Application Performance Characterization and Analysis on Blue Gene/Q

Bob Walkup (walkup@us.ibm.com)

• Click to add text
# Blue Gene/Q : Power-Efficient Computing

<table>
<thead>
<tr>
<th>System</th>
<th>date</th>
<th>GHz</th>
<th>cores/rack</th>
<th>largest-system</th>
<th>peak-PFlops</th>
</tr>
</thead>
<tbody>
<tr>
<td>Blue Gene/L</td>
<td>~2004</td>
<td>0.70</td>
<td>2K</td>
<td>104 racks</td>
<td>~0.6</td>
</tr>
<tr>
<td>Blue Gene/P</td>
<td>~2008</td>
<td>0.85</td>
<td>4K</td>
<td>72 racks</td>
<td>~1.0</td>
</tr>
<tr>
<td>Blue Gene/Q</td>
<td>~2012</td>
<td>1.60</td>
<td>16K</td>
<td>96 racks</td>
<td>~20.1</td>
</tr>
</tbody>
</table>

#1 Top 500 List 06/2012 : 16.3 PFlops  96-rack Sequoia system LLNL
#2 Top 500 List 11/2011  ... the #1 system recorded 17.6 PFlops

#1 Green 500 List 06/2012 : 2.1GFlops/Watt
#5 Green 500 List 11/2012  ... #1 system recorded 2.5 GFlops/Watt

Blue Gene/Q : 4 threads/core * 16K cores/rack * 96 racks = 6,291,456 threads

How about applications … how can you tell if you are using the cores efficiently?
Instrument the code with hardware counters … MPI profiling interface is handy.
Measure the instruction mix and instruction throughput.
IPC = instructions per cycle per core is a good metric.

Some lessons learned from jobs with more than one million processes.
Blue Gene/Q Hardware Overview

16 cores/node, 16 GB memory/node, 1024 nodes/rack
5D torus network, 2GB/sec per link, 40 GB/sec off-node bandwidth
System on a chip: cores, L2 cache, network devices are integrated on the chip
A2 cores: simple in-order execution 1.6 GHz frequency, no ILP
  Two execution units: XU for Integer/Load/Store, AXU for Floating-Point
  Six cycle latency, single-cycle throughput for floating-point operations.
  Four hardware threads … four sets of registers … the key to performance.
  At most one instruction can be completed per cycle per thread.
  At most two instructions can be completed per cycle per core, one from each of the two execution units.
  QPX unit for 4-wide SIMD operations => peak is 8*1.6 = 12.8 GFlops/core
  16 KB L1 D-cache, 4KB prefetch buffer per core
  32 MB shared L2 cache with a full crossbar switch connecting all cores
BG/Q Daxpy  \( y(\cdot) = a \times x(\cdot) + y(\cdot) \)
Stream Benchmark

- Copy - QPX
- Scale - QPX
- Add - QPX
- Triad - QPX

Bandwidth (MB/sec) vs. #Active Cores

- Copy
- Scale
- Add
- Triad
Blue Gene/Q Software Overview

Light-weight kernel on compute nodes, no context switches.

GNU and IBM XL compilers, Fortran, C, C++.

MPI optionally with OpenMP or Pthreads (and other comm methods).

File I/O is handled by separate I/O nodes; ratio is 1:32-128 io:compute

Real memory only, no paging, 16 GBytes per node.

<table>
<thead>
<tr>
<th>Processes/node</th>
<th>MB/process</th>
<th>%hardware</th>
</tr>
</thead>
<tbody>
<tr>
<td>64</td>
<td>206</td>
<td>80.4%</td>
</tr>
<tr>
<td>32</td>
<td>460</td>
<td>89.8%</td>
</tr>
<tr>
<td>16</td>
<td>970</td>
<td>94.7%</td>
</tr>
<tr>
<td>8</td>
<td>1929</td>
<td>94.2%</td>
</tr>
<tr>
<td>4</td>
<td>3969</td>
<td>96.9%</td>
</tr>
</tbody>
</table>

Most applications will use 4-32 processes per node.

Threading makes more flexible use of system resources.
Instrumentation : Hardware Counters, MPI Data, Statement-level Profiling Data

Strategy : do data-reduction on the fly, save key information
Example: at the end of program execution (MPI_Finalize) one has information about the work distribution, MPI timing, etc. ... use it.

BGPM : Blue Gene Performance Monitor provides many counters for the A2 cores, caches, memory, and network devices.
Aggregate counts at the process level, node level, and/or job level.

PMPI interface : collect cumulative information for MPI routines
Optionally collect the detailed time-history of MPI events.

Statement-level profiling : use support for the profil() routine in GNU libc.a.
Get basic histogram data : #hits at each program counter, map hits to source lines using methods provided by GNU binary-file descriptor library.

Static linking with the instrumentation library is the default on BGQ.
Save Data from Selected Processes

Way back when : write one small file per process

Now : don’t want a million files … best to be selective about what you save.

Simple strategy for MPI applications : when the app reaches MPI_Finalize(), one can determine the ranks with the minimum, median, and maximum times in MPI … save detailed data for those ranks … histogram the distribution.

Optionally save all data in one file.

In most cases, the rank that spent the least time in MPI did the most work.

Can use the same strategy based on hardware-counter data.

Can maintain low overhead non-intrusive performance monitoring at full scale.

Scaling limitation = memory! Any data-structure with size proportional to #ranks will eventually be a problem.
### MPI profile data for LAMMPS, 1024K MPI ranks

Data for MPI rank 524474 of 1048576
Times and statistics from MPI_Init() to MPI_Finalize().

<table>
<thead>
<tr>
<th>MPI Routine</th>
<th>#calls</th>
<th>avg. bytes</th>
<th>time(sec)</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI_Comm_size</td>
<td>4</td>
<td>0.0</td>
<td>0.000</td>
</tr>
<tr>
<td>MPI_Comm_rank</td>
<td>10</td>
<td>0.0</td>
<td>0.000</td>
</tr>
<tr>
<td>MPI_Send</td>
<td>12618</td>
<td>25159.1</td>
<td>3.360</td>
</tr>
<tr>
<td>MPI_Irecv</td>
<td>12618</td>
<td>25154.3</td>
<td>0.069</td>
</tr>
<tr>
<td>MPI_Sendrecv</td>
<td>1188</td>
<td>4.0</td>
<td>0.248</td>
</tr>
<tr>
<td>MPI_Wait</td>
<td>12618</td>
<td>0.0</td>
<td>1.308</td>
</tr>
<tr>
<td>MPI_Bcast</td>
<td>69</td>
<td>183.0</td>
<td>0.083</td>
</tr>
<tr>
<td>MPI_Barrier</td>
<td>2</td>
<td>0.0</td>
<td>0.001</td>
</tr>
<tr>
<td>MPI_Allreduce</td>
<td>191</td>
<td>7.2</td>
<td>0.428</td>
</tr>
</tbody>
</table>

MPI task 524474 of 1048576 had the minimum communication time.
**total communication time** = 5.497 seconds.
**total elapsed time**       = 143.969 seconds.
**heap memory used**         = 64.301 MBytes.

This LAMMPS problem scales nearly perfectly beyond 1M processes. The fraction of time in messaging remains 3-4% from a single node to 72 racks.
**MPI profile data : LAMMPS, 1024K MPI ranks**

Histogram of times spent in MPI

<table>
<thead>
<tr>
<th>time-bin</th>
<th>#ranks</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.497</td>
<td>10</td>
</tr>
<tr>
<td>6.783</td>
<td>53</td>
</tr>
<tr>
<td>8.070</td>
<td>1199</td>
</tr>
<tr>
<td>9.357</td>
<td>13983</td>
</tr>
<tr>
<td>10.644</td>
<td>65604</td>
</tr>
<tr>
<td>11.931</td>
<td>131666</td>
</tr>
<tr>
<td>13.218</td>
<td>230704</td>
</tr>
<tr>
<td>14.505</td>
<td>238892</td>
</tr>
<tr>
<td>15.791</td>
<td>205515</td>
</tr>
<tr>
<td>17.078</td>
<td>89530</td>
</tr>
<tr>
<td>18.365</td>
<td>43006</td>
</tr>
<tr>
<td>19.652</td>
<td>19365</td>
</tr>
<tr>
<td>20.939</td>
<td>7149</td>
</tr>
<tr>
<td>22.226</td>
<td>1738</td>
</tr>
<tr>
<td>23.513</td>
<td>162</td>
</tr>
</tbody>
</table>

Roughly “normal” distribution of times spent in MPI over all ranks. The computational load is approximately balanced.
MPI profile data: DNS3D, 3072^3 grid, 768K MPI Ranks

Data for MPI rank 0 of 786432
Times and statistics from summary_start() to summary_stop().

<table>
<thead>
<tr>
<th>MPI Routine</th>
<th>#calls</th>
<th>avg. bytes</th>
<th>time (sec)</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI_Allreduce</td>
<td>894</td>
<td>53.3</td>
<td>0.393</td>
</tr>
<tr>
<td>MPI_Alltoallv</td>
<td>10728</td>
<td>384.0</td>
<td>91.164</td>
</tr>
</tbody>
</table>

Total communication time = 91.557 seconds.
Total elapsed time = 137.843 seconds.
Heap memory used = 38.371 MBytes.
Heap memory available = 783.617 MBytes.

Message size distributions:

<table>
<thead>
<tr>
<th>MPI_Allreduce</th>
<th>#calls</th>
<th>avg. bytes</th>
<th>time (sec)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>596</td>
<td>16.0</td>
<td>0.092</td>
</tr>
<tr>
<td></td>
<td>298</td>
<td>128.0</td>
<td>0.301</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>MPI_Alltoallv</th>
<th>#calls</th>
<th>avg. bytes</th>
<th>time (sec)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>5364</td>
<td>192.0</td>
<td>34.062</td>
</tr>
<tr>
<td></td>
<td>5364</td>
<td>576.0</td>
<td>57.102</td>
</tr>
</tbody>
</table>

Parallel 3D FFTs using the p3dffft library and MPI_Alltoallv with 2D topology
MPI profile data: DNS3D, 3072^3 grid, 768K ranks

512x1536 process grid, 786432 MPI ranks

elapsed time = 137.84 seconds

<table>
<thead>
<tr>
<th>MPI Time</th>
<th>#ranks</th>
<th>FP op count</th>
<th>flop-bin</th>
<th>#ranks</th>
</tr>
</thead>
<tbody>
<tr>
<td>time-bin</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>79.192</td>
<td>1468</td>
<td>1.862e+10</td>
<td>1.862e+10</td>
<td>1741</td>
</tr>
<tr>
<td>80.366</td>
<td>68</td>
<td>1.912e+10</td>
<td>1.912e+10</td>
<td>413445</td>
</tr>
<tr>
<td>81.540</td>
<td>0</td>
<td>1.962e+10</td>
<td>1.962e+10</td>
<td>369710</td>
</tr>
<tr>
<td>82.714</td>
<td>0</td>
<td>2.012e+10</td>
<td>2.012e+10</td>
<td>0</td>
</tr>
<tr>
<td>83.888</td>
<td>0</td>
<td>2.062e+10</td>
<td>2.062e+10</td>
<td>0</td>
</tr>
<tr>
<td>85.062</td>
<td>0</td>
<td>2.112e+10</td>
<td>2.112e+10</td>
<td>0</td>
</tr>
<tr>
<td>86.235</td>
<td>0</td>
<td>2.163e+10</td>
<td>2.163e+10</td>
<td>0</td>
</tr>
<tr>
<td>87.409</td>
<td>0</td>
<td>2.213e+10</td>
<td>2.213e+10</td>
<td>0</td>
</tr>
<tr>
<td>88.583</td>
<td>0</td>
<td>2.263e+10</td>
<td>2.263e+10</td>
<td>0</td>
</tr>
<tr>
<td>89.757</td>
<td>0</td>
<td>2.313e+10</td>
<td>2.313e+10</td>
<td>225</td>
</tr>
<tr>
<td>90.931</td>
<td>784895</td>
<td>2.363e+10</td>
<td>2.363e+10</td>
<td>1307</td>
</tr>
<tr>
<td>92.105</td>
<td>1</td>
<td>2.413e+10</td>
<td>2.413e+10</td>
<td>4</td>
</tr>
</tbody>
</table>

Some load-imbalance: a total of 1536 MPI ranks have about 20% more floating-point work, and all other MPI ranks wait for them. The ranks with extra work are ranks with pex = 511, where the 2D coords are (pex, pey).
Total time in MPI
~32K MPI ranks
Blue = smallest time
Red = largest time
Load Imbalance
GFDL atmosphere model – Chris Kerr.

Yellow bands arise from ranks that have one extra grid point.

Ranks in the red squares have one extra grid point in each of two dimensions.
Time-history of MPI events: GTC at 128K MPI ranks

Must be selective, for example capture data for just a few time steps, otherwise data volume is excessive. Can do event tracing at scale.
Statement-Level Profiling with the profil() Routine

Interrupt 100 times/sec; histogram = #hits at each program counter.

Use the same criteria as before to save selected profile data.

The profil() routine is a useful relic … it needs updating … histogram buffer is unsigned short, 16-bits, enough for 64K samples per address.

```plaintext
tics | Source
1265| disc = bb*bb - aa*cc
151 | if (disc .lt. 0.0) go to 4
6629| d = sqrt(disc)
4346| l1 = (d - bb)/aa
145 | if (l1 .ge. 1.0d-10) then
177 | z1 = z + w*l1
921 | zzidks = zz(id,ks)
874 | zzidks1= zz(id,ks+1)
337 | isign = (z1.lt.zzidks .and. z1.lt.zzidks1 ) .or.
    | & (z1.gt.zzidks1 .and. z1.gt.zzidks )
53  | if (isign) l1 = 1000.0d0
140 | if (l1 .lt. lmin) lmin = l1
    | endif
5278| l2 = (-bb - d)/aa
```
Some problems occur with millions of processes

32-bit integers overflow; two victims were DNS3D and GTC. It is past time for 64-bit default integer size. Current fix is to find where integer overflow occurs and use 8-byte integer types where it matters.

64-bit integers are not always adequate. Example: sum 64-bit counters on enough processors: 1 GHz for 1 day on 1M cores => ~9E19 counts, but a 64-bit integer can hold only ~2E19 counts. Current fix is to do sums with 64-bit floating-point types.

More than a million core files is a bad idea … I know from experience.

Memory utilization often grows with increasing #processes. Any data structure linear in #processes will eventually spell trouble.

Reducing the number of processes and using more threads can help. Mixed distributed-memory/shared-memory programming is here to stay.

Applications with excellent locality, no global data structures, have an advantage when it comes to scaling to millions of processes.
Example of code tuning for BG/Q : GYRO

General Atomics code : https://fusion.gat.com/theory/Gyro

Objectives : Extend OpenMP coverage and scaling

Make minor adjustments to improve computation and communication performance.

Blue = original code
Red = tuned code
Performance is shown relative to orig. code with 1 OpenMP thread.
GYRO : Optimization example - original code

```
p_nek_loc = 0
do p_nek=1+i_proc_1,n_nek_1,n_proc_1
   do is = 1, n_gk
      p_nek_loc = p_nek_loc+1
      
      gyro_uv(:,:,p_nek_loc,is,1) = (0.0,0.0)
      kyro_uv(:,:,p_nek_loc,is,1) = (0.0,0.0)
   !$omp parallel do default(shared) private(i_diff,m)
   do i=1,n_x
      do i_diff=-m_gyro,m_gyro-i_gyro
         do m=1,n_stack
            gyro_uv(m,i,p_nek_loc,is,1) = gyro_uv(m,i,p_nek_loc,is,1) +&
            w_gyro(m,i_diff,i,p_nek_loc,is)*vf(m,i+i_diff,1)
            kyro_uv(m,i,p_nek_loc,is,1) = kyro_uv(m,i,p_nek_loc,is,1) +&
            w_gyro_rot(m,i_diff,i,p_nek_loc,is)*vf(m,i+i_diff,1)
         enddo
      enddo
   enddo
   !$omp end parallel do
end do
```

Issues: OpenMP parallel region is inside nested loops => repeat the overhead.

Large arrays are set to zero outside the OpenMP parallel region.
GYRO: Optimization example - tuned code

```c
!$omp parallel private(p_nek_loc, . . .)
  p_nek_loc = 0
  do p_nek=1+i_proc_1,n_nek_1,n_proc_1
    do is = 1, n_gk
      p_nek_loc = p_nek_loc+1
    . . .
    do i=ibeg, iend
      gyro_uv(:,i,p_nek_loc,is,1) = (0.0,0.0)
      kyro_uv(:,i,p_nek_loc,is,1) = (0.0,0.0)
      do i_diff=-m_gyro,m_gyro-i_gyro
        do m=1,n_stack
          gyro_uv(m,i,p_nek_loc,is,1) = gyro_uv(m,i,p_nek_loc,is,1) +&
            w_gyro(m,i_diff,i,p_nek_loc,is)*vf(m,i+i_diff,1)
          kyro_uv(m,i,p_nek_loc,is,1) = kyro_uv(m,i,p_nek_loc,is,1) +&
            w_gyro_rot(m,i_diff,i,p_nek_loc,is)*vf(m,i+i_diff,1)
        enddo
      enddo
      . . .
    enddo
  end do
!$omp end parallel
```

OpenMP parallel region is outside the nested loops, block partitioned “i” loop.
Large arrays are set to zero inside the OpenMP parallel region.
GYRO: Optimization example – transpose operation

Original code: transpose (alltoall) is called in a loop using short messages

```
|     call rTRANSP_INIT(n_i,n_j,n_k,NEW_COMM_1)
|     do m=1,n_stack
374|        call rTRANSP_DO(f_coll(m,:,:),h_C(m,:,:))
|     enddo
|     call rTRANSP_CLEANUP
```

Tuned code uses one alltoall and all memory accesses are stride-1

```
|     call rTRANSP_INIT(n_i,n_j,n_k,n_stack,NEW_COMM_1)
|     call rTRANSP_DO(f_coll,h_C)
|     call rTRANSP_CLEANUP
```

Result is far fewer calls to MPI_Alltoall, using larger messages. Eliminates array-section copies at “bad” stride. Roughly 3x improvement for the collision code-section.
Can use 24 counters per A2 core, so just 6 counters per hardware thread when counting on all four hardware threads. 64-bit counters.

Good default choice of A2 counters:
- PEVT_LSU_COMMIT_CACHEABLE_LDS: load instructions
- PEVT_L1P_BAS_MISS: load missed L1P buffer
- PEVT_INST_XU_ALL: XU instructions : int/ld/st/br
- PEVT_INST_QFPU_ALL: AXU = FPU instructions
- PEVT_INST_QFPU_FPGRP1: weighted floating-point ops

Use along with L2 counters:
- PEVT_L2_HITS: L2 hits
- PEVT_L2_MISSES: L2 misses
- PEVT_L2_FETCH_LINE: 128-byte lines loaded from memory
- PEVT_L2_STORE_LINE: 128-byte lines stored to memory

The A2 counters are hardware-thread specific. The L2 counters are shared across the node. These counters give instruction throughput, instruction mix, information about load misses at all levels of cache/memory, and the load/store traffic to memory. Other counters are needed to get more details.
### SPHOT : Instruction Mix, 16K cores

<table>
<thead>
<tr>
<th>SPHOT</th>
<th>XU</th>
<th>AXU</th>
<th>Floating-Point</th>
</tr>
</thead>
<tbody>
<tr>
<td>Int/Ld/St/Br</td>
<td>61.57</td>
<td>38.43</td>
<td></td>
</tr>
<tr>
<td>FP Loads</td>
<td>17.94</td>
<td>26.82</td>
<td>FP single</td>
</tr>
<tr>
<td>FP Stores</td>
<td>1.82</td>
<td>47.25</td>
<td>FP madd</td>
</tr>
<tr>
<td>Quad Loads</td>
<td>0.00</td>
<td>0.42</td>
<td>FP div</td>
</tr>
<tr>
<td>Quad Stores</td>
<td>0.00</td>
<td>0.12</td>
<td>FP sqrt</td>
</tr>
<tr>
<td>Int Loads</td>
<td>11.61</td>
<td>19.97</td>
<td>FP other</td>
</tr>
<tr>
<td>Int Stores</td>
<td>7.74</td>
<td>2.76</td>
<td>FP move</td>
</tr>
<tr>
<td>Branch</td>
<td>14.82</td>
<td>0.00</td>
<td>Quad single</td>
</tr>
<tr>
<td>Int Arithmetic</td>
<td>45.32</td>
<td>0.00</td>
<td>Quad madd</td>
</tr>
<tr>
<td>Int Other</td>
<td>0.74</td>
<td>2.67</td>
<td>Quad other</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0.00</td>
<td>Quad move</td>
</tr>
<tr>
<td><strong>Sum</strong></td>
<td>100.00</td>
<td>100.00</td>
<td><strong>Sum</strong></td>
</tr>
</tbody>
</table>

Instruction mix is dominated by integer, load, store, branch operations.
### SPHOT: Speed-up using multiple threads per core

<table>
<thead>
<tr>
<th>threads/core</th>
<th>1</th>
<th>2</th>
<th>4</th>
<th>units</th>
</tr>
</thead>
<tbody>
<tr>
<td>performance</td>
<td>1.00</td>
<td>1.88</td>
<td>2.94</td>
<td>relative</td>
</tr>
<tr>
<td>total instr</td>
<td>1.00</td>
<td>1.00</td>
<td>1.01</td>
<td>relative</td>
</tr>
<tr>
<td>issue rate</td>
<td>0.32</td>
<td>0.60</td>
<td>0.94</td>
<td>instr/cycle</td>
</tr>
<tr>
<td>GFlops/node</td>
<td>4.9</td>
<td>9.2</td>
<td>14.5</td>
<td></td>
</tr>
<tr>
<td>L1</td>
<td>91.2</td>
<td>91.2</td>
<td>88.8</td>
<td>%</td>
</tr>
<tr>
<td>L1P</td>
<td>0.6</td>
<td>0.2</td>
<td>0.1</td>
<td>%</td>
</tr>
<tr>
<td>L2</td>
<td>8.1</td>
<td>8.5</td>
<td>11.0</td>
<td>%</td>
</tr>
<tr>
<td>DDR</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>%</td>
</tr>
<tr>
<td>LD-BW</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>Bytes/cycle</td>
</tr>
<tr>
<td>ST-BW</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>Bytes/cycle</td>
</tr>
<tr>
<td>TOT-BW</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>Bytes/cycle</td>
</tr>
</tbody>
</table>

SPHOT has the highest speed-up ~3x for 4 threads per core. The main performance issue is pipeline stalls, not data loads/stores.
GTC : Instruction Mix , main loop, 8K cores

<table>
<thead>
<tr>
<th>GTC</th>
<th>XU</th>
<th>AXU</th>
<th>Floating-Point</th>
</tr>
</thead>
<tbody>
<tr>
<td>Int/Ld/St/Br</td>
<td>65.3</td>
<td>34.7</td>
<td></td>
</tr>
<tr>
<td>FP Loads</td>
<td>27.3</td>
<td>37.5</td>
<td>FP single</td>
</tr>
<tr>
<td>FP Stores</td>
<td>8.3</td>
<td>38.5</td>
<td>FP madd</td>
</tr>
<tr>
<td>Quad Loads</td>
<td>0.0</td>
<td>0.3</td>
<td>FP div</td>
</tr>
<tr>
<td>Quad Stores</td>
<td>0.0</td>
<td>0.2</td>
<td>FP sqrt</td>
</tr>
<tr>
<td>Int Loads</td>
<td>18.5</td>
<td>20.8</td>
<td>FP other</td>
</tr>
<tr>
<td>Int Stores</td>
<td>6.4</td>
<td>2.3</td>
<td>FP move</td>
</tr>
<tr>
<td>Branch</td>
<td>8.4</td>
<td>0.0</td>
<td>Quad single</td>
</tr>
<tr>
<td>Int Arithmetic</td>
<td>29.6</td>
<td>0.0</td>
<td>Quad madd</td>
</tr>
<tr>
<td>Int Other</td>
<td>1.5</td>
<td>0.4</td>
<td>Quad other</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0.0</td>
<td>Quad move</td>
</tr>
<tr>
<td>Sum</td>
<td>100.0</td>
<td>100.0</td>
<td>Sum</td>
</tr>
</tbody>
</table>

Roughly 2:1 ratio of integer/load/store/branch operations to floating-point.
GTC : Speed-up using multiple threads per core.

<table>
<thead>
<tr>
<th>threads/core</th>
<th>1</th>
<th>2</th>
<th>4</th>
<th>units</th>
</tr>
</thead>
<tbody>
<tr>
<td>performance</td>
<td>1.00</td>
<td>1.64</td>
<td>2.25</td>
<td>relative</td>
</tr>
<tr>
<td>total instr</td>
<td>1.00</td>
<td>1.01</td>
<td>1.06</td>
<td>relative</td>
</tr>
<tr>
<td>issue rate</td>
<td>0.30</td>
<td>0.50</td>
<td>0.71</td>
<td>instr/cycle</td>
</tr>
<tr>
<td>GFlops/node</td>
<td>3.8</td>
<td>6.2</td>
<td>8.5</td>
<td>GFlops/node</td>
</tr>
<tr>
<td>L1</td>
<td>94.3</td>
<td>94.3</td>
<td>94.3</td>
<td>%</td>
</tr>
<tr>
<td>L1P</td>
<td>1.9</td>
<td>1.6</td>
<td>1.4</td>
<td>%</td>
</tr>
<tr>
<td>L2</td>
<td>3.1</td>
<td>3.2</td>
<td>3.2</td>
<td>%</td>
</tr>
<tr>
<td>DDR</td>
<td>0.8</td>
<td>1.0</td>
<td>1.2</td>
<td>%</td>
</tr>
<tr>
<td>LD-BW</td>
<td>2.0</td>
<td>3.5</td>
<td>5.9</td>
<td>Bytes/cycle</td>
</tr>
<tr>
<td>ST-BW</td>
<td>0.8</td>
<td>1.5</td>
<td>2.8</td>
<td>Bytes/cycle</td>
</tr>
<tr>
<td>TOT-BW</td>
<td>2.9</td>
<td>4.9</td>
<td>8.7</td>
<td>Bytes/cycle</td>
</tr>
</tbody>
</table>

Get 2.25x speedup using 4 threads/core, very little contention for caches, modest memory bandwidth requirement, good total instruction throughput => efficient use of the cores.
LAMMPS : Instruction Mix, main loop, 16K cores

<table>
<thead>
<tr>
<th>LAMMPS</th>
<th>XU</th>
<th>AXU</th>
<th>Floating-Point</th>
</tr>
</thead>
<tbody>
<tr>
<td>Int/Ld/St/Br</td>
<td>69.7</td>
<td>30.3</td>
<td></td>
</tr>
<tr>
<td>FP Loads</td>
<td>23.4</td>
<td>43.6</td>
<td>FP single</td>
</tr>
<tr>
<td>FP Stores</td>
<td>7.1</td>
<td>43.9</td>
<td>FP madd</td>
</tr>
<tr>
<td>Quad Loads</td>
<td>0.0</td>
<td>0.1</td>
<td>FP div</td>
</tr>
<tr>
<td>Quad Stores</td>
<td>0.1</td>
<td>0.0</td>
<td>FP sqrt</td>
</tr>
<tr>
<td>Int Loads</td>
<td>26.7</td>
<td>12.4</td>
<td>FP other</td>
</tr>
<tr>
<td>Int Stores</td>
<td>2.7</td>
<td>0.0</td>
<td>FP move</td>
</tr>
<tr>
<td>Branch</td>
<td>8.6</td>
<td>0.0</td>
<td>Quad single</td>
</tr>
<tr>
<td>Int Arithmetic</td>
<td>31.0</td>
<td>0.0</td>
<td>Quad madd</td>
</tr>
<tr>
<td>Int Other</td>
<td>0.3</td>
<td>0.0</td>
<td>Quad other</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Quad move</td>
</tr>
<tr>
<td>Sum</td>
<td>100.0</td>
<td>100.0</td>
<td>Sum</td>
</tr>
</tbody>
</table>

More than 2:1 ratio of integer/load/store/branch instructions to floating-point.
LAMMPS : Speed-up using multiple threads per core

<table>
<thead>
<tr>
<th>threads/core</th>
<th>1</th>
<th>2</th>
<th>4</th>
<th>units</th>
</tr>
</thead>
<tbody>
<tr>
<td>performance</td>
<td>1.00</td>
<td>1.61</td>
<td>2.39</td>
<td>relative</td>
</tr>
<tr>
<td>total instr</td>
<td>1.00</td>
<td>0.99</td>
<td>1.05</td>
<td>relative</td>
</tr>
<tr>
<td>issue rate</td>
<td>0.25</td>
<td>0.39</td>
<td>0.62</td>
<td>instr/cycle</td>
</tr>
<tr>
<td>GFlops/node</td>
<td>2.8</td>
<td>4.5</td>
<td>6.6</td>
<td></td>
</tr>
<tr>
<td>L1</td>
<td>92.8</td>
<td>89.9</td>
<td>87.5</td>
<td>%</td>
</tr>
<tr>
<td>L1P</td>
<td>0.8</td>
<td>1.2</td>
<td>1.2</td>
<td>%</td>
</tr>
<tr>
<td>L2</td>
<td>5.9</td>
<td>8.5</td>
<td>10.8</td>
<td>%</td>
</tr>
<tr>
<td>DDR</td>
<td>0.5</td>
<td>0.5</td>
<td>0.5</td>
<td>%</td>
</tr>
<tr>
<td>LD-BW</td>
<td>1.2</td>
<td>1.9</td>
<td>2.9</td>
<td>Bytes/cycle</td>
</tr>
<tr>
<td>ST-BW</td>
<td>0.4</td>
<td>0.7</td>
<td>1.2</td>
<td>Bytes/cycle</td>
</tr>
<tr>
<td>TOT-BW</td>
<td>1.6</td>
<td>2.6</td>
<td>4.1</td>
<td>Bytes/cycle</td>
</tr>
</tbody>
</table>

Get ~2.4x speed-up with four threads/core, in spite of clear evidence of contention for L1 D-Cache. Memory bandwidth requirement is low, instruction issue rate is good.
Held-Suarez : Instruction Mix, main loop, 32K cores

<table>
<thead>
<tr>
<th>Held-Suarez</th>
<th>XU</th>
<th>AXU</th>
<th>Floating-Point</th>
</tr>
</thead>
<tbody>
<tr>
<td>Int/Ld/St/Br</td>
<td>56.7</td>
<td>43.3</td>
<td>Floating-Point</td>
</tr>
<tr>
<td>FP Loads</td>
<td>17.9</td>
<td>48.7</td>
<td>FP single</td>
</tr>
<tr>
<td>FP Stores</td>
<td>13.4</td>
<td>15.5</td>
<td>FP madd</td>
</tr>
<tr>
<td>Quad Loads</td>
<td>0.7</td>
<td>1.4</td>
<td>FP div</td>
</tr>
<tr>
<td>Quad Stores</td>
<td>0.3</td>
<td>0.0</td>
<td>FP sqrt</td>
</tr>
<tr>
<td>Int Loads</td>
<td>12.6</td>
<td>29.1</td>
<td>FP other</td>
</tr>
<tr>
<td>Int Stores</td>
<td>6.7</td>
<td>4.0</td>
<td>FP move</td>
</tr>
<tr>
<td>Branch</td>
<td>10.4</td>
<td>0.0</td>
<td>Quad single</td>
</tr>
<tr>
<td>Int Arithmetic</td>
<td>35.5</td>
<td>0.0</td>
<td>Quad madd</td>
</tr>
<tr>
<td>Int Other</td>
<td>2.3</td>
<td>1.3</td>
<td>Quad other</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0.0</td>
<td>Quad move</td>
</tr>
<tr>
<td>Sum</td>
<td>100.0</td>
<td>100.0</td>
<td>Sum</td>
</tr>
</tbody>
</table>

Closer balance for the two execution units, but still more Int/Ld/St/Br.
Held-Suarez: Speed-up using multiple threads per core.

<table>
<thead>
<tr>
<th>threads/core</th>
<th>1</th>
<th>2</th>
<th>4</th>
<th>units</th>
</tr>
</thead>
<tbody>
<tr>
<td>performance</td>
<td>1.00</td>
<td>1.72</td>
<td>2.18</td>
<td>relative</td>
</tr>
<tr>
<td>total instr</td>
<td>1.00</td>
<td>1.03</td>
<td>1.10</td>
<td>relative</td>
</tr>
<tr>
<td>issue rate</td>
<td>0.37</td>
<td>0.66</td>
<td>0.89</td>
<td>instr/cycle</td>
</tr>
<tr>
<td>GFlops/node</td>
<td>5.3</td>
<td>9.2</td>
<td>11.9</td>
<td>GFlops/node</td>
</tr>
<tr>
<td>L1</td>
<td>93.0</td>
<td>93.4</td>
<td>93.0</td>
<td>%</td>
</tr>
<tr>
<td>L1P</td>
<td>6.4</td>
<td>5.8</td>
<td>5.2</td>
<td>%</td>
</tr>
<tr>
<td>L2</td>
<td>0.1</td>
<td>0.0</td>
<td>0.5</td>
<td>%</td>
</tr>
<tr>
<td>DDR</td>
<td>0.5</td>
<td>0.8</td>
<td>1.2</td>
<td>%</td>
</tr>
<tr>
<td>LD-BW</td>
<td>1.1</td>
<td>3.1</td>
<td>6.6</td>
<td>Bytes/cycle</td>
</tr>
<tr>
<td>ST-BW</td>
<td>1.1</td>
<td>2.9</td>
<td>4.8</td>
<td>Bytes/cycle</td>
</tr>
<tr>
<td>TOT-BW</td>
<td>2.3</td>
<td>6.0</td>
<td>11.4</td>
<td>Bytes/cycle</td>
</tr>
</tbody>
</table>

Get ~2.18x speed-up with four threads per core. There is some instruction inflation, and significant requirement for memory bandwidth. The total instruction issue rate is very good.
### NEK : Instruction Mix, 64K cores

<table>
<thead>
<tr>
<th>NEK</th>
<th>XU</th>
<th>AXU</th>
<th>Floating-Point</th>
</tr>
</thead>
<tbody>
<tr>
<td>Int/Ld/St/Br</td>
<td>72.9</td>
<td>27.1</td>
<td></td>
</tr>
<tr>
<td>FP Loads</td>
<td>28.3</td>
<td>11.8</td>
<td>FP single</td>
</tr>
<tr>
<td>FP Stores</td>
<td>8.7</td>
<td>41.6</td>
<td>FP madd</td>
</tr>
<tr>
<td>Quad Loads</td>
<td>4.5</td>
<td>0.0</td>
<td>FP div</td>
</tr>
<tr>
<td>Quad Stores</td>
<td>2.1</td>
<td>0.0</td>
<td>FP sqrt</td>
</tr>
<tr>
<td>Int Loads</td>
<td>11.0</td>
<td>1.7</td>
<td>FP other</td>
</tr>
<tr>
<td>Int Stores</td>
<td>5.7</td>
<td>0.4</td>
<td>FP move</td>
</tr>
<tr>
<td>Branch</td>
<td>11.3</td>
<td>1.7</td>
<td>Quad single</td>
</tr>
<tr>
<td>Int Arithmetic</td>
<td>26.9</td>
<td>28.5</td>
<td>Quad madd</td>
</tr>
<tr>
<td>Int Other</td>
<td>1.6</td>
<td>0.0</td>
<td>Quad other</td>
</tr>
<tr>
<td></td>
<td></td>
<td>14.3</td>
<td>Quad move</td>
</tr>
<tr>
<td><strong>Sum</strong></td>
<td>100.0</td>
<td>100.0</td>
<td><strong>Sum</strong></td>
</tr>
</tbody>
</table>

QPX multiply-add instructions are mainly from matrix-matrix multiplication routines, integer/load/store/branch instructions dominate.
NEK : Speed-up using multiple MPI ranks per core

<table>
<thead>
<tr>
<th>threads/core</th>
<th>1</th>
<th>2</th>
<th>4</th>
<th>units</th>
</tr>
</thead>
<tbody>
<tr>
<td>performance</td>
<td>1.00</td>
<td>1.39</td>
<td>1.46</td>
<td>relative</td>
</tr>
<tr>
<td>total instr</td>
<td>1.00</td>
<td>1.08</td>
<td>1.20</td>
<td>relative</td>
</tr>
<tr>
<td>issue rate</td>
<td>0.32</td>
<td>0.50</td>
<td>0.57</td>
<td>instr/cycle</td>
</tr>
<tr>
<td>GFlops/node</td>
<td>7.5</td>
<td>10.5</td>
<td>11.0</td>
<td>GFlops/node</td>
</tr>
<tr>
<td>L1</td>
<td>92.4</td>
<td>91.0</td>
<td>88.9</td>
<td>%</td>
</tr>
<tr>
<td>L1P</td>
<td>6.3</td>
<td>6.8</td>
<td>6.8</td>
<td>%</td>
</tr>
<tr>
<td>L2</td>
<td>0.6</td>
<td>1.2</td>
<td>3.0</td>
<td>%</td>
</tr>
<tr>
<td>DDR</td>
<td>0.7</td>
<td>0.9</td>
<td>1.3</td>
<td>%</td>
</tr>
<tr>
<td>LD-BW</td>
<td>4.2</td>
<td>6.2</td>
<td>7.7</td>
<td>Bytes/cycle</td>
</tr>
<tr>
<td>ST-BW</td>
<td>1.8</td>
<td>2.7</td>
<td>3.3</td>
<td>Bytes/cycle</td>
</tr>
<tr>
<td>TOT-BW</td>
<td>6.1</td>
<td>8.9</td>
<td>11.0</td>
<td>Bytes/cycle</td>
</tr>
</tbody>
</table>

The total instruction count increases (near the strong-scaling limit) and the memory-bandwidth requirement is significant. The speed-up is limited, but the instruction throughput is still good.
### UMT: Instruction Mix, 16K cores

<table>
<thead>
<tr>
<th>UMT</th>
<th>XU</th>
<th>AXU</th>
<th>Floating-Point</th>
</tr>
</thead>
<tbody>
<tr>
<td>Int/Ld/St/Br</td>
<td>79.0</td>
<td>21.0</td>
<td></td>
</tr>
<tr>
<td>FP Loads</td>
<td>15.8</td>
<td>24.8</td>
<td>FP single</td>
</tr>
<tr>
<td>FP Stores</td>
<td>6.6</td>
<td>18.5</td>
<td>FP madd</td>
</tr>
<tr>
<td>Quad Loads</td>
<td>7.4</td>
<td>0.2</td>
<td>FP div</td>
</tr>
<tr>
<td>Quad Stores</td>
<td>4.6</td>
<td>0.0</td>
<td>FP sqrt</td>
</tr>
<tr>
<td>Int Loads</td>
<td>12.9</td>
<td>3.5</td>
<td>FP other</td>
</tr>
<tr>
<td>Int Stores</td>
<td>5.7</td>
<td>0.1</td>
<td>FP move</td>
</tr>
<tr>
<td>Branch</td>
<td>11.0</td>
<td>19.1</td>
<td>Quad single</td>
</tr>
<tr>
<td>Int Arithmetic</td>
<td>34.4</td>
<td>28.2</td>
<td>Quad madd</td>
</tr>
<tr>
<td>Int Other</td>
<td>1.7</td>
<td>2.3</td>
<td>Quad other</td>
</tr>
<tr>
<td></td>
<td></td>
<td>3.4</td>
<td>Quad move</td>
</tr>
<tr>
<td>Sum</td>
<td>100.0</td>
<td>100.0</td>
<td>Sum</td>
</tr>
</tbody>
</table>

Good QPX code generation by the compiler; integer, load, store, branch instructions dominate the mix.
**UMT : Speed-up using multiple threads per core**

<table>
<thead>
<tr>
<th>threads/core</th>
<th>1</th>
<th>2</th>
<th>4</th>
<th>units</th>
</tr>
</thead>
<tbody>
<tr>
<td>performance</td>
<td>1.00</td>
<td>1.32</td>
<td>1.30</td>
<td>relative</td>
</tr>
<tr>
<td>total instr</td>
<td>1.00</td>
<td>1.00</td>
<td>1.02</td>
<td>relative</td>
</tr>
<tr>
<td>issue rate</td>
<td>0.28</td>
<td>0.38</td>
<td>0.38</td>
<td>instr/cycle</td>
</tr>
<tr>
<td>GFlops/node</td>
<td>5.8</td>
<td>7.6</td>
<td>7.5</td>
<td></td>
</tr>
<tr>
<td>L1</td>
<td>93.1</td>
<td>92.4</td>
<td>89.1</td>
<td>%</td>
</tr>
<tr>
<td>L1P</td>
<td>5.4</td>
<td>5.3</td>
<td>5.5</td>
<td>%</td>
</tr>
<tr>
<td>L2</td>
<td>0.0</td>
<td>0.0</td>
<td>2.1</td>
<td>%</td>
</tr>
<tr>
<td>DDR</td>
<td>1.5</td>
<td>2.3</td>
<td>3.4</td>
<td>%</td>
</tr>
<tr>
<td>LD-BW</td>
<td>7.1</td>
<td>10.1</td>
<td>10.2</td>
<td>Bytes/cycle</td>
</tr>
<tr>
<td>ST-BW</td>
<td>2.6</td>
<td>3.4</td>
<td>3.4</td>
<td>Bytes/cycle</td>
</tr>
<tr>
<td>TOT-BW</td>
<td>9.7</td>
<td>13.5</td>
<td>13.6</td>
<td>Bytes/cycle</td>
</tr>
</tbody>
</table>

Speed-up is limited by bandwidth to memory.
Performance Data Repository

Collect performance data and store them into Mysql database
Help to characterize applications and machine usage efficiently
Uniform storage format to support queries and presentation

Chart from I-Hsin Chung, IBM Watson
## Average Application Characteristics vs. Key Benchmarks

<table>
<thead>
<tr>
<th></th>
<th>%FXU</th>
<th>%FPU</th>
<th>%Max Flops</th>
<th>%DDR BW</th>
<th>IPC</th>
</tr>
</thead>
<tbody>
<tr>
<td>App AVG</td>
<td>70.5</td>
<td>29.5</td>
<td>5.7</td>
<td>40.7</td>
<td>0.56</td>
</tr>
<tr>
<td>Linpack</td>
<td>43.8</td>
<td>56.2</td>
<td>74.9</td>
<td>22.6</td>
<td>1.34</td>
</tr>
<tr>
<td>Graph 500</td>
<td>100.0</td>
<td>0.0</td>
<td>0.0</td>
<td>75.9</td>
<td>0.34</td>
</tr>
</tbody>
</table>

Example algorithms:

- sparse matrix-vector multiplication: 80% Int/Ld/St/Br 20% FPU
- array update \( y(\cdot) = a \times x(\cdot) + y(\cdot) \) 78% Int/Ld/St/Br 22% FPU

IPC = instructions completed per cycle per core is a good indicator of how much work you are getting out of each core.

The general characteristics of most scientific applications are pretty similar, and are really different from some popular benchmarks.
Conclusions

The Blue Gene/Q design, low-power simple cores, four hardware threads per core, results in high instruction throughput, and thus exceptional power efficiency for applications. Can effectively fill in pipeline stalls and hide latencies in the memory subsystem.

The consequence is low performance per thread, so a high degree of parallelization is required for high application performance.

Traditional programming methods (MPI, OpenMP, Pthreads) hold up at very large scales. Memory costs can limit scaling when there are data-structures with size linear in the number of processes, threading helps by keeping the number of processes manageable.

Detailed performance analysis is viable at >10^6 processes but requires care. On-the-fly performance data reduction has merits.
Acknowledgements

IBM staff past and present worldwide

Livermore National Laboratory

Argonne National Laboratory

Many users who struggle to get excellent parallel performance.

U.S. Dept. of Energy    LLNL subcontract no. B554331