/rank calculation is "natural", this is often good enough: MPI runtime has done the placement

- **Use MPI topology coords/rank transformations:**
  - MPI_Cart_rank() and MPI_Cart_coords()
    - mainly convenience, makes program easier to read

- **Express neighborhood in app's coords not rank:**
  - MPI_Cart_shift() – again mainly convenience

- **Use collectives on cartesian sub-communicators:**
  - use MPI_Cart_sub() to create row or column sub-communicators (call similar to MPI_Comm_split)
  - then use these sub-comms in collectives
  - this may exploit special BG/L hardware features like **multicast along a torus axis**
BG/L MPI runtime support of MPI topologies

- Only cartesian topologies, no graphs
  - most apps are cartesian, and BG/L is a torus

- input communicator to MPI_Cart_Create() must be a rectangular part of the torus

- one- to three-dimensional topologies supported in CO and VN mode

- four-dimensional topologies only in VN mode, and with one dimension of size 2

- higher-dim. cartesian topologies and graphs are accepted, but result is a NO-OP
MPI_Cart_create()

MPI_CART_CREATE(comm_old, ndims, dims, periods, reorder, comm_cart)

**IN**
- **comm_old**: input communicator (handle)
- **ndims**: number of dimensions of cartesian grid (integer)
- **dims**: integer array of size ndims specifying the number of processes in each dimension
- **periods**: logical array of size ndims specifying whether the grid is periodic (true) or not (false) in each dimension
- **reorder**: ranking may be reordered (true) or not (false) (logical)

**OUT**
- **comm_cart**: communicator with new cartesian topology (handle)

```c
int MPI_Cart_create(MPI_Comm comm_old, int ndims, int *dims, int *periods,
            int reorder, MPI_Comm *comm_cart)
```

```c
MPI_CART_CREATE(COMM_OLD, NDIMS, DIMS, PERIODS, REORDER, COMM_CART, IERROR)
    INTEGER COMM_OLD, NDIMS, DIMS(*), COMM_CART, IERROR
    LOGICAL PERIODS(*), REORDER
```
MPI_Cart_create example (1 of 2)

```c
#include <stdio.h>
#include <stdlib.h>
#include <mpi.h>
#include <bglpersonality.h>

int main (int argc, char **argv)
{
    int world_size, world_rank, cart_size, cart_rank, ndims, reorder, rc;
    int dims[2], periods[2], coords[2];
    MPI_Comp cart_comm;
    char location[BGLPERSONALITY_MAX_LOCATION];
    BGLPersonality p;

    rc=MPI_Init(&argc, &argv);
    rc=MPI_Comm_size(MPI_COMM_WORLD, &world_size);
    rc=MPI_Comm_rank(MPI_COMM_WORLD, &world_rank);
    printf("W: %04i/%04i ", world_rank, world_size);

    ndims=2; dims[0]=7; dims[1]=3; periods[0]=1; periods[1]=1; reorder=1;
    rc=MPI_Cart_create(MPI_COMM_WORLD, ndims, dims, periods, reorder, &cart_comm);
```
MPI_Cart_create example (2 of 2)

if( cart_comm != MPI_COMM_NULL ) {
    rc=MPI_Comm_size(cart_comm, &cart_size);
    rc=MPI_Comm_rank(cart_comm, &cart_rank);
    rc=MPI_Cart_coords(cart_comm, cart_rank, ndims, coords);
    printf("C: %04i/%04i <%02i,%02i> ",
            cart_rank, cart_size, coords[0], coords[1]);
} else {
    printf("C: NULL/NULL <--,--> ");
}

rts_get_personality(&p, sizeof(p));
BGLPersonality_getLocationString(&p, location);
printf("T: <%02i,%02i,%02i>/<%02i,%02i,%02i> L: %s\n",
       BGLPersonality_xCoord(&p), BGLPersonality_yCoord(&p),
       BGLPersonality_zCoord(&p), BGLPersonality_xSize(&p),
       BGLPersonality_ySize(&p), BGLPersonality_zSize(&p), location);
MPI_Finalize();
exit(0);
mapping 7x3 mesh onto 4x4x2 node card

W: 0000/0032 C: 0000/0021 <00,00> T: <00,00,00>/<04,04,02> L: R00-M0-Ne-C:J16-U01
W: 0001/0032 C: 0001/0021 <00,01> T: <01,00,00>/<04,04,02> L: R00-M0-Ne-C:J12-U01
W: 0002/0032 C: 0002/0021 <00,02> T: <02,00,00>/<04,04,02> L: R00-M0-Ne-C:J08-U01
W: 0003/0032 C: NULL/NULL <--,--> T: <03,00,00>/<04,04,02> L: R00-M0-Ne-C:J04-U01
W: 0004/0032 C: 0003/0021 <01,00> T: <00,01,00>/<04,04,02> L: R00-M0-Ne-C:J16-U11
W: 0005/0032 C: 0004/0021 <01,01> T: <01,01,00>/<04,04,02> L: R00-M0-Ne-C:J12-U11
W: 0006/0032 C: 0005/0021 <01,02> T: <02,01,00>/<04,04,02> L: R00-M0-Ne-C:J08-U11
W: 0007/0032 C: NULL/NULL <--,--> T: <03,01,00>/<04,04,02> L: R00-M0-Ne-C:J04-U11
W: 0008/0032 C: 0006/0021 <02,00> T: <00,02,00>/<04,04,02> L: R00-M0-Ne-C:J17-U01
W: 0009/0032 C: 0007/0021 <02,01> T: <01,02,00>/<04,04,02> L: R00-M0-Ne-C:J13-U01
W: 0010/0032 C: 0008/0021 <02,02> T: <02,02,00>/<04,04,02> L: R00-M0-Ne-C:J09-U01
W: 0011/0032 C: NULL/NULL <--,--> T: <03,02,00>/<04,04,02> L: R00-M0-Ne-C:J05-U01
W: 0012/0032 C: 0009/0021 <03,00> T: <00,03,00>/<04,04,02> L: R00-M0-Ne-C:J17-U11
W: 0013/0032 C: 0010/0021 <03,01> T: <01,03,00>/<04,04,02> L: R00-M0-Ne-C:J13-U11
W: 0014/0032 C: 0011/0021 <03,02> T: <02,03,00>/<04,04,02> L: R00-M0-Ne-C:J09-U11
W: 0015/0032 C: NULL/NULL <--,--> T: <03,03,00>/<04,04,02> L: R00-M0-Ne-C:J05-U11
W: 0016/0032 C: NULL/NULL <--,--> T: <00,00,01>/<04,04,02> L: R00-M0-Ne-C:J14-U01
W: 0017/0032 C: NULL/NULL <--,--> T: <01,00,01>/<04,04,02> L: R00-M0-Ne-C:J10-U01
W: 0018/0032 C: NULL/NULL <--,--> T: <02,00,01>/<04,04,02> L: R00-M0-Ne-C:J06-U01
W: 0019/0032 C: NULL/NULL <--,--> T: <03,00,01>/<04,04,02> L: R00-M0-Ne-C:J02-U01
W: 0020/0032 C: 0018/0021 <06,00> T: <00,01,01>/<04,04,02> L: R00-M0-Ne-C:J14-U11
W: 0021/0032 C: 0019/0021 <06,01> T: <01,01,01>/<04,04,02> L: R00-M0-Ne-C:J10-U11
W: 0022/0032 C: 0020/0021 <06,02> T: <02,01,01>/<04,04,02> L: R00-M0-Ne-C:J06-U11
W: 0023/0032 C: NULL/NULL <--,--> T: <03,01,01>/<04,04,02> L: R00-M0-Ne-C:J02-U11
mapping 8x4 mesh onto 4x4x2 node card

<table>
<thead>
<tr>
<th>W</th>
<th>C</th>
<th>T</th>
<th>L</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000/0032</td>
<td>0000/0032</td>
<td>&lt;00,00&gt;</td>
<td>R00-M0-Ne-C:J16-U01</td>
</tr>
<tr>
<td>0001/0032</td>
<td>0001/0032</td>
<td>&lt;00,01&gt;</td>
<td>R00-M0-Ne-C:J12-U01</td>
</tr>
<tr>
<td>0002/0032</td>
<td>0002/0032</td>
<td>&lt;00,02&gt;</td>
<td>R00-M0-Ne-C:J08-U01</td>
</tr>
<tr>
<td>0003/0032</td>
<td>0003/0032</td>
<td>&lt;00,03&gt;</td>
<td>R00-M0-Ne-C:J04-U01</td>
</tr>
<tr>
<td>0004/0032</td>
<td>0004/0032</td>
<td>&lt;01,00&gt;</td>
<td>R00-M0-Ne-C:J16-U11</td>
</tr>
<tr>
<td>0005/0032</td>
<td>0005/0032</td>
<td>&lt;01,01&gt;</td>
<td>R00-M0-Ne-C:J12-U11</td>
</tr>
<tr>
<td>0006/0032</td>
<td>0006/0032</td>
<td>&lt;01,02&gt;</td>
<td>R00-M0-Ne-C:J08-U11</td>
</tr>
<tr>
<td>0007/0032</td>
<td>0007/0032</td>
<td>&lt;01,03&gt;</td>
<td>R00-M0-Ne-C:J04-U11</td>
</tr>
<tr>
<td>0008/0032</td>
<td>0008/0032</td>
<td>&lt;02,00&gt;</td>
<td>R00-M0-Ne-C:J17-U01</td>
</tr>
<tr>
<td>0009/0032</td>
<td>0009/0032</td>
<td>&lt;02,01&gt;</td>
<td>R00-M0-Ne-C:J13-U01</td>
</tr>
<tr>
<td>0010/0032</td>
<td>0010/0032</td>
<td>&lt;02,02&gt;</td>
<td>R00-M0-Ne-C:J09-U01</td>
</tr>
<tr>
<td>0011/0032</td>
<td>0011/0032</td>
<td>&lt;02,03&gt;</td>
<td>R00-M0-Ne-C:J05-U01</td>
</tr>
<tr>
<td>0012/0032</td>
<td>0012/0032</td>
<td>&lt;03,00&gt;</td>
<td>R00-M0-Ne-C:J17-U11</td>
</tr>
<tr>
<td>0013/0032</td>
<td>0013/0032</td>
<td>&lt;03,01&gt;</td>
<td>R00-M0-Ne-C:J13-U11</td>
</tr>
<tr>
<td>0014/0032</td>
<td>0014/0032</td>
<td>&lt;03,02&gt;</td>
<td>R00-M0-Ne-C:J09-U11</td>
</tr>
<tr>
<td>0015/0032</td>
<td>0015/0032</td>
<td>&lt;03,03&gt;</td>
<td>R00-M0-Ne-C:J05-U11</td>
</tr>
<tr>
<td>0016/0032</td>
<td>0016/0032</td>
<td>&lt;07,00&gt;</td>
<td>R00-M0-Ne-C:J14-U01</td>
</tr>
<tr>
<td>0017/0032</td>
<td>0017/0032</td>
<td>&lt;07,01&gt;</td>
<td>R00-M0-Ne-C:J10-U01</td>
</tr>
<tr>
<td>0018/0032</td>
<td>0018/0032</td>
<td>&lt;07,02&gt;</td>
<td>R00-M0-Ne-C:J06-U01</td>
</tr>
<tr>
<td>0019/0032</td>
<td>0019/0032</td>
<td>&lt;07,03&gt;</td>
<td>R00-M0-Ne-C:J02-U01</td>
</tr>
<tr>
<td>0020/0032</td>
<td>0020/0032</td>
<td>&lt;06,00&gt;</td>
<td>R00-M0-Ne-C:J14-U11</td>
</tr>
<tr>
<td>0021/0032</td>
<td>0021/0032</td>
<td>&lt;06,01&gt;</td>
<td>R00-M0-Ne-C:J10-U11</td>
</tr>
<tr>
<td>0022/0032</td>
<td>0022/0032</td>
<td>&lt;06,02&gt;</td>
<td>R00-M0-Ne-C:J06-U11</td>
</tr>
<tr>
<td>0023/0032</td>
<td>0023/0032</td>
<td>&lt;06,03&gt;</td>
<td>R00-M0-Ne-C:J02-U11</td>
</tr>
</tbody>
</table>
Questions?

(hennecke >at< de.ibm.com)
News with Blue Gene Software
v1r3
L1 Data Cache Mode Tuning in v1r3

- Default L1 data cache modes:
  - Store-with-Allocate and Write-Back

- To select Store-without-Allocate:
  - BGL_APP_L1_SWOA=1
    - For stores, if there is a L1 cache miss the L1 cache is bypassed (no cacheline is allocated); data is stored directly to lower levels of cache

- To select Write-Though:
  - BGL_APP_L1_WRITE_THROUGH=1
    - For stores, data is written to L1 and lower level caches and L1 is marked as clean (Write-Back writes only to L1, which is then dirty)
    - This is incompatible with the $dcbz()$ compiler intrinsic, remove it!

- Some applications may see improved performance
  - Some may also run slower ... your mileage will vary
STREAM on Blue Gene – different L1 settings

BlueGene/L STREAM Performance
Main Memory, N=2000000

Modes/Test

<table>
<thead>
<tr>
<th>Modes/Test</th>
<th>V1R2M2_060328</th>
<th>V1R3M0_060623</th>
</tr>
</thead>
<tbody>
<tr>
<td>Copy</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Scale</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Add</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Tridiad</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Copy</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Scale</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Add</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Tridiad</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Copy</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Scale</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Add</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Tridiad</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
New memory allocation functions in v1r3

- Allocating memory regions with specific L1 attributes (1 MB ... 14 MB)
  - `rts_get_dram_window()` system call, with options for L1:
    - Cache-inhibited or cached
    - Store without allocate or store with allocate
    - Write-through or write-back
  - Environment settings discussed before apply to whole environment
    - With this call, can have multiple regions with different attributes

- Allocating scratchpad memory (32 B ... 8 kB)
  - `rts_malloc_sram()` and `rts_free_sram()` system calls
  - Small SRAM memory region (16 kB), coherent between the 2 cores
    - MPI library normally uses 4 kB
    - some SRAM is used by the kernel and performance counters
Tuning L1 parity error recovery in v1r3

- Large number of CPUs in a Blue Gene system imply increased probability and impact of memory errors

- L1 I-Cache parity errors are always automatically recovered by CNK
  - Always clean since cachelines cannot be modified...

- For L1 D-Cache parity errors, three options exist:
  - Write-back mode with no recovery (fastest, default, errors are fatal)
  - Write-back mode with application assisted recovery: CNK can automatically recover parity errors *if the cache line is clean*
    - Requires setting `BGL_APP_L1_WRITEBACK_RECOVERY`
    - Performance impact over default write-back mode
    - Application can try to recover dirty cache lines by registering a handler, otherwise application is ended
  - Write-through mode with automatic recovery (use with SWOA, slowest)

- L2, L3, and DDR are ECC protected
Communication offloading for p2p messages

- **Blue Gene/L processors are slow relative to network speed**
  - Memory copy:
    - 1.0 to 2.0 Byte/cycle
  - Network bandwidth:
    - 2bit/cycle x 6 x 2 directions = 3.0 Byte/cycle
  - To get full network bandwidth, both processors are required
    - A single processor cannot saturate the network

- **Latency and $\sqrt{N}$ are very good**
Work distribution using coprocessor mode

- **Observation: injection is more expensive than reception**
  - Can send to 6 directions at full bandwidth
  - Can only receive from 4 directions at full bandwidth

- **Default approach in v1r1 and v1r2:**
  - Computation on the main processor
  - Sends on the main processor
  - Receives on the coprocessor
    - gives a better division of labor among the cores

- **This is still the default in v1r3**
  - but can be tuned, see below...
Network bandwidth in coprocessor mode

- **Better bandwidth in CO mode:**
  - Can saturate 4 receives
  - Can saturate 6 sends
  - Total aggregate bandwidth is now 1.8 Byte/cycle
    - if sends and receives are split between both processors

- **Disadvantage:**
  - Latency can be hurt
  - Sends are not overlapped
    - more on this later
  - Memory bandwidth may be hurt
    - cache flushes; synchronization between the cores
Packet injection is even more expensive
  - tree only supports a few basic ops in HW
    • sum, xor, or, ... on integers only
  - packet processing may reduce the BW

Option 1: CPU 0 talks to the tree, CPU 1 manages processing
  - Needs low latency shared memory
    • We use SRAM buffer
  - Used for some operations

Option 2: CPU 0 receives from the tree, CPU 1 sends to the tree, both preprocess
  - Used for some operations

Overlap possible?
  - MPI does not define non-blocking collectives
  - We do not implement any non-std routines
MPI_Allreduce results

Allreduce Performance

Bytes/Cycle

log msg size (bytes)

Unsigned MAX
Signed MAX No Coproc
Signed MAX Coproc
More on overlapping techniques

- In general, BG/L works *much* better when using alternating communication / computation phases ("lock-step" mode)
  - If you have a code which already takes advantage of overlap, *and* if your code is in the "overlap zone", overlapping may be worthwhile

- Ways to overlap communication and computation:
  - Put sends on the coprocessor
  - Put receives on the coprocessor (and solve the coherency issues)
    - This is the default in CO mode
  - Put sends *AND* receives on the coprocessor (bandwidth will suffer)
    - Can be enabled by setting `BGLMPI_COPRO_SENDS = Y`
  - Allow for *interrupt driven communication*
    (overall latency and bandwidth will suffer)
    - Can be enabled with `BGLMPI_INTERRUPT`, see below...
Overlap zone profile for Blue Gene
Interrupt-driven communication

- **Added in v1r3 to support one-sided communication**
  - ARMCI and Global Arrays

- **May also increase performance of non-blocking p2p calls**
  - MPI_Isend(), MPI_Irecv(), MPI_Wait()

- **Enabled through a new environment variable:**
  - BGLMPI_INTERRUPT = \{ Y | N | S | R \}
  - Four options:
    - Y - turn on both send and receive interrupts
    - N - turn off both send and receive interrupts
    - S - turn on only send interrupts
      - (e.g., MPI_Isend() can interrupt to copy data to the network)
    - R - turn on only receive interrupts
      - (e.g., incoming data can interrupt to copy data from the network)
  - Use profiling tools to check the effect on communication time
Interrupt-driven communication example

BlueGene/L Interrupt-Driven Performance
Application=MILC, 32 processors
Results obtained using the MPI_profiler tool

Mode / Range of Processor Performance

Interrupts Disabled (both S & R)  Interrupts Enabled (both S & R)
Questions?

( hennecke @ de.ibm.com )