



HPCG on the K computer

K.Kumahata, K.Minami, N.Maruyama



Tuesday, March 25, 2014

RIKEN Advanced Institute for
Computational Science



RIKEN ADVANCED INSTITUTE FOR COMPUTATIONAL SCIENCE



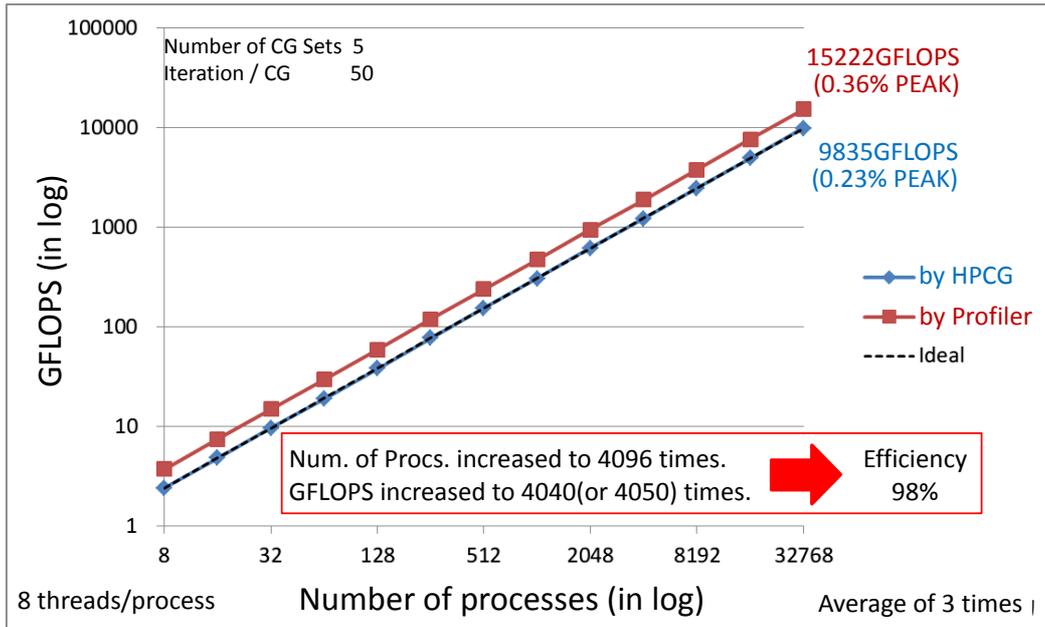
Today's menu

2

- Weak scaling measurement on the K computer
 - Performance
 - Communication costs trend
 - Hot spots
- Performance improvement on a single CPU
 - Continuous memory arrangement
 - SYMGS multi-threading
- Our requests for the HPCG

Weak scaling measurement on the K computer

- Code
 - As Is
 - Modified two kinds variables of “global_int_t” and “local_int_t”, for large number of processes
- Compile option
 - Typical compile option on the K
- Case
 - Local domain dimension was 104x104x104.
(by default hpcg.dat)
 - Number of CG sets was fixed 5 for test.



- Measured GFLOPS from 8 processes to 32768 processes
 - by HPCG Measured by HPCG code (from YAML file)
 - by Prof. Measured by profiler
- Good parallel performance was obtained until 32768 procs.

GFLOPS difference between HPCG and Profiler is 1.5 times.
↓
Cause of it is difference of floating point operation counting way.

By profiler

- All FLOP in measurement range will be counted.
- Division and SQRT are counted as multiple FLOP.

Over 10 FLOP

By HPCG

- Major FLOP of theoretical necessary will be counted.
- Minor FLOP was ignored.

```

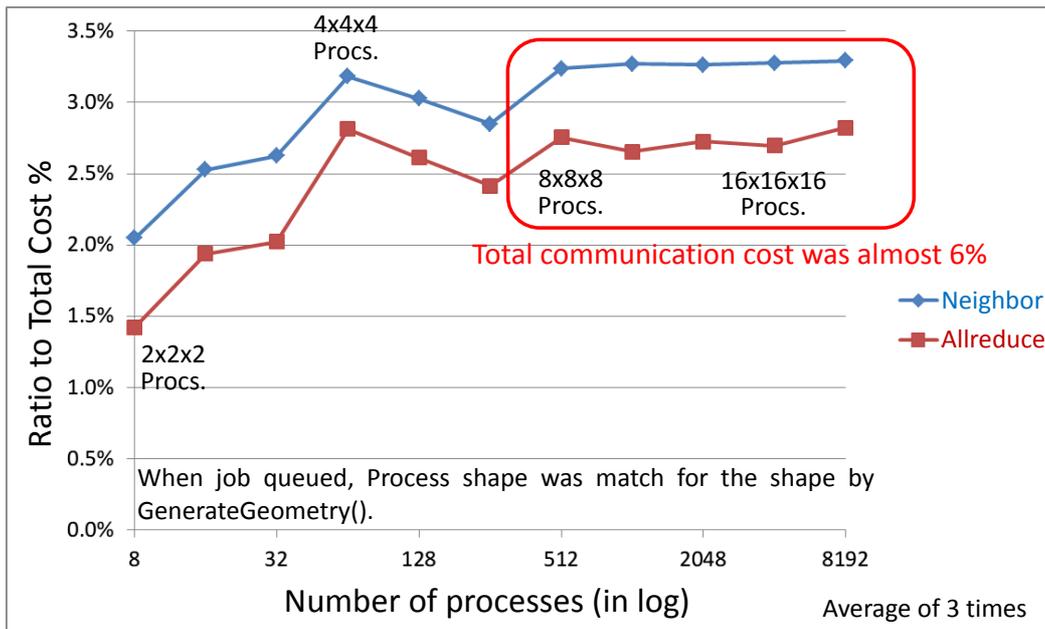
 $\vec{x} = \text{CG}(\mathbf{A}, \vec{b}, \vec{x}_0, \epsilon, \text{max})$ 
SPMV ()
WAXPY ()
DotProduct
for(k=1; k<max && !conv; k++){
  MG ()
  DotProduct ()
  WAXPY ()
  SPMV ()
  . . .
}
    
```

FLOP is ignored in an initial part

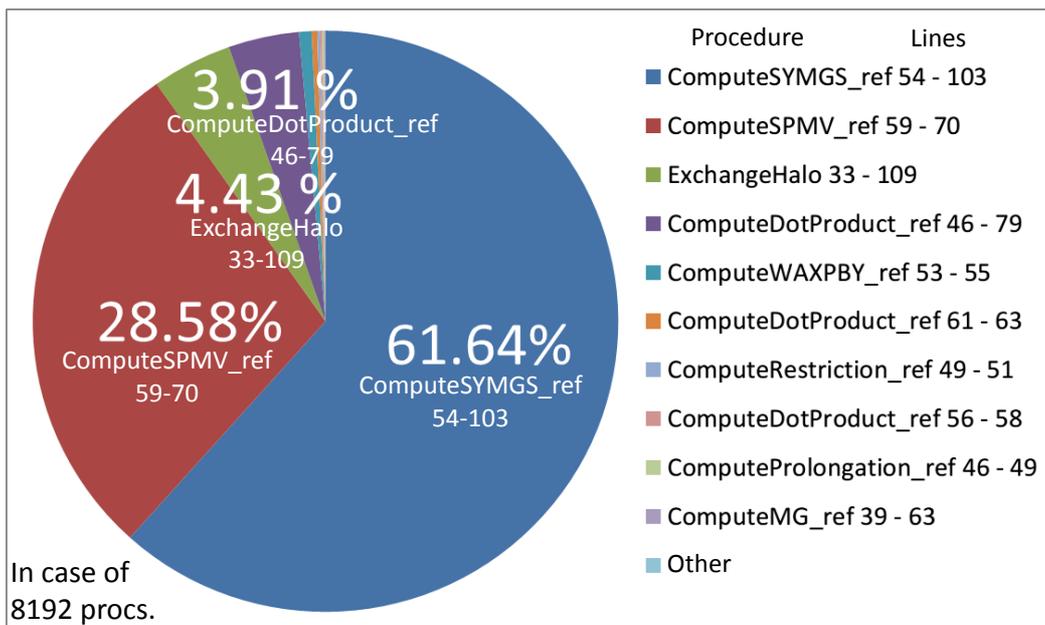
FLOP is counted in iterations

```

 $\vec{x} = \text{SYMGS}(\mathbf{A}, \vec{b}, \vec{x})$  ForwardLoop
for(i=0; i<nrow; i++){
  . . .
  for(j=0; j<nz[i]; j++){
    col = A.mtxIndL[i][j]
    Counted sum -= val[j] * x[col];
  }
  sum += x[i] * Diag[i];
  sum = sum / Diag[i];
}
Ignored
    
```



- MPI cost ratio to the total CG running cost by profiler
- Total communication ratio was saturated into about 6% with 512 processes or more.
- Study has been continue.



- Procedures costs ratio to the total CG running cost by profiler
- 98% of total cost consists of major 4 procedures (including communication)
 - Only 2 procedures occupy 90% of the total cost (only calculation)
- All cases of number of processes show such trend.

- Good parallel performance was obtained until 32768 processes.
 - Number of processes increased to 4096 times.
 - GFLOPS increased to 4050 times.
 - Efficiency was 98%.
- We expect the MPI cost will be “acceptable” in furthermore large processes.
 - MPI cost ratio of furthermore large processes is under studying.
- 90% of total cost consist of major 2 operational procedures.
 - We are trying to improve those procedures performance on a single CPU first.

Performance improvement on a single CPU Continuous memory arrangement

GenerateProblem.cpp (original)

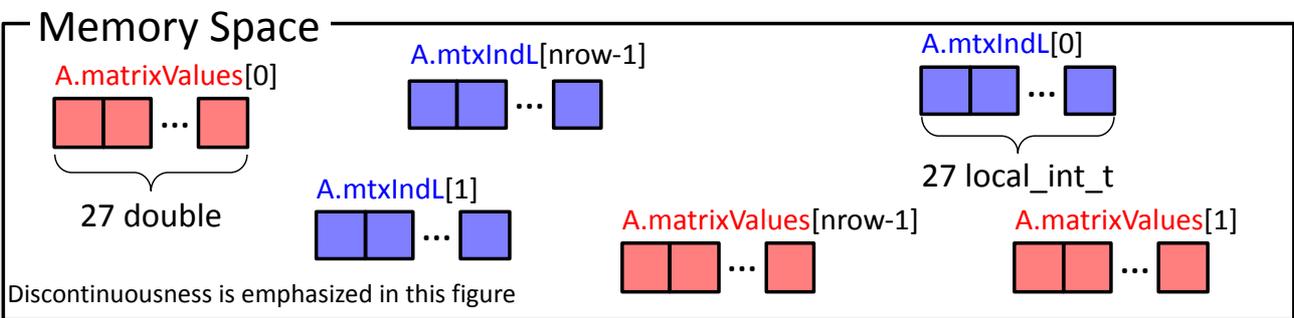
```

108: // Now allocate the arrays pointed to
109: for (local_int_t i=0; i< localNumberOfRows; ++i) {
110:     mtxIndL[i] = new local_int_t [numberOfNonzerosPerRow];
111:     matrixValues[i] = new double [numberOfNonzerosPerRow];
112:     mtxIndG[i] = new global_int_t [numberOfNonzerosPerRow];
113: }
    
```

=27



- Memory for storing a matrix rows is allocated separately in the source.
- Each row information are arranged discontinuous.



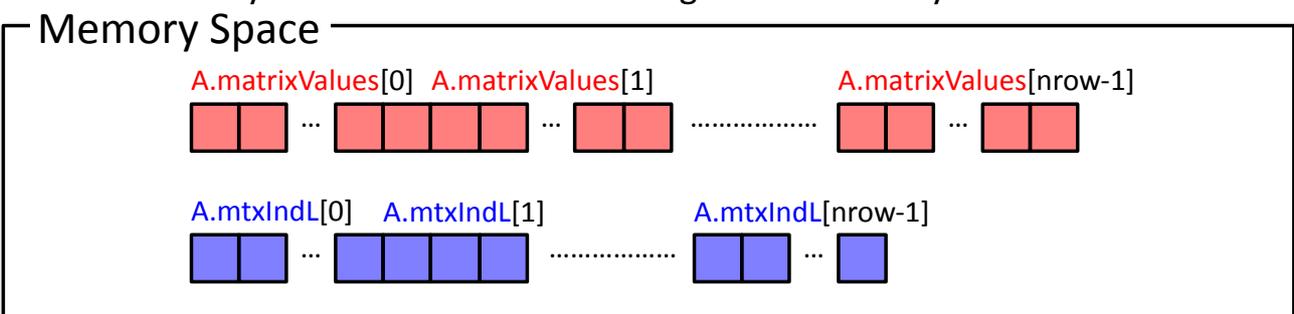
Modified

```

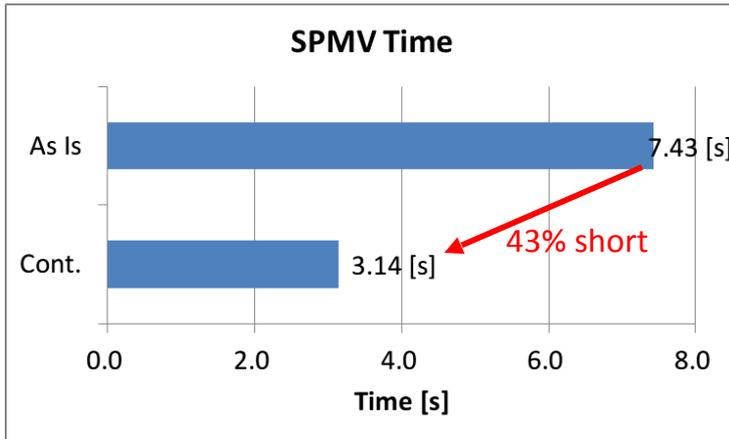
108: // Now allocate the arrays pointed to
109: int size = numberOfNonzerosPerRow * localNumberOfRows;
110: local_int_t*  tmp1 = new local_int_t [size];
111: double*      tmpd = new double      [size];  Allocate once all
112: global_int_t* tmpg = new global_int_t[size];
113:
114: for (local_int_t i=0; i< localNumberOfRows; ++i) {
115:     mtxIndL[i] = &(tmp1[i*numberOfNonzerosPerRow]);
116:     matrixValues[i] = &(tmpd[i*numberOfNonzerosPerRow]);
117:     mtxIndG[i] = &(tmpg[i*numberOfNonzerosPerRow]);
118: }
    
```



- Every row information are arranged continuously.



CASE	SPMV Time (sec)	Memory Throughput (GB/sec)	L2 Throughput (GB/sec)	L1D Miss Ratio (/Load, Store)	L2 Miss Ratio (/Load, Store)
As Is	7.429	25.99	25.72	5.59%	5.56%
Continuous	3.137	48.28	55.88	4.88%	4.13%



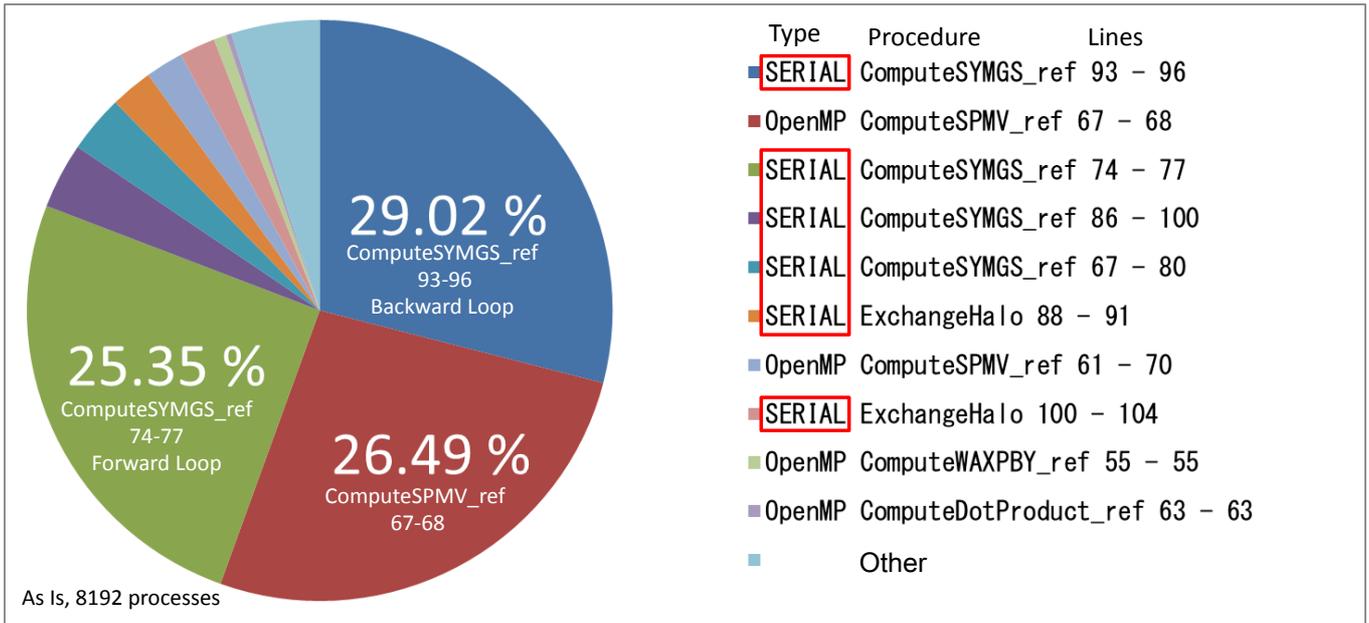
For SPMV called from CG directly, running time, throughput and cache miss ratio was measured. (except for called by MG)

Continuous memory arrangement improved the memory throughput and the cache miss ratio.

SPMV running time was decreased in 43%.

8 processes, 8 threads/process

Performance improvement on a single CPU SYMGS multi-threading



- The loops cost ratio to the total CG cost measured by profiler and loop running type
- SYMGS loops are not parallelized.



- SYMGS multi-threading is necessary.

Ex) ComputeSYMGS_ref () Forward loop

```

for(int i=0; i<nrow; 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;
}

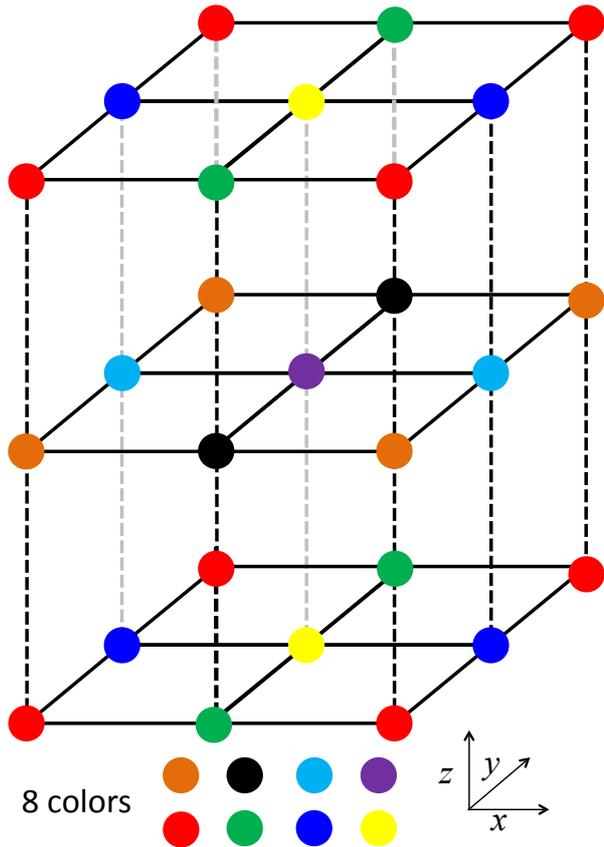
```

Recurrence!!
Can't parallelize

- There are recurrences between load and store for the vector xv.
- Cannot be parallelized by just insert a directive.



- It is necessary to separate load and store of same column by coloring.



- 27 points stencil (using diagonal points)
- 8 colors required for avoiding the recurrences

```
for(int i=0; i<nrow; i++){
    ...Innermost loop...
}
```

Loop structure of SYMGS

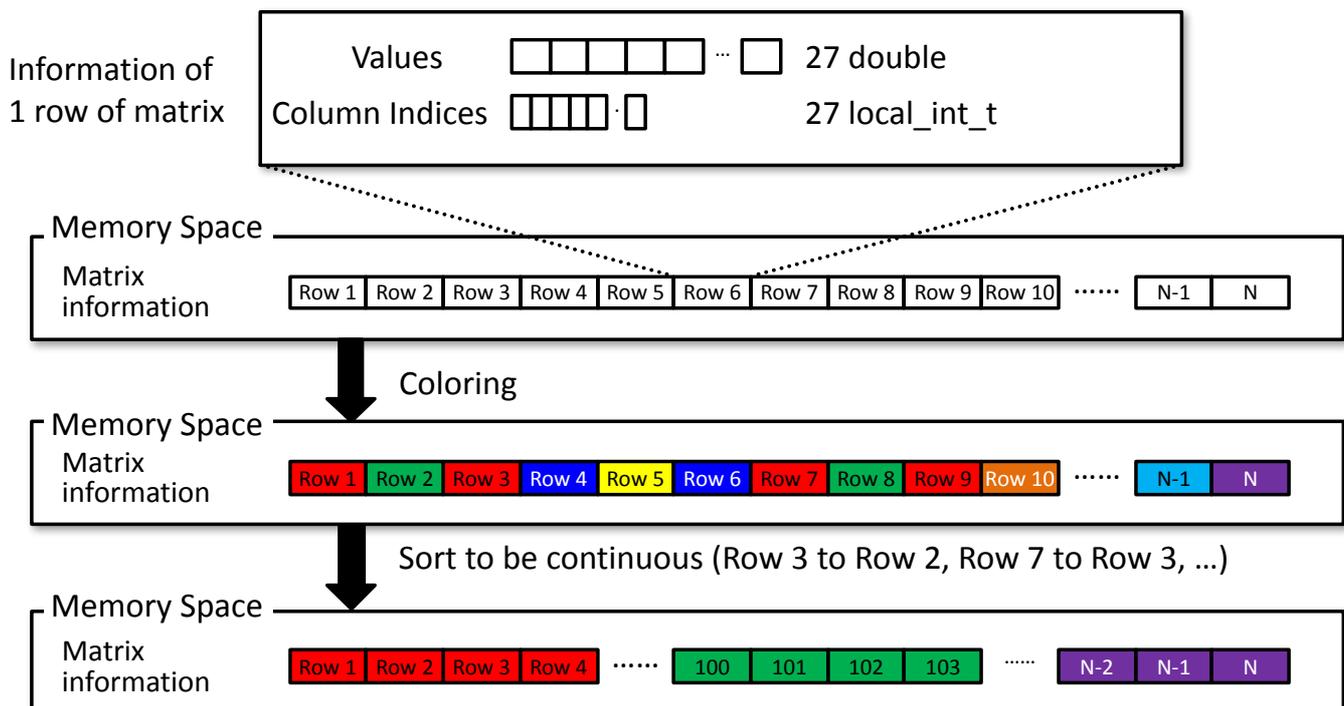


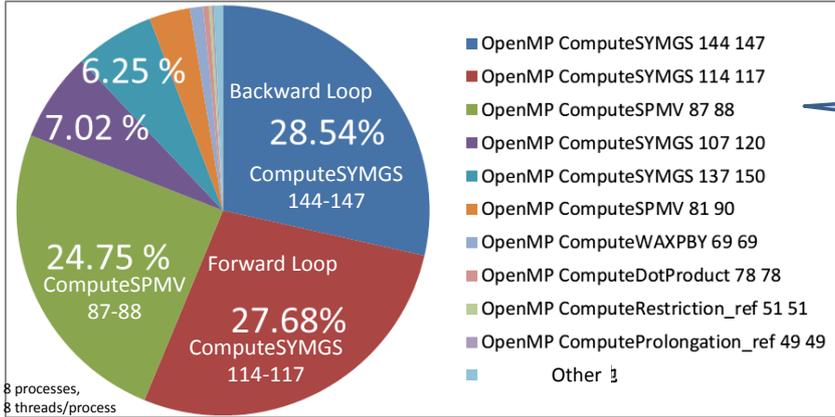
Modify for coloring

```
Add outer loop to iterate color
for(int ic=0; ic<8; ic++){
    Parallelize by directive
    #pragma omp parallel for
    for(int i=st; i<=ed ++){
        ...Innermost loop...
    }
}
```

Attention!!
 In this time, we are using the structural advantages of the test problem temporarily!
 Of course, we have the correct coloring way.

Corresponding to the coloring , memory accesses have been sequential by sorting row information in the memory.

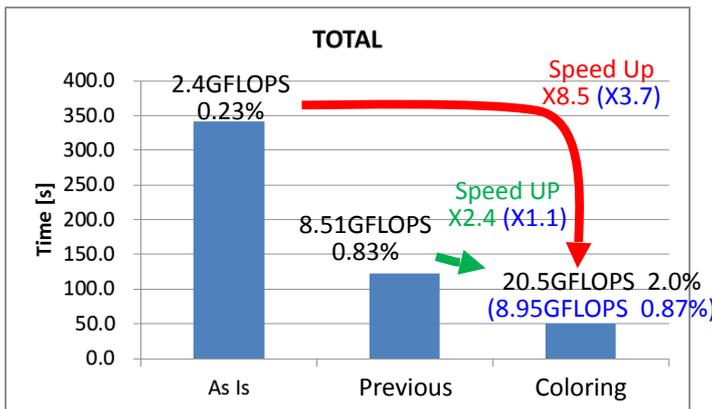
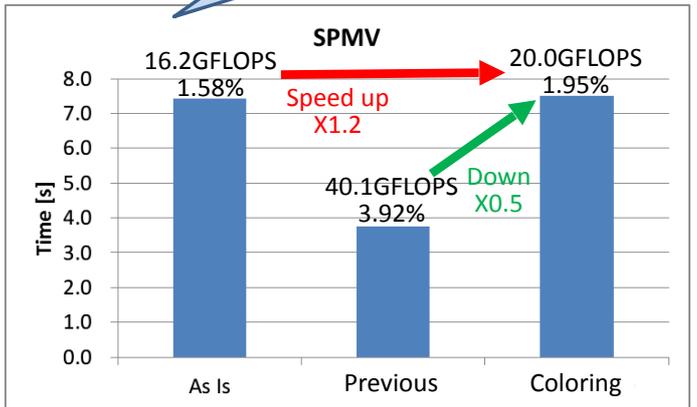
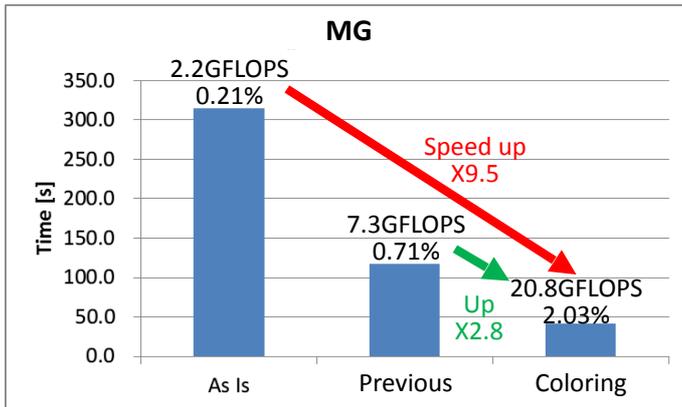




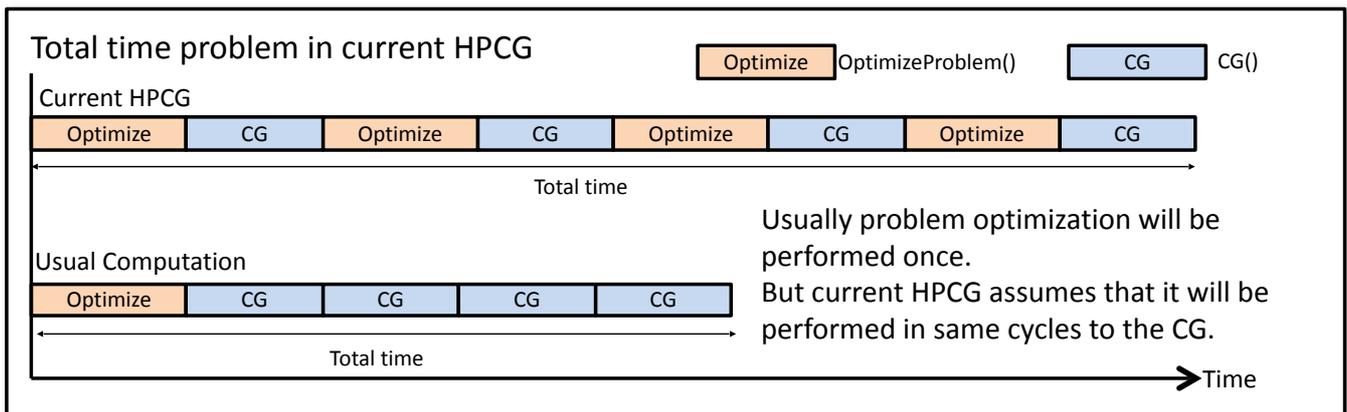
Problem size was change to 112^3 from 104^3 to adjust for the coloring

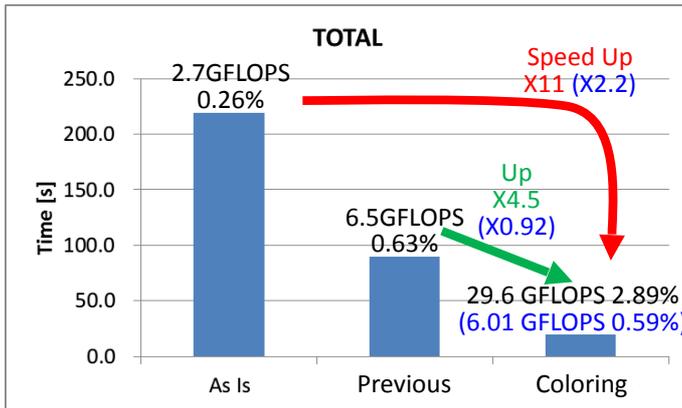
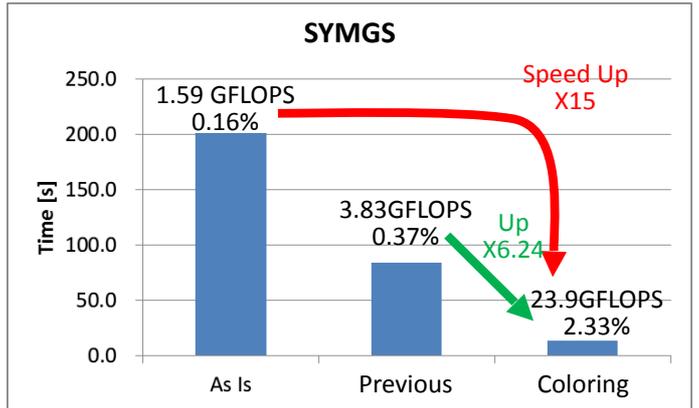
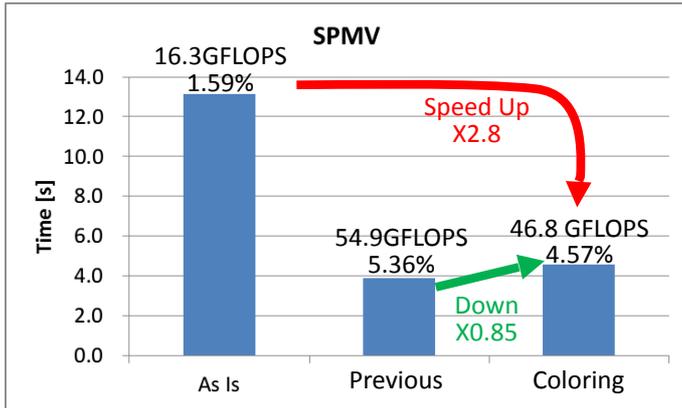
All loops were parallelized OK

- From previous tuning, MG was speed up 280% but parallel effect was low
- SPMV was worse. Why?



- TOTAL run time was improved 8.5 times.
- Coloring time was about 13 second. By dull implementation for coloring (It is able to be improved).
- Current HPCG code calculates a GFLOPS value based on the total time including with the coloring time. The time is defined assuming the optimization was performed every cycles with the CG.
- Therefore resultant GFLOPS value isn't improved 8.5 times.





- For previous version 1.1 (pre-conditioner was SYMGS)
- Same tuning way was applied.
 - Preconditioning performance was improved 15 times.
 - Total run time was improved 11 times, resultant GFLOPS value little low.
- Tuning effect on the Ver.2.1 was low.

- For preconditioning
 - In the Ver.1.1, the pre-conditioner performance was improved 6.24 times by the coloring
 - But in the Ver.2.1, the coloring effect was lower.
 - The ver.1.1 employ simple Gauss-Seidel pre-conditioner.
 - The ver.2.1 employ complicated Multi-Grid pre-conditioner.
- For SPMV
 - The SPMV performance got worse than without the coloring.
 - This degradation by the coloring has also occurred in the ver.1.1.
 - But in the ver.2.1, the degradation was larger than the Ver.1.1
 - Although the same coloring way was applied, why the degradation was larger than the Ver.1.1 in the Ver.2.1?

Our requests for the HPCG and questions

Feedbacks from Japanese HPC Centers²⁴

- Inconsistency due to the geometric MG preconditioner
 - Geometric yet not allowed to exploit the regular mesh structure in the solver
- Gauss-Seidel vs. Jacobi
 - GS without coloring is not parallelizable. Multi-coloring does not completely preserve the computation order of the serial implementation.
 - Assume an 8-core node. With multi-color GS, the residual with 8 MPI process run and 1 MPI process with 8 OpenMP threads will be different.
 - Jacobi is much simpler, no loop-carried dependency.

- Need for a well-tuned reference implementation (cf. HPL for Linpack)
 - Measurement until convergence
 - Why just 50 iterations?
 - Optimization time should not be included in the reported timing
-