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

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

#1 Top 500 List 06/2012 : 16.3 PFlops 96-rack Sequoia system LLNL
#2 Top 500 List 11/2101 … 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

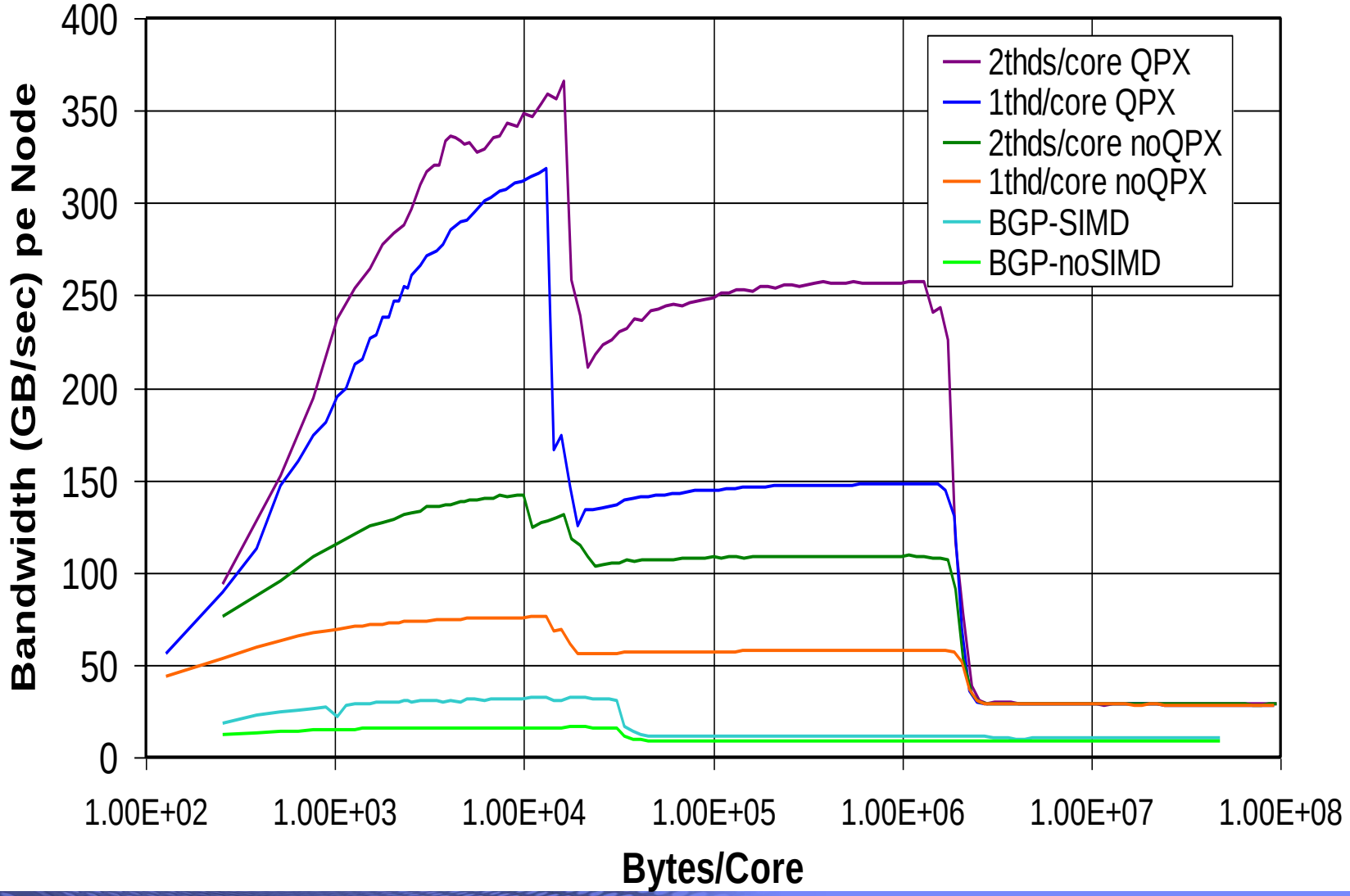
System on a chip : cores, L2 cache, network devices are integrated on the chip

A2 cores: simple in-order execution 1.6 GHz frequency, no ILP

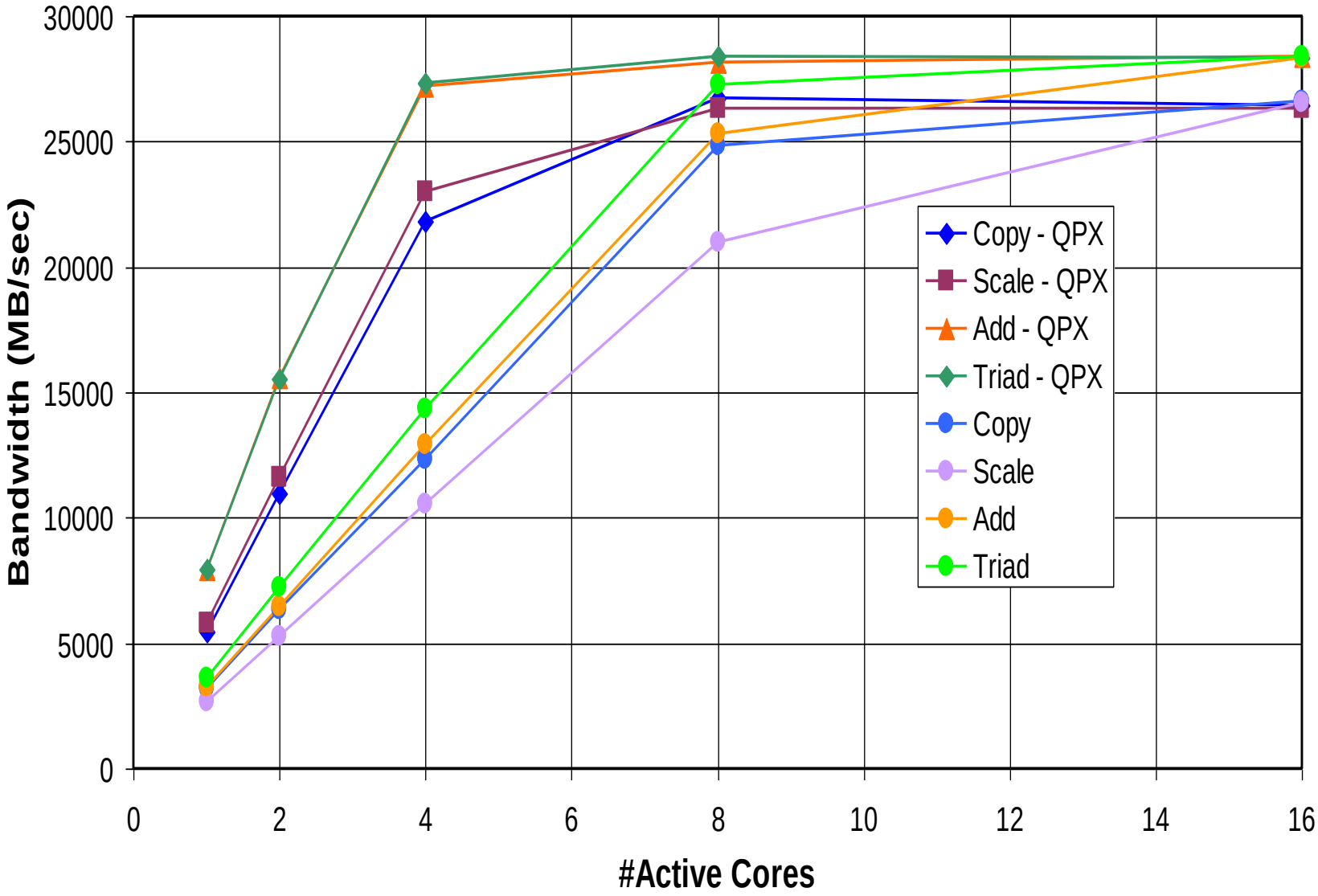
- Two execution units: XU for Integer/Load/Store, AXU for Floating-Point
- Six cycle latency, single-cycle throughput for floating-point operations.
- Four hardware threads … four sets of registers … the key to performance.
- At most one instruction can be completed per cycle per thread.
- At most two 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

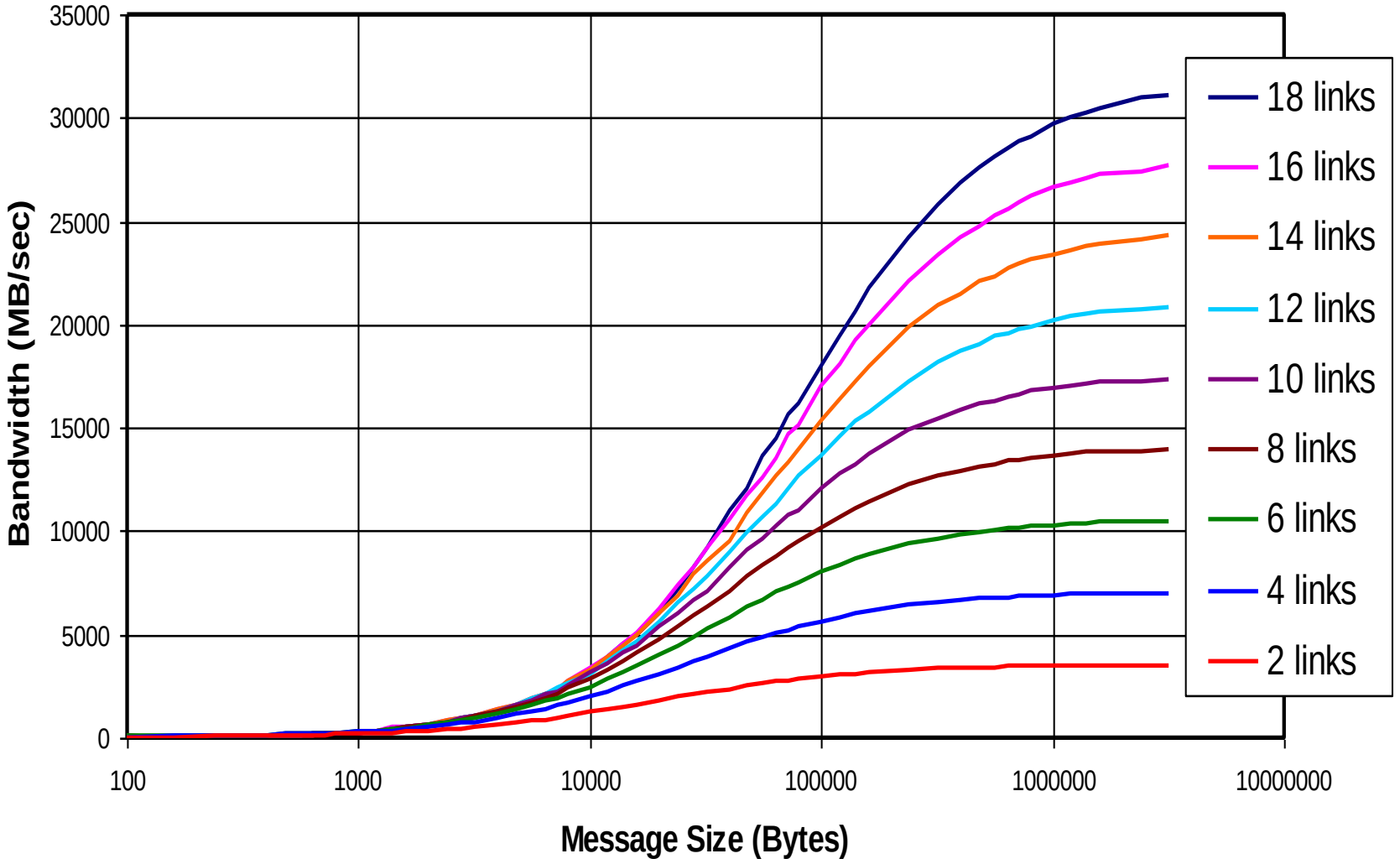
BG/Q Daxpy y(:) = a*x(:) + y(:)



Stream Benchmark



BGQ Link Bandwidth Test



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.

Processes/node	MB/process	%hardware
64	206	80.4%
32	460	89.8%
16	970	94.7%
8	1929	94.2%
4	3969	96.9%

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

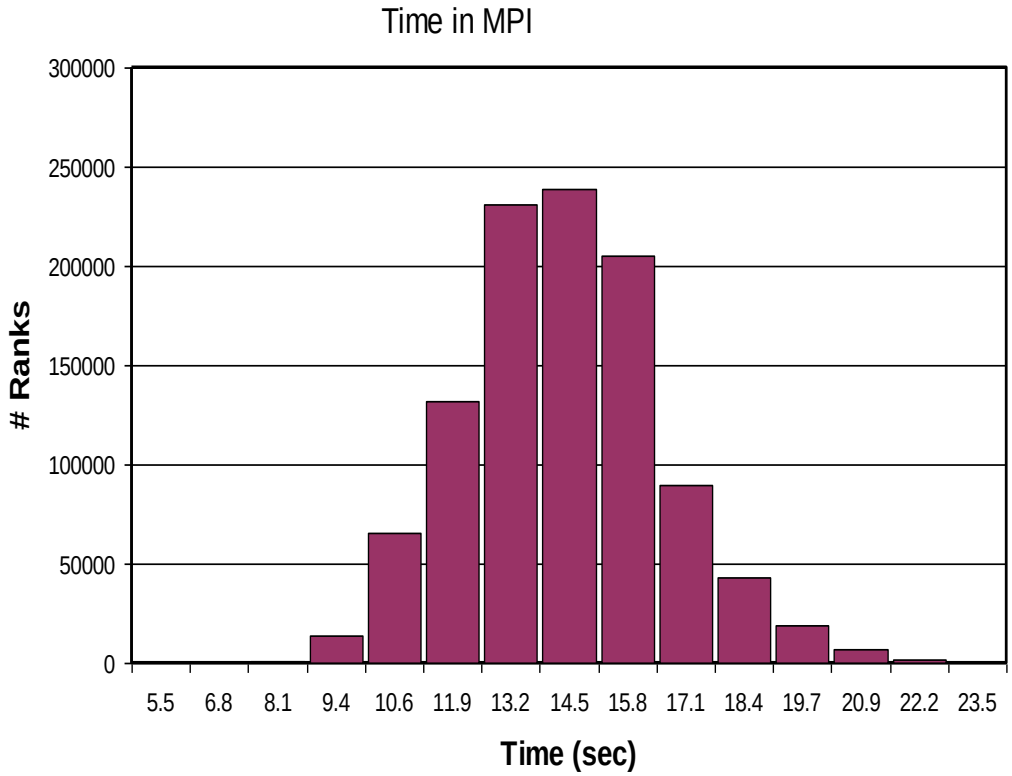
```
Data for MPI rank 524474 of 1048576
Times and statistics from MPI_Init() to MPI_Finalize().
---------------------------------------------------------------
MPI Routine                #calls     avg. bytes      time(sec)
---------------------------------------------------------------
MPI_Comm_size                   4            0.0          0.000
MPI_Comm_rank                  10            0.0          0.000
MPI_Send                    12618        25159.1          3.360
MPI_Irecv                   12618        25154.3          0.069
MPI_Sendrecv                 1188            4.0          0.248
MPI_Wait                    12618            0.0          1.308
MPI_Bcast                      69          183.0          0.083
MPI_Barrier                     2            0.0          0.001
MPI_Allreduce                 191            7.2          0.428
---------------------------------------------------------------
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

time-bin	#ranks
5.497	10
6.783	53
8.070	1199
9.357	13983
10.644	65604
11.931	131666
13.218	230704
14.505	238892
15.791	205515
17.078	89530
18.365	43006
19.652	19365
20.939	7149
22.226	1738
23.513	162



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

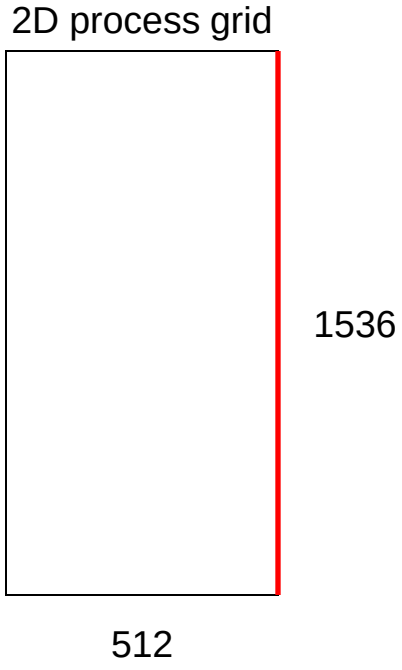
```
Data for MPI rank 0 of 786432
Times and statistics from summary_start() to summary_stop().
-----------------------------------------------------------
MPI Routine            #calls     avg. bytes      time(sec)
-----------------------------------------------------------
MPI_Allreduce             894           53.3         0.393
MPI_Alltoallv           10728          384.0        91.164
-----------------------------------------------------------
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:

MPI_Allreduce         #calls    avg. bytes      time(sec)
                         596          16.0         0.092
                         298         128.0         0.301

MPI_Alltoallv         #calls    avg. bytes      time(sec)
                        5364         192.0        34.062
                        5364         576.0        57.102
-----------------------------------------------------------
```

Parallel 3D FFTs using the p3dfft library and MPI_Alltoallv with 2D topology

MPI profile data : DNS3D, 3072^3 grid, 768K ranks

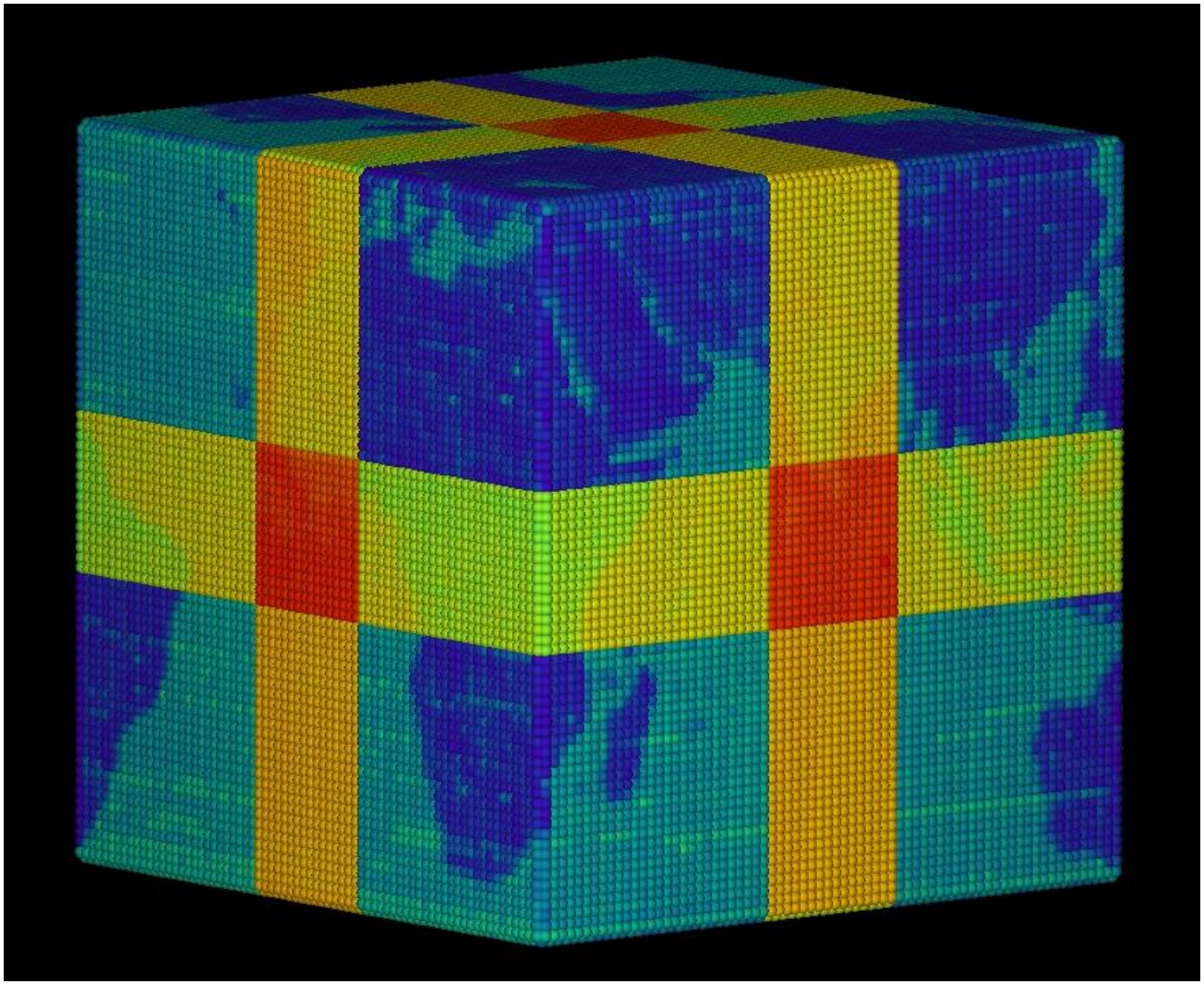
```
 512x1536 process grid, 786432 MPI ranks

 elapsed time = 137.84 seconds
```

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



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).

MPI Timing Data Mapped to the Simulation Domain



Total time in MPI

~32K MPI ranks

Blue = smallest time

Red = largest time

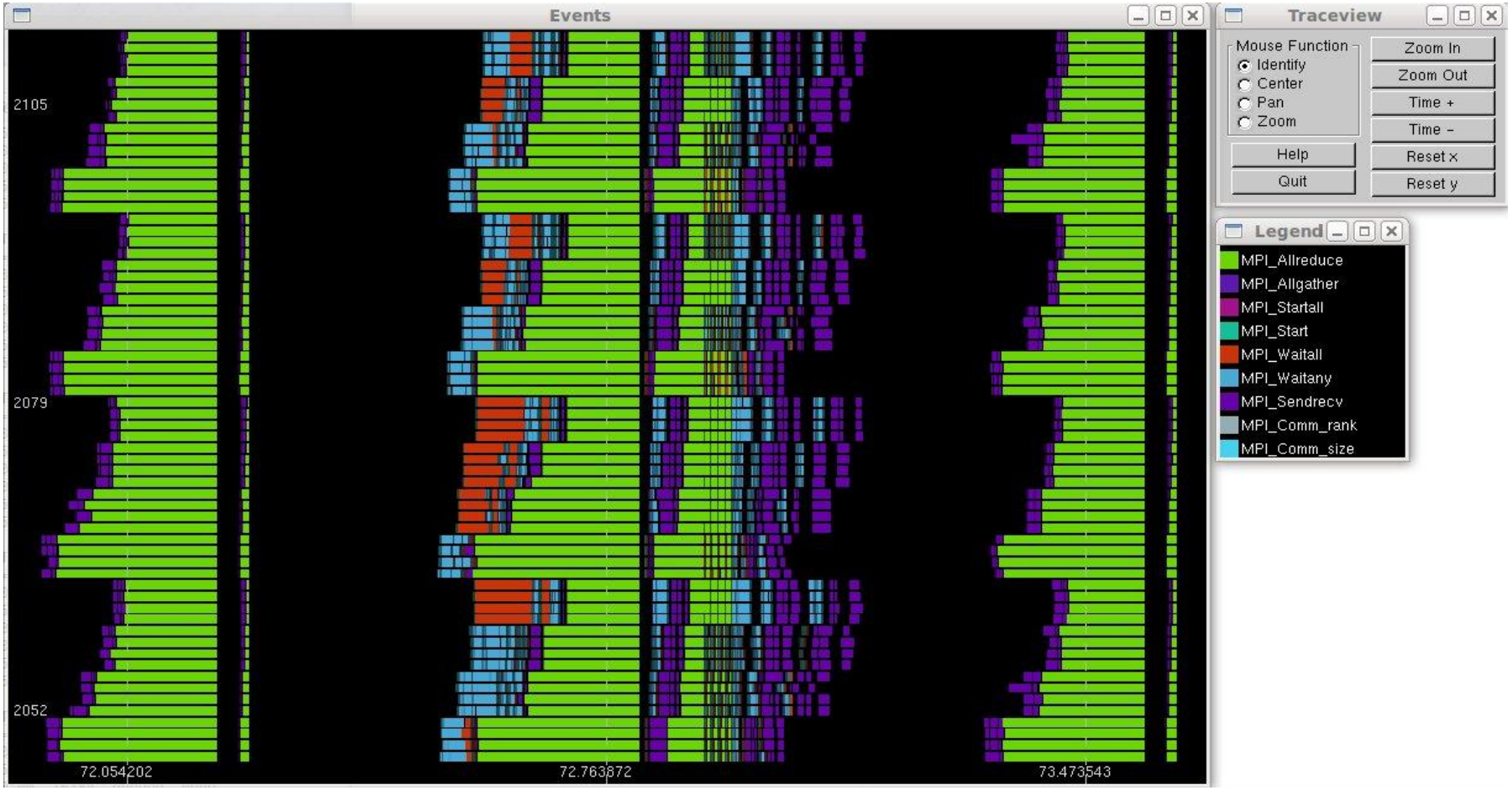
Load Imbalance

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.

```
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
```

Sequoia SPHOT Benchmark

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.

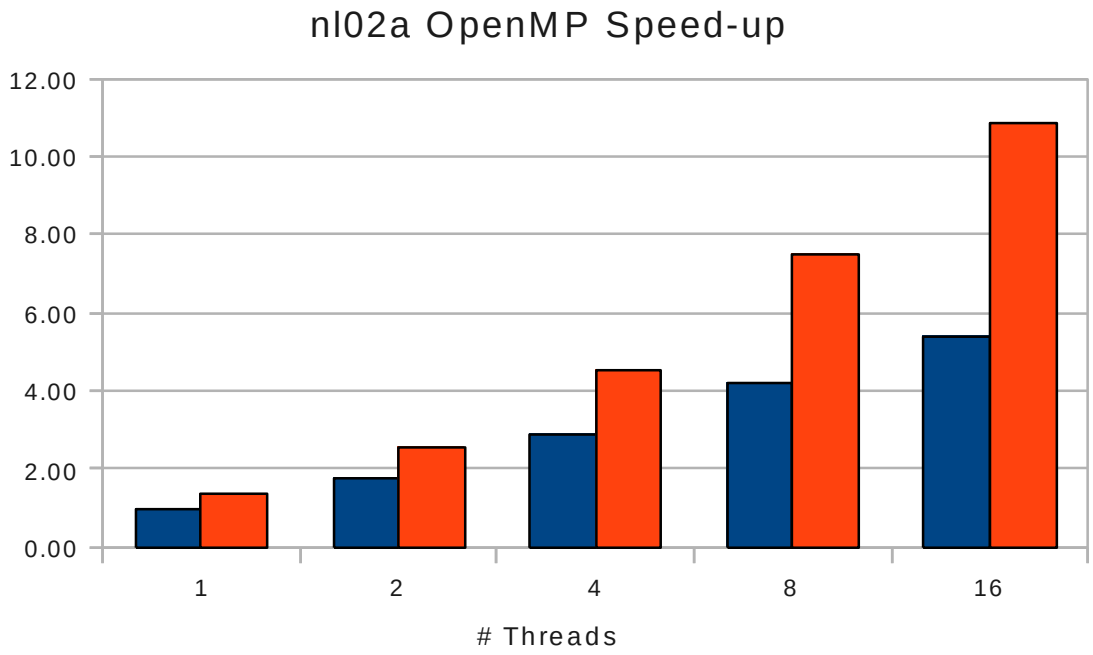
Applications with excellent locality, no global data structures, have an advantage when it comes to scaling to millions of processes.

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
  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

```
!$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
  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.

BGPM – Blue Gene Performance Monitor

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

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

Instruction mix is dominated by integer, load, store, branch operations.

SPHOT : Speed-up using multiple threads per core

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

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

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

Roughly 2:1 ratio of integer/load/store/branch operations to floating-point.

GTC : Speed-up using multiple threads per core.

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

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

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

More than 2:1 ratio of integer/load/store/branch instructions to floating-point.

LAMMPS : Speed-up using multiple threads per core

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

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

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

Closer balance for the two execution units, but still more Int/Ld/St/Br.

Held-Suarez : Speed-up using multiple threads per core.

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

Get ~2.18x speeed-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

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

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

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

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

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

Good QPX code generation by the compiler; integer, load, store, branch instructions dominate the mix.

UMT : Speed-up using multiple threads per core

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

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

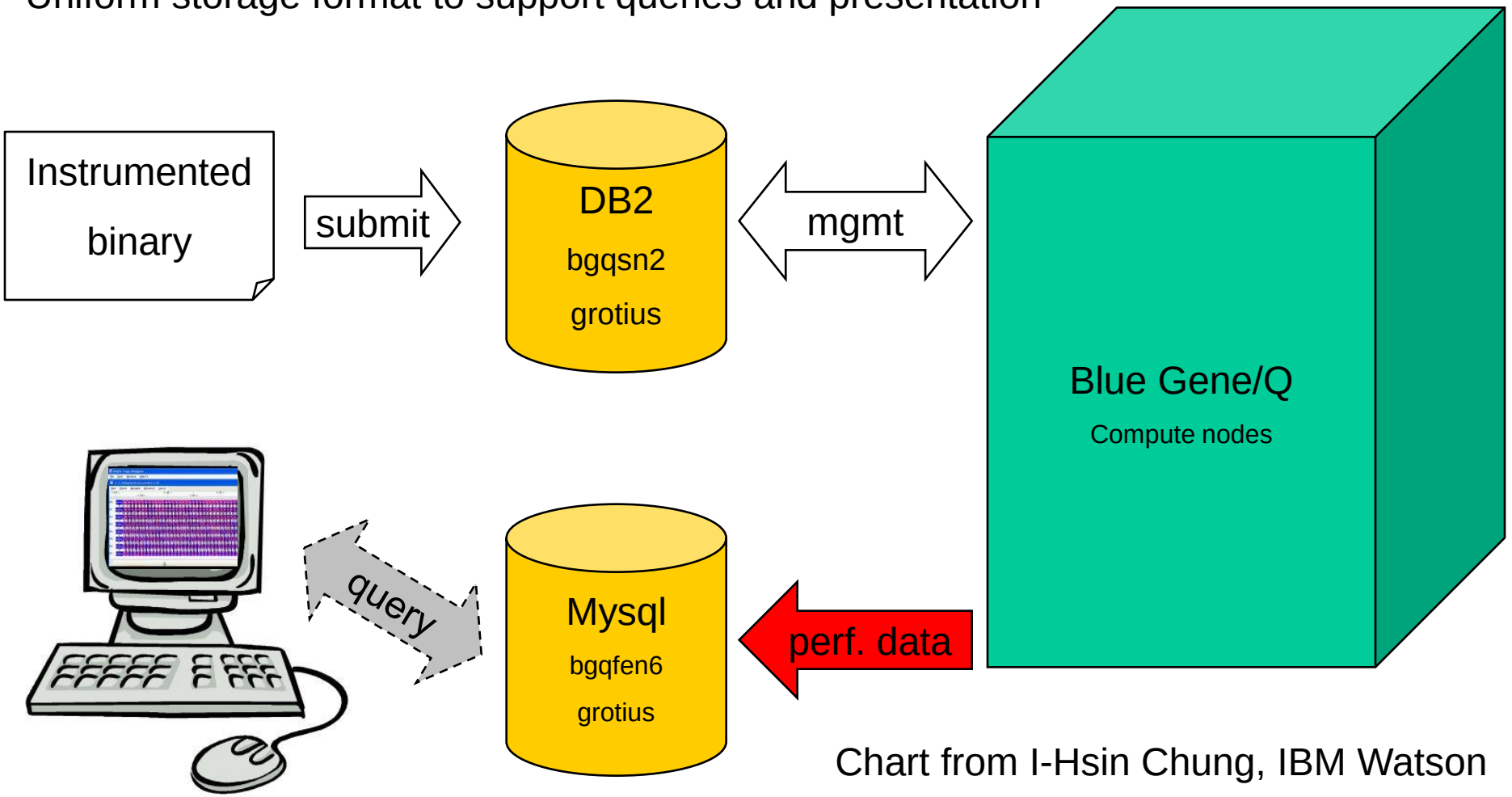


Chart from I-Hsin Chung, IBM Watson

Average Application Characteristics vs. Key Benchmarks

	%FXU	%FPU	%Max Flops	%DDR BW	IPC
App AVG	70.5	29.5	5.7	40.7	0.56
Linpack	43.8	56.2	74.9	22.6	1.34
Graph 500	100.0	0.0	0.0	75.9	0.34

Example algorithms:

sparse matrix-vector multiplication : 80% Int/Ld/St/Br 20% FPU

array update y(:) = a*x(:) + y(:) 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