Blue Gene/P System and Optimization Tips

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

(1) Some basic information for users

(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

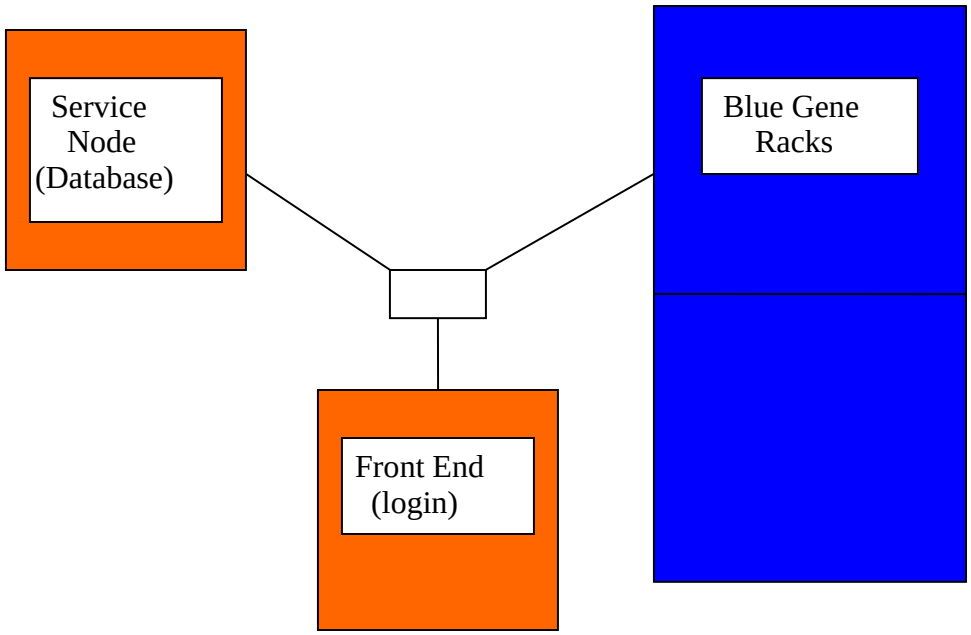
Blue Gene/P System

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:

2048 MB memory per node, 32-bit memory addressing

Compute-node kernel is not Linux (limited system calls)
examples: no fork() or system() calls



IBM Compilers for Blue Gene

Located on the front-end system in directories:

Fortran: /opt/ibmcmp/xlf/bg/11.1/bin (for version 11.1)
C: /opt/ibmcmp/vac/bg/9.0/bin (for version 9.0)
C++: /opt/ibmcmp/vacpp/bg/9.0/bin (for version 9.0)

Documentation is in /opt/ibmcmp/…/doc/en_US/pdf.

Fortran: bgxlf, bgxlf90, bgxlf95, …, and with added _r
C: bgxlc, bgcc, …, and with added _r
C++: bgxlC, …, and with added _r

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 bg compiler versions, or mpi scripts.

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

Fortran: /etc/opt/ibmcmp/xlf/bg/11.1/xlf.cfg
C/C++: /etc/opt/ibmcmp/vac/bg/9.0/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:

/bgsys/drivers/ppcfloor/…

GNU compilers for Blue Gene/P

GNU compilers for Blue Gene/P are in:

/bgsys/drivers/ppcfloor/gnu-linux/bin

powerpc-bgp-linux-gcc … BG/P gcc
powerpc-bgp-linux-g++ … BG/P g++
powerpc-bgp-linux-gfortran … BG/P gfortran

The IBM compilers tend to offer better performance, particularly for Fortran. 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.

Scripts that automatically use MPI

As part of the system software set, you will find scripts for GNU and IBM XL compilers in the directory:

/bgsys/drivers/ppcfloor/comm/bin

GNU : mpicc, mpicxx, mpif77, …

IBM: mpixlc, mpixlcxx, mpixlf77, mpixlf90, … (and _r)

MPI on Blue Gene/P

MPI implementation is based on MPICH-2

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

```
-I/bgsys/drivers/ppcfloor/com/include
```

Explicit list of libraries to link for MPI:

```
-L/bgsys/drivers/ppcfloor/comm/lib  \
    -lmpich.cnk -ldcmfcoll.cnk -ldcmf.cnk
-L/bgsys/drivers/ppcfloor/runtime/SPI  \
    -lSPI.cna -lrt -lpthread
```

It is simplest to use scripts mpixlc, mpixlf77, etc. but you can use explicit include paths and libraries.

Sample Makefile:

```
FC = mpixlf77
FFLAGS = -g -O2 -qarch=450 -qmaxmem=128000
LD = mpixlf77
LDFLAGS = -g

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

hello.o : hello.f
	$(FC) -c $(FFLAGS) hello.f
```

Submitting jobs with mpirun

You can use “mpirun” to submit jobs. The Blue Gene mpirun is in /bgsys/drivers/ppcfloor/bin

Typical use:

mpirun -np 2048 -cwd `pwd` -exe your.x

common options: -args “list of arguments”
-env “VARIABLE=value”
-mode SMP,DUAL,VN

SMP mode : one MPI process per node, 4GB memory, up to 4 threads

DUAL mode : two MPI processes per node, 2GB memory per process, up to two threads

virtual-node mode : four MPI processes per node, 1GB limit per process; can’t create additional threads.

More details: mpirun -h (for help)

Limitations: one job per partition, limited partition sizes

Brief Tour of BG/P Software

/bgsys/drivers/ppcfloor (is a soft link)

comm/bin : MPI compile/link script
comm/include : MPI header files, dcmf and armci headers
comm/lib : MPI and messaging libraries (dcmf and armci)

dcmf = deep computing messaging framework
armci = aggregate remote memory copy interface

arch/include/common : BGP personality structures

arch/include/spi : system programming interface

Kernel_GetPersonality(…);
Kernel_PhysicalProcessorID();
DMA routines
UPC counter routines

gnu-linux/bin : GNU compilers and bin-utils for BGP
python

gnu-linux/powerpc-bgp-linux/sys-include : BGP headers

BG/P Personality

BG/P machine-specific data is returned in a personality structure (torus dimensions, coordinates, etc.).

```
#include <spi/kernel_interface.h>
#include <common/bgp_personality.h>
#include <common/bgp_personality_inlines.h>

_BGP_Personality_t personality;
int myX, myY, myZ, coreID;

Kernel_GetPersonality(&personality, sizeof(personality));
myX = personality.Network_Config.Xcoord;
myY = personality.Network_Config.Ycoord;
myZ = personality.Network_Config.Zcoord;
coreID = Kernel_PhysicalProcessorID();
node_config = personality.Kernel_Config.ProcessConfig;
pset_size = personality.Network_Config.PSetSize;
pset_rank = personality.Network_Config.RankInPSet;
BGP_Personality_getLocationString(&personality, location);
printf("MPI rank %d has torus coords <%d,%d,%d>  cpu = %d,
location = %s\n", rank, myX, myY, myZ, coreID, location);
```

Use GNU compilers for BGP personality structures
mpicc or powerpc-bgp-linux-gcc and specify the include path:
 -I/bgsys/drivers/ppcfloor/arch/include

Some Survival Tips

addr2line can really help you identify problems – it is the first pass method for debugging. Many kinds of failures give you an instruction address; addr2line will take that and tell you the source file and line number – just make sure you compile and link with -g.

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

addr2line -e your.x hex_address

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

Use grep and word-count (wc) to examine core files :
grep hex_address “core.*” | wc -l

You can get the instruction that failed by using objdump:
powerpc-bgp-linux-objdump -d your.x >your.dump
You can locate the instruction address in the dump file, and can at least find the routine where failure occurred, even without –g.

If your application exits without leaving a core file,
set the env variable BG_COREDUMPONEXIT=1

CoreProcessor Tool

coreprocessor.pl … coreprocessor perl script in :

/bgsys/drivers/ppcfloor/tools/coreprocessor

online help via : coreprocessor.pl –help

Can analyze and sort text core files, and can attach to hung processes for deadlock determination.

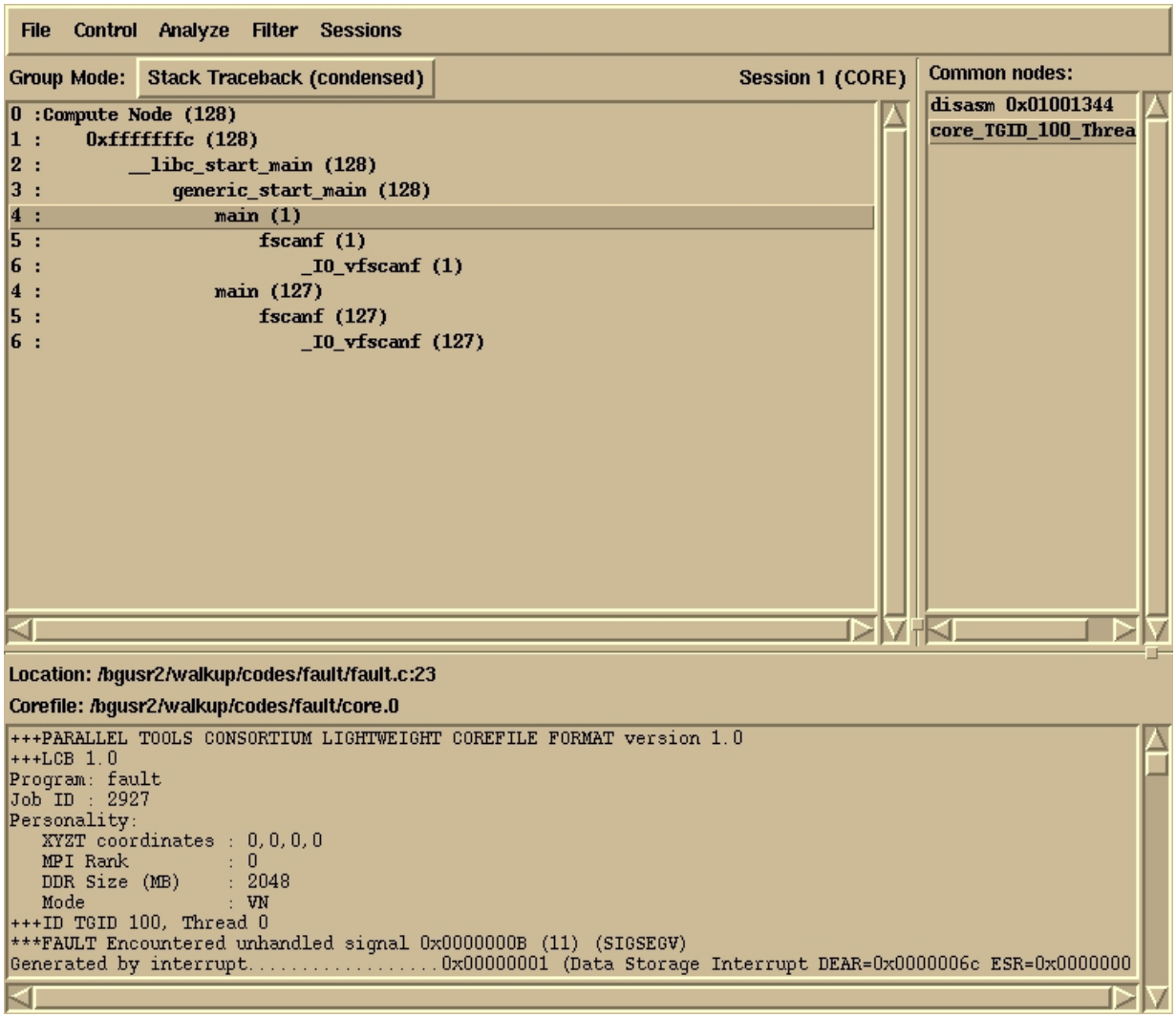
Normal use is “gui” mode, so set DISPLAY then:

coreprocessor.pl –c=/bgusr/username/rundir –b=your.x

click on “Select Grouping mode”,
select “Stack Trace (condensed)”
click on source statement of interest

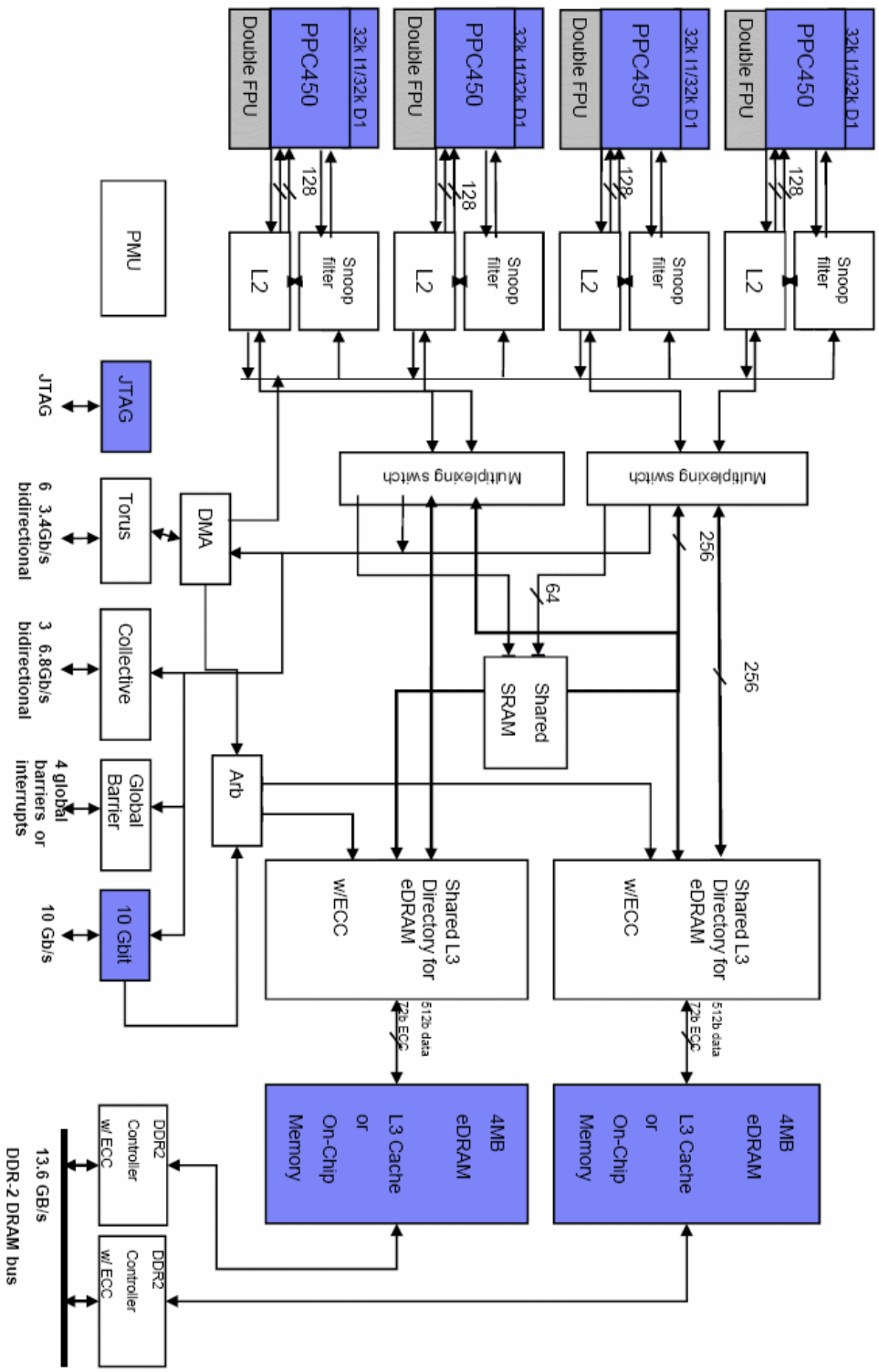
There is also a non-gui mode: coreprocessor.pl –help

Coreprocessor Example



In this example, one MPI rank failed in fault.c, line 23; the other 127 MPI ranks failed at a different source location.

BGP Chip Schematic Diagram



Powerpc-450 Processor

32-bit architecture at 850 MHz

one normal plus one multi-cycle integer unit

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, write-through for cache coherency, 4-cycle load to use

L2 Data cache : prefetch buffer, holds 15 128-byte lines
can prefetch up to 7 streams

L3 Data cache : 2x4 MB, ~50 cycles latency, on-chip

Memory : 2048 MB DDR2 at 425 MHz,
~104 cycles latency, ~16 GB/sec bandwidth limit

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, …)

Instruction Latencies and Throughputs

Instruction	latency	throughput/cycle
fadd	5 cycles	1
fmadd	5 cycles	1
fpmadd	5 cycles	1
fdiv	30 cycles	1/30

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

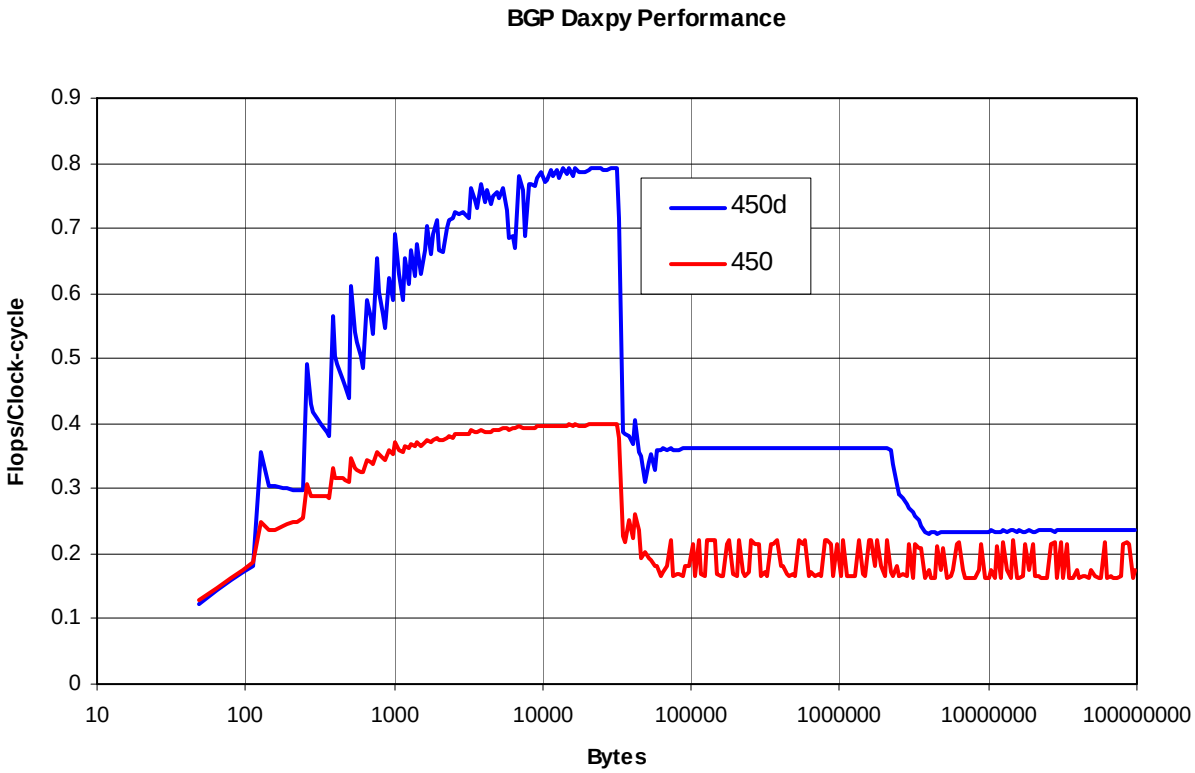
Practical limit is often loads and stores.

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. So the double-FPU instructions can help mainly for data in L1 cache, less help for data in L3 or memory.

Daxpy Performance on BG/P

```
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=450 => single FPU code, can also use 440
-qarch=450d => double FPU code, can use 440d

L1 cache edge at 32KB, L3 cache edge at about 2 MB
Data is for virtual-node mode, 4 processes per node.

Using IBM XL Compilers

Optimization levels:

Default optimization = none (very slow)
-O : good place to start, use with -qmaxmem=128000
-O2: same as -O
-O3 -qstrict : 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: BGP 450/450d; BGL 440/440d
-qarch=450 : generates standard powerpc floating-
point code, will use a single FPU
-qarch=450d : will try to generate double FPU code

On BG/P start with : -g -O -qarch=450 -qmaxmem=128000

Try : -O3 -qarch=450/450d
Try : -O4 -qarch=450/450d (or -O5 …)

-O4 = -O3 -qhot -qipa=level=1
-O5 = -O3 -qhot -qipa=level=2

OpenMP with IBM XL Compilers

OpenMP specification 2.5 is supported in C, C++, Fortran

Use -qsmp=omp to turn on OpenMP directive support.
Notes: -qsmp without qualifiers implies auto parallelization
-qsmp implies a minimum of -O2 and -qhot unless
unless you qualify it: -qsmp=omp:noopt

Use the *_r compiler invocation for compilation and linking: mpixlf_r, mpixlc_r, etc.; add -qsmp as a linker option.

SMP mode supports up to 4 threads
DUAL mode supports up to 2 threads
VIRTUAL-NODE (or QUAD) mode supports one thread

Threads are bound to hardware cores.

OpenMP uses a thread-private “stack”, taken from heap memory; default = 4MB, but can set this at run-time:

-env “XLSMPOPTS=stack=8000000” for stack = 8e6 bytes

See optimization and programming guide : opg.pdf
/opt/ibmcmp/xlf/bg/11.1/doc/en_US/pdf/opg.pdf

Some Blue Gene 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=450d -qlist –qsource
Read the assembler listing.

On BGP, alignment exceptions are fatal; can set BG_MAXALIGNEXP={-1,0, 1000=default}.

Easiest approach to double FPU is often use of 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=450d
(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.

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 primarily for data in L1 cache that can be accessed at stride-1.

One of the best potential improvements 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 = fpre(x) (good to 13 bits on BG/P)
> $x_{i+1} = x_i + x_i*(1.0 - a*x_i)$ (2 iterations for double)

The intrinsics are documented in the compiler pdf files:

Fortran language reference: lr.pdf
and : bg_using_xl_compilers.pdf

Example : Vector Reciprocal

```
#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

```
17|                                        CL.461:
23| 002304 lfpdx    7D284B9C   1     LFPL
24| 002308 lfpdx    7D08539C   1     LFPL
25| 00230C lfpdx    7CE85B9C   1     LFPL
26| 002310 lfpdx    7CC8639C   1     LFPL
29| 002314 fpre     03C0481C   1     FPRE
27| 002318 lfpdux   7CA8FBDC   1     LFPLU
30| 00231C fpre     03E0401C   1     FPRE
31| 002320 fpre     01A0381C   1     FPRE
32| 002324 fpre     0080301C   1     FPRE
33| 002328 fpre     0060281C   1     FPRE
35| 00232C fpnmsub  000957B8   1     FPNMSUB
36| 002330 fpnmsub  016857F8   1     FPNMSUB
37| 002334 fpnmsub  01875378   1     FPNMSUB
38| 002338 fpnmsub  00465138   1     FPNMSUB
39| 00233C fpnmsub  002550F8   1     FPNMSUB
41| 002340 fpmadd   001EF020   1     FPMADD
42| 002344 fpmadd   017FFAE0   1     FPMADD
43| 002348 fpmadd   018D6B20   1     FPMADD
44| 00234C fpmadd   004420A0   1     FPMADD
45| 002350 fpmadd   00231860   1     FPMADD
47| 002354 fpnmsub  01295038   1     FPNMSUB
48| 002358 fpnmsub  010852F8   1     FPNMSUB
49| 00235C fpnmsub  00E75338   1     FPNMSUB
50| 002360 fpnmsub  008650B8   1     FPNMSUB
51| 002364 fpnmsub  00655078   1     FPNMSUB
53| 002368 fpmadd   00000260   1     FPMADD
54| 00236C fpmadd   00AB5A20   1     FPMADD
55| 002370 fpmadd   00CC61E0   1     FPMADD
56| 002374 fpmadd   00421120   1     FPMADD
57| 002378 fpmadd   002108E0   1     FPMADD
59| 00237C stfpdx   7C074F9C   1     SFPL
60| 002380 stfpdx   7CA7579C   1     SFPL
61| 002384 stfpdx   7CC75F9C   1     SFPL
62| 002388 stfpdx   7C47679C   1     SFPL
63| 00238C stfpdux  7C27FFDC   1     SFPLU
 0| 002390 bc       4320FF74   0     BCT
```

Profile your Application

Standard profiling (prof, gprof) is available on BG/P, 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.

Tip: add –pg as a linker option (but not compiler option) to get a function-level profile with minimal overhead; analyze the gmon.out file using: gprof –p your.x gmon.out.0 > file. Adding –pg as a compiler option enables determination of the call graph, but adds overhead to each function call.

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

Gprof Example: GTC Flat profile

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

   …
```

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.

The tuning effort can focus on two main routines, and one should make use of a library for fast intrinsics, libmass.a.

Example : MPI Profile

```
Data for MPI rank 0 of 64, BGP in dual mode:
Times and statistics from MPI_Init() to MPI_Finalize().
---------------------------------------------------------------
MPI Routine                  #calls     avg. bytes      time(sec)
---------------------------------------------------------------
MPI_Comm_size                     2            0.0         0.000
MPI_Comm_rank                     2            0.0         0.000
MPI_Isend                       261      1208700.1         0.002
MPI_Irecv                       266      1194256.8         0.000
MPI_Wait                        520            0.0         1.037
MPI_Barrier                      11            0.0         0.000
MPI_Allreduce                    11           16.7         0.087
---------------------------------------------------------------
total communication time = 1.126 seconds.
total elapsed time       = 89.206 seconds.

---------------------------------------------------------------
Message size distributions:

MPI_Isend                  #calls    avg. bytes      time(sec)
                               26       40960.0          0.000
                               65      102400.0          0.000
                               10      947200.0          0.000
                              113     1377297.0          0.001
                               47     3034961.7          0.000

MPI_Irecv                  #calls    avg. bytes      time(sec)
                               26       40960.0          0.000
                               68      102400.0          0.000
                               12      947200.0          0.000
                              113     1377297.0          0.000
                               47     3034961.7          0.000

MPI_Allreduce              #calls    avg. bytes      time(sec)
                                5           8.0          0.082
                                6          24.0          0.005

---------------------------------------------------------------
Communication summary for all tasks:

  minimum communication time = 0.786 sec for task 31
  median  communication time = 1.249 sec for task 10
  maximum communication time = 4.651 sec for task 58
```

Link with libmpitrace.a, and run the application. You get a few small text files with a summary of times spent in MPI routines. Can optionally tag by instruction address, check communication locality, and record time-stamped traces showing the time-history of all MPI calls.

Scalar and Vector MASS Routines

Approximate cycle-counts per evaluation on BGL/BGP

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

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

Best performance is often with the vector routines, which 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.

/opt/ibmcmp/xlmass/bg/4.4/bglib/libmass.a
/opt/ibmcmp/xlmass/bg/4.4/bglib/libmassv.a

http://www.ibm.com/software/awdtools/mass/support

IBM ESSL Routines

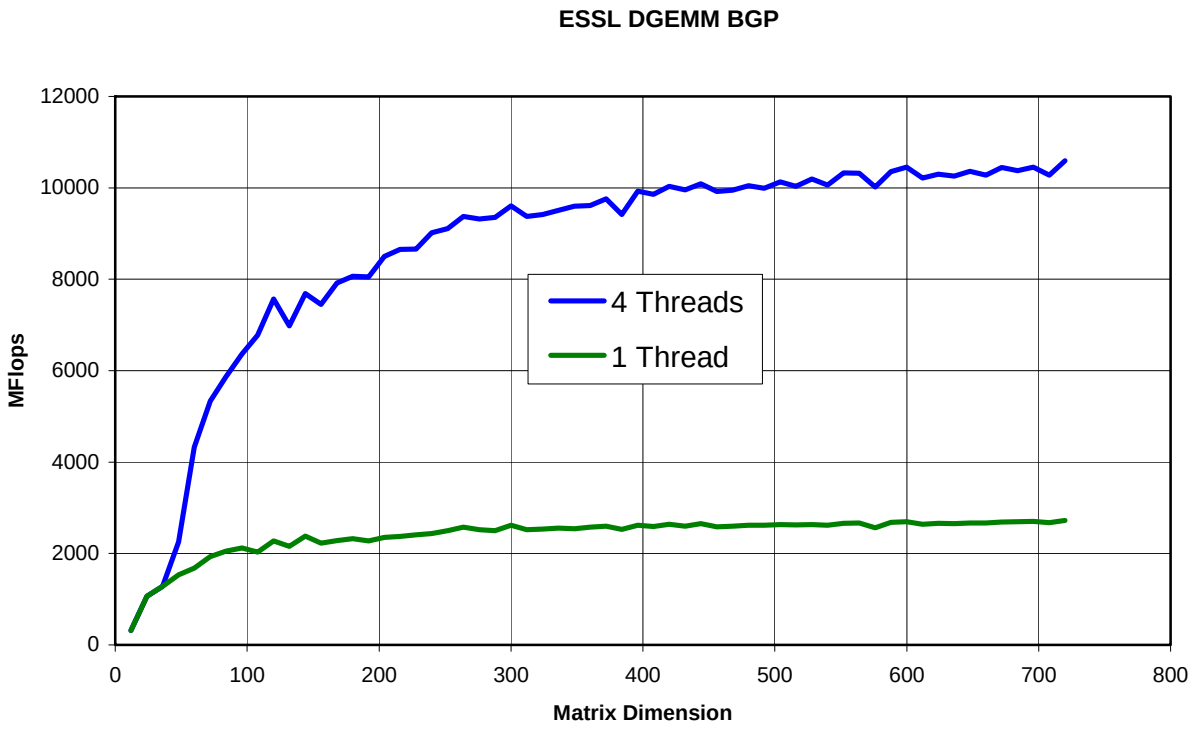
IBM ESSL for Blue Gene/P, normally installed in /opt/ibmmath/lib

libesslbg.a : single-threaded routines
libesslsmpbg.a : multi-threaded routines

For libesslbg.a, link with an _r version of XL compiler: mpixlf90_r, need to include threaded Fortran libraries

For libesslsmpbg.a, add –qsmp linker option.

Best for level 3 BLAS (matrix-matrix), and FFT routines.



Mapping onto the Torus Network

Default placement of MPI ranks is (x,y,z,t) order for all modes of operation: SMP, DUAL, Virtual-Node

You can specify a mapping at run time.

mpirun … -mapfile my.map (user-defined map file)
mpirun … -mapfile TXYZ (pre-defined ordering)

my.map has a line with "x y z t" for each MPI rank

Alternatively, env variable: BG_MAPPING can be used

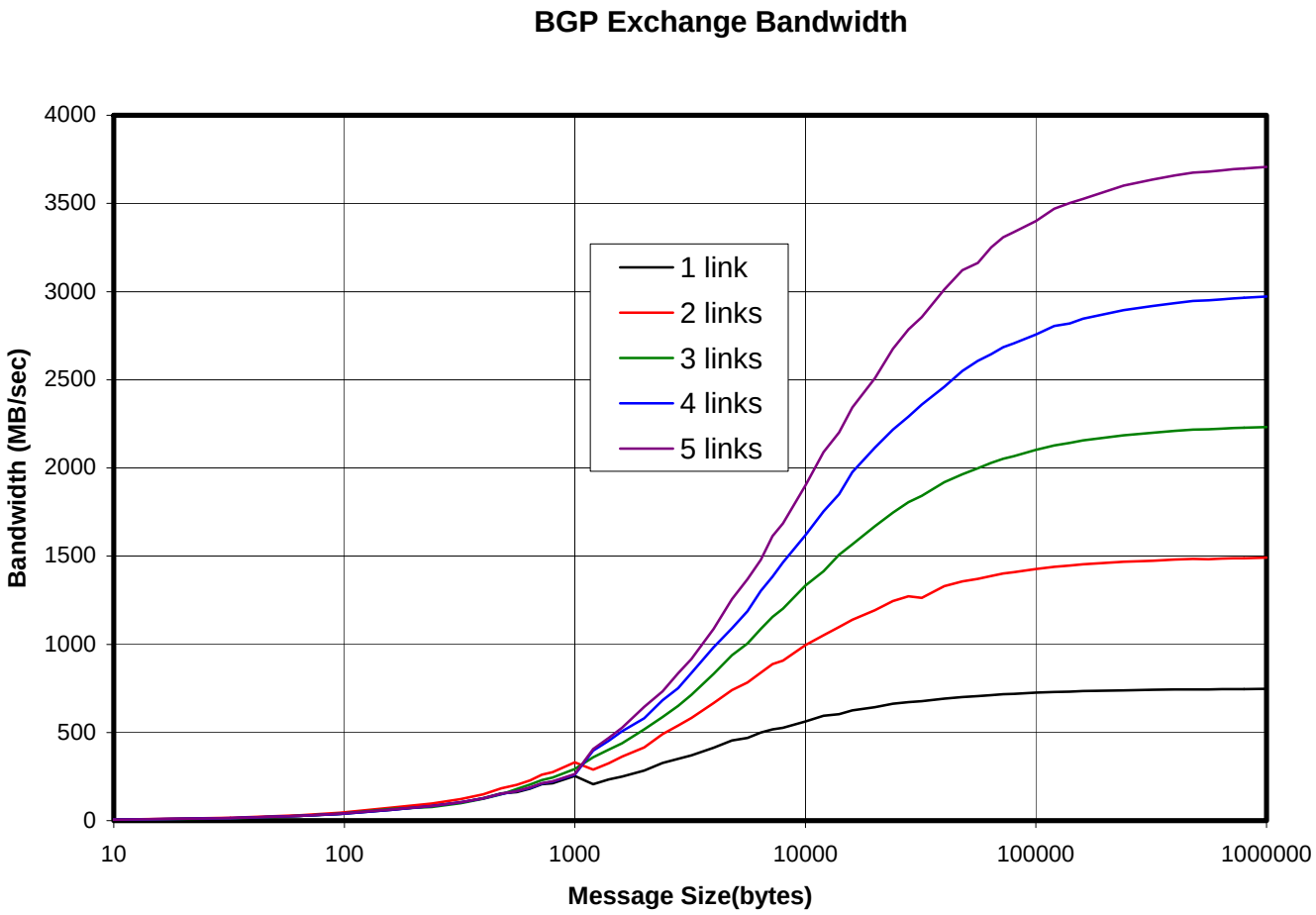
For dual-mode and virtual-node mode, it is frequently better to use TXYZ mapping, instead of the default. The TXYZ ordering fills up each node with MPI ranks, then moves to the next node.

For regular Cartesian-product logical process grids, you can often pick parameters to fit the machine perfectly:

2D example: 2048 MPI ranks, virtual-node mode
torus dimensions = 8x8x8 (one midplane)
TXYZ order gives effectively a 32x64 layout,
with one extra hop at the torus edges.

3D example: 8192 MPI ranks, virtual-node mode
torus dimensions = 8x16x16 (two racks)
can do 16x16x32, by doubling in x and then z

MPI Exchange on the Torus Network



The DMA on BGP makes it possible for MPI to get close to the hardware limit for communication on the torus.

MPI Environment variables:

DCMF_EAGER=20000 (sets the eager limit)
DCMF_COLLECTIVES=0 (disables all optimized collectives)
DCMF_{collective}=M; collective=BCAST,ALLREDUCE,etc.
DCMF_INTERRUPTS=1 turns on interrupt mode (not default)
DCMF_RECFIFO=bytes; default is 8 MB