HPCG Performance Improvement on the K computer
～short brief～

Kiyoshi Kumahata, Kazuo Minami
RIKEN AICS

HPCG BoF@Room15

SC15 Austin
Evaluate Original Code 1/2

Weak Scaling Measurement

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

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

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
• 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!!
Tune1: Continuous Memory 1/4

**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**

Discontinuousness is emphasized in this figure

### Diagram:
- `A.matrixValues[0]`
  - Red: double
  - Blue: local_int_t

### Code Explanation:
```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];
}
```
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 offset = 0;
for (local_int_t i=0; i<localNumberOfRows; ++i){
    mtxIndL[i] = tmpl + offset
    matrixValues[i] = tmpd + offset
    mtxIndG[i] = tempg + offset
    offset += 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]`

```
<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

max number of nonzeros for a row in matrix
Tune1: Continuous Memory 3/4

Essence of 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

On Cache
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.
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.

Add middle loop to iterate block.
And parallelize block loop by inserting a directive:

```c
#pragma omp parallel for
for(int ic=0; ic<ncolor; ic++){
    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)

```cpp
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

SYMGS also has same problem
Tune3: Loop Optimization 2/4

Memory Space (Original, Discontinuous)

Original Code
Discontinuous Memory Space

Array storing pointers

Fixed Skip

Tuned
Continuous Memory Space

Memory Space (Modified, Continuous)


Tuned Continuous Memory Space
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;
}

double* Val = A.matrixVal[0];
int*    Idx = A.matrixIdx[0];
int*    Nz = A.nonzerosInRow

for(local_int_t i=0; i<nrow; i++){
    double sum = 0.0;
    int cur_nnz = Nz[i];
    int top     = i*max_nnz;

    for(int j=0; j<nz; j++)
        sum += Val[top+j] * xv[Idx[top+j]];
    yv[i] = sum;
}
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;
}
```

**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
Other Tunings

1. Parameter adjustment
   • Running environment parameter on the K
   • Block size

2. Code Refinement for OptimizeProblem.cpp to decrease overhead

![Diagram showing time comparison before and after optimization]

- OptimizeProblem Time
- CG Time
- Optimization Overhead

<table>
<thead>
<tr>
<th>Time [s]</th>
<th>for ISC</th>
<th>after ISC</th>
</tr>
</thead>
<tbody>
<tr>
<td>Optimize Phase Time</td>
<td>5.0</td>
<td>1.5</td>
</tr>
</tbody>
</table>

65% faster
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