HPCG Performance Improvement on the K computer ~10min. brief~

Kiyoshi Kumahata, Kazuo Minami
RIKEN AICS

HPCG BoF #273

SC14 New Orleans
Evaluate Original Code 1/2

Weak Scaling Measurement

- Conditions
  - \(128^3\) DoF/CPU
  - 8 threads/CPU
  - 10min. for CG
  - Typical Compile Option

- Good scalability was obtained!!
- So tunings for parallel performance is not necessary!!

GFLOPS values are from “Total with convergence and optimization phase overhead” in the YAML file

9,500 GFLOPS in 32,768 CPUs (used 262,144 cores)
0.23% of Peak Performance
Parallel efficiency was 99%
Evaluate Original Code 2/2

- Procedures time ratio in the total CG running time by profiler
- 98% of total time consists by major 2 procedures (only computation)

Tuning for Single CPU performance is necessary!!
Essence of Matrix Memory Allocation Part (Original)

```c
for(local_int_t i=0; i< localNumberOfRows; ++i) {
    mtxIndL[i] = new local_int_t [numberOfNonzerosPerRow];
    matrixValues[i] = new double [numberOfNonzerosPerRow];
    mtxIndG[i] = new global_int_t [numberOfNonzerosPerRow];
}
```

- Memory for storing a matrix row is allocated separately
- Each row information are arranged discontinuous in memory space. It disturbs efficient cache memory utilization when computation

### Memory Space

- `A.matrixValues[0]`
- `A.mtxIndL[1]`
- `A.mtxIndL[nrow-1]`
- `A.matrixValues[nrow-1]`
- `A.matrixValues[1]`

Discontinuousness is emphasized in this figure.
Tune1: Continuous Memory 2/4

Essence of Matrix Memory Allocation Part (Modified)

```c
int total_size;
local_int_t* templ = new local_int_t [total_size];
double* tempd = new double [total_size];
global_int_t* tempg = new global_int_t[total_size];
int ofset = 0;
for (local_int_t i=0; i<localNumberOfRows; ++i) {
    mtxIndL[i] = tmpl + ofset
    matrixValues[i] = tmpd + ofset
    mtxIndG[i] = tempg + ofset
    ofset += max_nnz;
}
```

Allocate all once

Assign pointer for each row

- Every row information are arranged continuously

Memory Space

- `A.matrixValues[0]` to `A.matrixValues[nrow-1]`
- `A.mtxIndL[0]` to `A.mtxIndL[nrow-1]`
**Tune1: Continuous Memory 3/4**

**Essence of Backward Loop of SYMGS (Original)**

```c
for(int i=nrow-1; i>=0; i--){
    double* curValues = A.matrixValues[i];
    int*    curIndices = A.mtxIndL[i];
    int     curNZ = A.nonzerosInRow[i];
    double  curDiag = matrixDiagonal[i][0];

double sum = rv[i];
    for(int j=0; j<curNZ; j++){
        int curCol = curIndices[j];
        sum -= curValues[j]* xv[curCol];
    }
    sum   += xv[i] * curDiag;
xv[i] = sum / curDiag;
}
```

In original code, loop direction of **inner loop** in backward loop of SYMGS is reverse
- Outer ‘i’ loop goes backward direction
- Inner ‘j’ loop goes forward direction

After outer ‘i’ loop switched to next iteration, memory address referred by **inner ‘j’ loop** first iteration will not on cache because inner ‘j’ loop goes to reverse direction to outer ‘i’ loop
Tune1: Continuous Memory 4/4

for(int i=nrow-1; i>=0; i--){
    double* curValues = A.matrixValues[i];
    int* curIndices = A.mtxIndL[i];
    int curNZ = A.nonzerosInRow[i];
    double curDiag = matrixDiagonal[i][0];

    double sum = rv[i];
    for(int j=curNZ-1; j>=0; j--){
        int curCol = curIndices[j];
        sum -= curValues[j]* xv[curCol];
    }
    sum   += xv[i] * curDiag;
    xv[i] = sum / curDiag;
}

Essence of of Backward Loop of SYMGS (Modified)

Inner ‘j’ loop direction is not constraint to be forward direction. So it can be reversed
• Outer ‘i’ loop goes backward direction
• Inner ‘j’ loop goes backward direction

By reversing inner ‘j’ loop direction, memory address referred by inner ‘j’ loop first iteration will be on cache!

And prefetch mechanism can predict easily required memory address.
To avoid side effect of coloring (cache thrashing), we employed new way using block.  

1. Mesh structure is divided into lots of blocks. Each block has a lot of rows. And number of rows in block is as same as possible to avoid work imbalance.

2. In here, color is assigned to a block instead of a node. And different color is assigned into neighboring blocks.

3. The, there are no data dependencies between blocks in same color.
Tune2: Coloring for SYMGS 2/2

There are no data dependencies between blocks in same color.
But dependencies occur among the rows in same block.
Therefore, thread parallelism is applied to block, code is modified into right figure.

```java
for(int ic=0; ic<ncolor; ic++){
    #pragma omp parallel for
    for(int ib=0; ib<nblock[ic]; ib++){
        for(int i=st[ic][ib]; i<=ed[ic][ib]; ++i){
            ...Innermost loop...
        }
    }
}
```
Sample for SPMV (Original)

```c
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;
}
```

- Complicated access path for matrix nonzero information via pointer
- Software pipeline don’t work well for short loop

SYMGS also has same problem
double* \texttt{val} = A.matrixValues[0];
local_int_t* \texttt{index} = A.mtxIndL[0];
for(local_int_t \texttt{i}=0; \texttt{i}<\texttt{nrow}-1; \texttt{i}=\texttt{i}+2){
    \texttt{id1} = (\texttt{i} )*max_nnz;
    \texttt{id2} = (\texttt{i}+1)*max_nnz;
    \texttt{sum1} = 0.0;
    \texttt{sum2} = 0.0;
    for(\texttt{j}=0; \texttt{j}<\texttt{max_nnz}; \texttt{j}++){
        \texttt{sum1} += \texttt{val}[\texttt{id1}+\texttt{j}] * \texttt{xv}[\texttt{index}[\texttt{id1}+\texttt{j}]];
        \texttt{sum2} += \texttt{val}[\texttt{id2}+\texttt{j}] * \texttt{xv}[\texttt{index}[\texttt{id2}+\texttt{j}]];
    }
    \texttt{yv[\texttt{i}]} = \texttt{sum1};
    \texttt{yv[\texttt{i}+1]} = \texttt{sum2};
}
Modify Sample for SPMV

```c
double* val = A.matrixValues[0];
local_int_t* index = A.mtxIndL[0];
for(local_int_t i=0; i<nrow-1; i=i+2){
  id1 = (i )*max_nnz;
  id2 = (i+1)*max_nnz;
  sum1 = 0.0;
  sum2 = 0.0;
  for(int j=0; j<max_nnz; j++){
    sum1 += val[id1+j] * xv[index[id1+j]];
    sum2 += val[id2+j] * xv[index[id2+j]];
  }
  yv[i ] = sum1;
  yv[i+1] = sum2;
}
```

1. Unroll Full
2. Software Pipelined
3. Unroll 2

Avoiding short loop is necessary
- make several SPMV for various max_nnz
- Full unrolling innermost loop j

To increase software pipelined operations
- 2 unrolling loop i
Tuning after ISC14

1. Parameter adjustment
   - Running environment parameter on the K
   - Block size
2. Code Refinement for OptimizeProblem.cpp to decrease overhead
Summary: Latest Tuning Effect

Employ tuning ways
- Continuous Memory
- Coloring for SYMGS multithreading with blocking
- Loop optimization
- Parameter adjustment
- Code refinement for OptimizeProblem

Good improve obtained

Speed Up x19.6 !!