Blue Gene/L System and Optimization Tips

Bob Walkup
IBM Watson Research Center
(walkup@us.ibm.com, 914-945-1512)

(1) Some basic user information
(2) Characteristics of the hardware
(3) Getting the most out of IBM XL compilers
(4) Profiling to identify performance issues
(5) Using library routines for math kernels
Blue Gene System

You login to the front end to compile, submit jobs, analyze results, etc. The front end is an IBM p-Series system with Linux as the operating system; so both the processor architecture and OS are very different from the Blue Gene compute nodes.

The main limitations for the compute nodes are:

- 512 MB memory per node
- MPI only, no OpenMP or pthreads
- Compute-node kernel is not Linux (limited system calls)
- Executables must be statically linked (no shared libs)
Blue Gene System on a Chip

- IBM CU-11, 0.13 µm
- 11 x 11 mm die size
- 25 x 32 mm CBGA
- 474 pins, 328 signal
- 1.5/2.5 Volt
Blue Gene Networks

**3 Dimensional Torus**
- Interconnects all compute nodes
- Virtual cut-through hardware routing
- 1.4Gb/s on all 12 node links (2.1 GB/s per node)
- Communications backbone for computations

**Global Tree**
- One-to-all broadcast functionality
- Reduction operations functionality
- 2.8 Gb/s of bandwidth per link
- Latency of tree traversal in the order of 5 µs
- Interconnects all compute and I/O nodes

**Gigabit Ethernet**
- Incorporated into every node ASIC
- Active in the I/O nodes
- All external comm. (file I/O, user interaction, etc.)

**Low Latency Global Barrier and Interrupt**

**Control Network**
IBM Compilers for Blue Gene

Located on the front-end system in directories:

Fortran:  /opt/ibmcmp/xlf/9.1/bin  (for version 9.1)
C:       /opt/ibmcmp/vac/7.0/bin  (for version 7.0)
C++:     /opt/ibmcmp/vacpp/7.0/bin (for version 7.0)

Add the directories above to your PATH.

Fortran: blrts_xlf, blrts_xlf90, blrts_xlf95
C: blrts_xlc, blrts_cc
C++: blrts_xlC

Note: xlf, xlf90, xlc, xlC, etc. are for the front end, not for Blue Gene. To generate code for Blue Gene compute-nodes, use the blrts compiler versions.

Compiler config files are on the front-end node in:

Fortran: /etc/opt/ibmcmp/xlf/9.1/blrts-xlf.cfg
C/C++: /etc/opt/ibmcmp/vac/7.0/blrts-vac.cfg

For Blue Gene, you compile on the front end, which has a different architecture and different OS from the compute nodes. /usr is for the front-end; Blue Gene software is in:

/bg1/BlueLight/ppcfloor/bgl/sys/…
/bg1/BlueLight/ppcfloor/blrts-gnu/…
GNU compilers for Blue Gene

GNU compilers (not recommended) for Blue Gene are in:

/bgl/BlueLight/ppcfloor/blrts-gnu/bin

The IBM compilers tend to offer significantly higher performance. The GNU compilers offer more flexible support for things like inline assembler.

The GNU compilers in /usr/bin are for the front end, not for Blue Gene compute nodes.

As part of the system software set, you will find scripts mpif77, mpicc, mpicxx in the directory:

/bgl/BlueLight/ppccfloor/bglsys/bin

These scripts are for the Blue Gene GNU compilers, and are not recommended.

You can use convenience scripts: mpxlf, mpxlf90, mpxlc, mpcc, mpxlC that invoke the IBM compilers for Blue Gene. These “mp” scripts are locally-grown, not standard system software.
MPI on Blue Gene

MPI implementation based on MPICH-2 (Argonne)

Include path for `<mpi.h>`, mpif.h :

    -I/bgl/BlueLight/ppcfloor/bglsys/include

Libraries to link for MPI:

    -L/bgl/BlueLight/ppcfloor/bglsys/lib

    -lmpich.rts -lmsglayer.rts -lrts.rts -ldevices.rts

Sample Makefile:

FC = blrts_xlf
FFLAGS = -g -O -qarch=440 -qmaxmem=64000
MPI_INC = -I/bgl/BlueLight/ppcfloor/bglsys/include
MPI_LIB = -L/bgl/BlueLight/ppcfloor/bglsys/lib \
    -lmpich.rts -lmsglayer.rts -lrts.rts -ldevices.rts
LD = blrts_xlf
LDFLAGS = -g

hello.x : hello.o
    $(LD) $(LDFLAGS) hello.o $(MPI_LIB) -o hello.x

hello.o : hello.f
    $(FC) -c $(FFLAGS) $(MPI_INC) hello.f
Submitting jobs with mpirun

You can use “mpirun” to submit jobs. The Blue Gene
mpirun is in /bgl/BlueLight/ppcfloor/bglsys/bin

Typical use:

mpirun -np 512 -cwd `pwd` -exe your.x

common options: -args “list of arguments”
                -env “VARIABLE=value”
                -mode CO/VN (coprocessor/virtual-node)
                -connect TORUS (for 512/1024 nodes)

coprocessor mode : one MPI process per node, 512 MB
limit per process

virtual-node mode : two MPI processes per node, 256 MB
limit per process; L3 cache, memory, networks, are shared.

More details: mpirun -h (for help)

redbook: Blue Gene/L System Administration
(www.redbooks.ibm.com, sg247178)

Limitations: one job per partition, limited partition sizes
addr2line can really help you identify problems – it is the first pass method for debugging. Many kinds of failures give you an instruction address; and addr2line will take that and tell you the source file and line number.

On BG/L, core files are text. Look at the core file with a text editor, focus on the function call chain; feed the hex addresses to addr2line.

Examples:

addr2line -e your.x hex_address

tail -n 10 core.511 | addr2line -e your.x

Use grep and word-count (wc) to examine large numbers of core files: grep hex_address "core.*" | wc –l

You can get the instruction that failed by using objdump:
objdump -d your.x | grep hex_address

If your application exits without leaving a core file, get a traceback : link with libtraceback_exit.a (locally grown). This will give you instruction addresses.

If your Fortran application fails with a numerical runtime error message, such as “1525-108”, check the list of xlf messages: xlf/rte.messages (and complain, please).
Powerpc-440 Processor

32-bit architecture at 700 MHz

single integer unit (fxu)

single load/store unit

special double floating-point unit (dfpu)

L1 Data cache : 32 KB total size, 32-Byte line size, 64-way associative, round-robin replacement

L2 Data cache : prefetch buffer, holds 16 128-byte lines

L3 Data cache : 4 MB, ~35 cycles latency, on-chip

Memory : 512 MB DDR at 350 MHz, ~85 cycles latency

Double FPU has 32 primary floating-point registers, 32 secondary floating-point registers, and supports :
  (1) standard powerpc instructions, which execute on fpu0 (fadd, fmadd, fadds, fdiv, …), and
  (2) SIMD instructions for 64-bit floating-point numbers (fpadd, fpmadd, fpre, …)

Floating-point pipeline : 5 cycles

Floating-point load-to-use latency : 4 cycles
Instruction Latencies and Throughputs

<table>
<thead>
<tr>
<th>Instruction</th>
<th>latency</th>
<th>throughput/cycle</th>
</tr>
</thead>
<tbody>
<tr>
<td>fadd</td>
<td>5 cycles</td>
<td>1</td>
</tr>
<tr>
<td>fmadd</td>
<td>5 cycles</td>
<td>1</td>
</tr>
<tr>
<td>fpmadd</td>
<td>5 cycles</td>
<td>1</td>
</tr>
<tr>
<td>fdiv</td>
<td>30 cycles</td>
<td>1/30</td>
</tr>
</tbody>
</table>

Theoretical flop limit = 1 fpmadd per cycle => 4 floating-point ops per cycle.

Practical limit is often loads and stores.

No hardware square-root function. Default sqrt() is from GNU libm.a => ~100 cycles. With -O3 you get a Newton’s method for sqrt() inline, not a function call.

Efficient use of the double-FPU requires 16-byte alignment. There are quad-word load/store instructions (lfpd, stfpd) that can double the bandwidth between L1 and registers. In most applications loads and stores are at least as important as floating-point operations. You can saturate the bandwidth to L3 or memory with normal double-word load/store instructions. So the double-FPU instructions can help for data in L1, but not for data in L3 or memory.
Daxpy Performance on BG/L

call alignx(16, x(1))
call alignx(16, y(1))
do i = 1, n
    y(i) = a*x(i) + y(i)
end do

Performance of compiler-generated code is shown.

-qarch=440 => single FPU code, theoretical limit is 2/3 flops per cycle.

-qarch=440d => double FPU code, theoretical limit is 4/3 flops per cycle, data in-cache, 2/3 flops per cycle otherwise
Using IBM XL Compilers

Optimization levels:
  Default optimization = none (very slow)
  -O : good place to start, use with -qmaxmem=64000
  -O2: same as -O
  -O3 -qstrict : can try more aggressive optimization,
                  but must strictly obey program semantics
  -O3: aggressive, allows re-association, will replace
       division by multiplication with the inverse

-qhot : turns on high-order transformation module
       will add vector routines, unless -qhot=novector
       check listing: -qreport=hotlist
-qipa : inter-procedure analysis; many suboptions such as:
       -qipa=level=2

Architecture flags:
  -qarch=440  : generates standard powerpc floating-
                point code
  -qarch=440d : will try to generate double FPU code

On BG/L start with : -g -O -qarch=440 -qmaxmem=64000

Try : -O3 -qarch=440/440d
Try : -O4 -qarch=440d -qtune=440 (or -O5 …)

-O4 = -O3 -qhot -qipa=level=1 -qarch=auto
-O5 = -O3 -qhot -qipa=level=2 -qarch=auto
Some BG/L-Specific Items

For double FPU code generation, 16-byte alignment is required; may need alignment assertions:

Fortran:
    call alignx(16,x(1))
    call alignx(16,y(1))
!ibm* unroll(10)
    do i = 1, n
       y(i) = a*x(i) + y(i)
    end do

C:
    double * x, * y;
    #pragma disjoint (*x, *y)
    __alignx(16,x);
    __alignx(16,y);
    #pragma unroll(10)
    for (i=0; i<n; i++) y[i] = a*x[i] + y[i];

Try : -O3 -qarch=440d -qlist –qsource

Read the assembler listing.

Easiest approach to double FPU is to use optimized math library routines.
Generating SIMD Code

The XL compiler has two different components that can generate SIMD code:

1. the back-end optimizer with -O3 -qarch=440d
2. the TPO front-end, with -qhot or -O4, -O5

For TPO, you can add -qdebug=diagnostic to get some information about SIMD code generation.

Use -qlist -qsource to check assembler code.

If you use -O4 or -O5, add -qarch=440d -qtune=440

Try -O3 -qarch=440d (backend only).

Many things can inhibit SIMD code generation: unknown alignment, accesses that are not stride one, potential aliasing issues, etc.

In principle double-FPU code should help only for data in L1 cache.

Best potential improvement with SIMD is vectors of reciprocals or square-roots, where there are special fast parallel pipelined instructions that can help.
Double FPU Intrinsic Routines

In C/C++ : #include <complex.h>

double _Complex __lfpd (double * addr);
void __stfpd (double * addr, double _Complex);
double _Complex __fpadd (double _Complex, double _Complex);
double _Complex __fpre (double _Complex);
double _Complex __fprsqrte (double _Complex);
etc.

You can explicitly code calls to generate double FPU code, but the compiler may generate different assembler, and has control over instruction scheduling. Check the assembler code using -qlist -qsource.

If you want to control everything, write code in assembler.

Example: fast vector reciprocal

Use Newton’s method to solve \( f(x) = a - 1/x = 0 \)
\[ x_0 = \text{fpre}(x) \, \text{(good to 13 bits on BG/L)} \]
\[ x_{i+1} = x_i + x_i*(1.0 - a*x_i) \, \text{(2 iterations for double)} \]
Example: Vector Reciprocal

```c
#include <complex.h>

void aligned_vrec(double *y, double *x, int n) {
    complex double tx0, tx2, tx4, tx6, tx8;
    ...
    const complex double one = 1.0 + 1.0*I;
    #pragma disjoint(*x, *y)
    __alignx(16,x);
    __alignx(16,y);

    for (i=0; i<n-9; i+=10) {
        tx0 = __lfpd(&x[i   ]);
        tx2 = __lfpd(&x[i+ 2]);
        tx4 = __lfpd(&x[i+ 4]);
        tx6 = __lfpd(&x[i+ 6]);
        tx8 = __lfpd(&x[i+ 8]);

        rx0 = __fpre(tx0);
        rx2 = __fpre(tx2);
        rx4 = __fpre(tx4);
        rx6 = __fpre(tx6);
        rx8 = __fpre(tx8);

        sx0 = __fpnmsub(one, tx0 , rx0 );
        sx2 = __fpnmsub(one, tx2 , rx2 );
        sx4 = __fpnmsub(one, tx4 , rx4 );
        sx6 = __fpnmsub(one, tx6 , rx6 );
        sx8 = __fpnmsub(one, tx8 , rx8 );

        fx0 = __fpmadd(rx0 , rx0 , sx0 );
        fx2 = __fpmadd(rx2 , rx2 , sx2 );
        fx4 = __fpmadd(rx4 , rx4 , sx4 );
    ...
```
## Assembler Listing for Vector Reciprocal

<table>
<thead>
<tr>
<th>Line</th>
<th>Address</th>
<th>Instruction</th>
<th>Parameters</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>17</td>
<td>CL.461:</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>23</td>
<td>002304</td>
<td>lfpdx</td>
<td>7D284B9C</td>
<td>LFPL</td>
</tr>
<tr>
<td>24</td>
<td>002308</td>
<td>lfpdx</td>
<td>7D08539C</td>
<td>LFPL</td>
</tr>
<tr>
<td>25</td>
<td>002310</td>
<td>lfpdx</td>
<td>7CE85B9C</td>
<td>LFPL</td>
</tr>
<tr>
<td>26</td>
<td>002314</td>
<td>fpre</td>
<td>03C0481C</td>
<td>FPRE</td>
</tr>
<tr>
<td>27</td>
<td>002318</td>
<td>lfpdux</td>
<td>7CA8FBDC</td>
<td>LFPLU</td>
</tr>
<tr>
<td>30</td>
<td>00231C</td>
<td>fpre</td>
<td>03E0401C</td>
<td>FPRE</td>
</tr>
<tr>
<td>31</td>
<td>002320</td>
<td>fpre</td>
<td>01A0381C</td>
<td>FPRE</td>
</tr>
<tr>
<td>32</td>
<td>002324</td>
<td>fpre</td>
<td>0080301C</td>
<td>FPRE</td>
</tr>
<tr>
<td>33</td>
<td>002328</td>
<td>fpre</td>
<td>0060281C</td>
<td>FPRE</td>
</tr>
<tr>
<td>35</td>
<td>002332</td>
<td>fpnmsub</td>
<td>00957B8</td>
<td>FPNMSUB</td>
</tr>
<tr>
<td>36</td>
<td>002333</td>
<td>fpnmsub</td>
<td>016857F8</td>
<td>FPNMSUB</td>
</tr>
<tr>
<td>37</td>
<td>002334</td>
<td>fpnmsub</td>
<td>01875378</td>
<td>FPNMSUB</td>
</tr>
<tr>
<td>38</td>
<td>002338</td>
<td>fpnmsub</td>
<td>00465138</td>
<td>FPNMSUB</td>
</tr>
<tr>
<td>39</td>
<td>00233C</td>
<td>fpnmsub</td>
<td>002550F8</td>
<td>FPNMSUB</td>
</tr>
<tr>
<td>41</td>
<td>002340</td>
<td>fpmadd</td>
<td>001EF020</td>
<td>FPMADD</td>
</tr>
<tr>
<td>42</td>
<td>002344</td>
<td>fpmadd</td>
<td>017FFAE0</td>
<td>FPMADD</td>
</tr>
<tr>
<td>43</td>
<td>002348</td>
<td>fpmadd</td>
<td>018D6B20</td>
<td>FPMADD</td>
</tr>
<tr>
<td>44</td>
<td>00234C</td>
<td>fpmadd</td>
<td>004420A0</td>
<td>FPMADD</td>
</tr>
<tr>
<td>45</td>
<td>002350</td>
<td>fpmadd</td>
<td>00231860</td>
<td>FPMADD</td>
</tr>
<tr>
<td>47</td>
<td>002354</td>
<td>fpnmsub</td>
<td>01295038</td>
<td>FPNMSUB</td>
</tr>
<tr>
<td>48</td>
<td>002358</td>
<td>fpnmsub</td>
<td>010852F8</td>
<td>FPNMSUB</td>
</tr>
<tr>
<td>49</td>
<td>00235C</td>
<td>fpnmsub</td>
<td>00E75338</td>
<td>FPNMSUB</td>
</tr>
<tr>
<td>50</td>
<td>002360</td>
<td>fpnmsub</td>
<td>008650B8</td>
<td>FPNMSUB</td>
</tr>
<tr>
<td>51</td>
<td>002364</td>
<td>fpnmsub</td>
<td>00655078</td>
<td>FPNMSUB</td>
</tr>
<tr>
<td>53</td>
<td>002368</td>
<td>fpmadd</td>
<td>00000260</td>
<td>FPMADD</td>
</tr>
<tr>
<td>54</td>
<td>00236C</td>
<td>fpmadd</td>
<td>00AB5A20</td>
<td>FPMADD</td>
</tr>
<tr>
<td>55</td>
<td>002370</td>
<td>fpmadd</td>
<td>00CC61E0</td>
<td>FPMADD</td>
</tr>
<tr>
<td>56</td>
<td>002374</td>
<td>fpmadd</td>
<td>00421120</td>
<td>FPMADD</td>
</tr>
<tr>
<td>57</td>
<td>002378</td>
<td>fpmadd</td>
<td>002108E0</td>
<td>FPMADD</td>
</tr>
<tr>
<td>59</td>
<td>00237C</td>
<td>stfpdx</td>
<td>7C074F9C</td>
<td>SFPL</td>
</tr>
<tr>
<td>60</td>
<td>002380</td>
<td>stfpdx</td>
<td>7CA7579C</td>
<td>SFPL</td>
</tr>
<tr>
<td>61</td>
<td>002384</td>
<td>stfpdx</td>
<td>7CC75F9C</td>
<td>SFPL</td>
</tr>
<tr>
<td>62</td>
<td>002388</td>
<td>stfpdx</td>
<td>7C47679C</td>
<td>SFPL</td>
</tr>
<tr>
<td>63</td>
<td>00238C</td>
<td>stfpdux</td>
<td>7C27FFDC</td>
<td>SFPLU</td>
</tr>
<tr>
<td>0</td>
<td>002390</td>
<td>bc</td>
<td>4320FF74</td>
<td>BCT</td>
</tr>
</tbody>
</table>
Profile your Application

Standard profiling (prof, gprof) is now available on BG/L, so you can use the normal profiling options, -g and -pg, when you compile and link. Then run the application. When the job exits, it should create gmon.out files that can be analyzed with gprof on the front end:

```
gprof your.x gmon.out.0 > gprof_report.0
```

Gprof on the front end is OK for function (subroutine) timing information.

Xprofiler has been ported to Blue Gene (IBM High Performance Computing Toolkit), and can be used to obtain statement-level profiling data.

Use MPI profiling tools – there are many to choose from.

- mpi_trace : low overhead, text summary
- tau : [http://www.cs.uoregon.edu/research/paracomp/tau](http://www.cs.uoregon.edu/research/paracomp/tau)
- paraver : extensive data analysis capabilities
- mpe / jumpshot : standard with MPICH

For paraver : [http://www.cepba.upc.es/paraver/](http://www.cepba.upc.es/paraver/)
For kojak: [http://www.fz-juelich.de/zam/cojak/](http://www.fz-juelich.de/zam/cojak/)
Gprof Example: GTC Flat profile

Each sample counts as 0.01 seconds.

<table>
<thead>
<tr>
<th>%</th>
<th>cumulative</th>
<th>self</th>
<th>self</th>
<th>total</th>
<th>name</th>
</tr>
</thead>
<tbody>
<tr>
<td>time</td>
<td>seconds</td>
<td>seconds</td>
<td>calls</td>
<td>s/call</td>
<td></td>
</tr>
<tr>
<td>37.43</td>
<td>144.07</td>
<td>144.07</td>
<td>201</td>
<td>0.72</td>
<td>chargei</td>
</tr>
<tr>
<td>25.44</td>
<td>241.97</td>
<td>97.90</td>
<td>200</td>
<td>0.49</td>
<td>pushi</td>
</tr>
<tr>
<td>6.12</td>
<td>265.53</td>
<td>23.56</td>
<td></td>
<td></td>
<td>_xldintv</td>
</tr>
<tr>
<td>4.85</td>
<td>284.19</td>
<td>18.66</td>
<td></td>
<td></td>
<td>cos</td>
</tr>
<tr>
<td>4.49</td>
<td>301.47</td>
<td>17.28</td>
<td></td>
<td></td>
<td>sin1</td>
</tr>
<tr>
<td>4.19</td>
<td>317.61</td>
<td>16.14</td>
<td>200</td>
<td>0.08</td>
<td>poisson</td>
</tr>
<tr>
<td>3.79</td>
<td>332.18</td>
<td>14.57</td>
<td></td>
<td></td>
<td>_pxldmod</td>
</tr>
<tr>
<td>3.55</td>
<td>345.86</td>
<td>13.68</td>
<td></td>
<td></td>
<td>__ieee754_exp</td>
</tr>
<tr>
<td>2.76</td>
<td>356.48</td>
<td>10.62</td>
<td></td>
<td></td>
<td>BGLML_Messager_advance</td>
</tr>
<tr>
<td>2.23</td>
<td>365.05</td>
<td>8.57</td>
<td>200</td>
<td>0.04</td>
<td>shifti</td>
</tr>
<tr>
<td>1.23</td>
<td>369.80</td>
<td>4.75</td>
<td>400</td>
<td>0.01</td>
<td>smooth</td>
</tr>
<tr>
<td>0.79</td>
<td>372.85</td>
<td>3.05</td>
<td></td>
<td></td>
<td>exp</td>
</tr>
<tr>
<td>0.53</td>
<td>374.88</td>
<td>2.03</td>
<td>200</td>
<td>0.01</td>
<td>field</td>
</tr>
<tr>
<td>0.27</td>
<td>375.91</td>
<td>1.03</td>
<td></td>
<td></td>
<td>finitel</td>
</tr>
<tr>
<td>0.24</td>
<td>376.84</td>
<td>0.93</td>
<td></td>
<td></td>
<td>TreeAllreduce_fifo</td>
</tr>
<tr>
<td>0.24</td>
<td>377.77</td>
<td>0.93</td>
<td></td>
<td></td>
<td>__exp</td>
</tr>
<tr>
<td>0.20</td>
<td>378.54</td>
<td>0.77</td>
<td></td>
<td></td>
<td>__sin</td>
</tr>
<tr>
<td>0.15</td>
<td>379.12</td>
<td>0.58</td>
<td></td>
<td></td>
<td>readfun_torus</td>
</tr>
<tr>
<td>0.12</td>
<td>379.59</td>
<td>0.47</td>
<td>1</td>
<td>0.47</td>
<td>poisson_initial</td>
</tr>
</tbody>
</table>

Performance issues are mainly in two routines: chargei and pushi. There are lots of intrinsic functions, and expensive conversions to get the integer part of a floating-point number.
Function-Level Profiling – Experimental

IBM XL compilers, try -qdebug=function_trace

__func_trace_enter(char * routine, char * file, int lineno)
__func_trace_exit(char * routine, char * file, int lineno)

The compiler adds calls at entry and exit of every routine. Similar to -finstrument-functions in GNU compilers.

Makes it possible to do elapsed-time profiling at the function level; but there is overhead.

To try this method:

Compile with -qdebug=function_trace
Link with libmpitrace_f.a or libmpitrace_c.a
Specify env variable FLAT_PROFILE=yes at runtime.

An elapsed-time timer is started upon function entry, and stopped at function exit.
**Example: Flat Profile**

---

Elapsed-time flat profile for MPI task 0.
---

Self (seconds) does not include calls to other routines, Inclusive does.
---

<table>
<thead>
<tr>
<th>%Time</th>
<th>Self(seconds)</th>
<th>Inclusive</th>
<th>#Calls</th>
<th>Routine</th>
</tr>
</thead>
<tbody>
<tr>
<td>54.79</td>
<td>40.7349</td>
<td>58.4471</td>
<td>491520</td>
<td>sppm</td>
</tr>
<tr>
<td>11.49</td>
<td>8.5414</td>
<td>8.5414</td>
<td>491520</td>
<td>interf_sw</td>
</tr>
<tr>
<td>6.37</td>
<td>4.7344</td>
<td>4.7344</td>
<td>491520</td>
<td>difuze</td>
</tr>
<tr>
<td>5.97</td>
<td>4.4364</td>
<td>4.4364</td>
<td>983040</td>
<td>dintrf</td>
</tr>
<tr>
<td>2.59</td>
<td>1.9267</td>
<td>1.9267</td>
<td>5</td>
<td>glblmax</td>
</tr>
<tr>
<td>1.84</td>
<td>1.3642</td>
<td>1.3642</td>
<td>22</td>
<td>bdrys3r</td>
</tr>
<tr>
<td>1.73</td>
<td>1.2894</td>
<td>1.2894</td>
<td>10</td>
<td>bdrys3fr</td>
</tr>
<tr>
<td>1.69</td>
<td>1.2563</td>
<td>11.5589</td>
<td>5</td>
<td>hydzy</td>
</tr>
<tr>
<td>1.69</td>
<td>1.2553</td>
<td>11.6845</td>
<td>5</td>
<td>hydzy</td>
</tr>
<tr>
<td>1.60</td>
<td>1.1874</td>
<td>13.9504</td>
<td>5</td>
<td>hydx</td>
</tr>
<tr>
<td>1.33</td>
<td>0.9856</td>
<td>11.1903</td>
<td>5</td>
<td>hydx</td>
</tr>
<tr>
<td>1.24</td>
<td>0.9184</td>
<td>0.9184</td>
<td>1</td>
<td>setup</td>
</tr>
<tr>
<td>1.14</td>
<td>0.8506</td>
<td>0.8506</td>
<td>21</td>
<td>bdry3o2r</td>
</tr>
<tr>
<td>0.99</td>
<td>0.7329</td>
<td>0.7329</td>
<td>21</td>
<td>bdry3o1r</td>
</tr>
<tr>
<td>0.90</td>
<td>0.6676</td>
<td>0.6676</td>
<td>20</td>
<td>bdrys1r</td>
</tr>
<tr>
<td>0.84</td>
<td>0.6232</td>
<td>11.6437</td>
<td>5</td>
<td>hydxx</td>
</tr>
<tr>
<td>0.63</td>
<td>0.4649</td>
<td>0.4649</td>
<td>20</td>
<td>bdrys1s</td>
</tr>
<tr>
<td>0.62</td>
<td>0.4580</td>
<td>11.0800</td>
<td>5</td>
<td>hydzz</td>
</tr>
<tr>
<td>0.59</td>
<td>0.4350</td>
<td>0.4350</td>
<td>12</td>
<td>bdrys1ar</td>
</tr>
<tr>
<td>0.39</td>
<td>0.2899</td>
<td>0.2899</td>
<td>11</td>
<td>bdrys1br</td>
</tr>
<tr>
<td>0.25</td>
<td>0.1855</td>
<td>0.1855</td>
<td>10</td>
<td>bdrys2fr</td>
</tr>
<tr>
<td>0.20</td>
<td>0.1460</td>
<td>0.1460</td>
<td>12</td>
<td>bdrys1as</td>
</tr>
<tr>
<td>0.15</td>
<td>0.1145</td>
<td>0.1145</td>
<td>11</td>
<td>bdrys1bs</td>
</tr>
<tr>
<td>0.14</td>
<td>0.1064</td>
<td>0.1064</td>
<td>21</td>
<td>bdry2o1r</td>
</tr>
</tbody>
</table>
Example : MPI Profile

elapsed time from clock-cycles using freq = 700.0 MHz

<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>6</td>
<td>0.0</td>
<td>0.000</td>
</tr>
<tr>
<td>MPI_Comm_rank</td>
<td>1</td>
<td>0.0</td>
<td>0.000</td>
</tr>
<tr>
<td>MPI_Send</td>
<td>285196</td>
<td>6694.6</td>
<td>35.545</td>
</tr>
<tr>
<td>MPI_Recv</td>
<td>210284</td>
<td>698.5</td>
<td>20.959</td>
</tr>
<tr>
<td>MPI_Probe</td>
<td>81243</td>
<td>0.0</td>
<td>124.980</td>
</tr>
<tr>
<td>MPI_Iprobe</td>
<td>352732</td>
<td>0.0</td>
<td>0.358</td>
</tr>
<tr>
<td>MPI_Bcast</td>
<td>5</td>
<td>4.0</td>
<td>0.000</td>
</tr>
<tr>
<td>MPI_Barrier</td>
<td>10000</td>
<td>0.0</td>
<td>85.153</td>
</tr>
<tr>
<td>MPI_Gather</td>
<td>10002</td>
<td>8.0</td>
<td>0.803</td>
</tr>
<tr>
<td>MPI_Allgather</td>
<td>3</td>
<td>14.7</td>
<td>0.001</td>
</tr>
<tr>
<td>MPI_Allreduce</td>
<td>6</td>
<td>17.3</td>
<td>0.660</td>
</tr>
</tbody>
</table>

MPI task 0 of 512 had the minimum communication time.
total communication time = 268.460 seconds.
total elapsed time       = 434.298 seconds.
top of the heap address  = 47.293 MBytes.

Message size distributions:

<table>
<thead>
<tr>
<th>MPI_Send</th>
<th>#calls</th>
<th>avg. bytes</th>
<th>time(sec)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>73141</td>
<td>16.0</td>
<td>0.273</td>
</tr>
<tr>
<td></td>
<td>380</td>
<td>45.9</td>
<td>0.004</td>
</tr>
<tr>
<td></td>
<td>786</td>
<td>98.9</td>
<td>0.002</td>
</tr>
<tr>
<td></td>
<td>2325</td>
<td>196.8</td>
<td>0.016</td>
</tr>
<tr>
<td></td>
<td>5768</td>
<td>379.1</td>
<td>0.082</td>
</tr>
<tr>
<td></td>
<td>121925</td>
<td>998.1</td>
<td>22.893</td>
</tr>
<tr>
<td></td>
<td>1023</td>
<td>1600.2</td>
<td>0.011</td>
</tr>
<tr>
<td></td>
<td>643</td>
<td>4079.7</td>
<td>0.112</td>
</tr>
<tr>
<td></td>
<td>511</td>
<td>5721.0</td>
<td>0.019</td>
</tr>
<tr>
<td></td>
<td>77672</td>
<td>10240.0</td>
<td>5.937</td>
</tr>
<tr>
<td></td>
<td>1022</td>
<td>960000.0</td>
<td>6.198</td>
</tr>
</tbody>
</table>

-env BGLMPI_EAGER=900  (BG/L default = 1000)
to get adaptive routes for messages of ~1K, and better performance.
For exchange between random sites on the torus, adaptive routing helps. Current (December 2005) default is 1000 byte eager limit, with static routes for eager messages.

If all communication is collinear on the torus, adaptive routes can’t help, because there is only one minimal route on the linear path, and it may be better to increase the eager limit: BGLMPI_EAGER=10000, for example.
Scalar and Vector MASS Routines

Approximate cycle-counts per evaluation on BG/L

<table>
<thead>
<tr>
<th>Function</th>
<th>libm.a</th>
<th>libmass.a</th>
<th>libmassv.a</th>
</tr>
</thead>
<tbody>
<tr>
<td>exp</td>
<td>185</td>
<td>64</td>
<td>22</td>
</tr>
<tr>
<td>log</td>
<td>320</td>
<td>80</td>
<td>25</td>
</tr>
<tr>
<td>pow</td>
<td>460</td>
<td>176</td>
<td>29 – 48</td>
</tr>
<tr>
<td>sqrt</td>
<td>106</td>
<td>46</td>
<td>8-10</td>
</tr>
<tr>
<td>rsqrt</td>
<td>136</td>
<td>…</td>
<td>6-7</td>
</tr>
<tr>
<td>1/x</td>
<td>30 (fdiv)</td>
<td>…</td>
<td>4-5</td>
</tr>
</tbody>
</table>

Extensive set of both scalar and vector routines have been coded in C by IBM Toronto, and re-compiled for BG/L. The routines vrec(), vsqrt(), vrsqrt() use BG/L-specific double FPU instructions (fpre, fpr sqrte). The other routines make very little use of the double FPU.

Best performance in any case is with the vector routines, can be user-called or compiler generated (-qhot). Add linker option -Wl,--allow-multiple-definition to allow multiple definitions for the math routines – needed for libmass.a.

http://www.ibm.com/software/awdtools/mass/support
Optimized Math Routines

IBM ESSL for Blue Gene – recently released. Has many BLAS routines optimized for the double FPU. Best for level 3 BLAS (matrix-matrix).

FFTW with Vienna GEL SIMD-izer (from Stefan Kral) (bglfftwgel-2.1.5), good double FPU performance.

Use with FFTW_MEASURE for best performance.

Scalar and vector MASS libraries for intrinsic functions.

Double FPU code is often difficult to get from the compiler, so using optimized math libraries is one of the best ways to benefit from the double FPU.
The current compute-node kernel (December 2005) catches alignment exceptions, implements a fix and continues. This results in very poor SIMD performance unless the input data is 16-byte aligned. The alignment exceptions (if there were any) will appear in the RAS database.

For double-FPU optimized libraries, it is crucial to have 16-byte aligned inputs.
Virtual Node Mode

Can boot the partition in virtual node mode: two MPI processes per node; use both cpus for computation.

Advantage: can get more computational performance out of the hardware.

Disadvantages: have only $\frac{1}{2}$ the memory available per task, only 256 MB on most systems. Have more contention for network resources.

Typical speed-up is about 1.5x out of a maximum of 2x. There is contention for shared L3 and memory bandwidth.

Try virtual node mode when your app can fit inside 256 MB. With virtual node mode, experiment with

BGLMPI_MAPPING=TXYZ

This puts tasks 0 and 1 on the first node, tasks 2 and 3 on the next node, with nodes in x,y,z torus order. The default layout is XYZT, which is often less efficient than TXYZ. Also note that in the default TXYZ mode, you only get two tasks per node if you have $\#\text{tasks} = 2*\#\text{nodes}$; otherwise the XYZT layout will leave just one task on at least some nodes.