Optimizing IRS Benchmark for IBM Cell Processors

Balasubramanya Bhat

Suraj Kasi Satyanarayana

Project Goals

The goal of this project is to run the IRS (*Implicit Radiation Solver*) parallel benchmark [1][2][3] on multiple IBM cell processor [5] nodes using MPI and accelerate the runtime of the benchmark on each node using the multiprocessing capability available within the IBM cell processor (PPE and SPEs).

Purpose of IRS Benchmark

The name IRS stands for *Implicit Radiation Solver* [1][2][3]. IRS solves a diffusion equation on a three-dimensional, block structured mesh. The single CPU instruction mix for this benchmark is roughly 40%load/store, 18% floating point, 31% fixed-point, and 11% branch.

Current Status

We ported the IRS benchmark to run IBM Cell processor using MPI and profile the program using gprof. We also completed the initial identification of hotspot functions. The full set of log files are given at the website.

Porting of IRS benchmark code into IBM Cell processors

We downloaded the IRS benchmark version 1.0 from the LLNL website. We modified different Makefiles and some perl scripts to change the compilation to use powerpc compilers, compiler / linker options, include / library paths etc. Finally we could build the working IRS benchmark which could run on the PSXX machines. We also enabled MPI and the *gprof* [6] instrumentation on the IRS and got the execution profile of the benchmark on IBM Cell processor. Note that at each processor, the IRS runs only on the PPE.

Runtime Analysis of the benchmark on a single IBM Cell Processor

We ran the IRS benchmark on a single IBM Cell processor using ‘mpirun –np 2’, with *gprof* profiling enabled. We have an open issue in terms on running MPI programs on multiple PSXX nodes which is discussed in the later section.

Following command is used to run the benchmark using MPI on IBM cell processor:

```
mpirun -np 2 ./irs /home/skasisa/irs.1.0/decks/zrad3d -child_io_off -k 00008MPI -
def NDOMS=8
```

Following are the top 10 functions in terms of the execution times on the IBM cell processor as recorded by *gprof.*

```
Each sample counts as 0.01 seconds.
```

```
%     cumulative  self        self        total
time  seconds     seconds     calls       s/call      s/call      name
48.70 20.96       20.96       3304        0.01        0.01        rmatmult3
19.73 29.45       8.49        6           1.42        5.54        MatrixSolveDriver
4.00  31.17       1.72        3280        0.00        0.00        icdot
3.67  32.75       1.58        24          0.07        0.07        DiffCoef
3.14  34.10       1.35        76          0.02        0.02        volcal3d
2.32  35.10       1.00        4908        0.00        0.00        norml2
1.23  35.63       0.53        48          0.01        0.01        newMatrix
1.18  36.14        0.51       8284        0.00         0.00       setpz1
```

For a complete list of log files and commands used for running IRS, please see the following webpage http://sites.google.com/site/irsoncell/deliverables/on-ibm-cell

Based on the above timing details obtained from the gprof the functions were analyzed for parallelism. The analysis is as given below.

MatrixSolveDriver: The call graph for this function is as shown below.

```
[6]    79.5   14.61   36.50       6         MatrixSolveDriver [6]
              0.00   27.60    6608/6608        rmatmult [7]
              3.21    0.00    6560/6560        icdot [12]
              2.30    0.02    9816/9816        norml2 [14]
              1.37    0.03    3328/3436        rbndcom [18]
              1.08    0.00   16328/16568       setpz1 [22]
              0.87    0.00   16424/16664       setpz2 [24]
              0.00    0.03      48/48          cblkbc [102]
              0.00    0.00      60/19129       FunctionTimer_finalize [103]
              0.00    0.00      60/19129       FunctionTimer_initialize [116]
              0.00    0.00      48/152         TimeStepControl_find [217]
```

It is very clear that the majority of this function time corresponds to the rmatmult function that is shown in the gprof output.

1. The function rmatmult calls rmatmult3 or rmatmult2 based on the dimensions and this functions are responsible for matrix multiplication. In the above case rmatmult3 is called because the test ran with a dimension greater than 2.

   rmatmult is called multiples times as shown.

   ```
   for ( iblk = 0 ; iblk < my_nblk ; iblk++ ) {
       rmatmult( &domains[iblk], &rblk[iblk], x[iblk], r[iblk] ) ;
     }
   ```

   It is clearly seen that there is no relation between the iterations of the for loop and each iteration can be performed parallely. rmatmult is called 6608 times and hence it is a potential hot spot, by parallelizing this function we can get better performance.

2. icdot: Although called multiple times this cannot be parallelized because parameters that are passed on depends on the previous icdot calculation. Hence icdot is ruled out.

3. norml2: Even this one has dependencies between iterations hence cannot be used. There are loops inside the function that can be parallelized but the overhead of data transfer might supersede the benefits of parallelism.

4. rbndcom: Used for MPI communication hence ignored.

5. setpz1 and setpz2 : These functions are called multiple times inside MatrixSolveDriver as shown below.

```
for ( iblk = 0 ; iblk < my_nblk ; iblk++ ) {
      setpz1( p[iblk], 0.0, &domains[iblk] ) ;
      setpz1( t[iblk], 0.0, &domains[iblk] ) ;
      setpz2( t[iblk], 0.0, &domains[iblk] ) ;
    }
```

This can be easily parallelized

So from the above analysis 3 hot spots are identified.

- rmalmult
- setpz1
- setpz2

These 3 functions account for 40% of the total execution time, hence parallelizing these would lead to better performance.

Other optimization in rmalmult

Consider the code

```
for ( kk = kmin ; kk < kmax ; kk++ ) {
    for ( jj = jmin ; jj < jmax ; jj++ ) {
      for ( ii = imin ; ii < imax ; ii++ ) {
        i = ii + jj * jp + kk * kp ;

      }
    }
  }
```

The following loop is used for the matrix multiplication. The $2n^3$ multiplications can be reduced to (n^2 + n) multiplications by changing the code as shown below.

```
for ( kk = kmin ; kk < kmax ; kk++ ) {
   kk_intermediate = kk * kp;
    for ( jj = jmin ; jj < jmax ; jj++ ) {
      jj_intermediate = jj * jp
      for ( ii = imin ; ii < imax ; ii++ ) {
        i = ii + jj_intermediate + kk_intermediate.

      }
```

```
    }
  }
```

Inside the loop tiling can be used to improve the cache usage.

Timeline for the project

Dates	Task	Status
Oct25 – Nov2	Understanding the benchmark code	Completed
Nov3 – Nov9	Porting the benchmark onto the IBM cell processors without any modifications to the code.	Completed
Nov10-Nov15	Converting the hotspot functions into kernels and running it on the cell processor.	In Progress
Nov16- Nov24	Testing and debugging the code.	Not Started
Nov 24 -	Document the steps and write report.	Not Started

Deliverables

The deliverables for the project includes the modified source code for the IRS bench mark, project report, original and the improved timing metrics.

Please refer to the following website for deliverables related to HW5.
http://sites.google.com/site/irsoncell/deliverables/on-ibm-cell

References

[1] https://asc.llnl.gov/computing_resources/purple/archive/benchmarks/irs/
[2] http://moss.csc.ncsu.edu/~mueller/cluster/cluster03/g1/irs.pdf
[3] https://asc.llnl.gov/computing_resources/purple/archive/benchmarks/irs/irs.readme.html
[4] http://moss.csc.ncsu.edu/~mueller/cluster/ps3/
[5] http://www-03.ibm.com/technology/cell/index.html