BlueGene/Q Optimization

Bob Walkup
walkup@us.ibm.com
914-945-1512

Performance : what to expect

Using IBM XL compilers

A case study from SC13

Aternative compilers for BGQ

Libraries that can help performance

OpenMP and MPI

Some Useful Properties of BG/Q

You normally get dedicated resources for computation and communication (not counting the file-system).

Memory access is uniform … no worrying about NUMA.

Processes and threads are bound by CNK (compute node kernel).

No context switches => very low "system noise" => measure once.

Nice set of hardware counters => can really see what the hardware did.

Very strong network relative to compute performance => can scale.

Downside is very low performance per-core => must scale.

Rough Comparison : BGQ vs. Intel Sandy-Bridge

Intel Sandy-Bridge : max issue rate is **4** instr/cycle per core (2.6 GHz).

BGQ (A2 core) : at one thread/core, max issue rate is **1** instr/cycle per core.
with two or more threads, max issue rate is **2** instr/cycle per core
==> one instr/cycle from each of the two execution units (1.6 GHz) <==

	1 Node BGQ	4 Nodes BGQ	1 Node Sandy-Bridge
power consumption	~80 W	~320 W	~320 W
Memory bandwidth	~30 GB/sec	~120 GB/sec	~80 GB/sec
Number of cores	16	64	16
Hardware threads	64	256	32
Total max issue rate	51.2 G instr/sec	204.8 G instr/s	166.4 G instr/sec
Max DP Flops	204.8 GFlops	819.2 GFlops	332.8 GFlops

For typical apps, expect ~3-4 nodes BGQ = 1 node Intel SB in performance, or roughly a 3:1 to 4:1 ratio of performance per core for Intel SB: BGQ.

But … for equivalent performance, you will need 8x to 16x more threads on BGQ.

BGQ has simple low-power in-order cores. You normally need at least two threads per core for good utilization of the compute resources.

On BGQ : Use multiple hardware threads per core.

There are four hardware threads per core.

Each hardware thread has its own set of registers. User threads are bound to hardware threads. Instructions are dispatched to hardware threads in a balanced way.

The L1 D Cache and prefetch buffer are shared by the hardware threads, but they are private per-core.

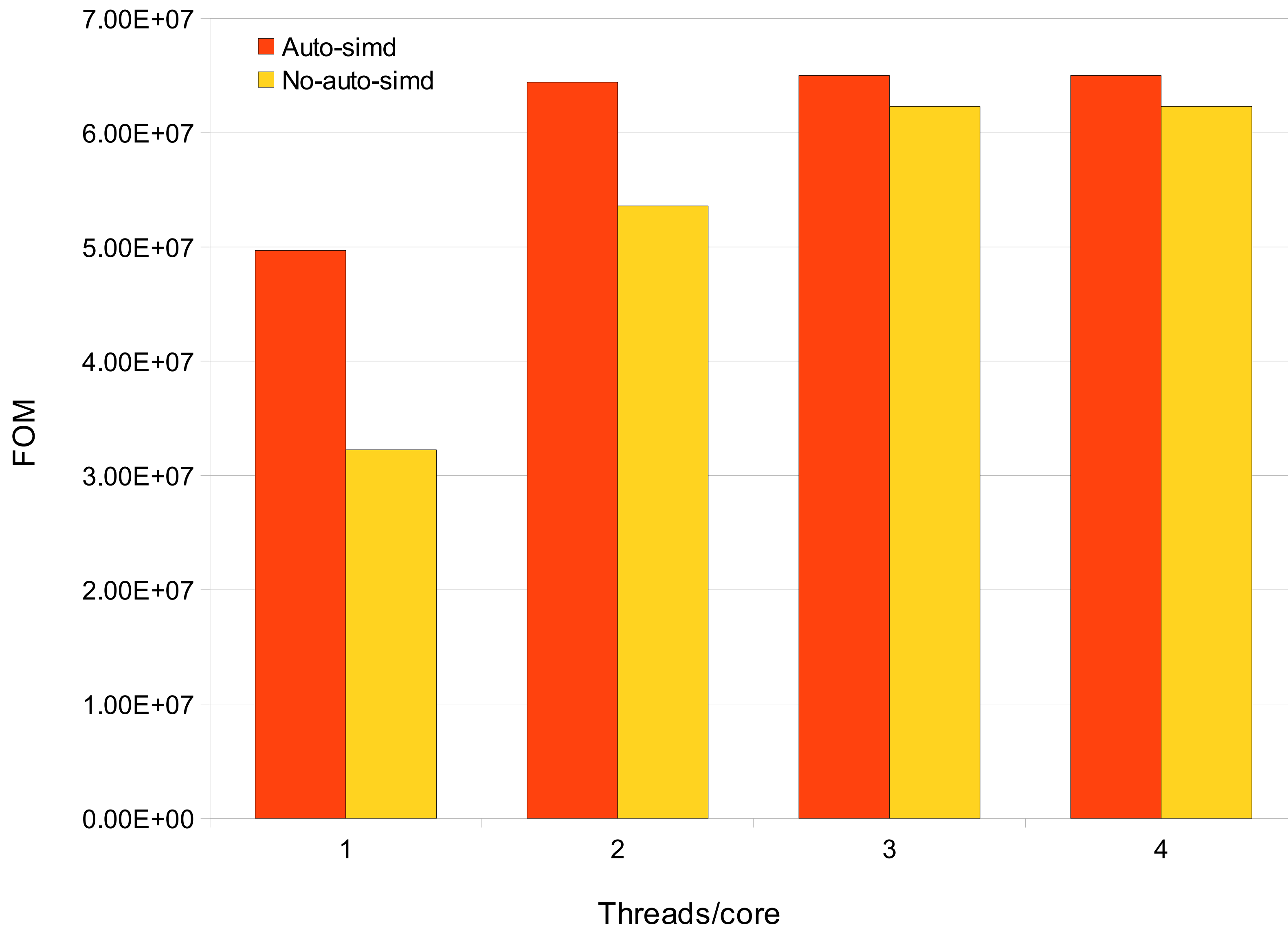
There is just one execution pipeline for integer/load/store/branch instructions, and a separate pipeline for floating-point instructions per core.

Multiple threads/core is your best bet for increased instruction throughput and for hiding latency.

Floating-point pipeline latency : 6 cycles

Latency to L1 D Cache : 6 cycles
Latency to the L1 Prefetch buffer : ~24 cycles
Latency to L2 : ~82 cycles
Latency to memory : ~350 cycles

See: www.redbooks.ibm.com ; IBM System Blue Gene Solution Blue Gene/Q Application Development (SG24-7948-01).


UMT 1-node Figure of Merit
Auto-simd
No-auto-simd
FOM
7.00E+07
6.00E+07
5.00E+07
4.00E+07
3.00E+07
2.00E+07
1.00E+07
0.00E+00
1
2
3
4
Threads/core

Can use “just MPI” or a mix of MPI + OpenMP and/or pthreads

Each node has 16 GB memory and 16 cores : 1 GB/core

Can use 1, 2, 4, 8, 16, 32, or 64 MPI ranks per node.

Memory is partitioned by the compute-node kernel based on #ranks/node.

Memory available per rank is quite limited when you have 32 or 64 ranks/node.

To check memory utilization : Kernel_GetMemorySize();

See: /bgsys/drivers/ppcfloor/spi/include/kernel/memory.h

Can also call getrusage(), but Kernel_GetMemorySize() provides more data.

The MPI-only approach can be very effective on BGQ if you can fit in the available memory … but I normally recommend using some form of threading, because that gives you more flexibility in how to use the resources.

Using IBM XL Compilers

Use profiling tools to identify hot spots, and look very carefully at the code generated by the compiler : add options -qlist -qsource and maybe -qreport.

XL compiler defaults : no optimization, auto-simd enabled, architecture set for BGQ

-O2 : a good place to start … add -qmaxmem=512000 (memory for the optimizer)
-O2 -qnostrict : allows some more aggressive optimizations
-O3 -qstrict : (-O3 implies -qhot=fastmath:level=0)
-O3   (-O3 implies -qhot=fastmath:level=0)
-O3 -qnohot   (turns off the high-order transformation step)
-O3 -qhot (same as adding -qhot=level=1)   hot = high-order transformations
-O4 = -O3 -qhot=level=1 -qipa=level=1   ipa = interprocedural analysis
-O5 = -O3 -qhot=level=2 -qipa=level=2

Default is -qsimd=auto.   Sometimes it can be better to have -qsimd=noauto.

For automatic SIMDization, there is some capability, but many issues limit what the compiler can/will do. Add -qreport and read the listing file for some insight.

I almost always add -g to preserve basic debug info.

Documentation : /soft/compilers/ibmcmp-aug2013/xlf/bg/14.1/doc/en_US/pdf
                        /soft/compilers/ibmcmp-aug2013/vacpp/bg/12.1/doc/en_US/pdf

Example : array update – single precision

```
void saxpy(int n, float alpha, float * x, float * y)
{
   int i;
   for (i=0; i<n; i++) y[i] = alpha*x[i] + y[i];
}
```

bgxlc_r -c -g -O2 -qlist -qsource -qreport => scalar (not SIMD) code, not effectively pipelined … read the listing file.

bgxlc_r -c -g -O3 -qlist -qsource -qreport => get SIMD code with loop versioning. One version for co-aligned arrays, and another with permute instructions, not effectively pipelined.

Add the “restrict” keyword : enables the compiler to unroll the loops and do better instruction pipelining … necessary to indicate that there is no pointer aliasing.

```
void saxpy(int n, float alpha, float * restrict x, float * restrict y)
```

Auto-SIMDization capability is in TPO … needs -O3 or -qhot and -qnostrict to enable.

-qreport => has information about loop transformations and SIMDization.

MASS Library for Math Intrinsic Functions

function	simd mass	scalar mass	libm.a
exp	61	103	395
log	83	118	602
sin	37	140	372
cos	37	136	385
tan	56	166	582
atan	76	200	413
sinh	71	115	555
cosh	59	112	507
atan2	74	229	770
pow	153	220	996

approx. cycles per eval.

The XL compiler can generate calls to its internal MASS library at -O3, and can call SIMD versions of exp(), log() …. vector4double arguments … in loops that are SIMDizable. Can check listings to see the version used by the compiler.

If you add “-lmass” before “-lm”, you should get the MASS library version. You can also use “-lmass” with GNU compilers. SIMD version is available : expd4().

Argonne systems : /soft/compilers/ibmcmp-aug2013/xlmass/bg/7.3/bglib64/

Special Considerations for 1/x, 1/sqrt(x), sqrt(x)

BGQ has fdiv and fsqrt instructions , ~30 cycles per evaluation, not pipelined.

However, XL C/C++ at -O2 will generate a library call for sqrt(x) => ~100 cycles.
XL Fortran will generate fsqrt instruction at -O2 or higher.

BGQ also has scalar and SIMD estimate instructions for 1/x, 1/sqrt(x). Those are pilelined and can provide very high throughput. Estimate instruction is good to 14 bits, and can be used with Newton's method to compute accurate 1/x, 1/sqrt(x), sqrt(x). Compiler may generate these at -O3 … but you can force it :

-O3 -qdebug=recipf:forcesqrt (not documented … but can be useful)

Example: pipelined SIMD estimate method for 1/x can reduce the cost to < 2 cycles per scalar evaluation for doubles.

BGQ Vector Intrinsics

BGQ has SIMD capability = instructions to process four doubles or floats.

All arithmetic work is done with doubles. QPX mult-add = 8 Flops/cycle.

Vector load and store operations can work with doubles or floats, and include automatic format conversion to/from float. Format is double in registers.

Vector intrinsics are provided by the XL compiler to make it possible for you to explicitly code SIMD instructions. Each vector intrinsic maps to one QPX instruction, callable from C, C++, or Fortran … also in bgclang.

Documentation in IBM XL compiler manuals :

> C/C++ Compiler Reference
> /opt/ibmcmp/vacpp/bg/12.1/doc/en_US/pdf/compiler.pdf
>
> Fortran Language Reference
> /opt/ibmcmp/xlf/bg/14.1/doc/en_US/pdf/langref.pdf

On Argonne systems, look in /soft/compilers instead of /opt.

http://pic.dhe.ibm.com/infocenter/compbg/v121v141/index.jsp

C example with load, store, and mult-add operations : doubles

Note: the vector load, vec_ld(0, &x[i]), will grab four contiguous doubles starting at the specified address rounded down to the nearest 32-byte boundary. It is absolutely critical to check or know the alignment.

```
//--------------------------------------------------
// vector version of y[i] = a*x[i] + y[i]
// where "x" and "y" are 32-byte aligned
//--------------------------------------------------
#include <stdio.h>    // C++ codes must #include <builtins.h>
#define NPTS 8
static double __attribute__((aligned(32))) x[NPTS], y[NPTS];

int main(int argc, char * argv[])
{
   int i;
   double a = 2.0;
   vector4double av, xv, yv;
   for (i=0; i<NPTS; i++) {
      x[i] = (double) i;
      y[i] = (double) i + 1;
   }
   if ((long) x & 0x1F ) printf("x is not 32-byte aligned\n");
   if ((long) y & 0x1F)  printf("y is not 32-byte aligned\n");
   av = vec_splats(a);            // replicate "a" in four vector slots
   for (i=0; i<NPTS; i+=4) {
      xv = vec_ld(0, &x[i]);      // load four contiguous elements of x[]
      yv = vec_ld(0, &y[i]);      // load four contiguous elements of y[]
      yv = vec_madd(av, xv, yv);  // yv = av*xv + yv
      vec_st(yv, 0, &y[i]);       // store four contiguous elements of y[]
   }
   for (i=0; i<NPTS; i++) printf("y[%d] = %.1lf\n", i, y[i]);
   return 0;
}
```

Data alignment is the number one concern.

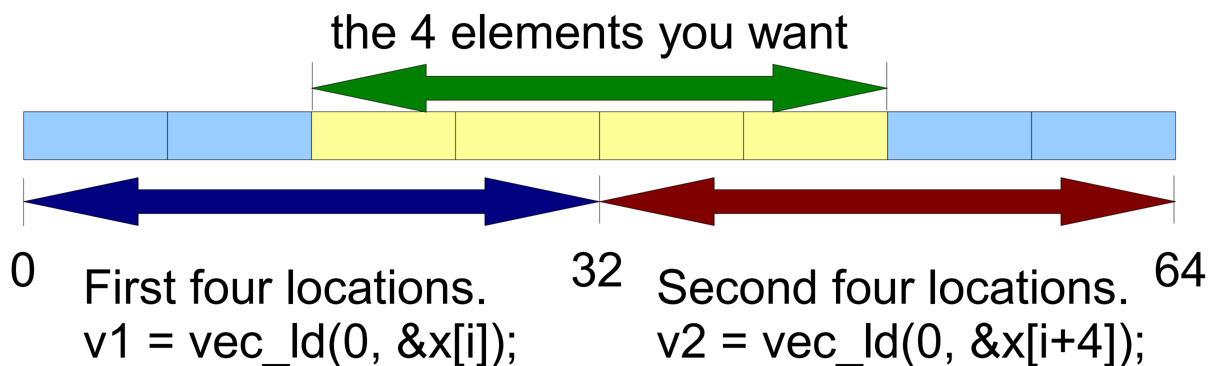
vec_ld() will always grab four contiguous items starting at the specified address rounded down to the nearest 32/16-byte boundary (doubles/floats).

vec_st() will always store four contiguous items starting at the specified address rounded down to the nearest 32/16 byte boundary (doubles/floats).

If there is any doubt about alignment, check the address :

```
C/C++ :  int misaligned = ((long) &x[i]) & 0x1F ;      // 0xF  for floats
Fortran : integer misaligned = iand(loc(x(i)), z'1F')  !  z'F'   for floats
```

To work with misaligned data, you have to load two "quads" and then select the appropriate four elements. There are several ways to do this, using shift or permute instructions.



Vector Permute Instructions for Mis-aligned Data

To handle misaligned data at runtime, use : pctl = vec_lvsl(0, &x[i]) :

```
double * x;
vector4double v1, v2, xv, pctl;
v1 = vec_ld(0, &x[i]);            // load 4 doubles at &x[i] rounded down to n*32
v2 = vec_ld(0, &x[i+4]);          // load 4 doubles at &x[i+4] rounded down to n*32
pctl = vec_lvsl(0, &x[i]);        // compute the permute control vector
xv = vec_perm(v1, v2, pctl);  // xv contains x[i], x[i+1], x[i+2], x[i+3]
```

Limitation : the maximum left shift is 3 … useful for loops.

General permutation extracts any four slots from the concatenation of two vectors :

```
double * x;
vector4double v1, v2, xv, pctl;
v1 = vec_ld(0, &x[i]);            // load 4 doubles at &x[i] rounded down to n*32
v2 = vec_ld(0, &x[i+4]);          // load 4 doubles at &x[i+4] rounded down to n*32
pctl = vec_gpci(0abcd);           // compute the permute control vector
xv = vec_perm(v1, v2, pctl);  // xv contains data from slots a,b,c,d
```

Note : a,b,c,d must be numbers in the range 0-7; leading zero for octal constant.
Fortran version : xv = vec_perm(o'abcd') … leading “o” for octal constant.

Case Study : SC13 Gordon Bell Paper

Diego Rossinelli, Babak Hejazialhosseini, Panagiotis Hadjidoukas, Costas Bekas, Alessandro Curioni, Adam Bertsch, Scott Futral, Steffen J. Schmidt, Nikolaus A. Adams, and Petros Koumoutsakos. 2013. 11 PFLOP/s simulations of cloud cavitation collapse. In Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis (SC '13). ACM, New York, NY, USA, , Article 3 , 13 pages. DOI=10.1145/2503210.2504565 http://doi.acm.org/10.1145/2503210.2504565

Got to 14.4 PFlops on Sequoia (20 PF peak)

https://github.com/cselab/CUBISM-MPCF Code repository

Highlights: finite volume with high-order discretization.

Extensive data re-ordering with data structures designed for cache and SIMD.

C++ code with extensive use of vector intrinsics for all key kernels.

Conversion between array-of-struct/struct-of-arrays as needed.

MPI + OpenMP with dynamic scheduling.

SIMD wavelet-based compression for data dumps.

Alternative Compilers for Blue Gene/Q

BGQ system software includes GNU 4.4.6 : powerpc64-bgq-linux-gcc, etc.
Try “-O3 -funroll-loops”. Experience has been that performance is often a bit lower than with XL compilers.

BGQ system software includes patch files for GNU 4.7.2 toolchain.
Try “-Ofast -funroll-loops”. Roughly comparable performance to XL compilers with “-O3 -qsimd=noauto” for C/C++. Has improved overall support for powerpc64 and BGQ, including -mcpu=a2 for the BGQ cores.
No built-in support for auto-simd or vector intrinsics.

On Argonne systems : bgclang. Includes support for auto-simd and vector intrinsics. http://www.alcf.anl.gov/user-guides/bgclang-compiler

IBM ESSL Library : BLAS, FFTs, etc.

Argonne systems : /soft/libraries/essl/current/lib64
    libesslbg.a : non-threaded version
    libesslsmpbg.a : has some threaded routines, uses XL OpenMP for threading
    Must also link with the threaded XL Fortran lib :
        /soft/compilers/ibmcmp-aug2013/xlf/bg/14.1/bglib64/libxlf90_r.a

Level-3 BLAS (dgemm, zgemm, etc.) : ~85% of peak Flops for large matrices

FFTs can provide a performance advantage over FFTW. Wrappers to use the FFTW version 3 interface with ESSL routines are available, but you must build.

ndim	FFTW-Est	FFTW-Meas	ESSL-FFTW	ESSL-DCFT
16	768	768	178	441
32	1009	1009	268	701
64	956	1184	307	762
128	1144	1144	595	1197
256	1097	1138	1022	1753
512	902	934	1226	1628
1024	693	679	1612	1826
2048	512	704	1643	1712
4096	563	734	1873	1705
8192	510	688	1744	1720

MFlops

Google tcmalloc library : libtcmalloc_minimal.a

malloc/free can be a significant performance problem, particularly for C++ codes that frequently use new/delete for small objects

Can link with libtcmalloc_minimal.a : no code change is required.

TC malloc = thread cache malloc … designed for multi-threaded use.

Included in Google performance tools … Google it !

Some C++ users report 20-30% performance improvement by linking.
Not expected to help if you have infrequent large memory allocations.

Also can be useful for memory diagnostics, such as warnings for large mallocs.

OpenMP : some performance considerations

OMP_WAIT_POLICY=active or passive
Experiment with it : active = busy wait ;   passive = yield (after some spins)
Passive policy can be better when there is load imbalance among threads,
Active policy can be better when there are many small parallel regions.

BG_THREADLAYOUT=1 the default, breadth first, round-robin over cores
BG_THREADLAYOUT=2 depth first, packs threads onto each core first.

XL OpenMP runtime has far less overhead than GNU OpenMP runtime.

If you need very low OpenMP overhead, there is LOMP, a light-weight prototype from IBM Research, which is coupled to specifically modified IBM XL compilers.

OpenMP provides flexible work scheduling, but it may be best to specify the work scheduling on the OpenMP directive, instead of using the default runtime scheduling method.

MPI : some performance considerations

On BGQ there are a number of messaging libraries to choose from:

ls /bgsys/drivers/ppcfloor/comm/bin :
gcc gcc.legacy xl xl.legacy xl.legacy.ndebug xl.ndebug

xl/gcc = which compiler was used for the build ; xl = lower latency, so use it
legacy = coarse-grained locks = lower latency
ndebug = no asserts in the messaging code = lower latency

The “legacy” version can provide lower latency, and may be the best choice if you have many MPI ranks per node and have short messages.

The non-legacy version is required for asynchronous progress, and can deliver higher messaging rates when there are only a few MPI ranks per node.

The “ndebug” versions have lower latency, but won't indicate internal errors.

If you encounter messsaging errors, set PAMID_VERBOSE=1, and run with a library that has asserts … not an “ndebug” version.

Use non-blocking pt-2-pt, post revcs early, use MPI_Waitall.

On BGQ, MPI_Allreduce is faster than MPI_Reduce.