

Introduction to Parallel Programming for Multicore/Manycore Clusters

Part IV: Introduction to Tuning

Kengo Nakajima
Information Technology Center
The University of Tokyo

- What is “tuning/optimization” ?
- Vector/Scalar Processors
- Example: Scalar Processors
- Bank Conflict/Cache Thrashing
- Examples

What is Tuning/Optimization

Optimization of Performance

- Purpose
 - Reduction of computation time
 - Optimization
- How to tune/optimize codes
 - Applying new algorithms
 - Modification/optimization according to property/parameters of H/W
 - Tuning/optimization should not affect results themselves.
 - or you have to recognize that results may change due to tuning/optimization

How do we tune/optimize codes ?

- When do we apply tuning ?
 - It's difficult if you apply tuning to the codes with $O(10^4)$ lines ...
- You have to be careful to write “efficient” codes.
 - Several tips.
- Simple, Readable codes
 - codes with few bugs
- Using optimized libraries
- Good parallel program = good serial program
- Tuning/optimization – faster computation – efficient research ...

Some References (in Japanese)

- スカラープロセッサ
 - 寒川「RISC超高速化プログラミング技法」, 共立出版, 1995.
 - Dowd(久良知訳)「ハイ・パフォーマンス・コンピューティング-RISCワークステーションで最高のパフォーマンスを引き出すための方法」, トムソン, 1994.
 - Goedecker, Hoisie “Performance Optimization for Numerically Intensive Codes”, SIAM, 2001.
- 自動チューニング
 - 片桐「ソフトウェア自動チューニング」, 慧文社, 2004.
- ベクトルプロセッサ
 - 長島, 田中「スーパーコンピュータ」, オーム社, 1992.
 - 奥田, 中島「並列有限要素解析」, 培風館, 2004.

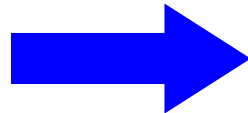
Tips for Tuning/Optimization

- Be careful about memory access patterns
- **Avoid calling functions in innermost loops**
 - Inline expansion of modern compilers might not work efficiently.
 - Avoid “if-clauses” in innermost loops.
- Avoid “too-multi-nested” loops
- Avoid many division operations, calling built-in-functions
- Avoid redundant operations
 - Storing in memory
 - Trade-off with memory capacity
- **Unfortunately, dependency on compilers and H/W is very significant !**
 - Optimum options/directives through empirical studies
 - Today’s content is very general remedy.

Example: Multi-Nested Loops

- Overhead for initialization of loop-counter occurs at every do-loop.
 - In the lower-left example (blue), innermost loop is reached 10^6 times. Therefore, 10^6 times initialization of loop-counter occurs.
 - In the lower-right example (yellow) with loop expansion, only one initialization of loop-counter occurs.

```
real*8 AMAT(3,1000000)
. . .
do j= 1, 1000000
  do i= 1, 3
    A(i,j)= A(i,j) + 1.0
  enddo
enddo
. . .
```



```
real*8 AMAT(3,1000000)
. . .
do j= 1, 1000000
  A(1,j)= A(1,j) + 1.0
  A(2,j)= A(2,j) + 1.0
  A(3,j)= A(3,j) + 1.0
enddo
. . .
```

Simple ways for measuring performance

- “time” command
- “timer” subroutines/functions
- Tools for “profiling”
 - Detection of “hot spots”
 - gprof (UNIX)
 - Tools for compilers/systems
 - pgprof: PGI
 - Vtune: Intel
 - https://oakleaf-www.cc.u-tokyo.ac.jp/cgi-bin/hpcportal_u.en/index.cgi

Files on FX10

```
>$ cd
```

```
>$ cp /home/z30088/omp/omp-c4.tar .
```

```
>$ cp /home/z30088/omp/omp-f4.tar .
```

```
>$ tar xvf omp-c4.tar
```

```
>$ tar xvf omp-f4.tar
```

```
>$ cd multicore/omp <$0-omp>
```

Confirm Directories:

```
>$ ls
```

```
dense run src src0 src10 src20 src2x
```

- What is “tuning/optimization” ?
- **Vector/Scalar Processors**
- Example: Scalar Processors
- Bank Conflict/Cache Thrashing
- Examples

Scalar/Vector Processors



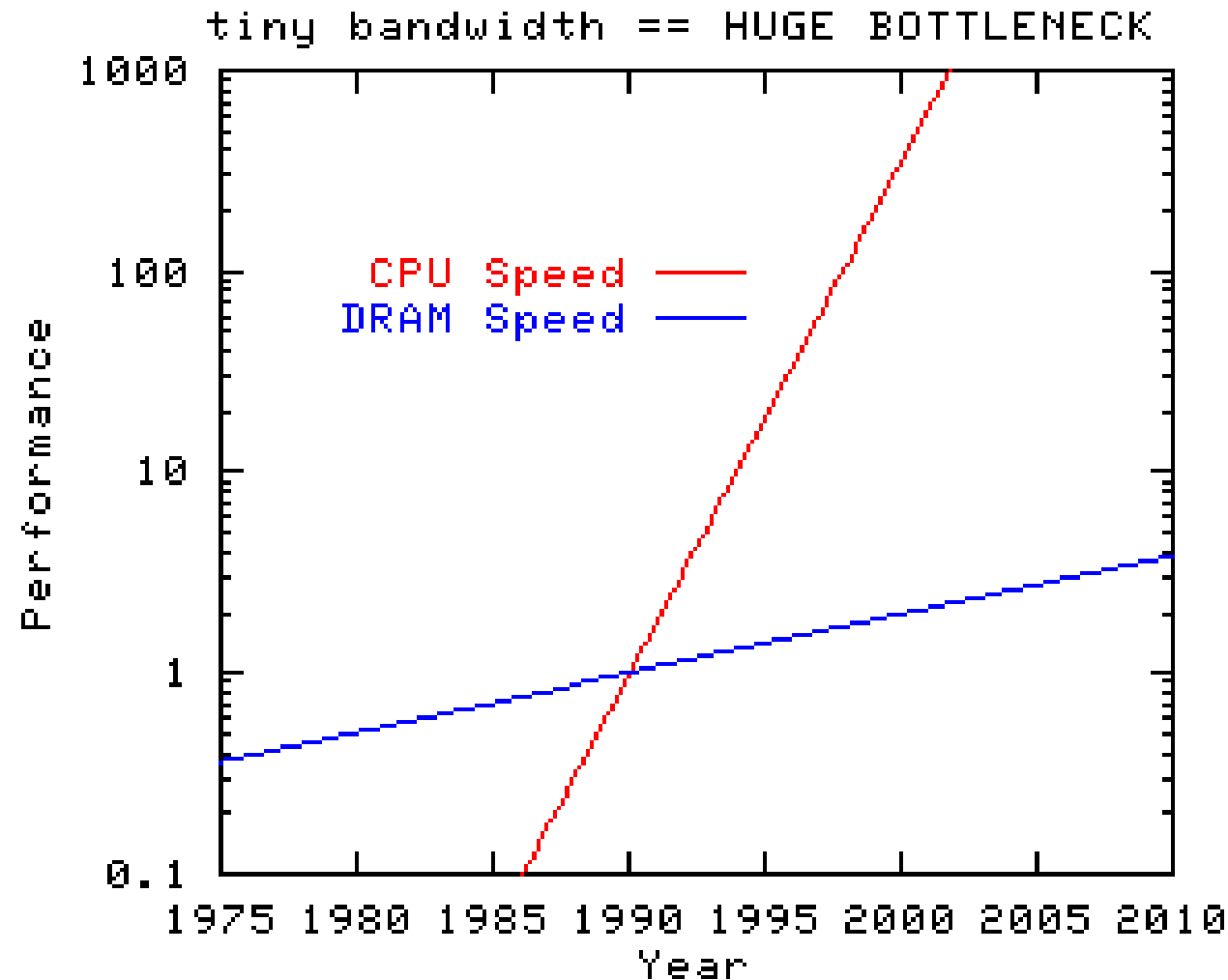
- Scalar Processors

- Gap between clock rate and memory bandwidth.
 - getting better, but multi/many-core architectures appear
- Low Peak-Performance Ratio
 - Ex.: IBM Power3/Power4, 5–8% in FEM applications

- Vector Processors

- High Peak-Performance Ratio
 - Ex.: Earth Simulator, 35% in FEM applications
- requires ...
 - very special tuning for vector processors
 - sufficiently long loop (problem size)
- Appropriate for rather simpler problems

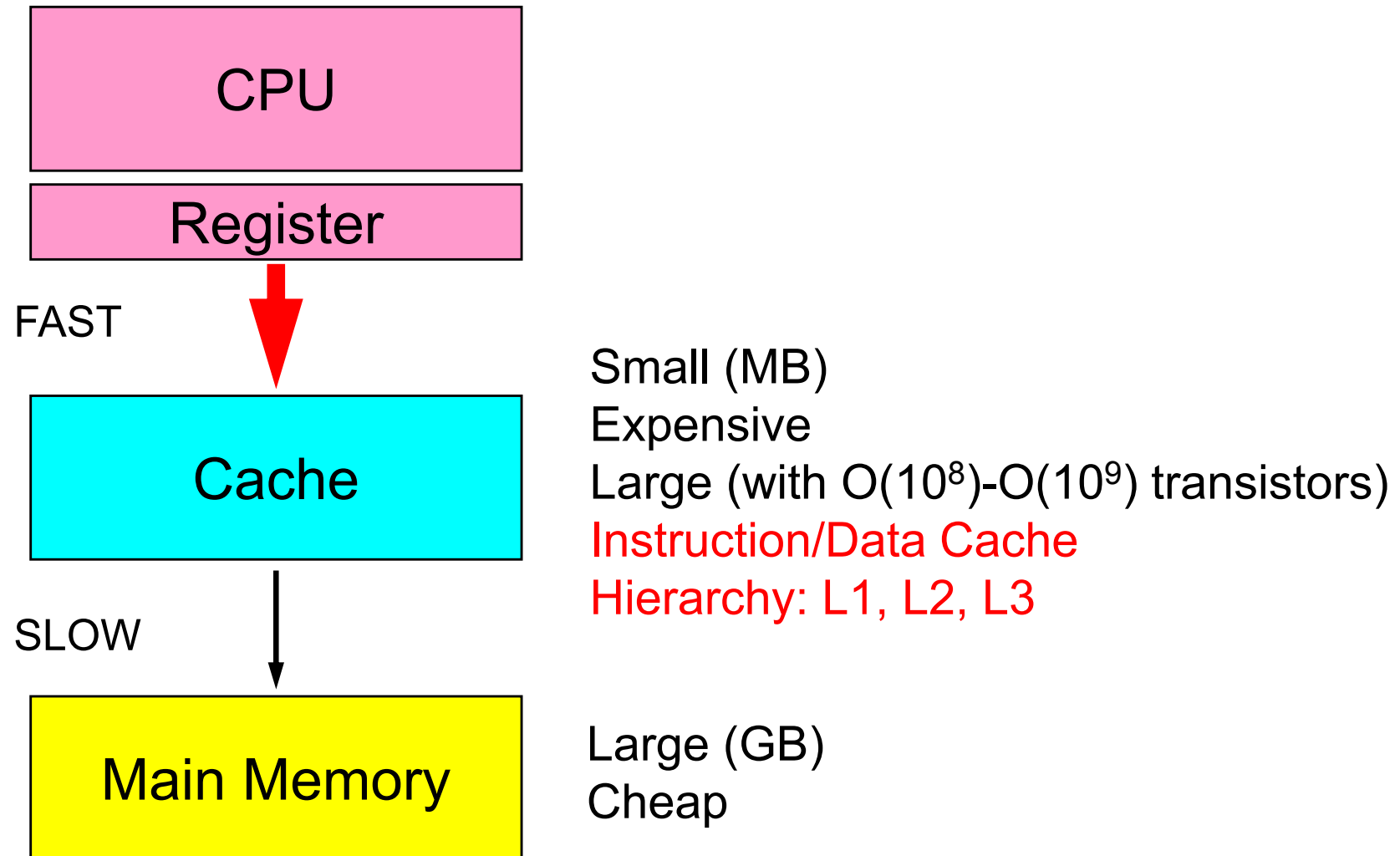
Gap between performance of CPU and Memory



<http://www.streambench.org/>

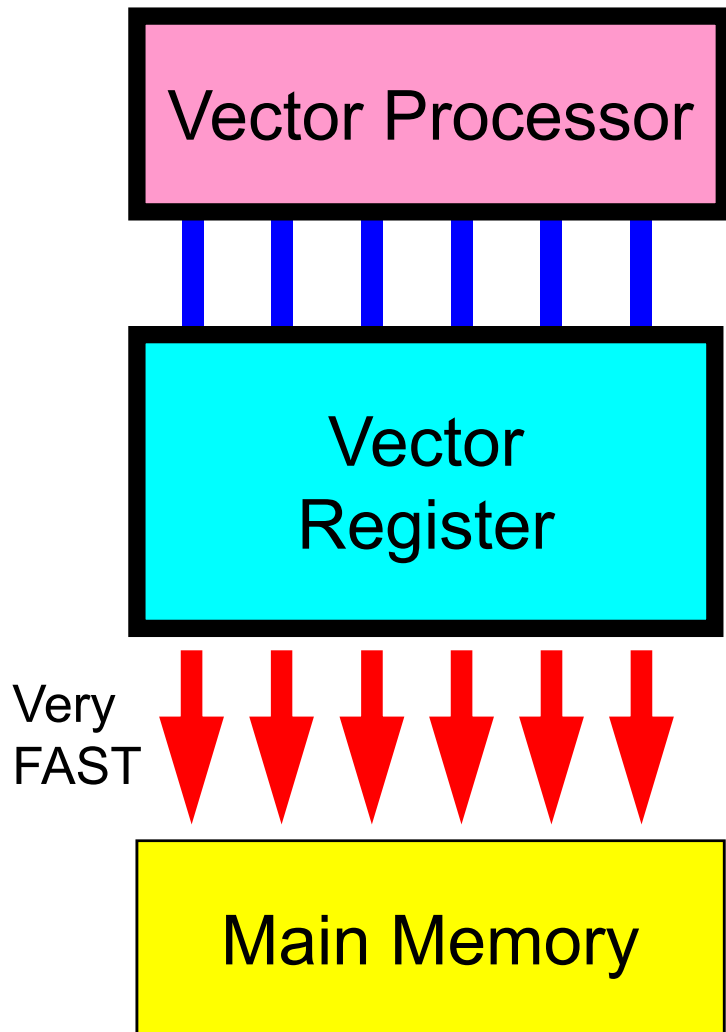
Scalar Processors

CPU-Cache: Hierarchical Structure



Vector Processors

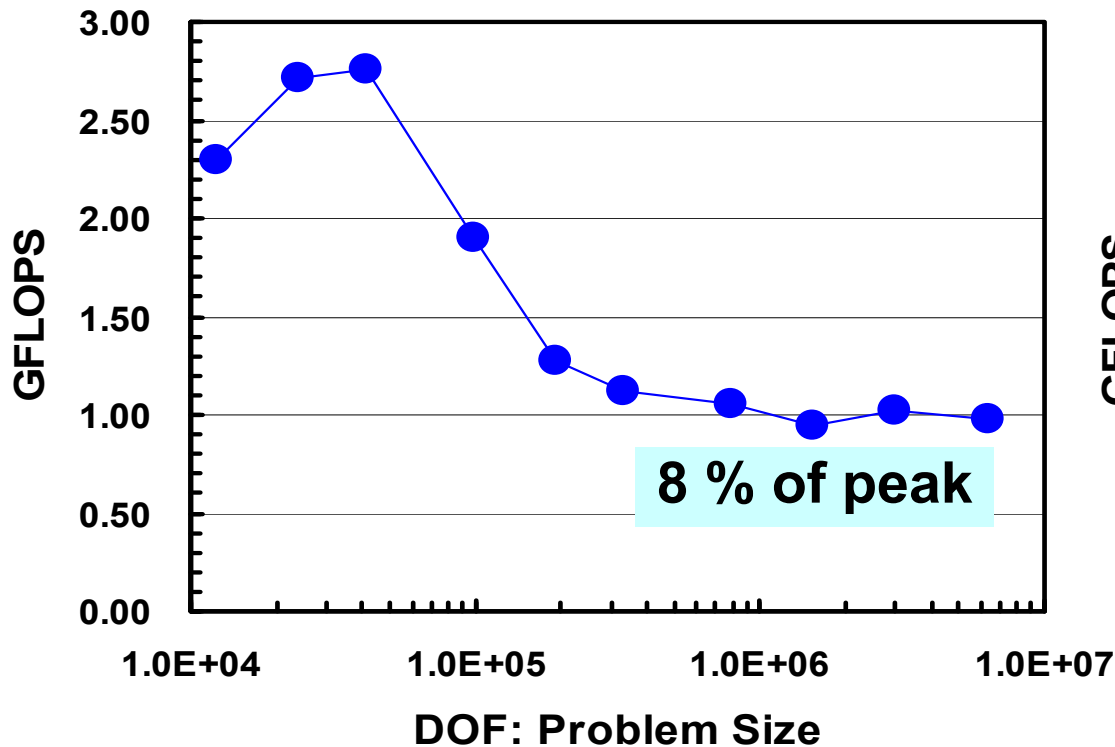
Vector Registers/Fast Memory



- Parallel operations for simple do-loops.
- Good for large-scale simple problems

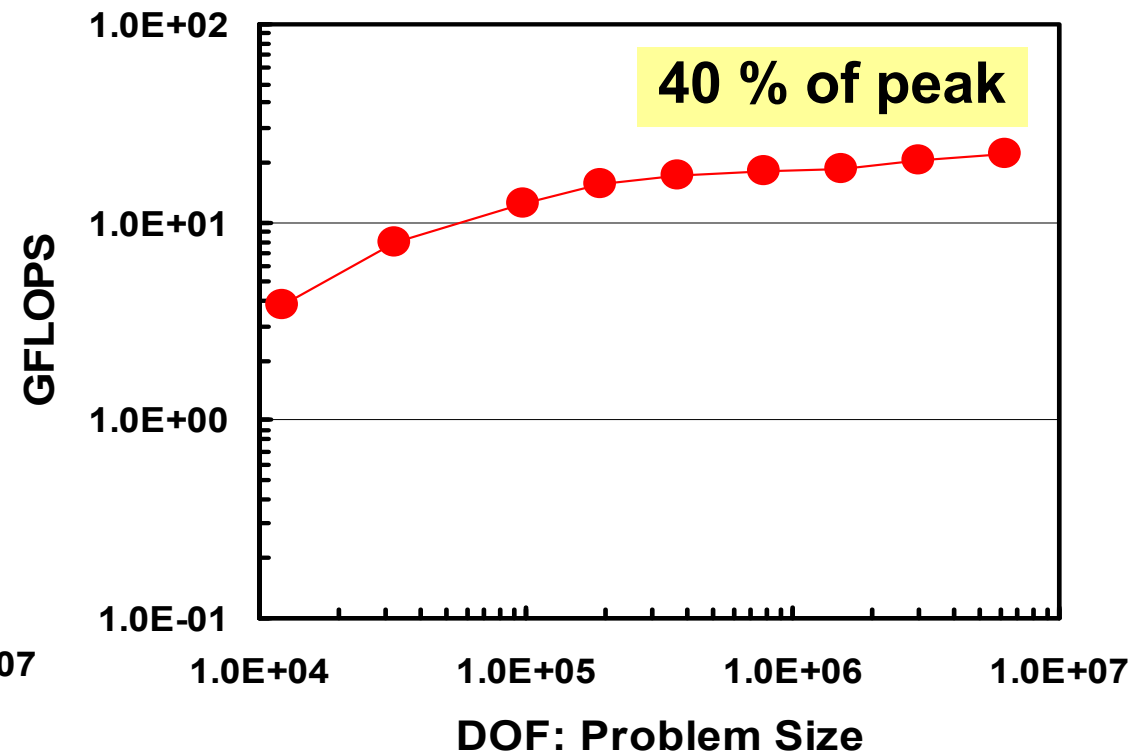
```
do i= 1, N  
  A(i)= B(i) + C(i)  
enddo
```

Typical Behaviors



IBM-SP3:

Higher performance for small problems, effect of cache



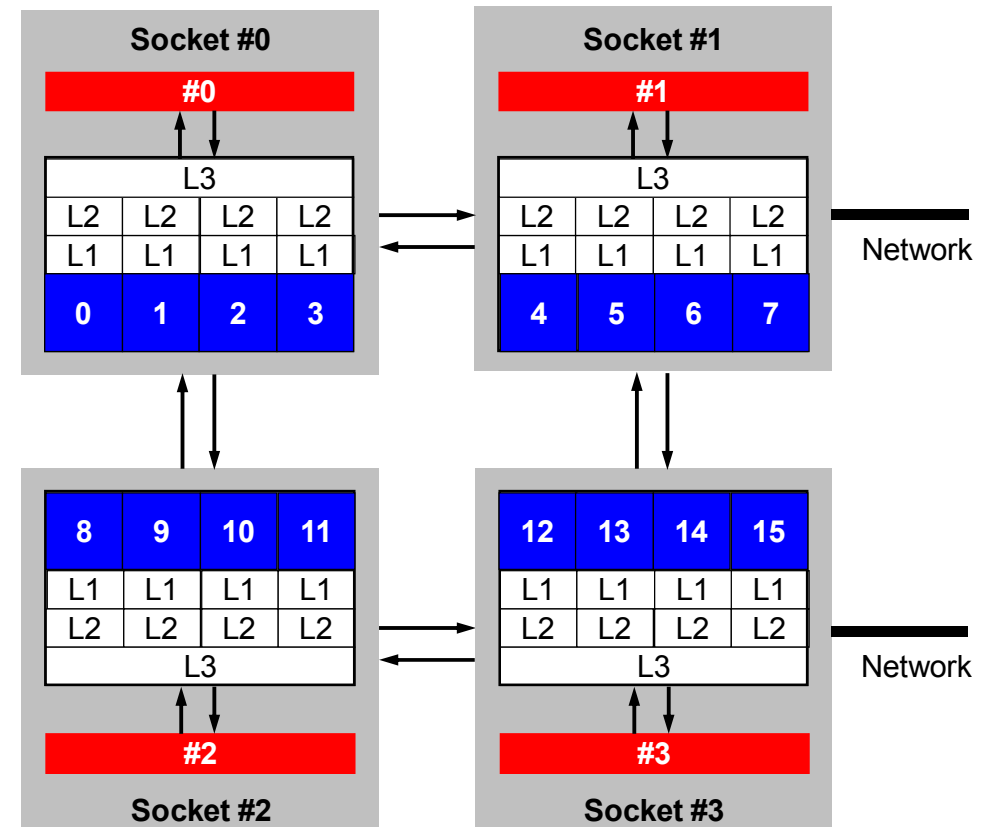
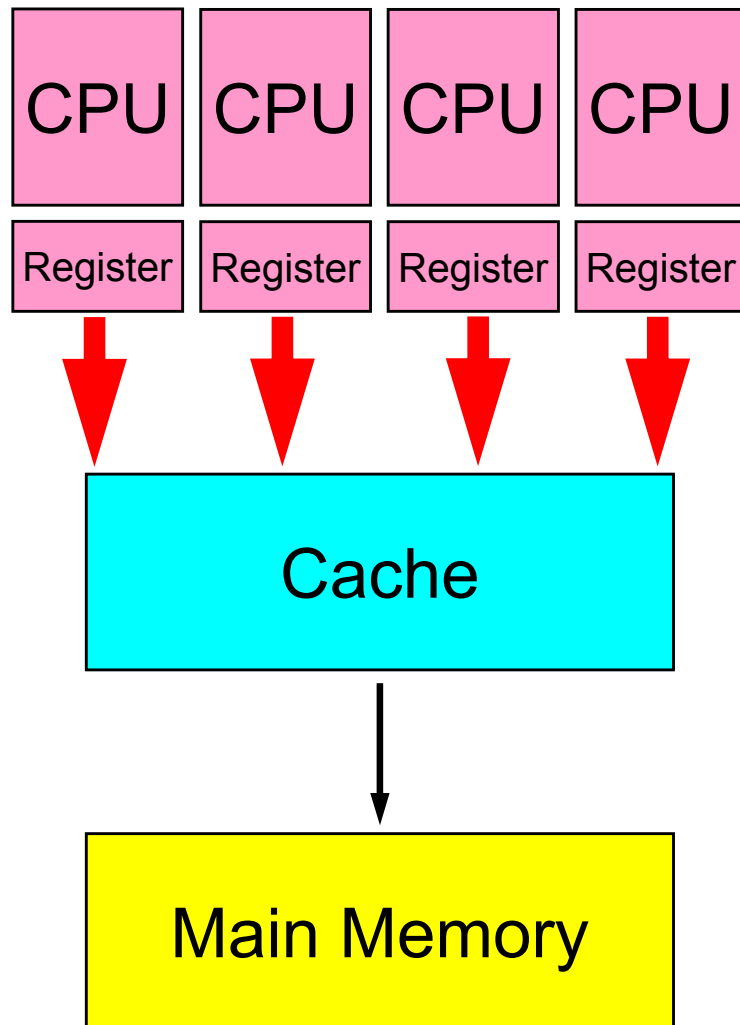
Earth Simulator:

Higher performance for large-scale problems with longer loops

Multicores/Manycores

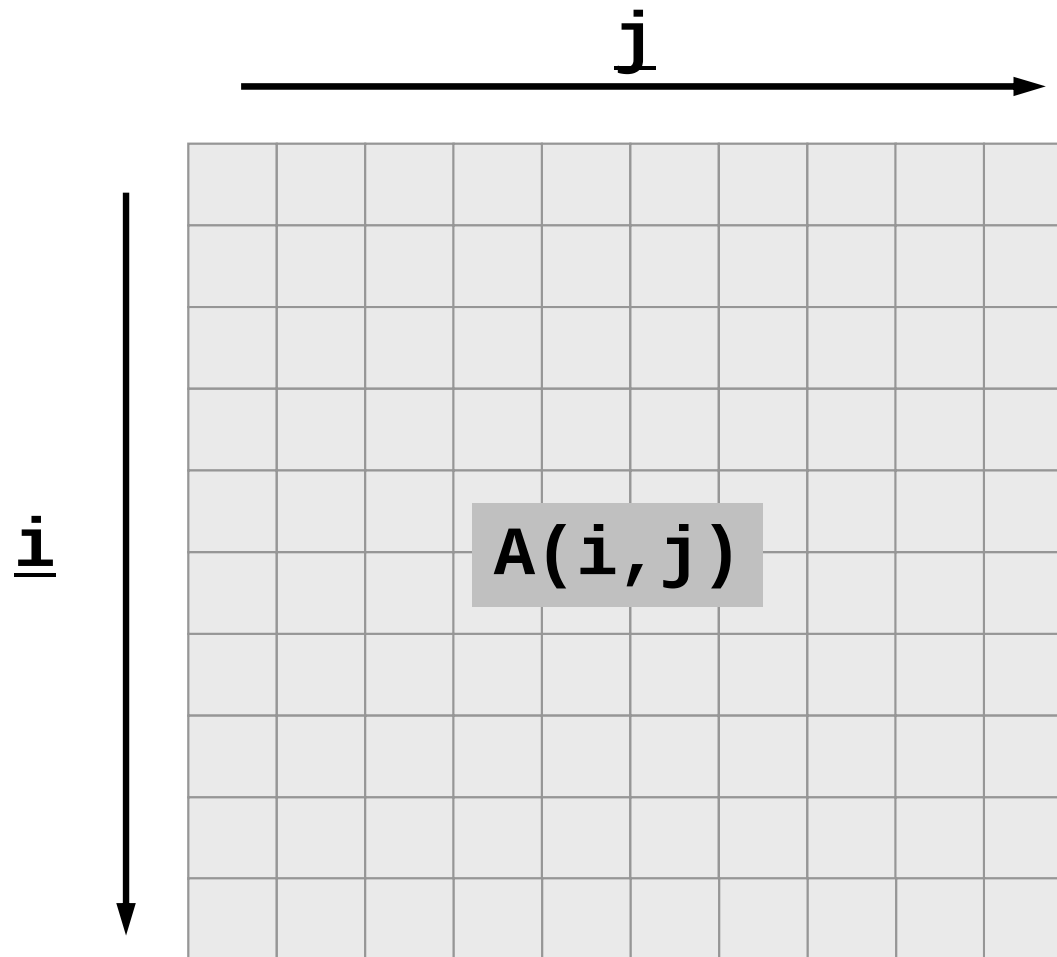
Multiple cores share memory/cache

→ Memory Contention



How to get opt. performance by Tuning ?

- = Optimization of memory access



How to get opt. performance by Tuning ? (cont.)

- Vector Processors
 - Long loops
- Scalar Processors
 - Utilization of cache, small chunks of data
- Common Issues
 - Continuous memory access
 - Localization
 - Changing sequence of computations might provide change of results.

- What is “tuning/optimization” ?
- Vector/Scalar Processors
- **Example: Scalar Processors**
- Bank Conflict/Cache Thrashing
- Examples

Typical Methods of Tuning for Scalar Processors

- Loop Unrolling
 - loop overhead
 - loading/storing
- Blocking
 - Cache miss

BLAS: Basic Linear Algebra Subprograms

- Library API for fundamental operations of vectors and (dense) matrices)
- Level 1: Vectors: dot products, DAXPY
- Level 2: Matrix x Vector
- Level 3: Matrix x Matrix
- LINPACK
 - DGEMM: Level 3 BLAS

Loop Unrolling

reduction of loading/storing (1/4)

- Ratio of computation increases

```
<$0-omp/dense>/t2.f
```

```
N= 10000
do j= 1, N
  do i= 1, N
    A(i)= A(i) + B(i)*C(i,j)
  enddo
enddo

do j= 1, N-1, 2
  do i= 1, N
    A(i)= A(i) + B(i)*C(i,j)
    A(i)= A(i) + B(i)*C(i,j+1)
  enddo
enddo

do j= 1, N-3, 4
  do i= 1, N
    A(i)= A(i) + B(i)*C(i,j)
    A(i)= A(i) + B(i)*C(i,j+1)
    A(i)= A(i) + B(i)*C(i,j+2)
    A(i)= A(i) + B(i)*C(i,j+3)
  enddo
enddo
```

```
$> cd <$0-omp>/dense
$> mpifrtpx -Kfast t2.f
```

```
<modify "go1.sh">
$> pjsub go1.sh (1 process)
```

```
1.560717E-01
1.012069E-01
7.471654E-02
```

go1.sh for a single core

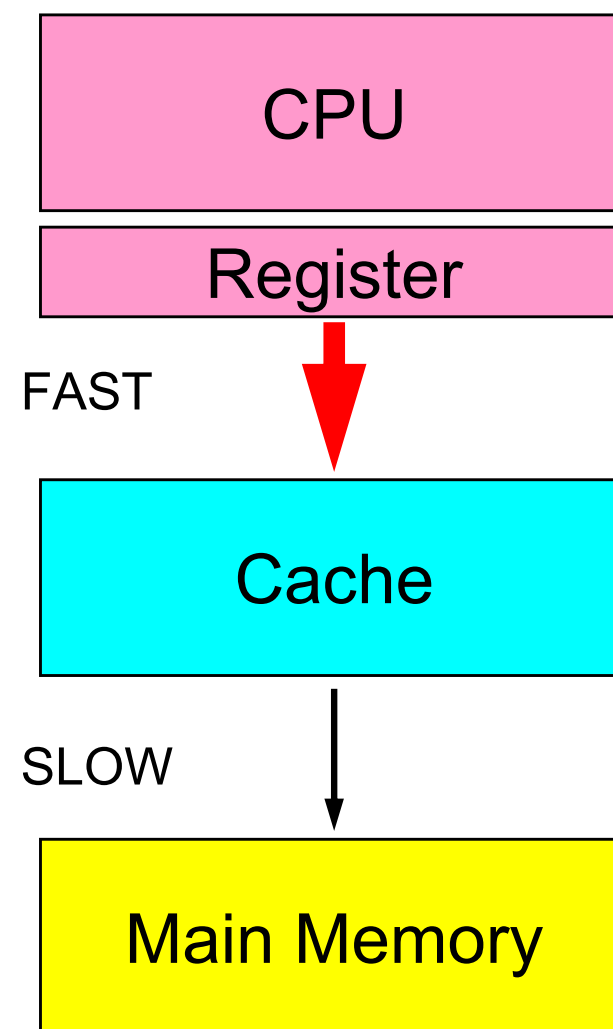
```
#!/bin/sh
#PJM -L "node=1"
#PJM -L "elapsed=00:10:00"
#PJM -L "rscgrp=lecture9"
#PJM -g "gt89"
#PJM -j
#PJM -o "test.lst"
#PJM --mpi "proc=1"

mpiexec ./a.out
```

Loop Unrolling

reduction of loading/storing (2/4)

- Load : Memory-Cache-Register
- Store: Register-Cache-Memory
- Fewer loading/storing, better performance



Loop Unrolling

reduction of loading/storing (3/4)

```
do j= 1, N
  do i= 1, N
    A(i)= A(i) + B(i)*C(i,j)
    Store Load Load Load
  enddo
enddo
```

- Loading/Storing for A(i), B(i), C(i,j) occurs in each loop.
- 1*S, 3*L: 2*C

Loop Unrolling

reduction of loading/storing (4/4)

```
do j= 1, N-3, 4
  do i= 1, N
    A(i)= A(i) + B(i)*C(i, j)
           Load Load Load
    A(i)= A(i) + B(i)*C(i, j+1) Load
    A(i)= A(i) + B(i)*C(i, j+2) Load
    A(i)= A(i) + B(i)*C(i, j+3) Load
           Store
  enddo
enddo
```

- Values of arrays are kept on register during each loop. Storing occurs only at the end of the loop.
- Ratio of memory access (loading/storing) to computation can be reduced (1*S, 6*L: 8*C)
- Be careful about sequence of computations.

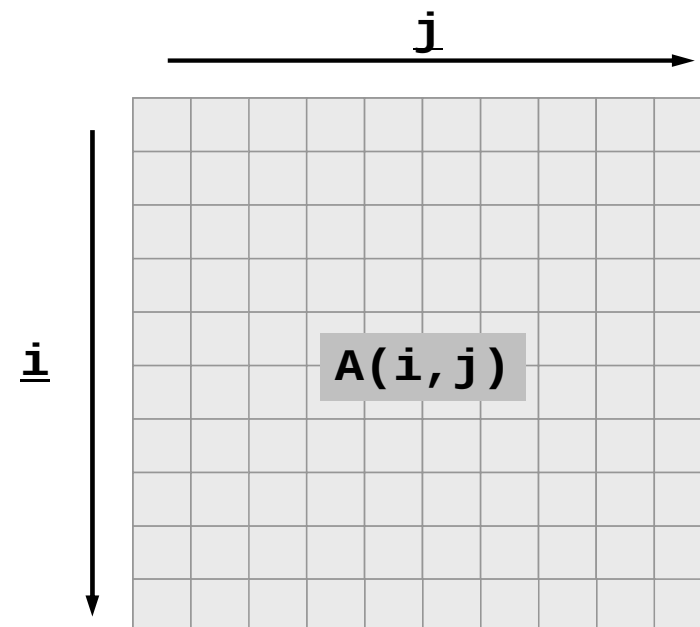
Loop Exchanging (1/2)

TYPE-A

```
do i= 1, N
  do j= 1, N
    A(i,j)= A(i,j) + B(i,j)
  enddo
enddo
```

TYPE-B

```
do j= 1, N
  do i= 1, N
    A(i,j)= A(i,j) + B(i,j)
  enddo
enddo
```



- In Fortran, component of $A(i,j)$ is aligned in the following way: $A(1,1)$, $A(2,1)$, $A(3,1)$, ..., $A(N,1)$, $A(1,2)$, $A(2,2)$, ..., $A(1,N)$, $A(2,N)$, ..., $A(N,N)$
 - In C: $A[0][0]$, $A[0][1]$, $A[0][2]$, ..., $A[N-1][0]$, $A[N-1][1]$, ..., $A[N-1][N-1]$
- Access must be according to this alignment for higher performance.

Loop Exchanging (2/2)

<\$0-omp>/dense/2d-1.f

```
TYPE-A
do i= 1, N
  do j= 1, N
    A(i,j)= A(i,j) + B(i,j)
  enddo
enddo
```

```
TYPE-B
do j= 1, N
  do i= 1, N
    A(i,j)= A(i,j) + B(i,j)
  enddo
enddo
```

```
TYPE-A
for (j=0; j<N; j++){
  for (i=0; i<N; i++){
    A[i][j]= A[i][j] + B[i][j];
  }
}
```

```
TYPE-B
for (i=0; i<N; i++){
  for (j=0; j<N; j++){
    A[i][j]= A[i][j] + B[i][j];
  }
}
```

```
$> cd <$0-omp>/dense
$> mpifrtpx -01 2d-1.f
$> pjsub go1.sh
### N ###      500
WORSE          3.730428E-03
BETTER          9.525069E-04
### N ###      1000
WORSE          1.642722E-02
BETTER          3.771729E-03
### N ###      1500
WORSE          4.304052E-02
BETTER          8.295263E-03
### N ###      2000
WORSE          6.198836E-02
BETTER          1.455921E-02
### N ###      2500
WORSE          1.180517E-01
BETTER          2.305677E-02
### N ###      3000
WORSE          1.675602E-01
BETTER          3.311505E-02
### N ###      3500
WORSE          2.324806E-01
BETTER          4.509363E-02
### N ###      4000
WORSE          2.594637E-01
BETTER          5.803503E-02
```

Loop Exchanging (2/2)

Performance is improved, and difference is smaller by option “-Kfast”

```
TYPE-A
do i= 1, N
  do j= 1, N
    A(i,j)= A(i,j) + B(i,j)
  enddo
enddo
```

```
TYPE-B
do j= 1, N
  do i= 1, N
    A(i,j)= A(i,j) + B(i,j)
  enddo
enddo
```

```
TYPE-A
for (j=0; j<N; j++){
  for (i=0; i<N; i++){
    A[i][j]= A[i][j] + B[i][j];
  }
}
```

```
TYPE-B
for (i=0; i<N; i++){
  for (j=0; j<N; j++){
    A[i][j]= A[i][j] + B[i][j];
  }
}
```

```
$> cd <$0-omp>/dense
$> mpifrtpx -Kfast 2d-1.f
$> pjsub go1.sh
### N ###      500
WORSE      3.017990E-04
BETTER      3.012992E-04
### N ###      1000
WORSE      1.213297E-03
BETTER      1.189598E-03
### N ###      1500
WORSE      2.675394E-03
BETTER      3.808392E-03
### N ###      2000
WORSE      4.778489E-03
BETTER      4.779590E-03
### N ###      2500
WORSE      7.354485E-03
BETTER      7.370183E-03
### N ###      3000
WORSE      1.060798E-02
BETTER      1.060808E-02
### N ###      3500
WORSE      1.444077E-02
BETTER      1.443657E-02
### N ###      4000
WORSE      1.977946E-02
BETTER      1.977636E-02
```

Blocking for Cache Miss (1/7)

```
do i= 1, NN
  do j= 1, NN
    A(j,i)= A(j,i) + B(i,j)
  enddo
enddo
```

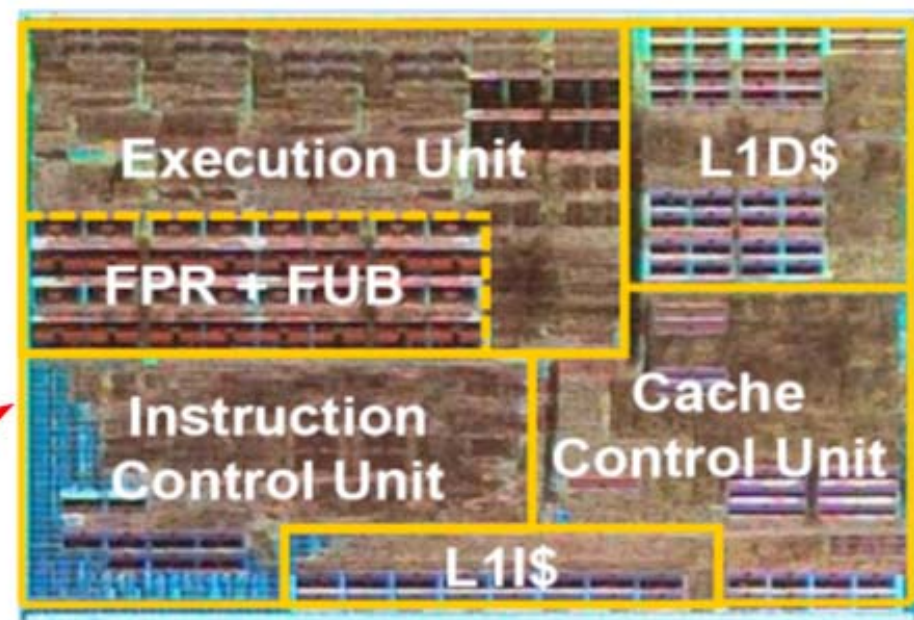
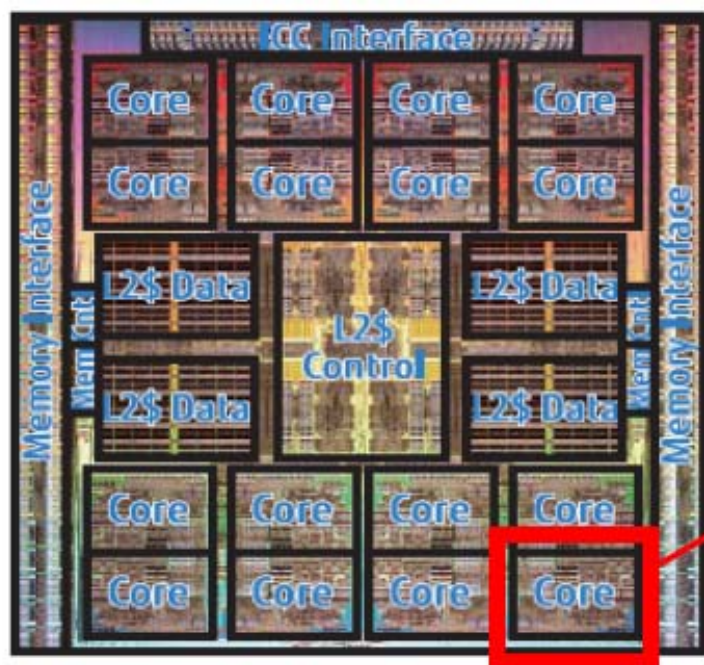
- Consider this situation.

Utilizing Cache

- Cache of **SPARC64™ IXfx**
 - L1: 32KB/core (Instruction, Data)
 - L2: 12MB/node
 - Size of “cache-line” is 128 byte
 - Request for memory occurs for each “cache-line”
- Sector Cache
 - Software Controllable Cache
 - Users can keep necessary data on cache
 - Capacity of cache is small, therefore data stored in the early stage of computation may be deleted from cache.
 - No lectures on this issue in this class

SPARC64™ IXfx

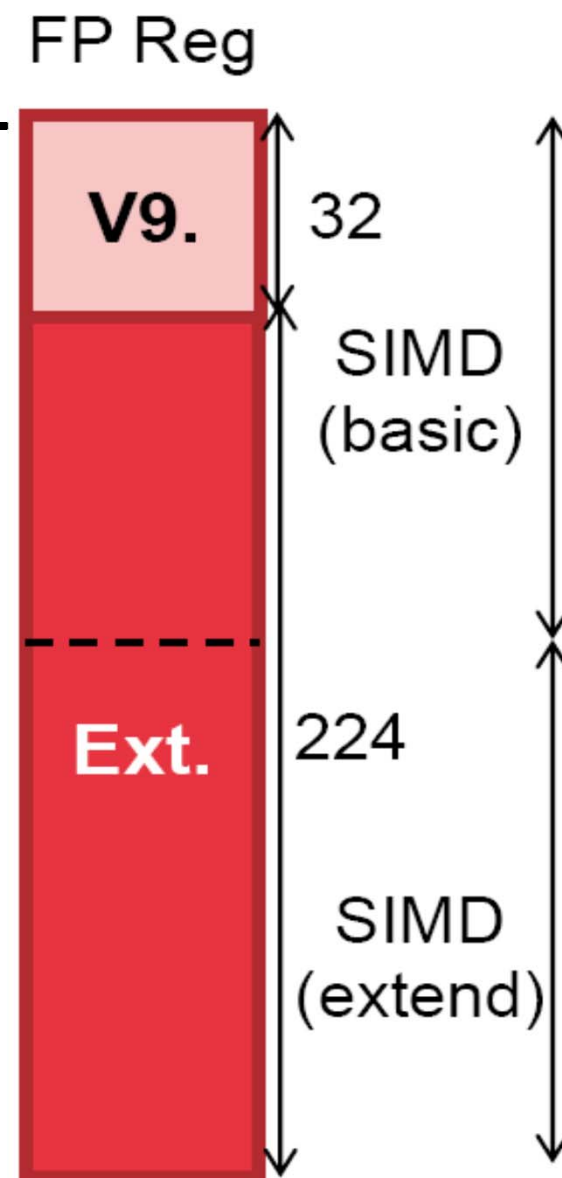
- HPC-ACE (High Performance Computing – Arithmetic Computational Extensions)
- UMA, not NUMA
- H/W barrier for high-speed synchronization of on-chip cores



HPC-ACE

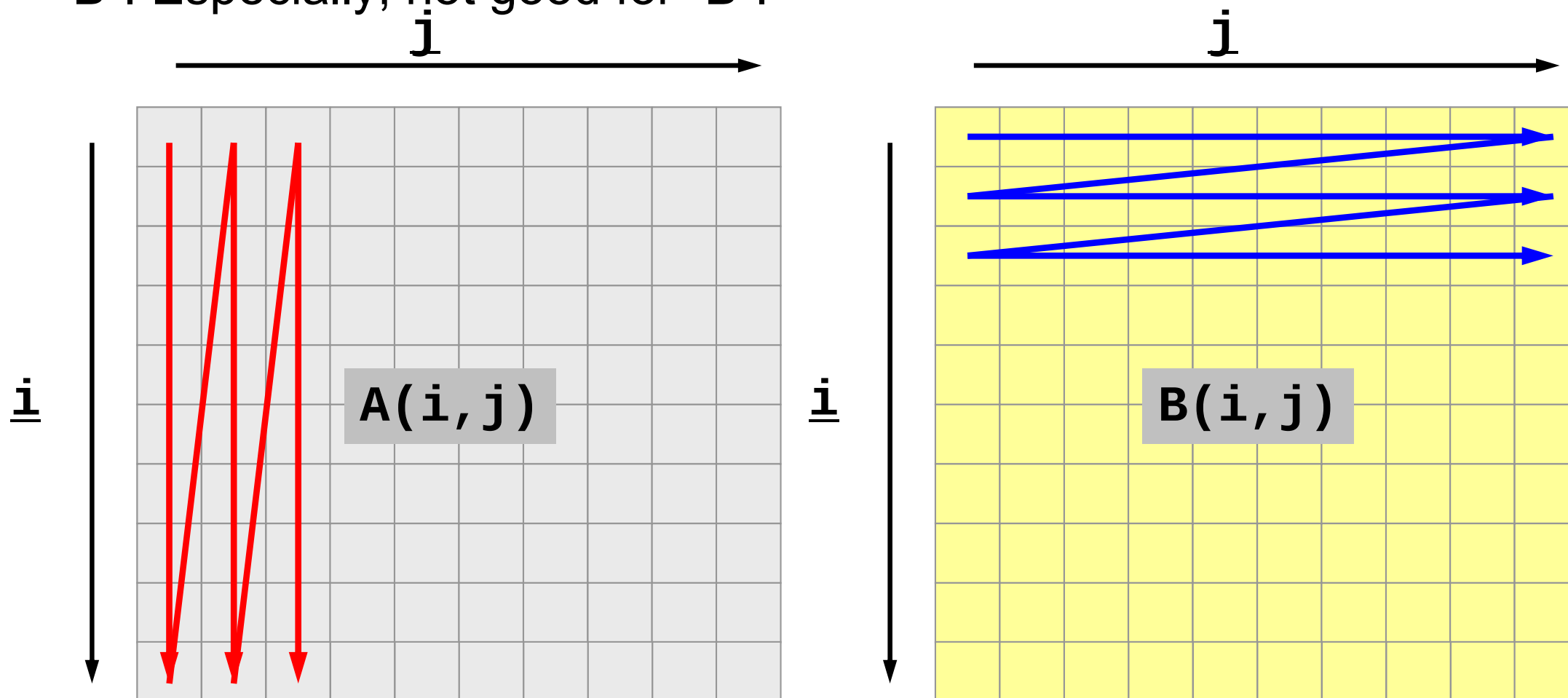
High Performance Computing – Arithmetic Computational Extensions

- Enhanced instruction set for the SPARC-V9 instruction set arch.
 - High-Performance & Power-Aware
- Extended Number of Registers
 - FP Registers: 32→256
- S/W Controllable Cache
 - “Sector Cache”
 - for keeping reusable data sets on cache
- High-Performance, Efficient
 - Optimized FP functions
 - Conditional Operation



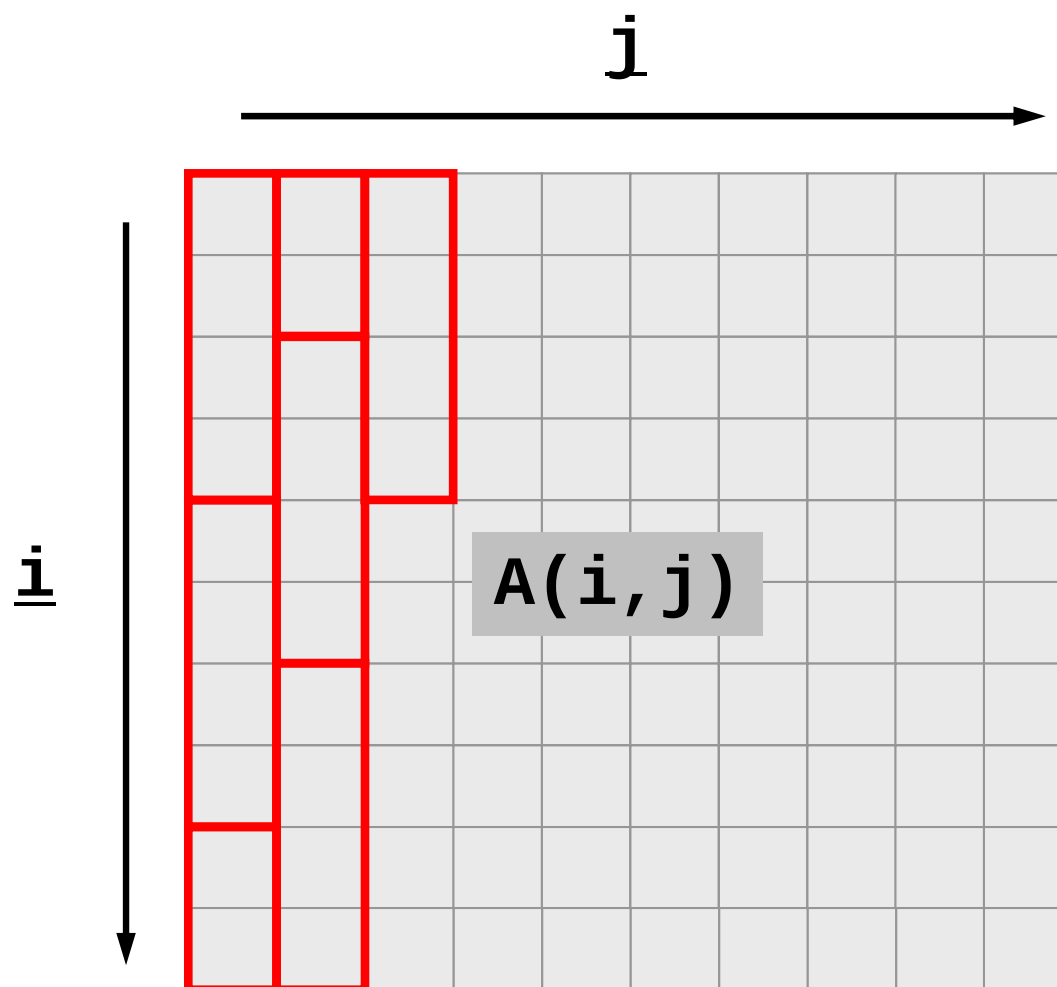
Blocking for Cache Miss (2/7)

- Direction of optimum memory access for “A” is different from that of “B”. Especially, not good for “B”.



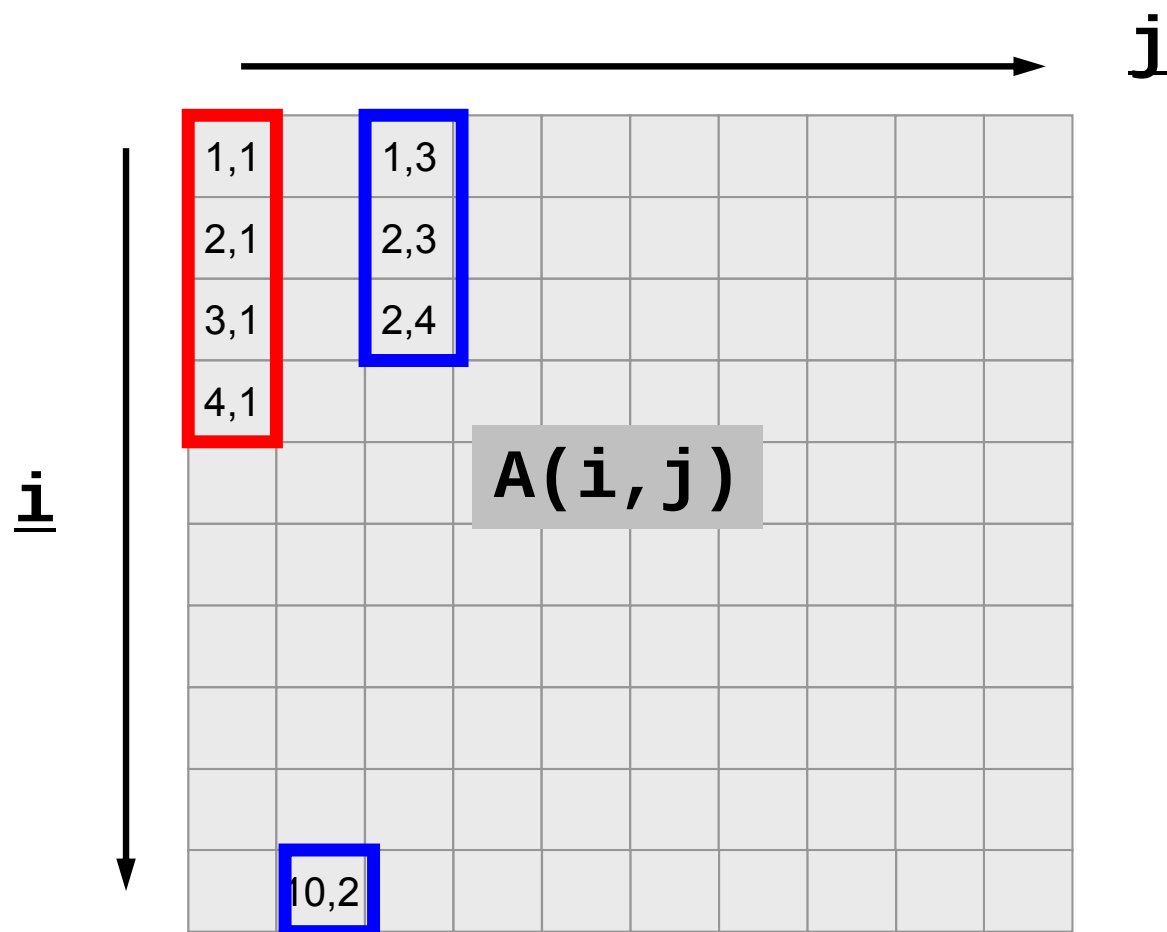
Blocking for Cache Miss (3/7)

- If the size of cache-line is 4-word, data on array is sent to cache from main memory in the following way:



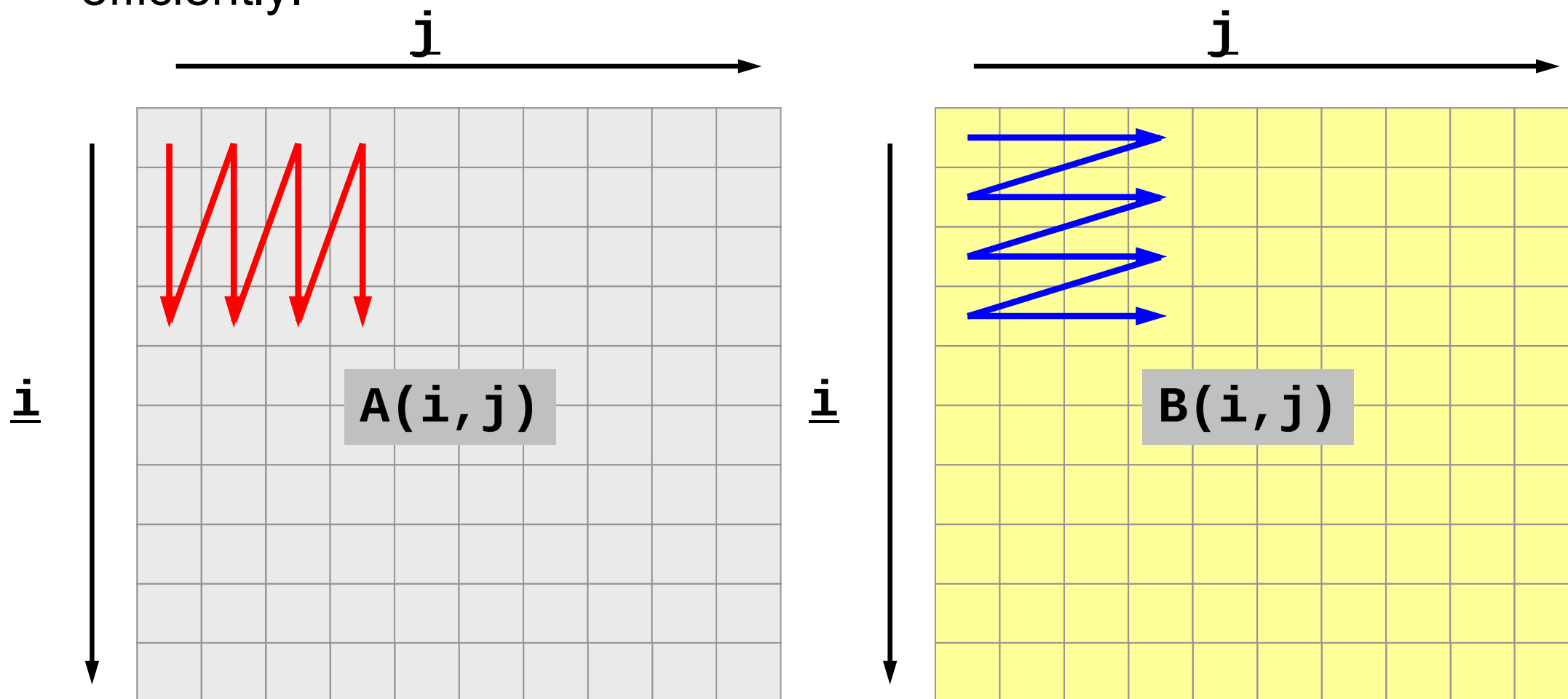
Blocking for Cache Miss (4/7)

- Therefore, if $A(1,1)$ is touched, $A(1,1)$, $A(2,1)$, $A(3,1)$, $A(4,1)$ are on cache. If $A(10,2)$ is touched $A(10,2)$, $A(1,3)$, $A(2,3)$, $A(3,3)$ are on cache.



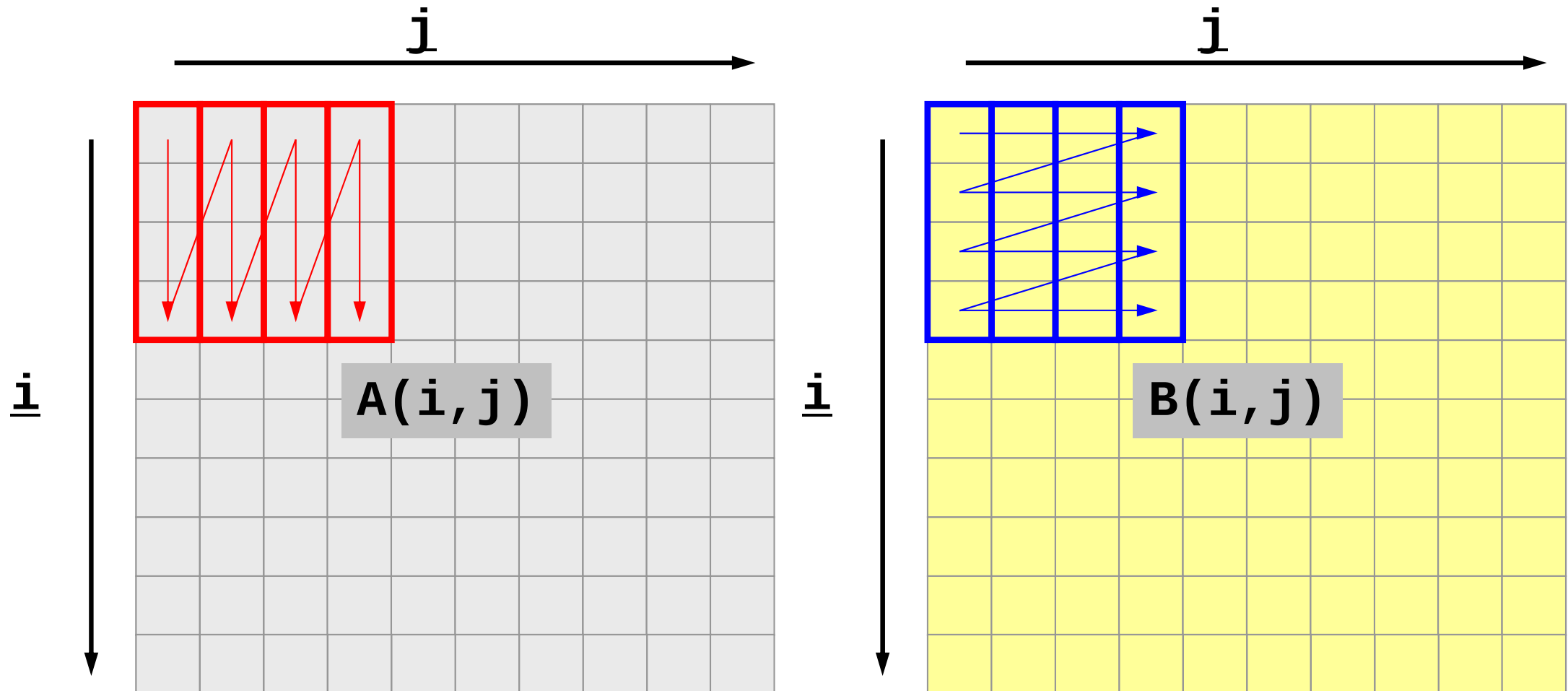
Blocking for Cache Miss (5/7)

- Therefore, following block-wise access pattern utilizes cache efficiently.



Blocking for Cache Miss (6/7)

- □, □ are on cache.



Blocking for Cache Miss (7/7)

- 2x2 block

<\$0-omp>/dense/2d-2.f

```
do i= 1, NN
  do j= 1, NN
    A(j,i)= A(j,i) + B(i,j)
  enddo
enddo

do i= 1, NN-1, 2
  do j= 1, NN-1, 2
    A(j,i)= A(j,i) + B(i,j)
    A(j+1,i)= A(j+1,i) + B(i,j+1)
    A(j,i+1)= A(j,i+1) + B(i+1,j)
    A(j+1,i+1)= A(j+1,i+1) + B(i+1,j+1)
  enddo
enddo

do i= 1, NN-1, 2
  do j= 1, NN/2, 2
    A(j,i)= A(j,i) + B(i,j)
    A(j+1,i)= A(j+1,i) + B(i,j+1)
    A(j,i+1)= A(j,i+1) + B(i+1,j)
    A(j+1,i+1)= A(j+1,i+1) + B(i+1,j+1)
  enddo
enddo

do i= 1, NN-1, 2
  do j= NN/2+1, NN-1, 2
    A(j,i)= A(j,i) + B(i,j)
    A(j+1,i)= A(j+1,i) + B(i,j+1)
    A(j,i+1)= A(j,i+1) + B(i+1,j)
    A(j+1,i+1)= A(j+1,i+1) + B(i+1,j+1)
  enddo
enddo
```

Loop-Fission is also effective for reduction of cache/TLB miss's.

```
$> cd <$0-omp>/dense
$> mpifrtpx -O1 2d-2.f
$> pjsub go1.sh

### N ###      500
BASIC      2.838309E-03
2x2        1.835505E-03
2x2-b      1.538004E-03
### N ###      1000
BASIC      1.167853E-02
2x2        9.495229E-03
2x2-b      9.299729E-03

...

### N ###      4000
BASIC      2.081746E-01
2x2        1.294938E-01
2x2-b      1.317760E-01
### N ###      4500
BASIC      3.203001E-01
2x2        2.335151E-01
2x2-b      2.354205E-01
### N ###      5000
BASIC      3.517879E-01
2x2        2.478577E-01
2x2-b      2.492195E-01
```

-Kfast: faster, no difference ...

- 2x2 block

<\$0-omp>/dense/2d-2.f

```
do i= 1, NN
  do j= 1, NN
    A(j,i)= A(j,i) + B(i,j)
  enddo
enddo
```

```
do i= 1, NN-1, 2
  do j= 1, NN-1, 2
    A(j,i)= A(j,i) + B(i,j)
    A(j+1,i)= A(j+1,i) + B(i,j+1)
    A(j,i+1)= A(j,i+1) + B(i+1,j)
    A(j+1,i+1)= A(j+1,i+1) + B(i+1,j+1)
  enddo
enddo
```

```
do i= 1, NN-1, 2
  do j= 1, NN/2, 2
    A(j,i)= A(j,i) + B(i,j)
    A(j+1,i)= A(j+1,i) + B(i,j+1)
    A(j,i+1)= A(j,i+1) + B(i+1,j)
    A(j+1,i+1)= A(j+1,i+1) + B(i+1,j+1)
  enddo
enddo
```

```
do i= 1, NN-1, 2
  do j= NN/2+1, NN-1, 2
    A(j,i)= A(j,i) + B(i,j)
    A(j+1,i)= A(j+1,i) + B(i,j+1)
    A(j,i+1)= A(j,i+1) + B(i+1,j)
    A(j+1,i+1)= A(j+1,i+1) + B(i+1,j+1)
  enddo
enddo
```

Loop-Fission is also effective for reduction of cache/TLB miss's.

```
$> cd <$0-omp>/dense
$> mpifrtpx -Kfast 2d-2.f
$> pjsub go1.sh
```

```
### N ###      500
BASIC      1.870605E-03
2x2        1.602004E-03
2x2-b      1.579705E-03
### N ###      1000
BASIC      7.570124E-03
2x2        7.771923E-03
2x2-b      9.585428E-03
```

...

```
### N ###      4000
BASIC      1.276466E-01
2x2        1.236477E-01
2x2-b      1.216945E-01
### N ###      4500
BASIC      1.625757E-01
2x2        1.761032E-01
2x2-b      1.732303E-01
### N ###      5000
BASIC      2.037693E-01
2x2        1.944027E-01
2x2-b      1.913424E-01
```

Summary: Tuning

- Scalar Processor
- Dense Matrices: BLAS
- Optimization of operations for sparse matrices (which appear in this class) is much more difficult (still research topics)
 - Basic idea is same as that for dense matrices.
 - Optimum memory access.

Sparse/Dense Matrices

```
do i= 1, N
  Y(i)= D(i)*X(i)
  do k= index(i-1)+1, index(i)
    Y(i)= Y(i) + AMAT(k)*X(item(k))
  enddo
enddo
```

```
do j= 1, N
  do i= 1, N
    Y(j)= Y(j) + A(i, j)*X(i)
  enddo
enddo
```

- “X” in RHS
 - Dense: continuous on memory, easy to utilize cache
 - Sparse: continuity is not assured, difficult to utilize cache
 - more “memory-bound”
- Effective method for sparse matrices: Blocking
 - Reordering: provides “block” features
 - Utilizing physical features of matrices: multiple DOF on each element/node.

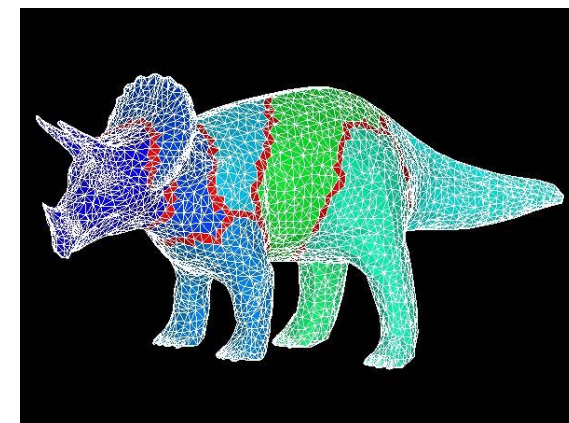
Summary: Tuning (cont.)

- Sparse Matrices

- Strategy for tuning may depends on alignment of data components.
- Program may change according to alignment of data components.

- Dense Matrices

- Structured, Regularity
- Performance mainly depends on machine parameters & mat. size.
 - Effect of options of compilers
 - Automatic tuning (AT) is applicable.
- Libraries
 - ATLAS (Automatic Tuning)
 - <http://math-atlas.sourceforge.net/>
 - GoToBLAS (Manual Tuning)
 - Kazushige Goto (Microsoft)



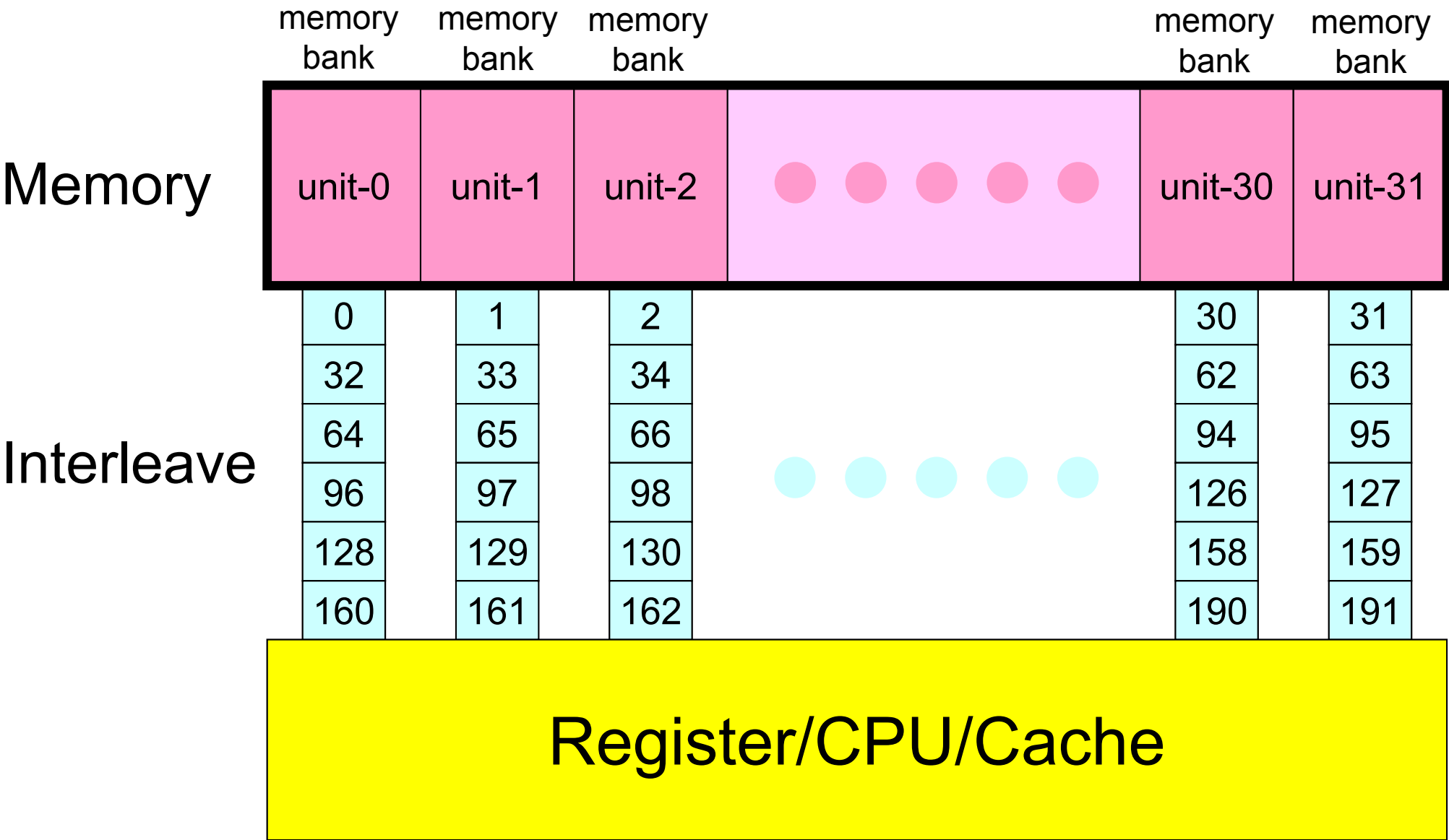
- What is “tuning/optimization” ?
- Vector/Scalar Processors
- Example: Scalar Processors
- **Bank Conflict/Cache Thrashing**
- Examples

Memory Interleaving/Bank Conflict

- Memory Interleaving
 - Method for fast data transfer to/from memory.
 - Parallel I/O for multiple memory banks.
- Memory Bank
 - Unit for memory management, small pieces of memory
 - Usually, there are 2^n independent modules.
 - Single bank can execute a single reading or writing at one time. Therefore, performance gets worse if data components on same bank are accessed simultaneously.
 - For example, “bank conflict” occurs if off-set of data access is 32 (in next page).
 - Remedy: Change of array size, loop exchange etc.

Bank Conflict

If off-set of data access is 32, only a single bank is utilized



Avoiding Bank Conflict

X

```
REAL*8 A(32,10000)
```

```
k= N
```

```
do i= 1, 10000
```

```
    A(k,i)= 0.d0
```

```
enddo
```

O

```
REAL*8 A(33,10000)
```

```
k= N
```

```
do i= 1, 10000
```

```
    A(k,i)= 0.d0
```

```
enddo
```

- Arrays with size of 2^n should be avoided.

If $N=2^m$, e.g. 128x128x128

- Bank conflict always occurs when non-zero off-diagonals are accessed, 16 threads
- Remedy for Bank Conflict
 - Padding by compiler
 - Reordering, such as CM, is effective
 - Hyperline, Hyperplane

385	386	387	388
257	258	259	260
129	130	131	132
1	2	3	4



10	14	19	25
6	9	13	18
3	5	8	12
1	2	4	7

Cache Thrashing

- FX10: L1D cache with 32KB for each core, 2-way
 - n-way set associative (n群連想記憶式)
 - Cache is divided into “n” banks
 - Each bank is divided into “cache lines”
 - Number of Cache Lines, Size of Cache Line (128 bytes for FX10) $\Rightarrow 2^m$
- This “2-way” cache is very harmful
 - If “ $N=2^m$ ”, memory addresses of $W(i, P)$, $W(i, Q)$, $W(i, R)$ map to the same cache address in CG computation.
 - Cache Thrashing: Lower Performance
 - $R=1$, $P=2$, $Q=3$
 - $X(i)$ is not affected

```
!$omp parallel do private(i)
  do i= 1, N
    X(i) = X(i) + ALPHA * W(i, P)
    W(i, R) = W(i, R) - ALPHA * W(i, Q)
  enddo
```

Remedy

- If the loop is split into 2 loops, up to 2 cache lines of W are referred. Therefore, cache thrashing does not occur (Remedy-1).

```
!$omp parallel do private(i)
  do i= 1, N
    X(i) = X (i)  + ALPHA * W(i,P)
  enddo

!$omp parallel do private(i)
  do i= 1, N
    W(i,R) = W(i,R) - ALPHA * W(i,Q)
  enddo
```

- If “ $N=2^m$ ”, certain numbers (e.g. 64, 128 ...) can be added to N . Thus, size of the array is not equal to 2^m , and cache thrashing is avoided (Remedy-2).
 - No such operation is needed for X

```
N2=128
allocate (W(N+N2, 4))
```

- What is “tuning/optimization” ?
- Vector/Scalar Processors
- Example: Scalar Processors
- Bank Conflict/Cache Thrashing
- **Examples**

Target Code

- FVM code
- Preconditioned CG solver: PCG
 - Diagonal Scaling, Point Jacobi
- **<\$0-omp>/src0**

Files on FX10

```
>$ cd <$0-omp>
```

```
>$ cd src10
```

```
>$ make
```

```
>$ ls ../run/sol10  
sol10
```

```
>$ cd src20
```

```
>$ make
```

```
>$ ls ../run/sol20  
sol20
```

```
>$ cd src2x
```

```
>$ make
```

```
>$ ls ../run/sol2x  
sol2x
```

Programs

Exec's	Summary
sol10	Original (L1-sol on PC)
sol20	CM Reordering
sol2x	CM Reordering + Remedy-2 for Cache Thrashing (equal to <\$0 - omp>/src0)

sol10: Original (L1-sol) (C)

poi_gen

```
OLDtoNEW = (int *) allocate_vector(sizeof(int), ICELTOT);  
NEWtoOLD = (int *) allocate_vector(sizeof(int), ICELTOT);  
  
for (i=0; i<ICELTOT; i++) {  
    OLDtoNEW[i]= i+1;  
    NEWtoOLD[i]= i+1;  
}
```

solver_PCG

```
N3= N;  
W = (double **)malloc(sizeof(double *)*4);  
...  
for (i=0; i<4; i++) {  
    W[i] = (double *)malloc(sizeof(double)*N3);  
    ...  
}
```

sol20: +CM reordering (C)

poi_gen

```

OLDtoNEW = (int *) allocate_vector(sizeof(int), ICELTOT);
NEWtoOLD = (int *) allocate_vector(sizeof(int), ICELTOT);

RCM(ICELTOT, NL, NU, INL, IAL, INU, IAU,
    &NCOLORtot, COLORindex, NEWtoOLD, OLDtoNEW);

```

solver_PCG

```

N3= N;
W = (double **)malloc(sizeof(double *)*4);
...
for (i=0; i<4; i++) {
    W[i] = (double *)malloc(sizeof(double)*N3);
    ...
}

```

sol2x: +CM reordering + Remedy-2 (C)

poi_gen

```

OLDtoNEW = (int *) allocate_vector(sizeof(int), ICELTOT);
NEWtoOLD = (int *) allocate_vector(sizeof(int), ICELTOT);

RCM(ICELTOT, NL, NU, INL, IAL, INU, IAU,
    &NCOLORtot, COLORindex, NEWtoOLD, OLDtoNEW);

```

solver_PCG

```

N3= N+N2;
W = (double **)malloc(sizeof(double *)*4);
...
for (i=0; i<4; i++) {
    W[i] = (double *)malloc(sizeof(double)*N3);
    ...
}

```

N2=128

sol10: Original (L1-sol) (F)

poi_gen

```
allocate (OLDtoNEW(ICELTOT), NEWtoOLD(ICELTOT))

do i= 1, ICELTOT
  OLDtoNEW(i)= i
  NEWtoOLD(i)= i
enddo
```

solver_PCG

```
allocate (W(N,4))
```

sol20: +CM reordering (C)

poi_gen

```
allocate (OLDtoNEW(ICELTOT), NEWtoOLD(ICELTOT))  
  
call RCM (ICELTOT, NL, NU, INL, IAL, INU, IAU,      &  
          NCOLORTot, COLORindex, NEWtoOLD, OLDtoNEW)
```

solver_PCG

```
allocate (W(N,4))
```

sol2x: +CM reordering + Remedy-2 (F)

poi_gen

```
allocate (OLDtoNEW(ICELTOT), NEWtoOLD(ICELTOT))
```

```
call RCM (ICELTOT, NL, NU, INL, IAL, INU, IAU,      &  
          NCOLORtot, COLORindex, NEWtoOLD, OLDtoNEW)
```

solver_PCG

```
allocate (W(N+N2, 4))
```

N2=128

Results Fortran

	NX=NY=NZ=128 2,097,152 meshes Load/Store= 8.28×10^{10}	NX=NY=NZ=129 2,146,689 meshes Load/Store= 8.53×10^{10}
sol10 (original)	19.50 sec. 24.11 GB/sec 13.59 %	9.15 sec. 52.98 GB/sec 3.97 %
sol20 (CM)	10.15 sec. 45.60 GB/sec 5.65 %	9.44 sec. 50.64 GB/sec 4.11 %
sol2x (CM+ Remedy-2)	9.69 sec. 47.77 GB/sec 4.20 %	9.54 sec. 50.12 GB/sec 4.11 %

C (N=128³)

- sol10: 20.03 sec.
- sol20: 13.72
- sol2x: 10.05

- Comp. Time
- Memory Throughput
- L1D Miss Ratio
(to Load/Store)

Cache Thrashing

- Problem is that the cache is 2-way on FX10
- Most of modern architectures based on 4-way or 8-way
- FX-100 (successor of FX10) has 4-way cache