Performance Tuning for BGQ : Case Studies

Bob Walkup (walkup@us.ibm.com)
IBM Watson Research Center

HPCG benchmark

- 27-point stencil, CG method, expect memory bandwidth limitation
- Symmetric Gauss-Seidel smoother
- Sparse matrix-vector multiplication
- Vector operations : daxpy, dot-products, norms

WRF 3.6.1 2.5 km Continental US problem

- Mix of many atmospheric dynamics and physics routines
- Choice of MPI + OpenMP for parallelization
- Compiler options and math libraries are important
- Limited potential for source-code changes
- I/O options can make a huge difference

HPCG-2.4 First Look

Start with MPI-only approach; compile and run a test problem 1 MPI rank/core

GFLOP/s Summary: (128 node partition, 1 rank/core, original code)
Raw DDOT: 73.9718
Raw WAXPBY: 322.85
Raw SpMV: 231.579
Raw MG: 182.727
Raw Total: 185.358
HPCG result is VALID with a GFLOP/s rating of: 189.141 => 1.48 GFlops/node

The multi-grid smoother is symmetric Gauss-Seidel, and has the same structure as sparse matrix-vector multiplication. HPCG spends most of the time in the MG smoother plus SpMV routines.

Sum += matrixValue[j] * vector[indx[j]]; 2 flops for load indx, vector, and matrix
Structured 27-point stencil : expect to get some re-use of vector data, need to load ~15 to 20 Bytes for every 2 flops => Bytes/flop = 7.5 to 10. BGQ has 30 GB/sec bandwidth to memory, so we expect ~3 - 4 GFlops/node for the main routines.

Something is holding performance back.
Suspects : compiler, messaging, issues with the source code.

Brute Force : use more hardware threads per core

MPI and/or OpenMP ?

Only the sparse matrix-vector multiplication routine is threaded.
The smoother has recursion (Gauss-Seidel), not easy to thread.
BGQ has excellent network => use two or four MPI ranks/core.

Ranks/Core	GFlops/node total (not just SpMV)
1	1.48
2	2.13
4	2.01

Expect ~3 - 4 GFlops/node … not getting there by more ranks/core.

Look at MPI cost :

Data using two MPI ranks/core, instrumenting just the CG solver :

```
MPI task 3125 of 4096 had the minimum communication time.
total communication time = 2.843 seconds.
total elapsed time       = 143.452 seconds.
```

Problem is not MPI … something is making computation slow.

HPCG : SpMV code generation

```
67 |    for (int j=0; j< cur_nnz; j++)
68 |      sum += cur_vals[j]*xv[cur_inds[j]];
```

```
 0|                    CL.77:
68| 0001B8 lwa         1     L4A     gr31=(int).rns6.(gr12,16)
68| 0001BC fmadd       1     FMA     fp0=fp0,fp2,fp7,fcr
68| 0001C0 rldicr      2     SLL8    gr11=gr30,3
68| 0001C4 lfdu        1     LFDU    fp2,gr10=(double).rns7.(gr10,32)
68| 0001C8 lfdx        1     LFL     fp7=(double).rns7.(gr3,gr11,0)
 0| 0001CC addi        1     AI      gr12=gr12,16
68| 0001D0 fmadd       1     FMA     fp1=fp1,fp5,fp6,fcr
68| 0001D4 lwa         1     L4A     gr30=(int).rns6.(gr12,4)
68| 0001D8 rldicr      1     SLL8    gr11=gr31,3
68| 0001DC lfd         1     LFL     fp5=(double).rns7.(gr10,8)
68| 0001E0 lfdx        1     LFL     fp6=(double).rns7.(gr3,gr11,0)
68| 0001E4 fmadd       1     FMA     fp4=fp4,fp2,fp7,fcr
68| 0001E8 lwa         1     L4A     gr11=(int).rns6.(gr12,8)
68| 0001EC rldicr      1     SLL8    gr31=gr30,3
68| 0001F0 lfd         1     LFL     fp2=(double).rns7.(gr10,16)
68| 0001F4 lfdx        1     LFL     fp7=(double).rns7.(gr3,gr31,0)
68| 0001F8 lwa         1     L4A     gr30=(int).rns6.(gr12,12)
68| 0001FC fmadd       1     FMA     fp3=fp3,fp5,fp6,fcr
68| 000200 rldicr      2     SLL8    gr11=gr11,3
68| 000204 lfd         1     LFL     fp5=(double).rns7.(gr10,24)
68| 000208 lfdx        1     LFL     fp6=(double).rns7.(gr3,gr11,0)
 0| 00020C bc          1     BCT     ctr=CL.77,taken=100%(100,0)
```

Code is not bad : loads matrix, index, and vector, one fmadd per loop iteration, four-way loop unrolling. XL compilers with -qlist -qsource.

HPCG : Make some measurements

In src/ComputeSPMV_ref.cpp : instrument the SpMV section of code

```
HPM_Start("spmv");
for (local_int_t i=0; i< nrow; i++)  {
  double sum = 0.0;
  const double * const cur_vals = A.matrixValues[i];
  const local_int_t * const cur_inds = A.mtxIndL[i];
  const int cur_nnz = A.nonzerosInRow[i];
  for (int j=0; j< cur_nnz; j++)
    sum += cur_vals[j]*xv[cur_inds[j]];
  yv[i] = sum;
}
HPM_Stop("spmv");
```

Data using 2 MPI ranks/core :

```
Derived metrics for code block "spmv" averaged over process(es) on node <0,2,0,1,1>:
Instruction mix:  FPU = 17.16 %,  FXU = 82.84 %
Instructions per cycle completed per core = 0.2982
Per cent of max issue rate per core = 24.70 %
Total weighted GFlops for this node = 2.196
Loads that hit in L1 d-cache =  91.23 %     <== lower than expected
                  L1P buffer =   4.10 %
                  L2 cache   =   2.72 %
                  DDR        =   1.95 %
DDR traffic for the node: ld = 14.624, st = 0.379, total = 15.003 (Bytes/cycle)
```

Getting more D-cache misses and more demand-loads to memory than expected.
Bytes/flop ~11 is too large. Let's look at the data layout for the sparse matrix.

HPCG : sparse matrix data layout

Code in : GenerateProblem.cpp

```
// Now allocate the arrays pointed to
  for (local_int_t i=0; i< localNumberOfRows; ++i) {
    mtxIndL[i] = new local_int_t[numberOfNonzerosPerRow];  <== bad idea
    matrixValues[i] = new double[numberOfNonzerosPerRow];  <== bad idea
    mtxIndG[i] = new global_int_t[numberOfNonzerosPerRow]; <== bad idea
  }
```

HPCG makes a novice mistake : uses “new/malloc” for every sparse row.
The result is that the matrix is not stored in a contiguous block, and so the hardware prefetch mechanism can't pull in the data before it is needed.

Fix the code :

```
  mtxIndL[0] = new local_int_t[numberOfNonzerosPerRow*localNumberOfRows];
  mtxIndG[0] = new global_int_t[numberOfNonzerosPerRow*localNumberOfRows];
  matrixValues[0] = new double[numberOfNonzerosPerRow*localNumberOfRows];

  // now assign the pointer values to ensure contiguous memory
  for (local_int_t i=1; i< localNumberOfRows; ++i) {
     mtxIndL[i] = mtxIndL[0] + i*numberOfNonzerosPerRow;
     mtxIndG[i] = mtxIndG[0] + i*numberOfNonzerosPerRow;
     matrixValues[i] =  matrixValues[0]   + i*numberOfNonzerosPerRow;
  }
```

Now the matrix values are stored in one contiguous block of memory.

HPCG with contiguous data layout

Repeat measurements using 1, 2, 4 MPI ranks per core :

Ranks/core	Original code	Contiguous memory
1	1.48 GFlops/node	1.82 GFlops/node
2	2.13	2.85
4	2.01	3.21

Now we are getting to the expected performance range : ~3 - 4 GFlops/node.

Some minor improvements are possible, for example one can merge some vector operations and SIMDize other vector operations. The symmetric Gauss-Seidel smoother has a backward sweep, but BGQ prefetch hardware does not detect a backward stream. On can add instructions (dcbt) to help a little with prefetching in the backward direction.

After additional tuning : GFlops/node = 3.53, with near linear scaling.

The most important optimization by far was fixing the data layout so that the data is in a contiguous block, to enable efficient hardware prefetching.

HPCG data after tuning

GFLOP/s Summary: (128 node partition, 2 ranks/core, tuned code)
Raw DDOT: 303.017
Raw WAXPBY: 388.985
Raw SpMV: 426.389
Raw MG: 439.584
Raw Total: 433.518
Total with convergence overhead: 451.581
HPCG result is VALID with a GFLOP/s rating of: 451.581 => 3.53 Gflops/node

```
Derived metrics for code block "spmv" averaged over process(es) on node <0,0,2,1,1>:
Instruction mix:  FPU = 17.60 %,  FXU = 82.40 %
Instructions per cycle completed per core = 0.5148
Per cent of max issue rate per core = 42.43 %
Total weighted GFlops for this node = 3.888
Loads that hit in L1 d-cache =  94.86 %     <== better
                  L1P buffer =   3.65 %
                  L2 cache   =   0.36 %
                  DDR        =   1.13 %
DDR traffic for the node: ld = 14.310, st = 0.580, total = 14.890 (Bytes/cycle)
```

The data for the sparse matrix-vector part is good, and overall performance is OK.

WRF 3.6.1

WRF = Weather Research Framework version 3.6.1 (current)
Test case 2.5 km Continental US (1500x1200x35 grid dimensions)

First challenge : build the code … there is no BGQ recipe, so choose BGP (distributed-memory + shared-memory) as a guide, and make adjustments to the configure.wrf file (that file controls how WRF is built).

Build fails with undefined external for flush; that is flush_() with XL compilers.
Re-compile with -qextname=flush … build succeeds … get wrf.exe.

First job submission : 8 MPI ranks/node and one midplane = 512 nodes
- job fails with SEGV in long-wave radiation
- get SEGV with one to eight OpenMP threads
- SEGV persists even after bumping up OMP_STACKSIZE
- Not a memory limitation … must be a compiler or code problem.
- Core files not helpful in this case; => try less aggressive compiler options

BGP options: FCOPTIM = -O3 -qnoipa -qarch=auto -qcache=auto -qtune=auto
replace with : FCOPTIM = -O3 -qstrict (just a guess)

Code runs, but I/O is really slow and the computation is also very slow.

WRF I/O Options

namelist.input has io_form = 2 => NetCDF I/O funneled through one MPI rank.

NetCDF is slow for several reasons : transpose is required for reads and writes, NetCDF for file writes will use the “fill” option by default : file is first written with blank data, then re-written with real data. Work is done by one MPI rank.

=> use parallel NetCDF : io_form = 11 in WRF namelist.input

Data for one midplane, 8 MPI ranks/node, 8 OpenMP threads, conus 2.5km:

file	operation	NetCDF	pNetCDF
restart	read	1244 sec	71 sec
lateral bc	read	361 sec	40 sec
history	write	206 sec	30 sec

WRF includes asynchronous I/O options, including “pNetCDF-quilting”, which is recommended on clusters.

BGQ suggestion : use parallel NetCDF and fewer MPI ranks/node with more OpenMP threads: 2 ranks/node with 32 OpenMP threads for example.

WRF Profile with -O3 -qstrict

Function-level profile is dominated by __ieee754_log, __ieee754_pow, atan, __ieee754_exp, … slow math intrinsic functions.

Linking with the IBM MASS library will not help, because libmass.a contains entry points with names : log, pow, exp, etc. … not __ieee754_log, __ieee754_pow, etc.

XL Fortran compiler uses name-mangling to control the selection of math intrinsics. Use option “-qstrict=nolibrary”. That way you get the fast library routines, but keep the -qstrict rules for source code.

Note : -O3 without -qstrict will map log(x) => __xl_log(x) etc., where the “xl” entry points are in libxlopt.a, and are equivalent to libmass.a.

FCOPTIM = -O3 -qstrict => computation time = 305 sec

FCOPTIM = -O3 -qstrict=nolibrary -qdebug=recipf:forcesqrt -qsimd=noauto
=> computation time = 141 sec more than 2x improvement.

FCOPTIM = -O3 -qdebug=recipf:forcesqrt -qsimd=noauto
=> computation time = 141 sec … same as with -qstrict=nolibrary

WRF Compiler/Runtime Options

-qhot : high-order transformations, implies “fastmath” and “vector MASS” , scalar (exp, log, …) and vector (array) (vexp, vlog, ...) math intrinsics.

-qhot is not recommended for WRF on BGQ, because it will try to replace division in loops with calls to vdiv() or vrec(). It is faster to generate inline code using the reciprocal-estimate instruction: -qdebug=recipf:forcesqrt.

Could improve performance a little by explicit use of vector math routines in selected places, vector log for example.

I ended up with options :

FCOPTIM = -O3 -qdebug=recipf:forcesqrt -qsimd=noauto

And link with -lmass (for scalar routines that were not name-shifted).

I also tried -qsimd=auto (default option) and WRF failed with SEGV.

Using OMP_WAIT_POLICY=active improves performance significantly default policy on some systems is OMP_WAIT_POLICY=passive.

WRF Domain Decomposition Considerations

Continental US 2.5 km problem : grid dimensions are 1500x1200x35

MPI 2D decomposition nproc_x, nproc_y

Basic storage order : array(i,k,j) in WRF xzy order

OpenMP default strategy is to tile over the “j” index (y dimension).

BGQ midplane : 512 nodes.

8 ranks/node => 4096 MPI ranks x/i y/j
64X64 decomposition => local domain ~ (23-24) x (18-19)
with 8 OpenMP threads, each thread gets 2 or 3 “j” values
=> guaranteed imbalance between work per thread

Concurrency is limited, and you have to pay careful attention to the way MPI ranks and OpenMP threads are mapped onto the simulation domain.

2 ranks/node => 1024 MPI ranks x/i y/j
32X32 decomposition => local domain ~(46-47) x (37-38)
with 32 OpenMP threads, each thread gets 1 or 2 “j values”
=> even more load imbalance

WRF Performance Data

BGQ midplane, 8 ranks/node, 8 thds, OMP_WAIT_POLICY=active

```
MPI task 2765 of 4096 had the minimum communication time.
total communication time = 9.440 seconds.
total elapsed time       = 143.936 seconds.

Derived metrics for code block "step" averaged over process(es) on node <2,2,3,0,1>:
Instruction mix:  FPU = 38.33 %,  FXU = 61.67 %
Instructions per cycle completed per core = 0.6707
Per cent of max issue rate per core = 41.36 %
Total weighted GFlops for this node = 9.264
Loads that hit in L1 d-cache =  94.77 %
                  L1P buffer =   2.22 %
                  L2 cache   =   2.66 %
                  DDR        =   0.34 %
DDR traffic for the node: ld = 1.653, st = 1.229, total = 2.882 (Bytes/cycle)
```

Communication performance is very good … the problem is compute-bound.

Memory bandwidth is not a limiting factor : the node can support ~6x more bandwidth.

IPC is ~41% of the theoretical limit. L1 D-cache hits are not bad. Latency to the L2 cache is a factor. Load imbalance limits the SMT speedup, and the MPI scaling.

Take Home Messages

A performance model or expectation can be very useful.

There are hardware limits to performance.

BGQ has useful hardware counters … you can see what is going on.

Data layout really matters for performance purposes.

Spend some time learning compiler/runtime options.

I/O is a difficult subject … talk to someone with experience.

Expect to be humbled, but don't take “no” for an answer.