Performance Tools and Optimization for Blue Gene/L
Example: GTC (Princeton Plasma Physics Lab)

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

With special thanks to Stephane Ethier
Princeton Plasma Physics Laboratory

1. Check computation vs. communication time
   Tools: MPI wrappers such as libmpitrace_c/f.a
   or libmpihpm_c/f.a (adds perf counters)

2. If compute bound, get more detailed information
   Tools: Xprofiler, gprof, libmpitrace_c/f.a

3. Check code generation (-qlist -qsource), and tune the code
   Tools: the compiler, libraries (MASS, MASSV)
Communication vs. Computation

Link with libmpitrace_c/f.a; run the code.

or link with libmpihpm_c/f.a -lbgl_perfctr.rts; run the code
(starts counters in mpi_init, stops them in mpi_finalizer)

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>3</td>
<td>0.0</td>
<td>0.000</td>
</tr>
<tr>
<td>MPI_Comm_rank</td>
<td>3</td>
<td>0.0</td>
<td>0.000</td>
</tr>
<tr>
<td>MPI_Recv</td>
<td>24</td>
<td>24.0</td>
<td>0.000</td>
</tr>
<tr>
<td>MPI_Sendrecv</td>
<td>2800</td>
<td>208777.8</td>
<td>4.438</td>
</tr>
<tr>
<td>MPI_Bcast</td>
<td>3</td>
<td>85.3</td>
<td>0.000</td>
</tr>
<tr>
<td>MPI_Gather</td>
<td>1626</td>
<td>104.2</td>
<td>0.253</td>
</tr>
<tr>
<td>MPI_Reduce</td>
<td>36</td>
<td>207.2</td>
<td>0.002</td>
</tr>
<tr>
<td>MPI_Allreduce</td>
<td>675</td>
<td>77040.1</td>
<td>8.507</td>
</tr>
</tbody>
</table>

total communication time = 13.201 seconds.
total elapsed time = 384.498 seconds.
top of the heap address = 81.703 MBytes.

...  

BG/L counters avg Flops = 146.738 MFlops

Communication summary for all tasks:

  minimum communication time = 6.665 sec for task 222
  median communication time = 9.656 sec for task 371
  maximum communication time = 14.052 sec for task 40

For this application, the focus is on computation.
### BGL Counter Statistics

**Link with libmpihpm_c/f.a -lbgl_perfctr.rts**

<table>
<thead>
<tr>
<th>Counter</th>
<th>Avg_Value</th>
<th>Min_Value</th>
<th>Min_Rank</th>
<th>Max_Value</th>
<th>Max_Rank</th>
</tr>
</thead>
<tbody>
<tr>
<td>CYCLES</td>
<td>2.6871e+11</td>
<td>2.6870e+11</td>
<td>207</td>
<td>2.6915e+11</td>
<td>0</td>
</tr>
<tr>
<td>ADD+SUB</td>
<td>8.7109e+09</td>
<td>8.6736e+09</td>
<td>500</td>
<td>8.7519e+09</td>
<td>176</td>
</tr>
<tr>
<td>MUL+DIV</td>
<td>1.3175e+10</td>
<td>1.3153e+10</td>
<td>389</td>
<td>1.3201e+10</td>
<td>225</td>
</tr>
<tr>
<td>FMADD</td>
<td>1.7267e+10</td>
<td>1.7241e+10</td>
<td>414</td>
<td>1.7293e+10</td>
<td>170</td>
</tr>
<tr>
<td>FPMADD</td>
<td>1.0704e+05</td>
<td>1.0704e+05</td>
<td>3</td>
<td>1.0704e+05</td>
<td>3</td>
</tr>
<tr>
<td>LD_DOUBLE</td>
<td>7.4458e+09</td>
<td>7.3934e+09</td>
<td>500</td>
<td>7.5010e+09</td>
<td>176</td>
</tr>
<tr>
<td>ST_DOUBLE</td>
<td>3.8258e+09</td>
<td>3.8110e+09</td>
<td>389</td>
<td>3.8417e+09</td>
<td>225</td>
</tr>
<tr>
<td>LD_QUAD</td>
<td>9.6443e+09</td>
<td>9.6221e+09</td>
<td>222</td>
<td>9.6661e+09</td>
<td>74</td>
</tr>
<tr>
<td>ST_QUAD</td>
<td>9.7205e+09</td>
<td>9.7094e+09</td>
<td>499</td>
<td>9.7314e+09</td>
<td>47</td>
</tr>
<tr>
<td>L3_HIT</td>
<td>2.4224e+09</td>
<td>2.2870e+09</td>
<td>240</td>
<td>2.5209e+09</td>
<td>342</td>
</tr>
<tr>
<td>L3_MISS</td>
<td>3.8211e+08</td>
<td>3.8035e+08</td>
<td>419</td>
<td>3.8307e+08</td>
<td>176</td>
</tr>
<tr>
<td>XM_PKTS</td>
<td>1.9073e+05</td>
<td>8.5200e+02</td>
<td>0</td>
<td>2.1803e+05</td>
<td>1</td>
</tr>
<tr>
<td>XP_PKTS</td>
<td>1.9069e+05</td>
<td>8.8000e+02</td>
<td>7</td>
<td>2.1797e+05</td>
<td>4</td>
</tr>
<tr>
<td>YM_PKTS</td>
<td>1.1751e+06</td>
<td>1.1406e+06</td>
<td>391</td>
<td>1.2342e+06</td>
<td>0</td>
</tr>
<tr>
<td>YP_PKTS</td>
<td>1.5518e+06</td>
<td>1.5193e+06</td>
<td>120</td>
<td>1.6583e+06</td>
<td>504</td>
</tr>
<tr>
<td>ZM_PKTS</td>
<td>1.5031e+05</td>
<td>3.0520e+03</td>
<td>8</td>
<td>8.3984e+05</td>
<td>322</td>
</tr>
<tr>
<td>ZP_PKTS</td>
<td>1.9601e+05</td>
<td>1.9550e+03</td>
<td>480</td>
<td>1.2993e+06</td>
<td>378</td>
</tr>
</tbody>
</table>

Average MFlops = 146.980 per MPI task  
Total GFlops = 75.254  
BGL FP op count = (add+sub) + (mul+div) + 2*fmadd + 4*fpmadd  
MFlops/GFlops values are suspect due to counter limitations.  
FPU and LSU averages are per MPI process.  
L3 and packet averages are per BGL node.

Blue Gene has many more counters, but interpretation is not simple. You can use other tools, such as Jim Sexton’s APC, to get more counter data.
Overall “report card” – from Power5 Counters

CPI = 1.29 clock cycles per instruction completed
IPC = 0.78 instructions completed per clock cycle

Instruction mix:
29.16 % loads (load references)/(instructions completed)
19.15 % floating-point loads (fp loads)/(instructions completed)
9.64 % stores (store references)/(instructions completed)
5.42 % floating-point stores (fp stores)/(instructions completed)
33.92 % floating-point operations (fp ops)/(instructions completed)
36.57 % integer operations (fx ops)/(instructions completed)
3.71 % branches (branches issued)/(instructions completed)

Cache and TLB:
3.15 % loads that miss L1 Data cache / (load refs)
19.60 % stores that miss L1 Data cache / (store refs)
1.74 % loads from L2 / (load refs)
0.06 % loads from L3 / (load refs)
0.13 % loads from memory / (load refs)
2.51 % cycles in data "table-walks" / (run_cycles)

Floating Point:
700.4 MFlops per process
10.57 % theoretical floating-point peak
13.84 % hardware floating-point peak (fp_ops / 2*run_cycles)
38.95 % fp ops that are fmadds (fmadds/fp_ops)
0.78 % fp ops that are divides (fp divides/fp_ops)
6.41 % max fraction of cycles in division
0.33 % fp ops that are square_roots (square-roots/fp_ops)
2.74 % max fraction of cycles in square_roots

Prefetchs:
 0.59 % L1 prefetches / (load refs)
 0.37 % L2 prefetches / (load refs)

Branches:
5.39 % branches mispredicted (mispredictions/branches issued)

Stalls:
25.48 % cycles stalled for loads/stores
14.10 % cycles stalled for L1 data cache miss
39.44 % cycles stalled in the floating-point pipeline
1.59 % cycles stalled for floating-point division or square-roots
4.41 % cycles stalled in the integer unit
Profiling: gprof and Xprofiler

Compile and link with -g -pg

Optionally link with libmpitrace_c/f.a, to limit profiler output. Get gmon.out for rank 0, and ranks with minimum, median, and maximum communication times. Would get one gmon.out for each app unless you use the MPI wrappers or directly call mondisable().

```
gprof: gprof your.exe gmon.out.n >gprof_n.txt
       Produces a “flat profile” … subroutine-level, and a call-graph section.
```

Xprofiler can provide statement-level timing data: clock ticks tied to source lines.

```
Set your DISPLAY variable (requires X-windows)
Xprofiler your.exe gmon.out.n
Select “Report” then “Flat Profile”
Click on a routine and select “Code Display”, then “Show Source Code”
```
Gprof Example: GTC Flat profile

Each sample counts as 0.01 seconds.

<table>
<thead>
<tr>
<th>% cumulative</th>
<th>time</th>
<th>seconds</th>
<th>self</th>
<th>calls</th>
<th>s/call</th>
<th>self</th>
<th>total</th>
<th>s/call</th>
<th>name</th>
</tr>
</thead>
<tbody>
<tr>
<td>37.43</td>
<td>144.07</td>
<td>144.07</td>
<td>201</td>
<td>0.72</td>
<td>0.72</td>
<td>chargei</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>25.44</td>
<td>241.97</td>
<td>97.90</td>
<td>200</td>
<td>0.49</td>
<td>0.49</td>
<td>pushi</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6.12</td>
<td>265.53</td>
<td>23.56</td>
<td>_xldintv</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4.85</td>
<td>284.19</td>
<td>10.66</td>
<td>cos</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4.49</td>
<td>301.47</td>
<td>17.28</td>
<td>sinl</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4.19</td>
<td>317.61</td>
<td>16.14</td>
<td>200</td>
<td>0.08</td>
<td>0.08</td>
<td>poisson</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3.79</td>
<td>332.18</td>
<td>14.57</td>
<td>_pxldmod</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3.55</td>
<td>345.86</td>
<td>13.68</td>
<td>__ieee754_exp</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2.76</td>
<td>356.48</td>
<td>10.62</td>
<td>BGLML_Messager_advance</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2.23</td>
<td>365.05</td>
<td>8.57</td>
<td>200</td>
<td>0.04</td>
<td>0.04</td>
<td>shifti</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1.23</td>
<td>369.80</td>
<td>4.75</td>
<td>400</td>
<td>0.01</td>
<td>0.01</td>
<td>smooth</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0.79</td>
<td>372.85</td>
<td>3.05</td>
<td>exp</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0.53</td>
<td>374.88</td>
<td>2.03</td>
<td>200</td>
<td>0.01</td>
<td>0.01</td>
<td>field</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0.27</td>
<td>375.91</td>
<td>1.03</td>
<td>finitel</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0.24</td>
<td>376.84</td>
<td>0.93</td>
<td>TreeAllreduce_fifo</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0.24</td>
<td>377.77</td>
<td>0.93</td>
<td>_exp</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0.20</td>
<td>378.54</td>
<td>0.77</td>
<td>_sin</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0.15</td>
<td>379.12</td>
<td>0.58</td>
<td>readfun_torus</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0.12</td>
<td>379.59</td>
<td>0.47</td>
<td>1</td>
<td>0.47</td>
<td>0.47</td>
<td>poisson_initial</td>
<td></td>
<td></td>
<td></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.
Flat Profile using MPI wrapper libraries

Compile with -qdebug=function_trace
Link with libmpitrace_c/f.a
Run with env variable FLAT_PROFILE=yes

Elapsed-time flat profile for MPI task 0.

<table>
<thead>
<tr>
<th>%Time</th>
<th>Self(seconds)</th>
<th>Inclusive</th>
<th>#Calls</th>
<th>Routine</th>
</tr>
</thead>
<tbody>
<tr>
<td>45.05</td>
<td>173.2084</td>
<td>173.2084</td>
<td>201</td>
<td>chargei</td>
</tr>
<tr>
<td>43.92</td>
<td>168.8568</td>
<td>168.8568</td>
<td>200</td>
<td>pushi</td>
</tr>
<tr>
<td>4.24</td>
<td>16.2962</td>
<td>16.8397</td>
<td>200</td>
<td>poisson</td>
</tr>
<tr>
<td>3.64</td>
<td>13.9843</td>
<td>13.9843</td>
<td>200</td>
<td>shifti</td>
</tr>
<tr>
<td>1.55</td>
<td>5.9536</td>
<td>5.9883</td>
<td>400</td>
<td>smooth</td>
</tr>
<tr>
<td>0.75</td>
<td>2.8856</td>
<td>2.8856</td>
<td>200</td>
<td>field</td>
</tr>
<tr>
<td>0.44</td>
<td>1.6768</td>
<td>1.7237</td>
<td>1</td>
<td>load</td>
</tr>
<tr>
<td>0.20</td>
<td>0.7638</td>
<td>0.8712</td>
<td>1</td>
<td>snapshot</td>
</tr>
<tr>
<td>0.14</td>
<td>0.5435</td>
<td>0.5435</td>
<td>1</td>
<td>poisson_initial</td>
</tr>
<tr>
<td>0.03</td>
<td>0.1301</td>
<td>384.6211</td>
<td>1</td>
<td>gtc</td>
</tr>
<tr>
<td>0.01</td>
<td>0.0548</td>
<td>0.0852</td>
<td>1</td>
<td>setup</td>
</tr>
<tr>
<td>0.01</td>
<td>0.0462</td>
<td>0.0462</td>
<td>1</td>
<td>set_random_zion</td>
</tr>
<tr>
<td>0.01</td>
<td>0.0327</td>
<td>0.0428</td>
<td>32155</td>
<td>boozer2x</td>
</tr>
<tr>
<td>0.01</td>
<td>0.0321</td>
<td>0.0433</td>
<td>32155</td>
<td>boozer2z</td>
</tr>
<tr>
<td>0.01</td>
<td>0.0308</td>
<td>0.0308</td>
<td>2375</td>
<td>spcpft</td>
</tr>
<tr>
<td>0.01</td>
<td>0.0268</td>
<td>0.0268</td>
<td>1</td>
<td>read_input_params</td>
</tr>
</tbody>
</table>

Uses compiler-generated hooks for entry/exit of each routine. Get elapsed-time data for routines compiled with the function trace option. No interrupts.

This method can be useful because you get inclusive elapsed times, but Xprofiler is recommended because it provides a lot more information.
Xprofiler Example : GTC pushi routine

Line   ticks source
115         do m=1,mi 
116          657    r=sqrt(2.0*zion(1,m))
117          136    rinv=1.0/r 
118          34     ii=max(0,min(mpsi-1,int((r-a0)*delr)))
119          55     ip=max(1,min(mflux,1+int((r-a0)*d_inv)))
120          194    wp0=real(ii+1)-(r-a0)*delr 
121          52     wp1=1.0-wp0 
122          104    tem=wp0*temp(ii)+wp1*temp(ii+1)
123          86     q=q0+q1*r*a inv+q2*r*r*a inv*a inv
124          166    qinv=1.0/q 
125          68     cost=cos(zion(2,m))
126          18     sint=sin(zion(2,m))
127          104    b=1.0/(1.0+r*cost) 
130          1     g=1.0 
131          3     gp=0.0 
134          1     ri=0.0 
135          3     rip=0.0 
136          97     dbdp=-b*b*cost*rinv 
137          18     dbdt=b*b*r*sint 
... 

clock ticks are 100 per second

Lots of expensive operations : sqrt, division, cos, sin, …
Blue Gene default routines for sqrt, cos, sin, exp, … are very slow,
from GNU libm.a. Use libmass.a, but better to vectorize and use
libmassv.a routines.

For scalar MASS:
   link with libmass.a -Wl,--allow-multiple-definition

For vector MASSV, can try -qhot or hand-code.
Adding Vector MASS Routines

```c
#ifdef USE_VECTOR
  do m = 1, mi
    r_vec(m) = 2.0*zion(1,m)
    z_vec(m) = zion(2,m)
  end do
  call vssqrt(r_vec, r_vec, mi)
  call vsrec(rinv_vec, r_vec, mi)
  call vssincos(sin_vec, cos_vec, z_vec, mi)
  do m = 1, mi
    rfac = rw*(r_vec(m) - rc)
    z_vec(m) = - rfac**6
  end do
  call vsexp(z_vec, z_vec, mi)
#endif

do m=1,mi
  #ifdef USE_VECTOR
    r = r_vec(m)
    rinv = rinv_vec(m)
  #else
    r=sqrt(2.0*zion(1,m))
    rinv=1.0/r
  #endif
  ii=max(0,min(mpsi-1,int((r-a0)*delr)))
  ip=max(1,min(mflux,1+int((r-a0)*d_inv)))
  wp0=real(ii+1)-(r-a0)*delr
  wp1=1.0-wp0
  temp=wp0*temp(ii)+wp1*temp(ii+1)
  q=q0+q1*r*ainv+q2*r*r*ainv*ainv
  qinv=1.0/q
  #ifdef USE_VECTOR
    cost = cos_vec(m)
    sint = sin_vec(m)
  #else
    cost=cos(zion(2,m))
    sint=sin(zion(2,m))
  #endif
  ...
```
Xprofiler Example : GTC chargei routine

```fortran
  46         do larmor=1,4
  47             rdum=delr*max(0.0,min(a1-a0,r+rhoi*pgyro(larmor,ipjt)-a0))
  48             ii=max(0,min(mpsi-1,int(rdum)))
  49             wp1=rdum-real(ii)
  50             wpion(larmor,m)=wp1
  51
  52          ! particle position in theta
  53             tflr=thetatmp+rhoi*tgyro(larmor,ipjt)
  54
  55          ! inner flux surface
  56             im=ii
  57             tdum=pi2_inv*(tflr-zetatmp*qtinv(im))+10.0
  58             tdum=(tdum-aint(tdum))*delt(im)
  59             j00=max(0,min(mtheta(im)-1,int(tdum)))
  60             jtion0(larmor,m)=igrid(im)+j00
  61             wtion0(larmor,m)=tdum-real(j00)
  62
  63          ! outer flux surface
  64             im=ii+1
  65             tdum=pi2_inv*(tflr-zetatmp*qtinv(im))+10.0
  66             tdum=(tdum-aint(tdum))*delt(im)
  67             j01=max(0,min(mtheta(im)-1,int(tdum)))
  68             jtion1(larmor,m)=igrid(im)+j01
  69             wtion1(larmor,m)=tdum-real(j01)
  70         enddo
```

Issues : pipelining, type-conversion, register spills, ...

Tuning : check the compiler listing!
Compiler Listing Example : chargei routine

Source section
55 | ! inner flux surface
56 | im=ii
57 | tdum=pi2_inv*(tflr-zetatmp*qtinv(im))+10.0
58 | tdum=(tdum-aint(tdum))*delt(im)
59 | j00=max(0,min(mtheta(im)-1,int(tdum)))
60 | jtion0(larmor,m)=igrid(im)+j00
61 | wtion0(larmor,m)=tdum-real(j00)

Register section
GPR's set/used:   ssss ssss ssss s-ss  ssss ssss ssss ssss
FPR's set/used:   ssss ssss ssss ssss  ssss ssss ssss ssss
                      ssss ssss ssss ss--  ---- ---- ---s s--s
CCR's set/used:   ssss ssss

Assembler section
58| 000DDC fmr       3     LRFL     fp1=fp15
69| 000DE0 frsp      1     CVLS     fp3=fp3,fcr
69| 000DE4 fmsubs    4     FMSS     fp0=fp3,fp2,fp0,fcr
69| 000DE8 stfsux    0     STFSU    gr3,wtion1(gr3,gr20,0)=fp0
69| 000DEC stw       0     ST4A     #SPILL29(gr31,428)=gr3
49| 000DF0 lfd       0     LFL      fp2=#MX_CONVS1_4(gr31,216)
49| 000DF4 fadd      1     AFL      fp0=fp2,fp17,fcr
49| 000DF8 frsp      4     CVLS     fp0=fp0,fcr
49| 000DFC fmsubs    4     FMSS     fp0=fp0,fp30,fp4,fcr
50| 000E00 stfsux    0     STFSU    gr4,wpion(gr4,gr5,0)=fp0
50| 000E04 stw       0     ST4A     #SPILL52(gr31,520)=gr4
58| 000E08 bl        0     CALLN    fp1=_xldintv,0,fp1,gr1,gr31,...
59| 000E0C mullw     2     M        gr3=gr19,gr15
58| 000E10 rlwinm    1     SLL4     gr5=gr15,2
60| 000E14 mullw     2     M        gr4=gr18,gr15
64| 000E18 addi      1     AI       gr15=gr15,1,ca"
60| 000E1C lwzx      1     L4A     gr0=igrid(gr24,gr4,0)
...

Issues : pipelining, function call for aint(x), register spills

Tuning : replace aint(x) with real(int(x)), eliminates function call, improves pipelining
GTC Performance on Blue Gene/L

Original code: main loop time = 384 sec (512 nodes, coprocessor)
Tuned code: main loop time = 244 sec (512 nodes, coprocessor)

Factor of ~1.6 performance improvement by tuning.

Scaling measurements on BG/W (Blue Gene at Watson)

Weak scaling, relative performance per processor:

<table>
<thead>
<tr>
<th>#nodes</th>
<th>coprocessor</th>
<th>virtual-node</th>
</tr>
</thead>
<tbody>
<tr>
<td>512</td>
<td>1.000</td>
<td>0.974</td>
</tr>
<tr>
<td>1024</td>
<td>1.002</td>
<td>0.961</td>
</tr>
<tr>
<td>2048</td>
<td>0.985</td>
<td>0.963</td>
</tr>
<tr>
<td>4096</td>
<td>1.002</td>
<td>0.956</td>
</tr>
<tr>
<td>8192</td>
<td>1.009</td>
<td>0.935</td>
</tr>
<tr>
<td>16384</td>
<td>0.968</td>
<td>NAN</td>
</tr>
</tbody>
</table>

Outstanding scaling and exceptionally good use of the second cpu.

See an increase of time spent in mpi_sendrecv and mpi_allreduce at the largest process counts.

Remaining issue: results are NANs for 32K cpus