Blue Gene/Q
Power-Efficient Parallel Computation

Bob Walkup (walkup@us.ibm.com)
IBM Research, Yorktown Heights NY

Hardware Highlights

Programming Strategies

Optimization Resources

Performance Tools

Special Features

BG/Q Hardware Highlights

16 “A2” cores on each node, 1.6 GHz, 4 hardware threads/core
16 GB memory on each node, evenly partitioned at boot time
5D torus integrated network, 20 links/node, 2 Gbits/sec per link

Simple low-power cores, key feature is 4 hardware threads per core.
In-order execution, no instruction-level parallelism.
One execution unit (XU) for integer, load, store, branch, etc.
One auxiliary execution unit (AXU) for floating-point operations.
Support for 4-way SIMD instructions … QPX

Max issue rate is one instruction/cycle when there is one thread/core,
and one instruction/cycle from each of XU and AXU for two or more threads/core.

Key idea: use multiple threads per core to maximize instruction throughput.
Can effectively fill in cycles that would be stalled due to memory access
or due to pipeline dependencies.

BG/Q Power Efficiency

On BG/Q power efficiency comes from squeezing as many instructions as possible through each low-power core by using multiple threads per core.

Performance per thread is very low relative to current high-end cores, but you have a huge number of threads (~10^6) available.

For example, IBM Power7 performance per core is about ~3.6x faster, and performance per thread is ~4x - >10x faster, but BG/Q has higher Flops/Watt

BG/Q is tops on the green 500 list, based on Linpack, and is in practice more efficient in terms of application throughput per Watt for a wide range of applications.

Low performance per thread limits time to solution unless the application supports a large number of threads and/or processes.

BG/Q Conventional Programming Methods

Most common parallel methods : MPI optionally with OpenMP or Pthreads.

The choice may be determined by the memory requirement per process:
The compute-node kernel (CNK) partitions memory “evenly” at boot time.

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%

Results are with BG_MAPALIGN16=1 for 16-64 ranks/node using OMP_STACKSIZE=1M and a small BG/Q partition, and a small program text region. The memory cost can grow with a larger number of processes. Data from Sept 30, 2012.

Programming Methods
Mira Science Benchmarks

MILC	MPI only	32 ranks/node	2 threads/core
GFMC	MPI + OpenMP	8 ranks/node	2 threads/core
NEK	MPI only	32 ranks/node	2 threads/core
GTC	MPI + OpenMP	16 ranks/node	4 threads/core
LS3DF	MPI + OpenMP	32 ranks/node	4 threads/core
GFDL	MPI + OpenMP	8 ranks/node	4 threads/core
DNS	MPI only	32 ranks/node	2 threads/core
FLASH	MPI + OpenMP	16 ranks/node	4 threads/core
NAMD	PAMI + Pthreads	4 ranks/node	4 threads/core
GPAW	MPI + ESSLSMP	32 ranks/node	4 threads/core

Threading was added to FLASH and NAMD, and OpenMP was enabled for GTC in order to make better use of BG/Q resources. MILC, NEK, and DNS distribute the memory requirement and scale well without requiring threading.

BG/Q Hardware Counter Data
Mira Science Benchmarks - Instruction Mix

Code	thds/core	FPU%	FXU%	FP ops/inst	Issue rate	%max rate	GFlops/node
MILC	2	43.3	56.7	1.44	0.63	35.6	10.1
GFMC	2	30.1	69.9	3.16	0.29	20.2	7.0
NEK	2	22.4	77.6	3.32	0.50	39.1	9.6
GTC	4	32.0	68.0	1.42	0.69	47.2	8.1
LS3DF	4	49.2	50.9	6.25	0.51	25.9	40.1
GFDL	4	40.3	59.7	1.29	0.89	52.8	11.8
DNS	2	19.7	80.3	1.32	0.59	47.6	3.9
FLASH	4	20.7	79.3	1.45	0.45	35.8	3.5
NAMD	2	20.1	79.9	2.25	0.48	38.6	5.6
GPAW	4	17.1	82.9	6.79	0.60	49.9	17.9
AVG	**3**	**29.5**	**70.5**	**2.87**	**0.56**	**39.3**	**11.8**

These codes use 2-4 hardware threads per core and achieve an average of roughly ~40% of the maximum possible issue rate from each core. Five of the ten applications have FP ops/inst > 2, indicating significant SIMD instructions. Notice that GFlops/node is not strongly correlated with instruction issue rate.

BG/Q Hardware Counter Data
Mira Science Benchmarks – Cache and Memory

Code	L1%	L1P%	L2%	DDR%	LD B/cycle	ST B/cycle	TOT B/cycle
MILC	94.8	3.8	0.3	1.2	8.4	2.6	11.0
GFMC	72.9	22.4	3.0	1.8	7.1	3.3	10.4
NEK	91.1	6.1	2.1	0.8	4.6	1.9	6.5
GTC	94.4	1.5	3.0	1.2	5.6	2.7	8.3
LS3DF	81.3	15.1	0.8	2.8	8.4	3.8	12.2
GFDL	92.7	5.3	0.7	1.2	6.4	4.8	11.3
DNS	94.5	2.6	1.8	1.1	2.6	2.5	5.1
FLASH	91.2	1.4	6.7	0.7	1.7	1.6	3.4
NAMD	89.2	0.0	10.9	0.0	0.2	0.2	0.4
GPAW	91.9	5.7	1.9	0.5	3.8	1.0	4.8
AVG	**89.4**	**6.4**	**3.1**	**1.1**	**4.9**	**2.4**	**7.3**

Four of the ten applications have an average requirement of >10 Bytes/cycle for bandwidth to memory … the hardware limit is ~18 Bytes/cycle per node. Several codes have a relatively low hit rate in the L1 Data-Cache, but benefit from the prefetch buffer, L1P. NAMD is an exception … low memory bandwidth requirement but high percentage of hits in the L2 cache.

BG/Q Resources for Developers

IBM XL compilers have good documentation; the normal install path is:
/opt/ibmcmp/xlf/bg/14.1/doc/en_US/pdf/*.pdf
/opt/ibmcmp/vacpp/bg/12.1/doc/en_US/pdf/*.pdf

BG/Q vector intrinsics are documented in the Fortran language reference and the C/C++ compiler guide.

Special considerations for the OpenMP runtime for BG/Q are described in the compiler's Optimization and Programming Guide … see above directories.

Many details are covered in the BG/Q Application Development Redbook:

www.redbooks.ibm.com
IBM System Blue Gene Solution: Blue Gene/Q Application Development

Performance Tools

Disclaimer … I tend to use my own tools … many others have been ported to BlueGene/Q. Many tools use the MPI profiling interface, and are therefore limited to MPI applications. Some other tools have a wider range of applicability.

Key capabilities … all must work at the largest scale :

statement-level profiling at scale … clock tics ties to source lines
used to identify hot-spots and specific code blocks

hardware counters … used to provide information on resource utilization for cores and the memory hierarchy

MPI timing data … used to check time spent in MPI and to get a picture of load balance

Statement-Level Profiling - FLASH example

```
tics| Source
   6| do i=i0-2,imax+2
    |   !! ============ x-direction ========================
    |   ! YZ cross derivatives for X states
1446|   call  upwindTransverseFlux&
    |        (hy_transOrder,sig(DIR_Z,:,i,j-2:j+2,k),
    |         lambda(DIR_Y,i,j,k,:),leig(DIR_Y,i,j,k,:,:),&
    |         reig(DIR_Y,i,j,k,:,:),TransFluxYZ(:))
    |
    |   ! ZY cross derivatives for X states
1147|   call upwindTransverseFlux&
    |        (hy_transOrder,sig(DIR_Y,:,i,j,k-2:k+2),
    |         lambda(DIR_Z,i,j,k,:),leig(DIR_Z,i,j,k,:,:),&
    |         reig(DIR_Z,i,j,k,:,:),TransFluxZY(:))
     ...
```

In this case there is huge overhead in making the function calls, because it requires copying non-contiguous array sections. It might be better to order the arrays differently to eliminate copies with bad-stride.

Each “tic” corresponds to 0.01 sec of cpu time.

MPI Timing Data - GFMC

```
total elapsed time       = 1328.197 seconds.
Communication summary for all tasks:

minimum communication time = 45.996 sec for task 2074
median  communication time = 77.764 sec for task 5622
maximum communication time = 1275.763 sec for task 0

Rank 8054 had the largest heap memory used    : 1256.67 MBytes

Histogram of times spent in MPI
    time-bin    #ranks
      45.996      7361
     133.837       766
     221.677         0
     309.518         0
     397.358         0
     485.199         0
     573.039         0
     660.880         0
     748.720         0
     836.561        64
     924.401         0
    1012.241         0
    1100.082         0
    1187.922         0
    1275.763         1
```

The histogram shows that the vast majority of workers spend a very small fraction of time in MPI, so overall parallel efficiency is high.

If there were not enough manager processes, one would see the workers waiting in MPI instead of working.

BG/Q List Prefetch Feature

BG/Q has an API to record a sequence of miss addresses, and prefetch that sequence upon playback. This requires rather reproducible sequences of miss addresses, and in practice is beneficial mainly when there are no available methods to make effective use of the multiple hardware threads.

Threads Per Core	Time(sec) List disabled	Time(sec) List enabled	L1P Misses List disabled	L1P Misses List enabled
1	52.33	32.19	23.4E9	0.32E9
4	21.18	28.65	26.1E9	10.03E9

Data for NAS Parallel Benchmark CG, class B 16 MPI ranks, 1 node.

The list was recorded every 4 CG iterations, and played back each iteration. The overall speed-up was 1.62 when using one thread per core. However, adding OpenMP to engage all four hardware threads gave a further improvement in performance.

List Prefetch – CG Example Code

Use hardware counters to measure L1 D-cache misses, and use that to configure the size of the list prefetch buffer.

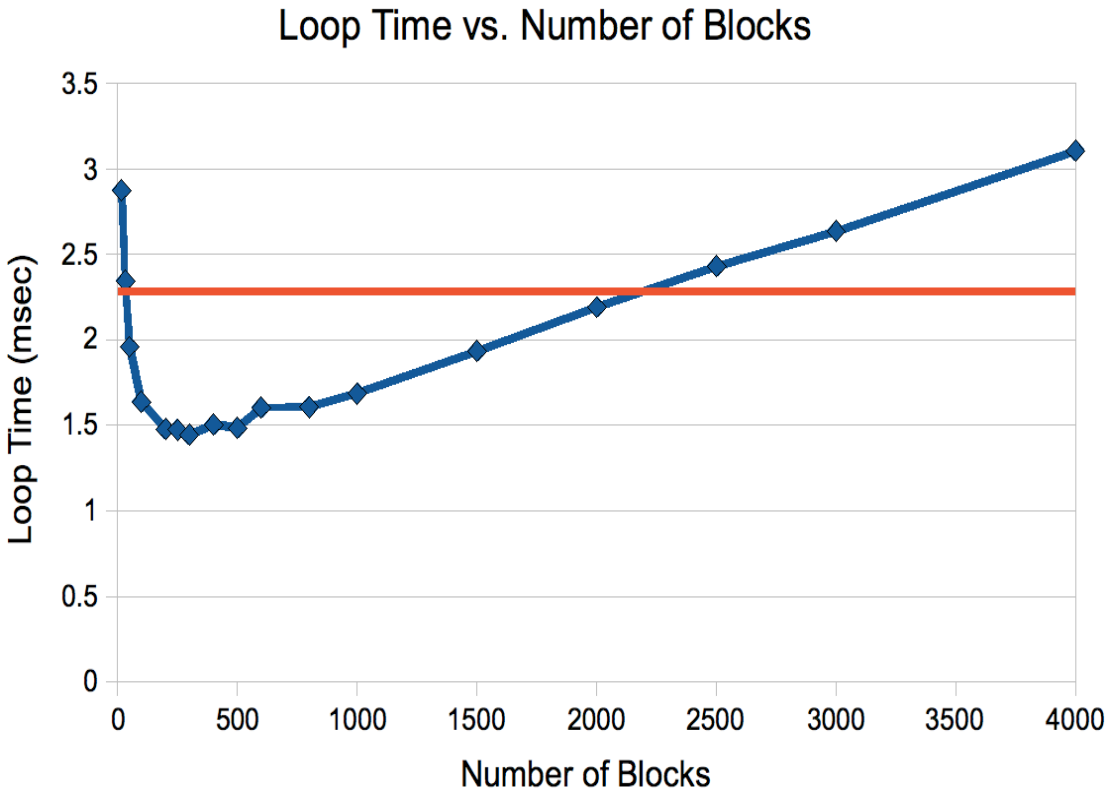
```
call l1p_patternconfigure(1200000)
…
do cgit = 1, cgitmax
   call l1p_patternstart(1)     ! Record/playback each iter
...  The usual cg iteration work …
   call l1p_patternstop()
end do
```

This example uses Fortran wrappers for the L1P API, defined in :

/bgsys/drivers/ppcfloor/spi/include/l1p/pprefetch.h

If using OpenMP or Pthreads, each thread must configure, start, and stop recording the miss pattern.

BG/Q Transactional Memory



The XL compiler has support for

#pragma tm_atomic

to mark a transaction region for atomic updates.

Use option –qtm along with –qsmp.

```
for (face = 0; face < nfaces; face++) {
    ii = ii_list[face];
    jj = jj_list[face];
    y[ii] += A[face] * x[jj];
    y[jj] += A[face] * x[ii];
 }
```

For effective use of TM, one must add loop-blocking and adjust the size of the transaction region.

Transactional Memory Code Example

```
#pragma omp parallel for private(block,fbeg,fend,face,ii,jj) schedule(static)
  for (block = 0; block < numblocks; block++) {
    if (block < leftover) {
      fbeg = block*(blocksize + 1);
      fend = fbeg + blocksize + 1;
    }
    else {
      fbeg = leftover + block*blocksize;
      fend = fbeg + blocksize;
    }
    #pragma tm_atomic
    {
       for (face=fbeg; face<fend; face++) {
          ii = ii_list[face];
          jj = jj_list[face];
          y[ii] += A[face] * x[jj];
          y[jj] += A[face] * x[ii];
       }
    }
  }
```

Adjust the size of the transaction region to optimize performance.

OpenMP is used for thread management.