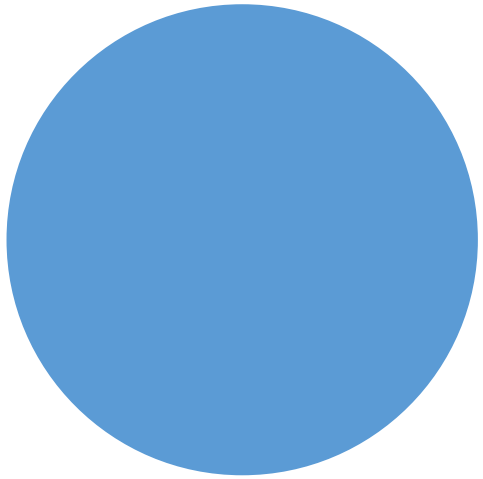
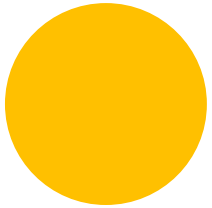



OpenMP Day 2

WorkSharing, Schedule, Synchronization and OMP best practices

Recap of Day 1

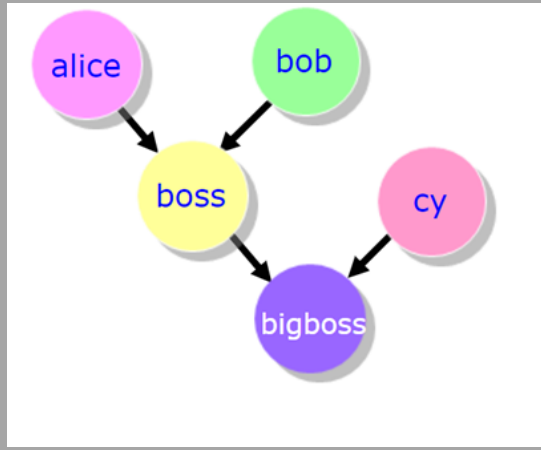
- ✓ What is OPENMP?
- ✓ Fork/Join Programming model
- ✓ OPENMP Core Elements
- ✓ #pragma omp parallel OR Parallel construct
- ✓ run time variables
- ✓ environment variables
- ✓ data scoping (private, shared…)
- ✓ compile and run openmp program in c++ and fortran
- ✓ work sharing constructs
  - #pragma omp for
  - sections
  - tasks
- ➢ schedule clause
- ➢ synchronization

Work Sharing: sections

SECTIONS directive is a non-iterative work-sharing construct.

It specifies that the enclosed section(s) of code are to be divided among the threads in the team.

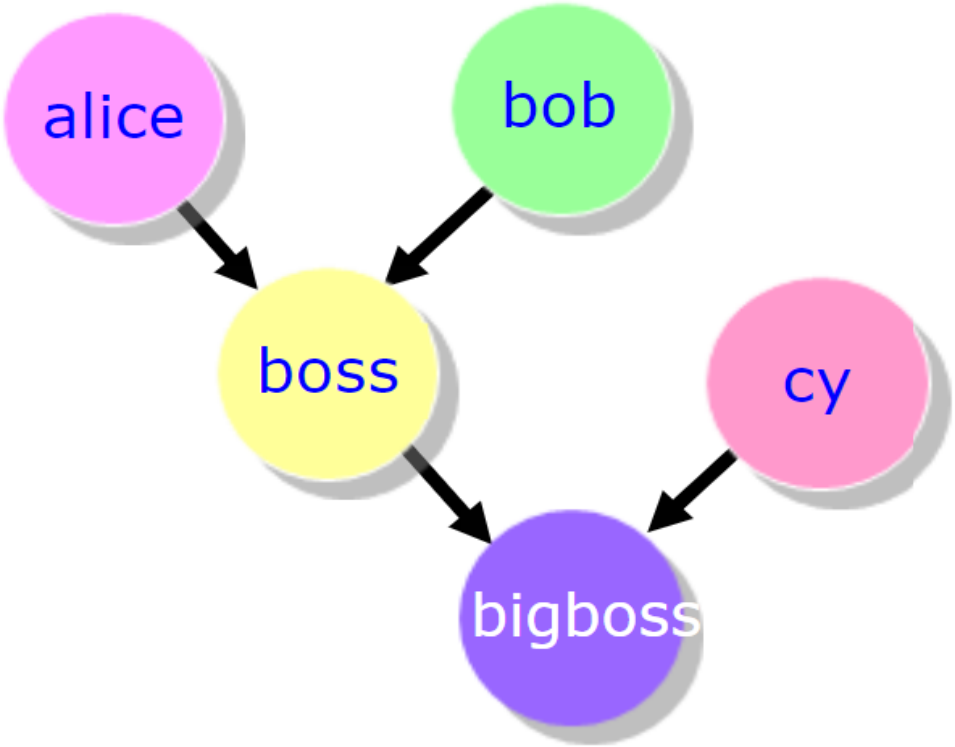
Each SECTION is executed ONCE by a thread in the team.




```
a = alice();
b = bob();
s = boss(a, b);
c = cy();
printf ("%6.2f\n", bigboss(s,c));
```

Work Sharing: sections

```
#pragma omp parallel sections
{
#pragma omp section
    double a = alice();
#pragma omp section
    double b = bob();
#pragma omp section
    double c = cy();
}

 double s = boss(a, b);
 printf ("%6.2f\n", bigboss(s,c));
```



OpenMP: lastprivate Clause

```
!$OMP DO PRIVATE(I)
LASTPRIVATE(B)
DO i = 1, 1000
B = i
ENDDO
!$OMP END DO
!—value of B here is
1000
```

```
!$OMP SECTIONS
LASTPRIVATE(B)
!$OMP SECTION
      B = 2
!$OMP SECTION
      B = 4
!$OMP SECTION
      D = 6
!$OMP END SECTIONS
```

- Creates private memory location for each thread.
- Does not initialize the private variable.
- The sequentially last iteration of the associated loops, or the lexically last section construct [...] to the original list item.

Work Sharing: tasks

#pragma omp task [clauses]……

- Tasks allow to parallelize irregular problems (Unbounded loops & Recursive algorithms)
- A task has - Code to execute – Data environment (It owns its data) – Internal control variables – An assigned thread that executes the code and the data
- Each encountering thread packages a new instance of a task (code and data)
- Some thread in the team executes the task at some later time

Work Sharing: tasks

Fibonacci series:

f(1) = 1

f(2) = 1

f(n) = f(n-1) + f(n-2)

```
/* serial code to compute Fibonacci */
int fib(int n)
{
   int i, j;
   if(n < 2) return n;
   i = fib(n-1);
   j = fib(n-2);
   return (i+j);
}
int main(){
   int n = 8;
   printf("fib(%d) = %d\n", n, fib(n);
}
```

```
Static int fib(int n){
   int i, j, id;
   if(n < 2)
      return n;
   #pragma omp task shared (i) private (id)
   {
      i = fib(n-1);
   }
   #pragma omp task shared (j) private (id)
   {
      j = fib(n-2);
   }
   return (i+j);
}
```

#pragma omp taskwait

Work Sharing: single

- The SINGLE directive specifies that the enclosed code is to be executed by only one thread in the team.

- May be useful when dealing with sections of code that are not thread safe (such as I/O)

```
!$OMP SINGLE [clause ...]
    PRIVATE (list)
    FIRSTPRIVATE (list)
   block
!$OMP END SINGLE [ NOWAIT ]
```

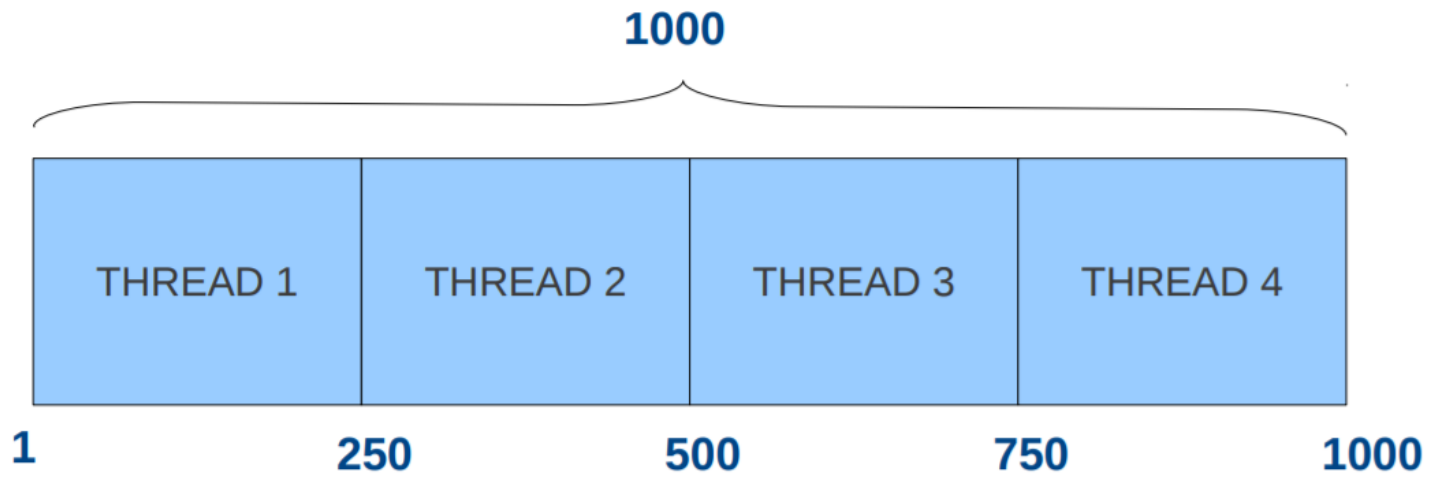
```
#pragma omp single [clause ...] newline private (list)
firstprivate (list) nowait structured_block
```

Schedule Clause

How is the work is divided among threads? Directives for work distribution

Although the OpenMP standard does not specify how a loop should be partitioned most compilers split the loop in N/p (N #iterations, p #threads) chunks by default. This is called a static schedule (with chunk size N/p)

For example, suppose we have a loop with 1000 iterations and 4 omp threads. The loop is partitioned as follows:



Schedule Clause: Types

A schedule kind is passed to an OpenMP loop schedule clause:

- Provides a hint for how iterations of the corresponding OpenMP loop should be assigned to threads in the team of the OpenMP region surrounding the loop.
- Five kinds of schedules for OpenMP loop1:

  static
  dynamic
  guided
  auto
  runtime

- The OpenMP implementation and/or runtime defines how to assign chunks to threads of a team given the kind of schedule specified by as a hint.

Schedule Clause

STATIC: Iterations of a loop are divided into chunks of size **ceiling**(*iterations/threads*). Each thread is assigned a separate chunk.
STATIC, N: Iterations of a loop are divided into chunks of size ***N***. Each chunk is assigned to a thread in *round-robin* fashion. ***N*** >= 1 (integer expression)

DYNAMIC: Iterations of a loop are divided into chunks of size 1.
Chunks are assigned to threads on a first-come, first-serve basis as threads become available. This continues until all work is completed.
DYNAMIC, N: Same as above, all chunks are set to size **N**

GUIDED: Chunks are made progressively smaller until a chunk size of one is reached. The first chunk is of size **ceiling**(*iterations/threads*). Remaining chunks are of size **ceiling**(*iterations_remaining/threads*).Chunks are assigned to threads on a first-come, first-serve basis as threads become available. This continues until all work is completed.
GUIDED, N: Minimum chunk size is N

AUTO: Delegated the decision of the scheduling to the compiler and/or runtime system
RUNTIME: Scheduling policy is determined at run time. OMP_SCHEDULE/ OMP_SET_SCHEDULE

OpenMP: Synchronization

- The programmer needs finer control over how variables are shared.
- The programmer must ensure that threads do not interfere with each other so that the output does not depend on how the individual threads are scheduled.
- In particular, the programmer must manage threads so that they read the correct values of a variable and that multiple threads do not try to write to a variable at the same time.
- Data dependencies and Task Dependencies

- MASTER, CRITICAL, BARRIER, FLUSH, TASKWAIT, ORDERED, NOWAIT

Data Dependencies

OpenMP assumes that there is NO data-dependency across jobs running in parallel

When the **omp parallel** directive is placed around a code block, it is the programmer's responsibility to make sure data dependency is ruled out

Synchronization Constructs

1) Mutual Exclusion (Data Dependencies)

Critical Sections : Protect access to shared & modifiable data, allowing ONLY ONE thread to enter it at a given time

#pragma omp critical

#pragma omp atomic – special case of **critical**, less overhead

Locks

Only one thread updates this at a time

```
float dot_prod(float* a, float* b, int N)
{
    float sum = 0.0;
#pragma omp parallel for shared(sum)
    for(int i=0; i<N; i++) {
#pragma omp critical
        sum += a[i] * b[i];
    }
    return sum;
}
```

Synchronization Constructs

To impose order constraints and protect shared data.

Achieved by **Mutual Exclusion & Barriers**

2) Barriers (Task Dependencies)

Implicit : Sync points exist at the end of

parallel –necessary barrier – cant be removed

for – can be removed by using the **nowait** clause

sections – can be removed by using the **nowait** clause

single – can be removed by using the **nowait** clause

Explicit : Must be used when ordering is required

#pragma omp barrier

each thread waits until all threads arrive at the barrier

Synchronization: Barrier

```
#pragma omp parallel private(id)
{
   id=omp_get_thread_num();
   A[id] = big_calc1(id);
   #pragma omp barrier

   #pragma omp for
   for(i=0;i<N;i++)
   {
      C[i]=big_calc3(i,A);
   }

   #pragma omp for nowait
   for(i=0;i<N;i++)
   {
      B[i]=big_calc2(C, i);
   }
   A[id] = big_calc4(id);
}
```

Explicit Barrier

Implicit Barrier at end of parallel region

No Barrier
nowait cancels barrier creation

OPENMP Synchronization: review

PRAGMA	DESCRIPTION
#pragma omp taskwait !$OMP TASKWAIT	Specifies a wait on the completion of child tasks generated since the beginning of the current task
#pragma omp critical !$OMP CRITICAL !$OMP END CRITICAL	Code within the block or pragma is only executed on one thread at a time.
#pragma omp critical !$OMP ATOMIC !$OMP END ATOMIC	Provides a mini-CRITICAL section. specific memory location must be updated atomically (Atomic statements)
#pragma omp barrier !$OMP BARRIER !$OMP END BARRIER	Synchronizes all threads in a team; all threads pause at the barrier, until all threads execute the barrier.

OPENMP Synchronization: review

PRAGMA	DESCRIPTION
#pragma omp for **ordered** *[clauses...] (loop region)* #pragma omp **ordered** *structured_block*	Used within a DO / for loop Iterations of the enclosed loop will be executed in the same order as if they were executed on a serial processor. Threads will need to wait before executing their chunk of iterations if previous iterations haven't completed yet.
#pragma omp flush *(list)*	Synchronization point at which all threads have the same view of memory for all shared objects. FLUSH is implied for barrier parallel - upon entry and exit critical - upon entry and exit ordered - upon entry and exit for - upon exit sections - upon exit single - upon exi

Performance in OPENMP Programs

Performance in OPENMP programs

Might not change speed much or break code!

Must understand application and use wisely

Performance of single threaded code

Percentage of code that is run in parallel and scalability

CPU utilization, effective data sharing, data locality and load balancing

Amount of synchronization and communication

Overhead to create, resume, manage, suspend, destroy and synchronize threads

Memory conflicts due to shared memory or falsely shared memory

Performance limitations of shared resources e.g memory, bus bandwidth, CPU execution units

Computing Efficiency of Parallel Code

$\frac{T_{serial}}{T_{parallel}}$ = Speed Up

$\frac{T_{serial}}{P \times T_{parallel}}$ = Efficiency, P = number of processors

$$T_{CPU}(P) = (1 + O_P \cdot P) \cdot T_{serial}$$

$$T_{Elapsed}(P) = (\frac{f}{P} - f + 1 + O_P \cdot P) \cdot T_{serial}$$

$$Speedup(P) = \frac{T_{Serial}(P)}{T_{Elapsed}(P)} = \frac{1}{\frac{f}{P} - f + 1 + O_P \cdot P}$$

$$Efficiency(P) = \frac{Speedup(P)}{P}$$

Key Steps in Parallel Algorithms

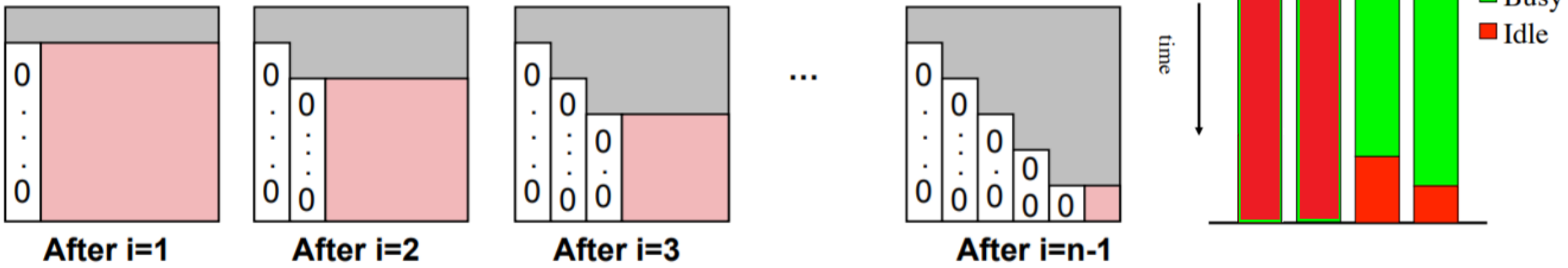
- Dividing a computation into smaller computations
- Assign them to different processors for parallel execution
- The process of dividing a computation into smaller parts, some or all of which may potentially be executed in parallel, is called ***decomposition***
- The number and size of tasks into which a problem is decomposed determines the ***granularity*** of the decomposition.
- Decomposition into a large number of small tasks is called ***fine-grained*** and a decomposition into a small number of large tasks is called ***coarse-grained***
- Decomposition for matrix-vector multiplication is fine-grained because each of a large number of tasks performs a single dot-product.
- A coarse-grained decomposition of the same problem into 4 tasks, where each task computes $n/4$ of the entries of the output vector of length n.
- The mechanism by which tasks are assigned to processes for execution is called ***mapping***.

Data decomposition & mapping to Processors
OR
Task decomposition & mapping to Processors
Objective
All tasks complete in the shortest amount of elapsed time
How to achieve the objective?
Reduce overheads:
Time spent in inter-process interaction/ Overheads of data sharing between processes.
Time that some processes may spend being idle.

Uneven load distribution may cause some processes to finish earlier than others.
Unfinished tasks mapped onto a process could be waiting for tasks mapped onto other processes to finish in order to satisfy the constraints imposed by the task-dependency graph.

Load Balancing: Gaussian Elimination

Conversion of a Matrix into its Upper Triangular Equivalent
Simple Data Partitioning method for parallel processing –
1D vertical strip partitioning
Each process owns N/P columns of data
The ▭ represents outstanding work in successive K iterations



When eliminating a column, processors to the left of are idle
Each processor is active for only part of the computation

Several Data & Task Decomposition and mapping techniques (Beyond the scope of this talk)

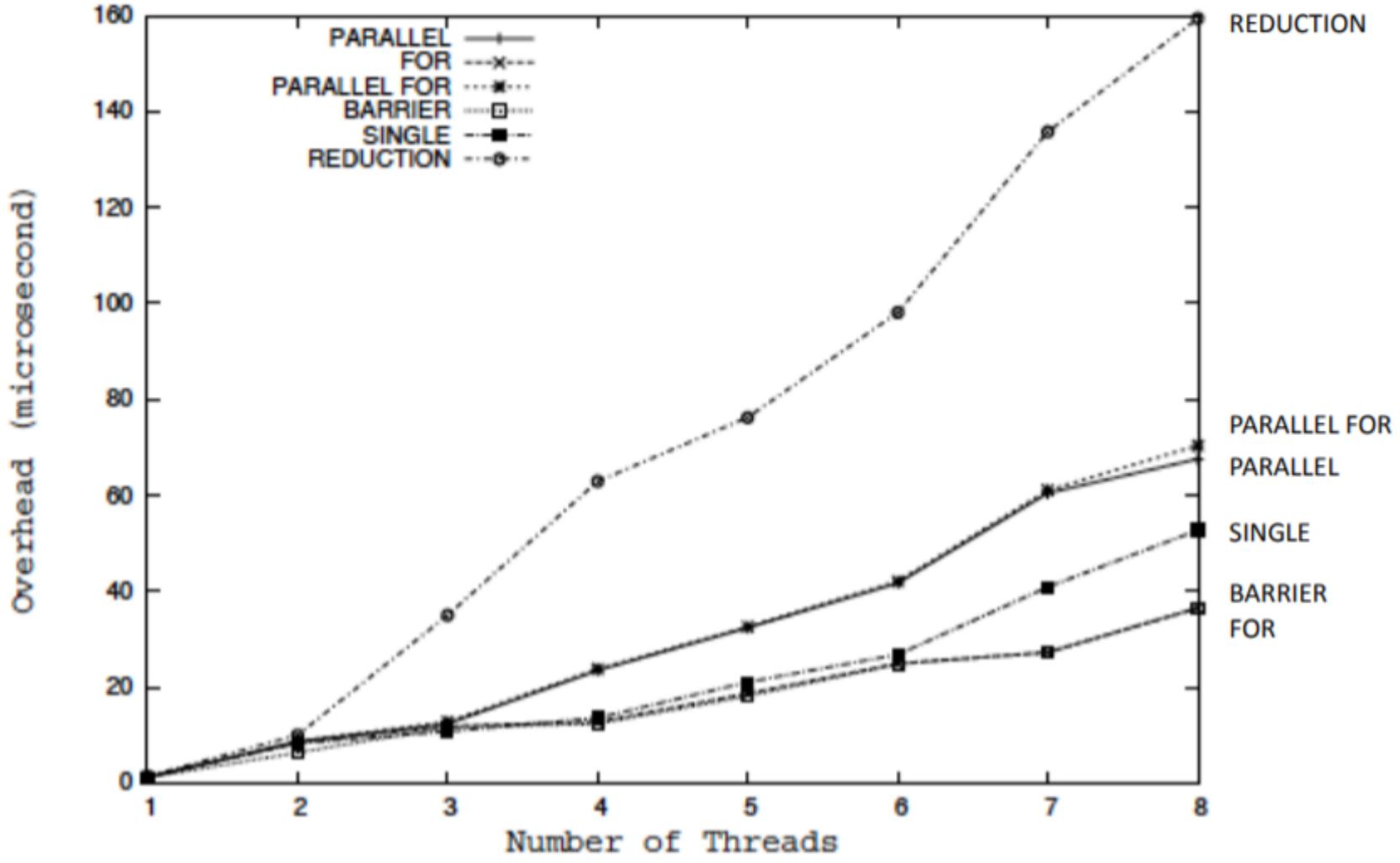
Some simple techniques to avoid overheads.

Parallel Overhead

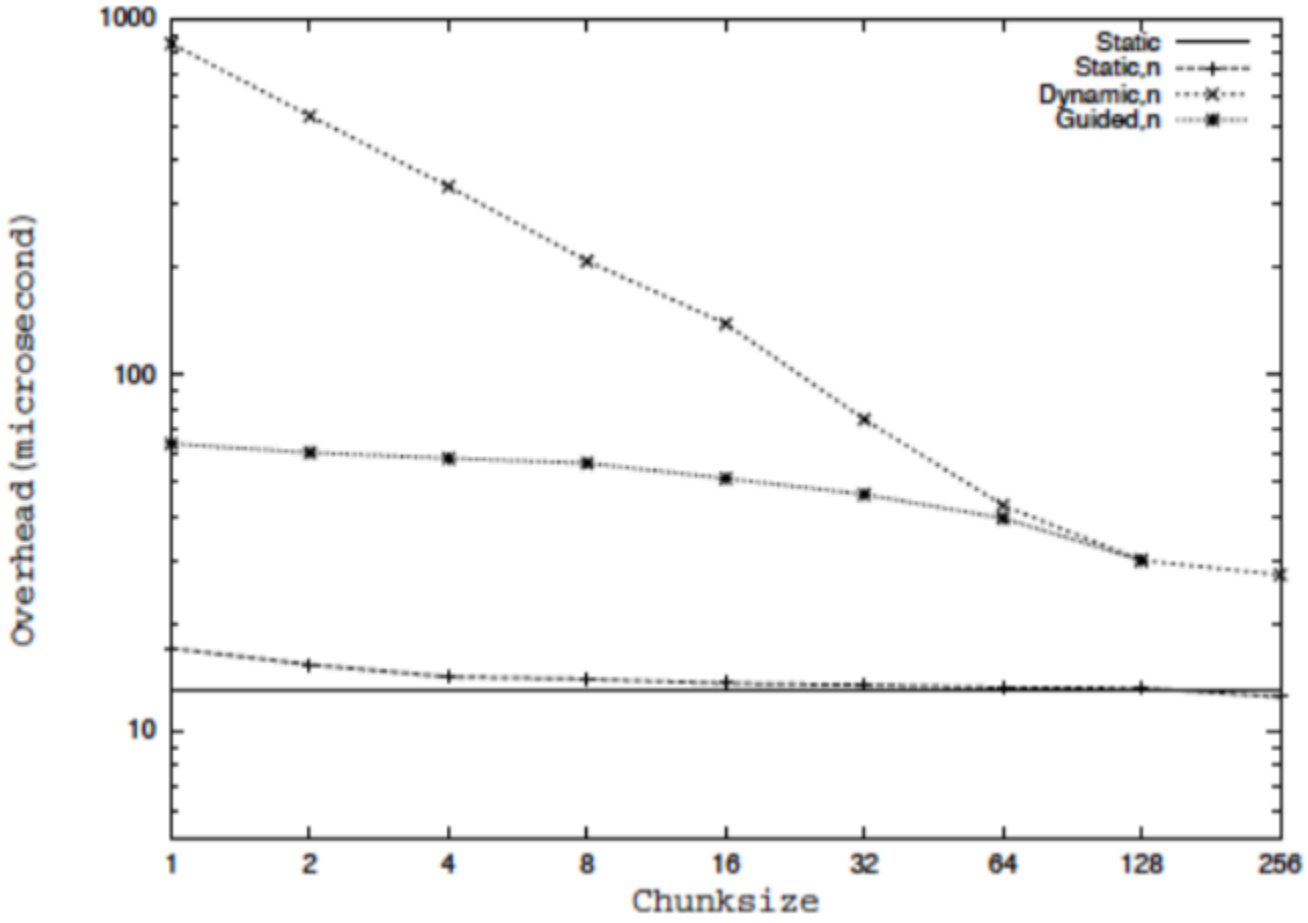
The amount of time required to coordinate parallel threads, as opposed to doing useful work.

Thread start-up time
Synchronization
Software overhead imposed by parallel compilers, libraries, tools, operating system, etc.
Thread termination time

Overheads of Parallel Directives



Overheads of Scheduling

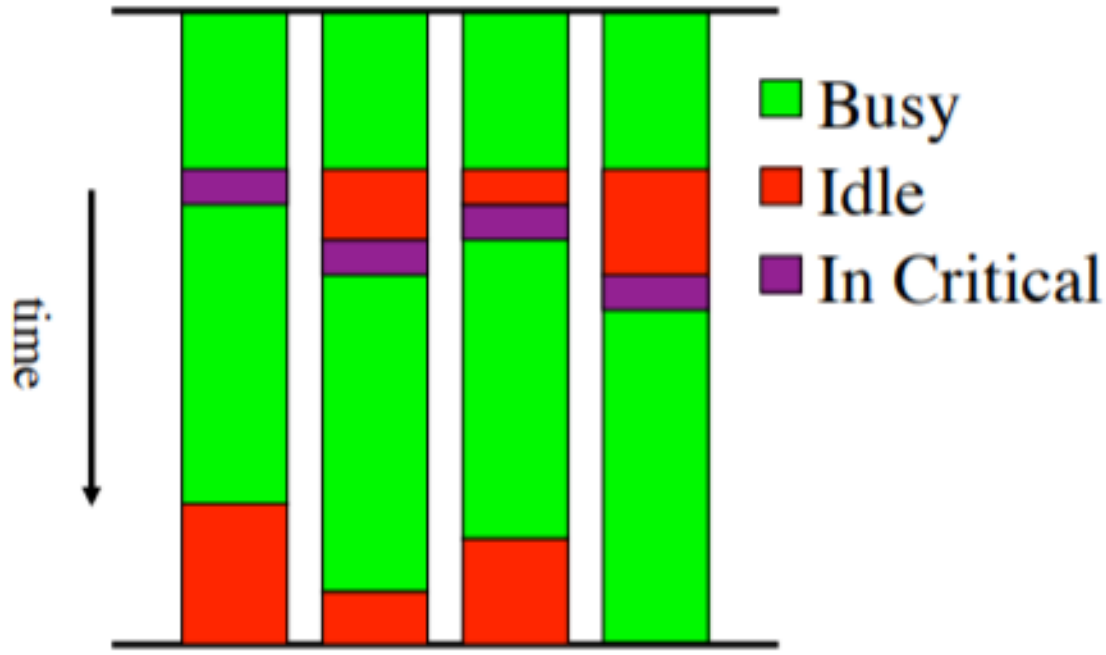


Optimize the use of barriers

```
#pragma omp parallel default(none) \
        shared(n,a,b,c,d,sum) private(i)
{
   #pragma omp for nowait
   for (i=0; i<n; i++)
      a[i] += b[i];

   #pragma omp for nowait
   for (i=0; i<n; i++)
      c[i] += d[i];

   #pragma omp barrier

   #pragma omp for nowait reduction(+:sum)
   for (i=0; i<n; i++)
      sum += a[i] + c[i];
} /*-- End of parallel region --*/
```

Prefer Atomic to Critical & Avoid Large critical regions

```
#pragma omp parallel
{

  #pragma omp critical
  {
    ...
  }
  ...
}
```



Maximize Parallel Regions

```
#pragma omp parallel
{
    #pragma omp for  /*-- Work-sharing loop 1 --*/
    { ..... }

    #pragma omp for  /*-- Work-sharing loop 2 --*/
    { ..... }

       .........

    #pragma omp for  /*-- Work-sharing loop N --*/
    { ..... }
}
```

Single parallel region enclosing all work sharing for loops.

Avoid Parallel Regions in inner loops

```
for (i=0; i<n; i++)
    for (j=0; j<n; j++)
        #pragma omp parallel for
        for (k=0; k<n; k++)
            { .........}
```

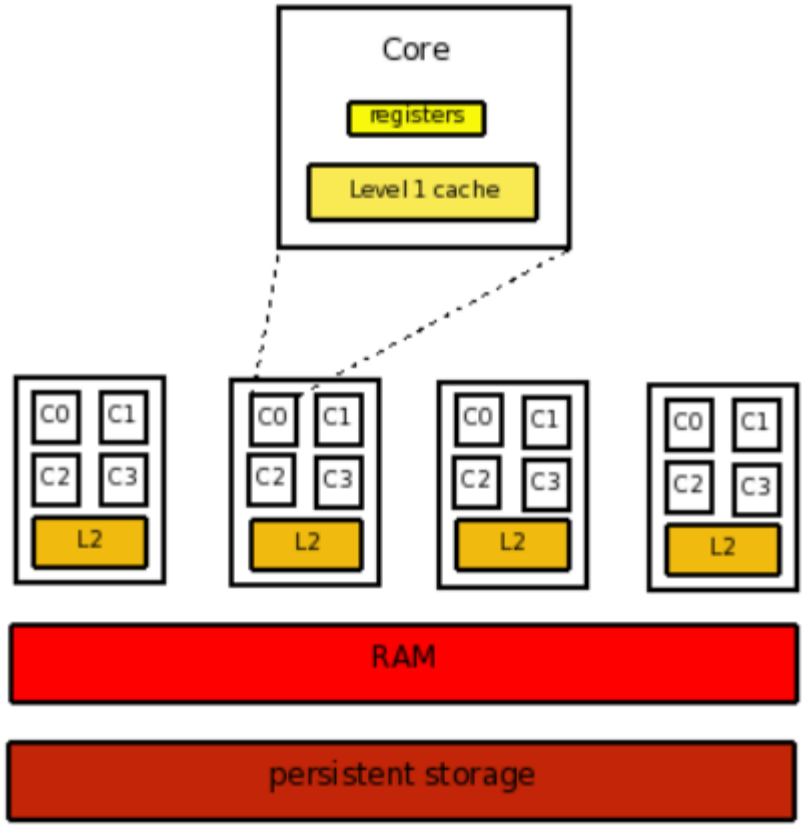
Figure 5.25: **Parallel region embedded in a loop nest** – The overheads of the parallel region are incurred n^2 times.

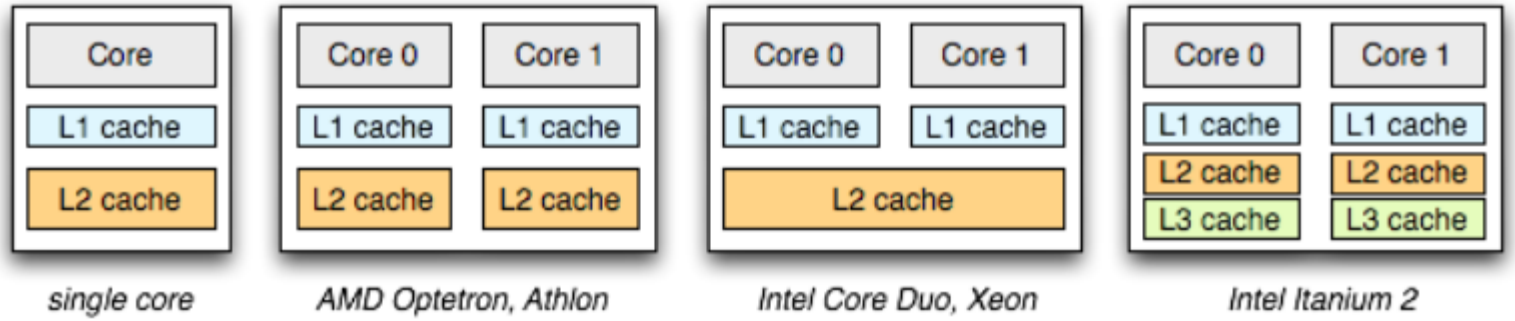
```
#pragma omp parallel
   for (i=0; i<n; i++)
       for (j=0; j<n; j++)
           #pragma omp for
           for (k=0; k<n; k++)
               { .........}
```

Figure 5.26: **Parallel region moved outside of the loop nest** – The parallel construct overheads are minimized.

Caching issues in multicore performance

Memory Architecture





Caching issues in multicore performance

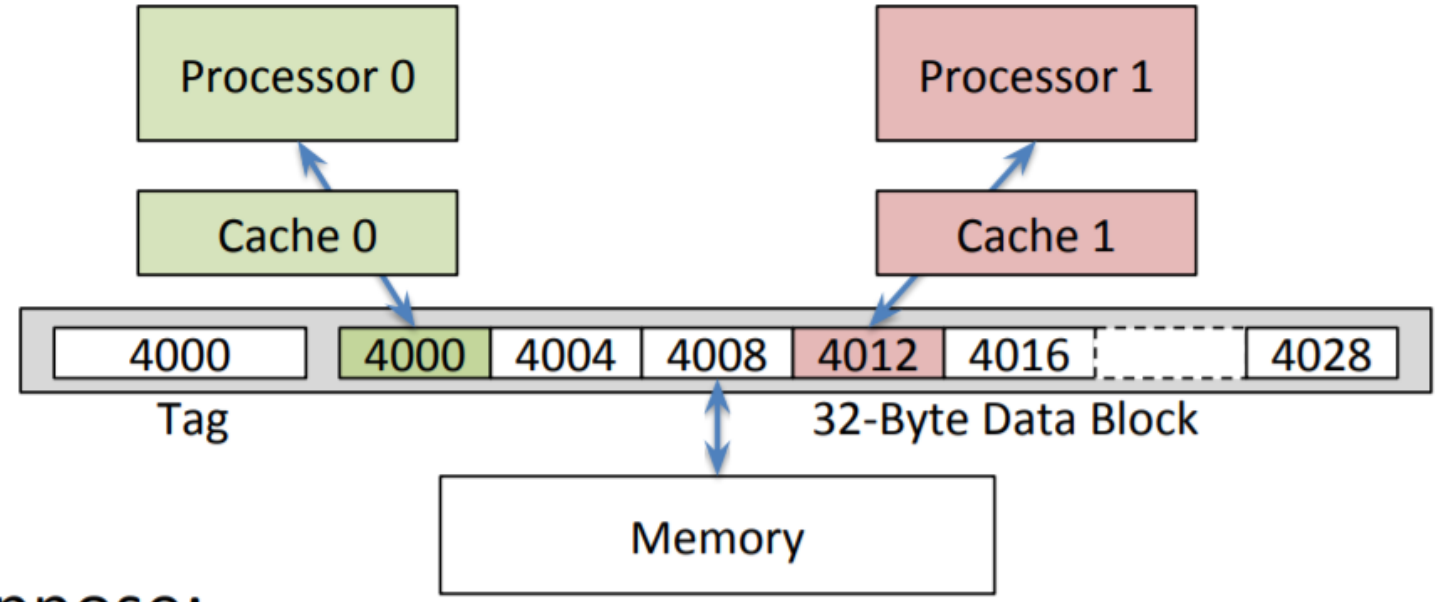
Each core has its own separate L2 cache, but a write by one can impact the state of the others.

For example, if one core writes a value into one of its own cache lines, any other core using a copy of that same cache line can no longer count on its values being up-to-date. In order to regain that confidence, the core that wrote must flush that cache line back to memory and the other core must then reload its copy of that cache line.

To maintain this organization, each core's L2 cache has 4 states (***MESI***):

1. **M**odified
2. **E**xclusive
3. **S**hared
4. **I**nvalid

Caching issues in multicore performance



- Suppose:
  - Block size is 32 bytes
  - P0 reading and writing variable X, P1 reading and writing variable Y
  - X in location 4000, Y in 4012
- What will happen?

False Sharing

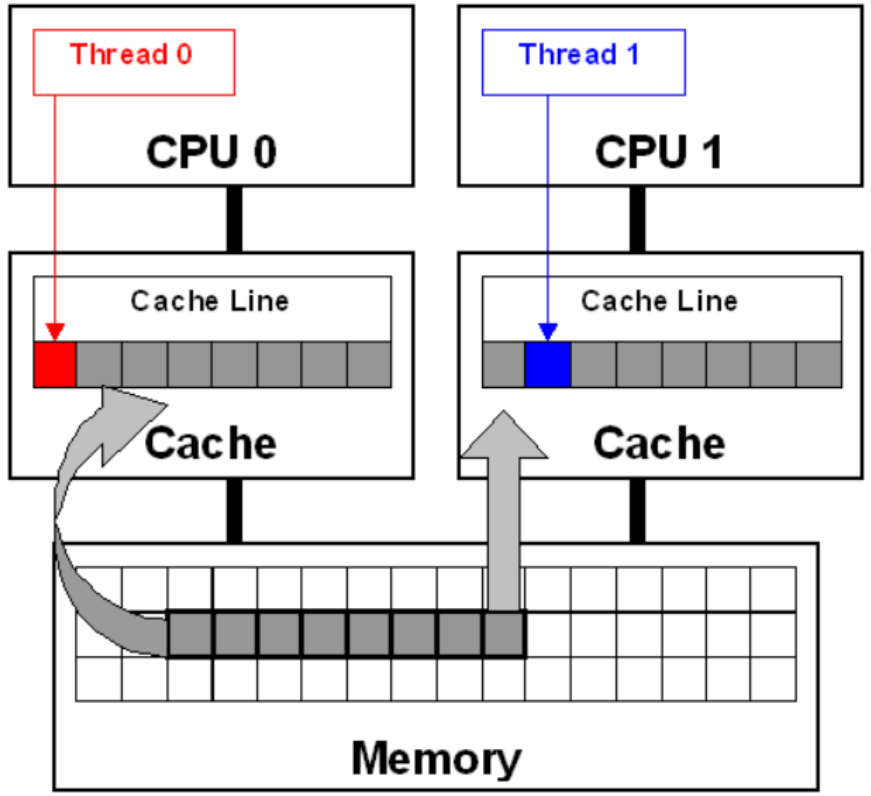
1. Core A reads a value. Those values are brought into its cache. That cache line is now tagged **Exclusive**.
2. Core B reads a value from the same area of memory. Those values are brought into its cache, and now both cache lines are re-tagged **Shared**.
3. If Core B writes into that value. Its cache line is re-tagged **Modified** and Core A's cache line is re-tagged **Invalid**.
4. Core A tries to read a value from that same part of memory. But its cache line is tagged **Invalid**. So, *Core B's cache line is flushed back to memory and then Core A's cache line is re-loaded from memory*. Both cache lines are now tagged **Shared**.

Array[N]
N is large. A[1] is accessed by one processor & A[2] by another

Step	Cache Line A	Cache Line B
1	Exclusive	-----
2	Shared	Shared
3	Invalid	Modified
4	Shared	Shared

False Sharing

```
double sum=0.0, sum_local[NUM_THREADS];
#pragma omp parallel num_threads(NUM_THREADS)
{
 int me = omp_get_thread_num();
 sum_local[me] = 0.0;

 #pragma omp for
 for (i = 0; i < N; i++)
 sum_local[me] += x[i] * y[i];

 #pragma omp atomic
 sum += sum_local[me];
}
```

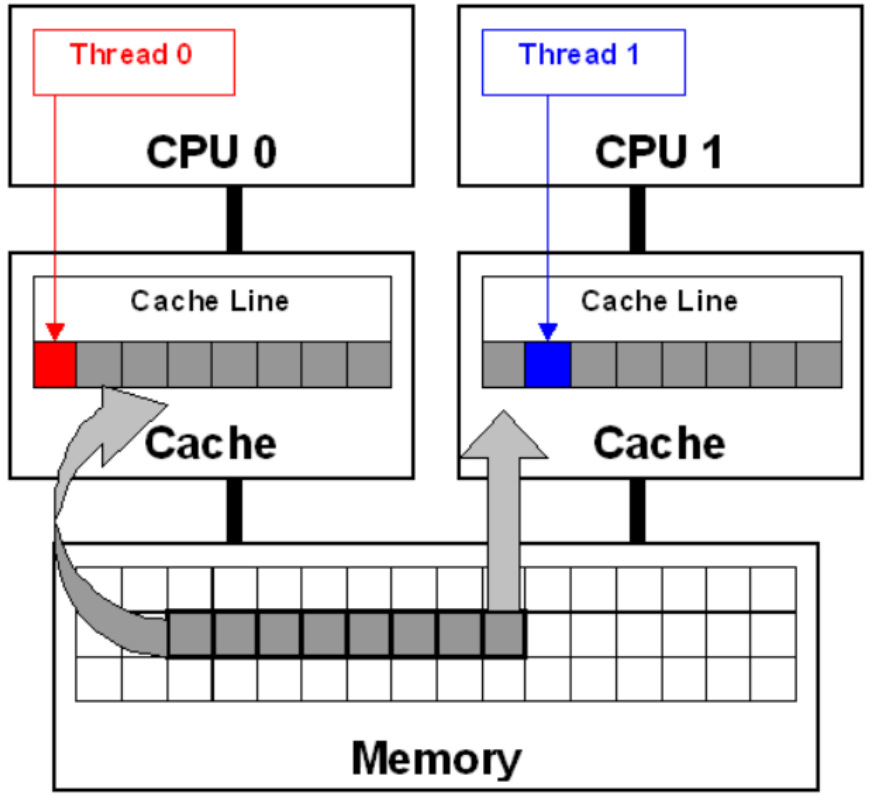


When threads on different processors modify variables that reside on the same cache line.
This invalidates the cache line and forces a memory update to maintain cache coherency.
Potential false sharing on the array **sum_local**.

"place" data on different blocks OR Reduce block size

False Sharing: Solution 1

```
double sum=0.0, sum_local[NUM_THREADS];
#pragma omp parallel num_threads(NUM_THREADS)
{
 int me = omp_get_thread_num();
 sum_local[me] = 0.0;

 #pragma omp for schedule(static,chunkSize)
 for (i = 0; i < N; i++)
 sum_local[me] += x[i] * y[i];

 #pragma omp atomic
 sum += sum_local[me];
}
```



Adding a schedule clause with chunksize that ensures that 2 threads do not step over the same cache line

False Sharing: Solution 2

```
double sum=0.0, sum_local[NUM_THREADS][cacheline];
#pragma omp parallel num_threads(NUM_THREADS)
{
 int me = omp_get_thread_num();
 sum_local[me] = 0.0;

 #pragma omp for
 for (i = 0; i < N; i++)
 sum_local[me][0] += x[i] * y[i];

 #pragma omp atomic
 sum += sum_local[me];
}
```

Use compiler directives to force individual variable alignment.

__declspec(align(n)) (**n** =64) (64 byte boundary) to align the individual variables on cache line boundaries.

```
__declspec (align(64)) int
thread1_global_variable;
__declspec (align(64)) int
thread2_global_variable;
```

Array padding and memory alignment to reduce false sharing. This works because successive Array elements are forced onto different cache lines, so less (or no) cache line conflicts exist

False Sharing: Solution 2

```
struct ThreadParams
{
  // For the following 4 variables: 4*4 = 16 bytes
  unsigned long thread_id;
  unsigned long v; // Frequent read/write access variable
  unsigned long start;
  unsigned long end;

  // expand to 64 bytes to avoid false-sharing
  // (4 unsigned long variables + 12 padding)*4 = 64
  int padding[12];
};

__declspec (align(64)) struct ThreadParams Array[10];
```

Padding a data structure to a cache line boundary
Ensuring the array is also aligned using the compiler

__declspec (align(n)) [**n** = 64 (64 byte boundary)]

Array of data structures

- Pad the structure to the end of a cache line to ensure that the array elements begin on a cache line boundary.
- If you cannot ensure that the array is aligned on a cache line boundary, pad the data structure to twice the size of a cache line.
- If the array is dynamically allocated, increase the allocation size and adjust the pointer to align with a cache line boundary.

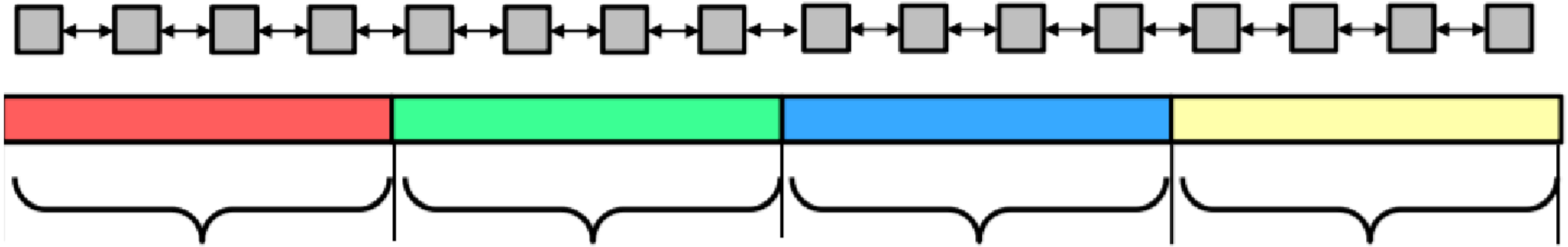
False Sharing: Solution 3

Use of private variables

Note: Shared data that is read-only in a loop does not lead to false sharing.

```
double sum=0.0, sum_local[NUM_THREADS];
#pragma omp parallel num_threads(NUM_THREADS)
{
 int me = omp_get_thread_num();
 sum_local[me] = 0.0;

 #pragma omp for private(ThreadLocalSum)
 for (i = 0; i < N; i++)
 ThreadLocalSum += x[i] * y[i];

 #pragma omp atomic
 sum += ThreadLocalSum;
}
```

False Sharing



One large global memory block – Shared
False Sharing?
Make sure each individual-block starts and ends at the cache boundary

Separate blocks each local to its own core (i.e. private)
No false sharing but detailed code to identify where each private block begins and ends.

Cache hits and misses

Cache could be KB or MB, but bytes are transferred in much smaller sizes. Typical size of cache line is 64MB
When CPU asks for a value from the memory
If the value is already in the cache -> **Cache Hit**
Value is not in the cache, has to be fetched from the memory -> **Cache Miss**

- Compulsory (cold start or process migration): - First access to a block in memory impossible to avoid
- Capacity: Cache cannot hold all blocks accessed by the program
- Conflict (collision): - Multiple memory locations map to same cache location

Cache hits and misses

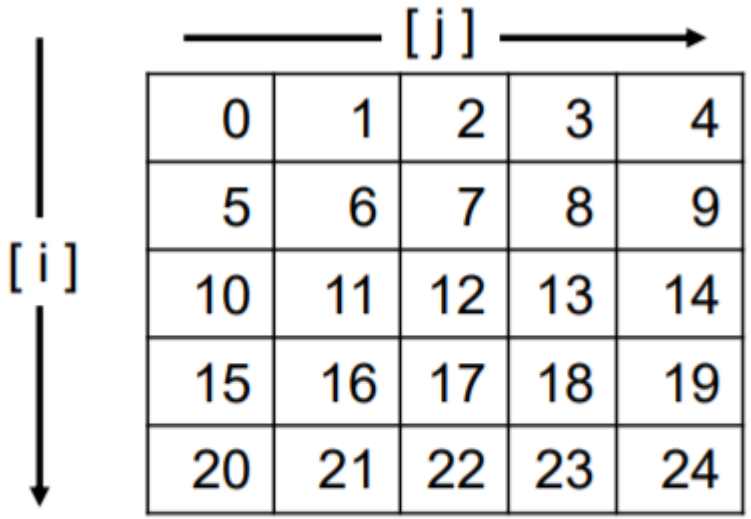
Coherence Misses: Misses caused by coherence traffic with other processor
Also known as communication misses because represents data moving between processors working together on a parallel program
For some parallel programs, coherence misses can dominate total misses

Spatial Coherence
"If you need one memory address's contents now, then you will probably also need the contents of some of the memory locations around it soon."

Temporal Coherence
"If you need one memory address's contents now, then you will probably also need its contents again soon."

Cache hits and misses

C and C++ store 2D arrays a row-at-a-time, like this, A[i][j]:



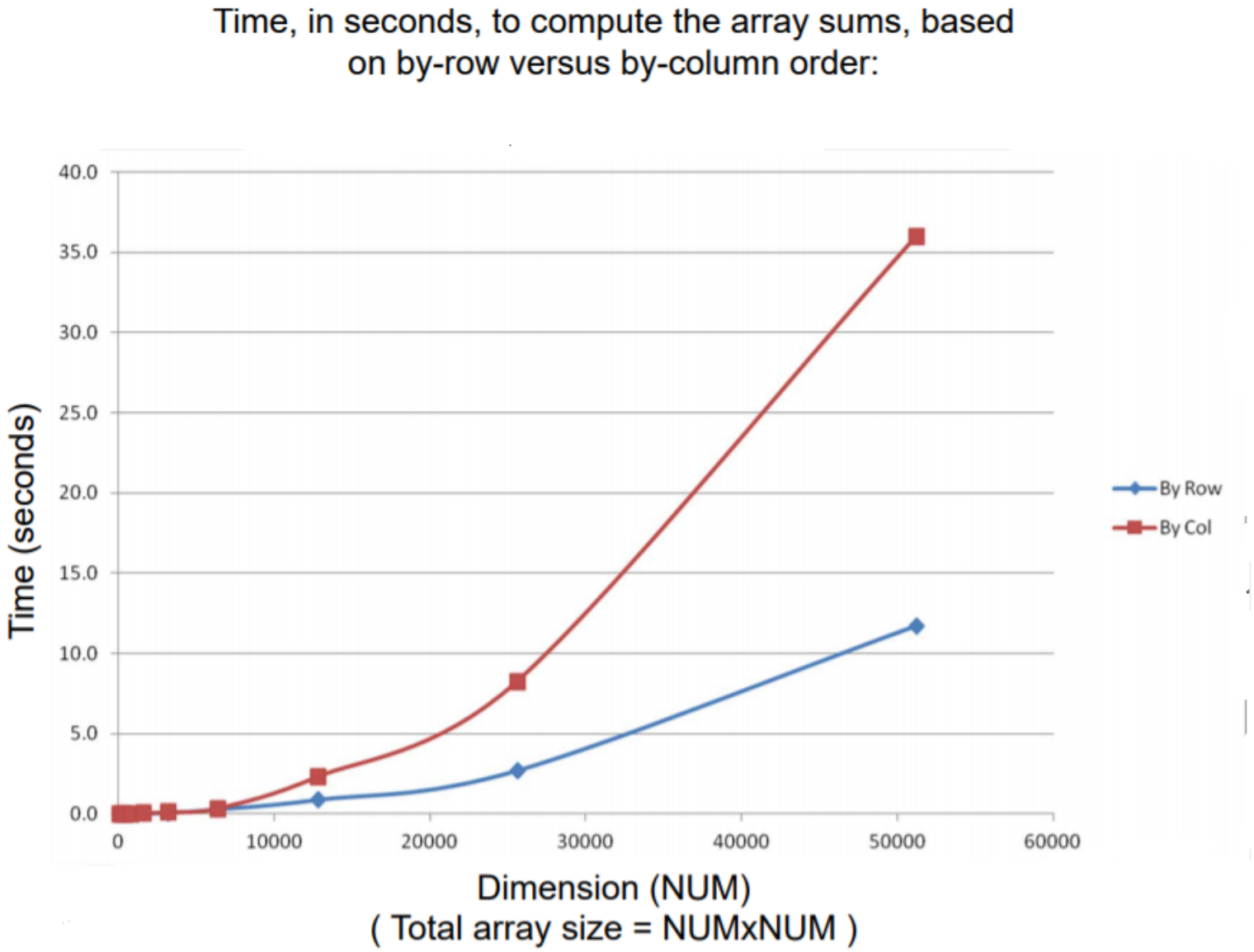

```
sum = 0.;
for( int i = 0; i < NUM; i++ )
{
        for( int j = 0; j < NUM; j++ )
        {
                float  f = ???
                sum += f;
        }
}
```

For large arrays, would it be better to add the elements by row, or by column? Which will avoid the most cache misses?

```
float  f  =  Array[ i ][ j ] ;
```
→ Sequential memory order

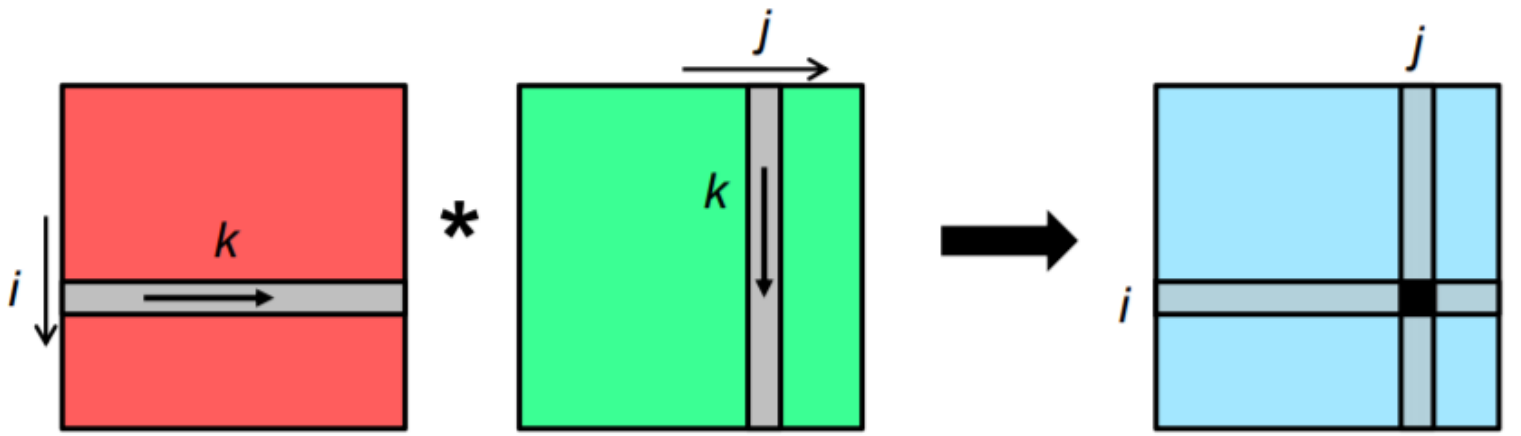
```
float  f  =  Array[ j][ i ] ;
```
→ Jump in memory order

Cache hits and misses



Where Cache Coherence Really Matters: Matrix Multiply

The usual approach is multiplying the entire A row * entire B column
This is equivalent to computing a single dot product

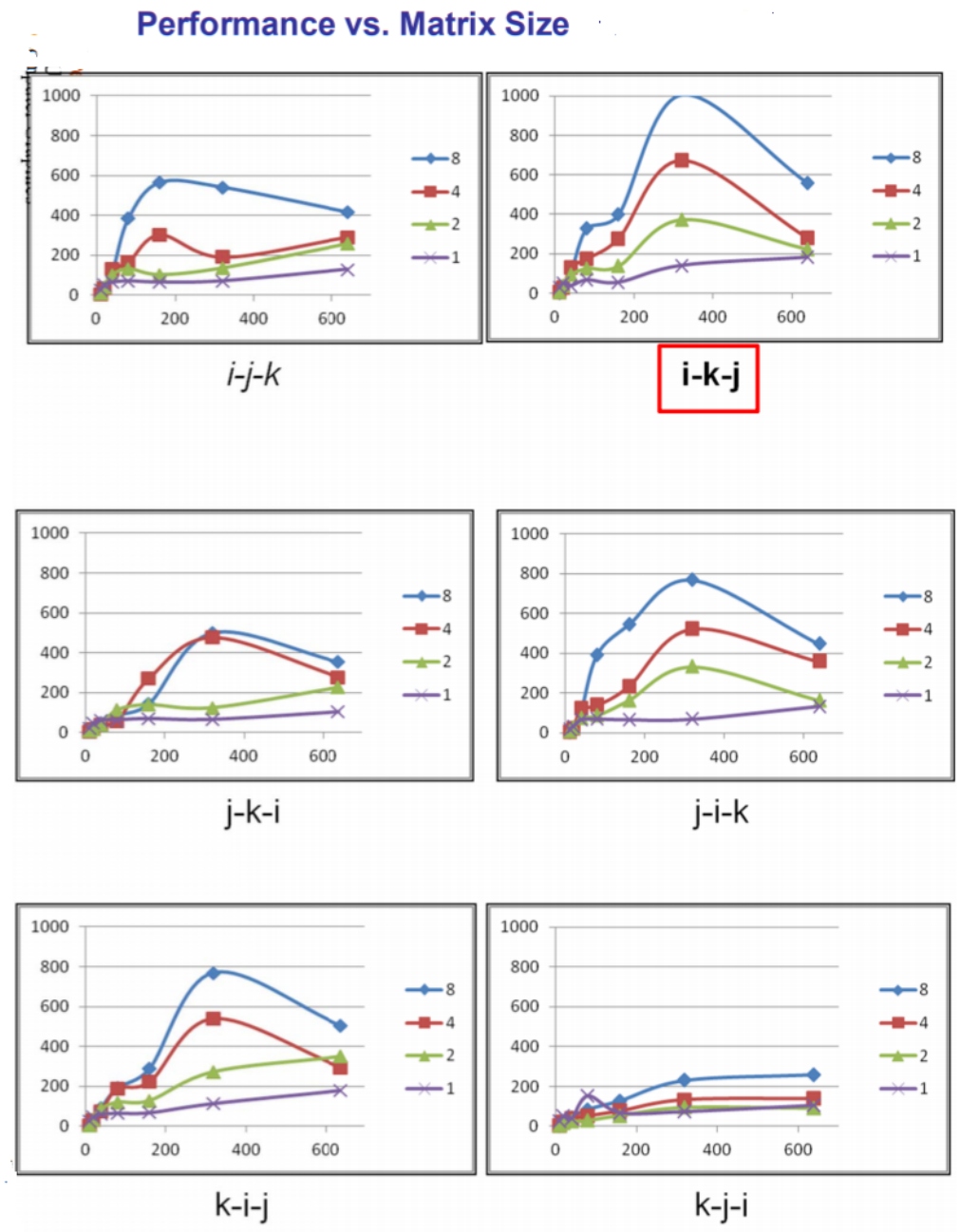



```
for( i = 0; i < SIZE; i++ )
    for( j = 0; j < SIZE; j++ )
        for( k = 0; k < SIZE; k++ )
```

$$\sum A[i][k] \quad * \quad B[k][j] \quad \xrightarrow{\text{Sum and store}} \quad C[i][j]$$

Code simplicity!
Blindly marches through memory (how does this affect the cache?)
This is a problem in a C /C++ program because B is not doing a unit stride

Where Cache Coherence Really Matters: Matrix Multiply

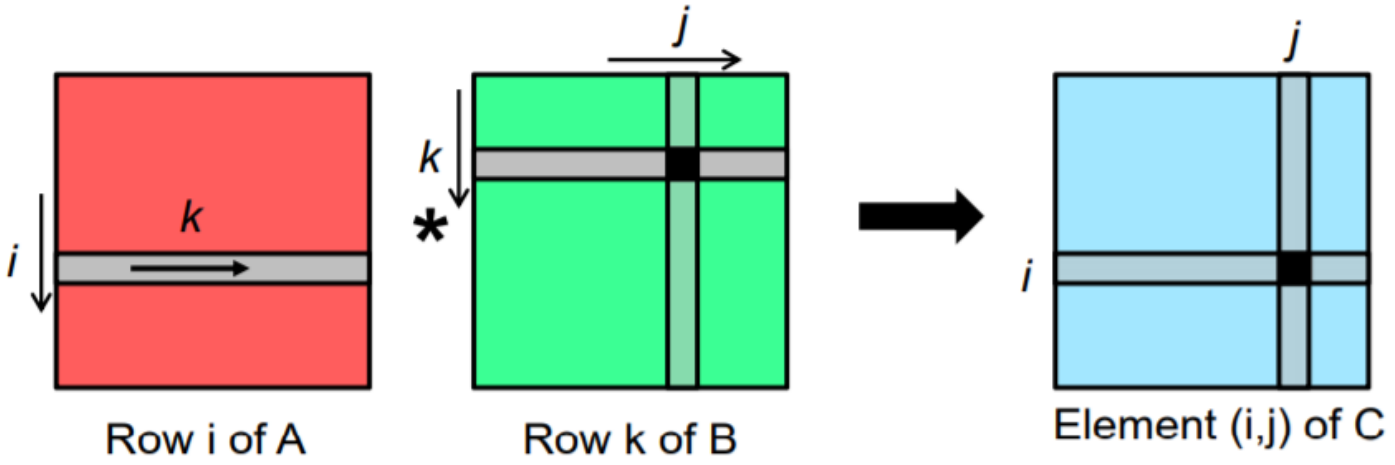


Where Cache Coherence Really Matters: Matrix Multiply

Scalable Universal Matrix Multiply Algorithm (SUMMA)

Entire A row * one element of B row

Equivalent to computing one item in many separate dot products



I, k, j

```
for( i = 0; i < SIZE; i++ )
    for( k = 0; k < SIZE; k++ )
        for( j = 0; j < SIZE; j++ )
            A[ i ][ k ]  *  B[ k ][ j ]  →  C[ i ][ j ]
```

Add to

Block Dense Matrix Multiplication

Usually size of matrices (N) much larger than number of processors (p).
Divide matrix into s^2 submatrices.
Each submatrix has **N/s x N/s** elements.

C_{11}	C_{12}	C_{13}	C_{14}
C_{21}	C_{22}	C_{23}	C_{24}
C_{31}	C_{32}	C_{43}	C_{34}
C_{41}	C_{42}	C_{43}	C_{44}

A_{11}	A_{12}	A_{13}	A_{14}
A_{21}	A_{22}	A_{23}	A_{24}
A_{31}	A_{32}	A_{33}	A_{34}
A_{41}	A_{42}	A_{43}	A_{144}

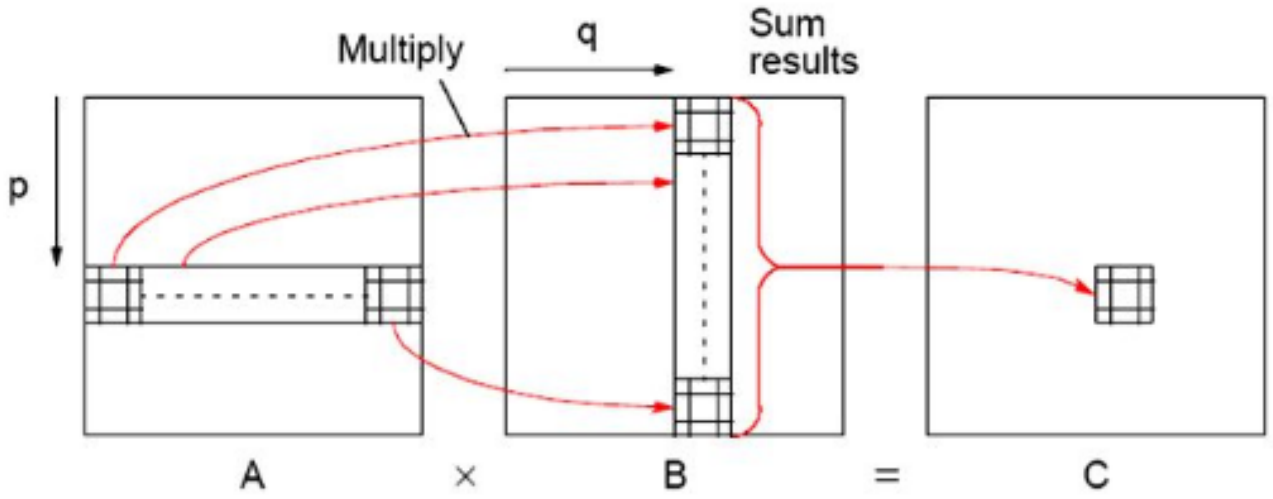
B_{11}	B_{12}	B_{13}	B_{14}
B_{21}	B_{22}	B_{23}	B_{24}
B_{32}	B_{32}	B_{33}	B_{34}
B_{41}	B_{42}	B_{43}	B_{44}

$$C_{22} = A_{21}B_{12} + A_{22}B_{22} + A_{23}B_{32} + A_{24}B_{42} = \sum_k A_{2k} * B_{k2}$$

Block Dense Matrix Multiplication

Usually size of matrices (n) much larger than number of processors (p).

Divide matrix into s^2 submatrices. Each submatrix has n/s x n/s elements.




```
for (p = 0; p < s; p++)
    for (q = 0; q < s; q++) {
      C[p,q] = 0;                          /* clear elements of submatrix*/
      for (r = 0; r < m; r++)               /* submatrix multiplication */
        C[p,q] = C[p,q] + A[p,r] * B[r,q];  /*add to accum. submatrix*/
    }
```

The line: $C_{p,q} = C_{p,q} + A_{p,r} * B_{r,q}$; means multiply submatrix $A_{p,r}$ and $B_{r,q}$ using matrix multiplication and add to submatrix $C_{p,q}$ using matrix addition.

Cache hits and misses

Mathematically calculating cache misses using cache block and line sizes and size of objects in the code.

Using some performance tools, like *perf*. One can identify cache hits and misses.

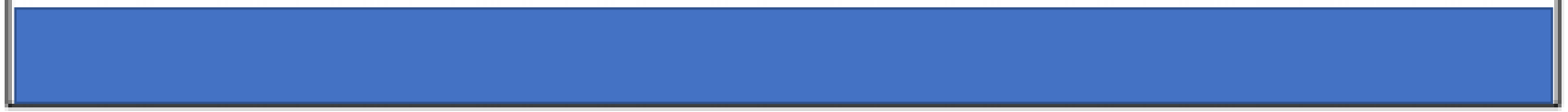
Perf comes with linux platforms.

```
$ perf stat -e task-clock,cycles,instructions,cache-references,cache-misses
./stream_c.exe
```

Exercise : Run perf on Matrix Vector multiplication code

Loop Unrolling

```
for (int i=1; i<n; i++) {
      a[i] = b[i] + 1;
      c[i] = a[i] + a[i-1] + b[i-1];
}
```



```
for (int i=1; i<n; i+=2) {
      a[i] = b[i] + 1;
      c[i] = a[i] + a[i-1] + b[i-1];
      a[i+1] = b[i+1] + 1;
      c[i+1] = a[i+1] + a[i] + b[i];
}
```

Loop Fusion

```
for (int i=0; i<n; i++)
     a[i] = b[i] * 2;
for (int i=0; i<n; i++)
   {
     x[i] = 2 * x[i];
     c[i] = a[i] + 2;
   }
```

Figure 5.10: **A pair of loops that both access array a** – The second loop reuses element a[i], but by the time it is executed, the cache line this element is part of may no longer be in the cache.

```
for (int i=0; i<n; i++)
 {
     a[i] = b[i] * 2;
     c[i] = a[i] + 2;
     x[i] = 2 * x[i];
 }
```

Figure 5.11: **An example of loop fusion** – The pair of loops of Figure 5.10 have been combined and the statements reordered. This permits the values of array a to be immediately reused.

OpenMP Parallel Programming

- Start with a parallelizable algorithm
  Loop level parallelism /tasks
- Implement Serially : Optimized Serial Program
- Test, Debug & Time to solution
- Annotate the code with parallelization and Synchronization directives
- Remove Race Conditions, False Sharing
- Test and Debug
- Measure speed-up (T-serial/T-parallel)