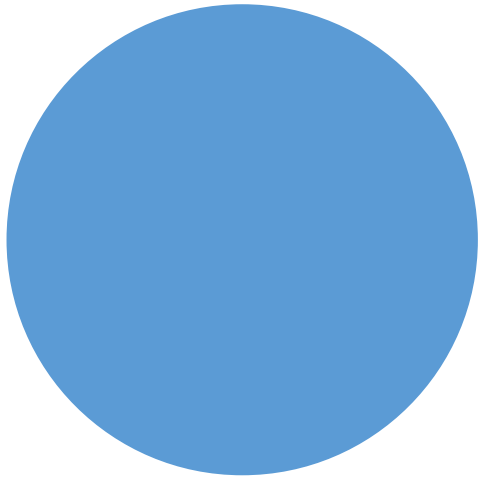
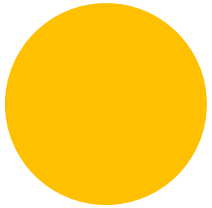



OpenMP Part 2

WorkSharing, Schedule, Synchronization and OMP best practices

Recap of Part 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…)
- ✓ work sharing constructs
  - #pragma omp for
- ✓ compile and run openmp program in c++ and fortran
- ✓ work sharing constructs
  - schedule clause
  - sections
  - tasks
- ➢ synchronization

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)

Installing and running C/C++/Fortran Programs on multicore machines

The GNU Project https://gcc.gnu.org/install/binaries.html
MacOS:
The Homebrew package manager : https://brew.sh/
MacPorts https://www.macports.org/

MS Windows
The Cygwin project https://sourceware.org/cygwin/
MinGW and mingw-w64 projects http://www.mingw.org/
http://mingw-w64.org/doku.php

Linux machines come pre-installed with gnu binaries

```
gcc --version
gfortran --version
```

Environment Variables

LINUX /UNIX bash /Cygwin:
List all environment variables: **printenv**
Update environment variable : **export OMP_NUM_THREADS=5**

LINUX/UNIX csh
List all environment variables: **printenv**
Update environment variable : **setenv OMP_NUM_THREADS 5**

Windows/DOS
List all environment variables: **set**
Update environment variable : **set** (/A) **OMP_NUM_THREADS= 5**

OSX
List all environment variables: **env**
Update environment variable : **export OMP_NUM_THREADS= 5**

Compiling and running OPENMP Code

Locally

```
$g++ -fopenmp Program.cpp -o <output_name>
$gfortran -fopenmp Program.f95 -o <output_name>
$./<output_name>
```

OpenMP environment variables

Display OPENMP environment upon execution of the program

$export OMP_DISPLAY_ENV=TRUE (bash)

$setenv OMP_DISPLAY_ENV TRUE (csh)

```
OPENMP DISPLAY ENVIRONMENT BEGIN
  _OPENMP = '201307'
  OMP_DYNAMIC = 'FALSE'
  OMP_NESTED = 'FALSE'
  OMP_NUM_THREADS = '32'
  OMP_SCHEDULE = 'DYNAMIC'
  OMP_PROC_BIND = 'FALSE'
  OMP_PLACES = ''
  OMP_STACKSIZE = '140729178218216'
  OMP_WAIT_POLICY = 'PASSIVE'
  OMP_THREAD_LIMIT = '4294967295'
  OMP_MAX_ACTIVE_LEVELS = '2147483647'
  OMP_CANCELLATION = 'FALSE'
  OMP_DEFAULT_DEVICE = '0'
OPENMP DISPLAY ENVIRONMENT END
```

Running OpenMP code

Controlling the number of threads at runtime

- The default number of threads = number of **online** processors on the machine.
- C shell : **setenv OMP_NUM_THREADS number**
- Bash shell: **export OMP_NUM_THREADS = number**
- Runtime OpenMP function omp_set_num_threads(4)
- Clause in #pragma for parallel region

Execution Timing

```
#include omp.h
stime = omp_get_wtime();
longfunction();
etime = omp_get_wtime();
total = etime-stime;
```

Compiling and running OPENMP Code

On SahasraT

$ssh <username>@sahasrat.serc.iisc.ernet.in
$password:

Copy the code onto the home area
Create make files for compiling code (clearing binaries, compiling, linking and creating executable)

Run **make**
Create a job script to submit the job in batch mode
qsub jobscript
qstat
Check output

2020JanOPENMP

Exercise 1 - HelloWorld

Exercise 2 – Data Scoping

Exercise 3 – Compute PI

Exercise 4- Schedule Clause

Exercise 5- Schedule Clause

Exercise 6 – Matrix Multiplication

Exercise 7 – Jacobi Iteration

Exercise 8 – Mandelbrot Set

```
$vi <code>.cpp
$make
$module load pbs
$module load craype-haswell
$qsub jobscript
$qstat
$vi jobscript
```

Recap of Part 1

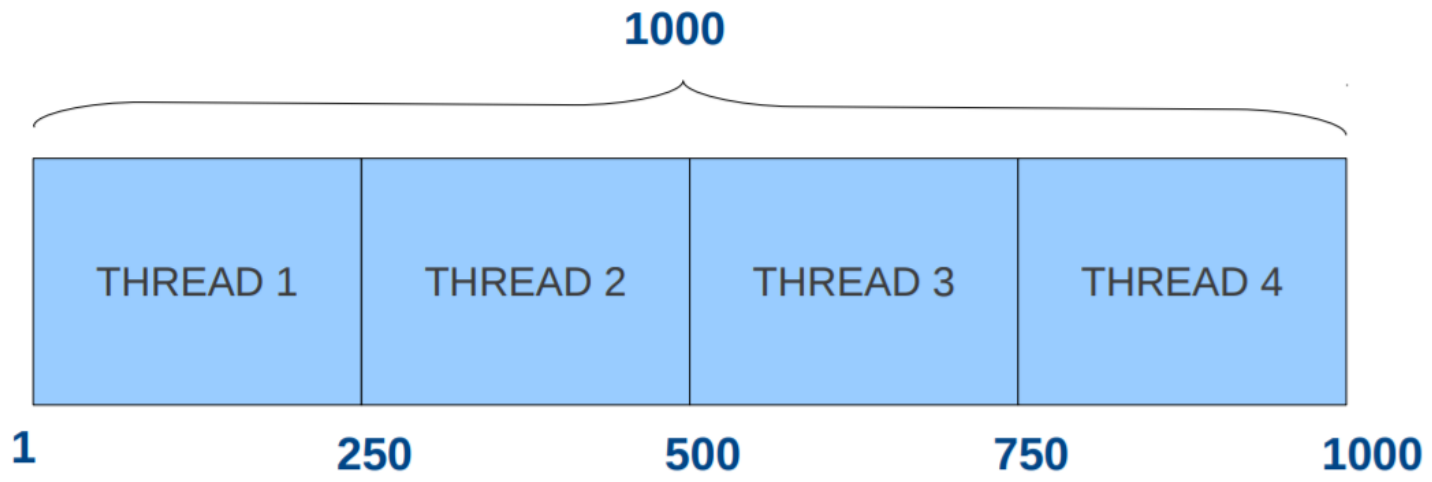
- ✓ What is OPENMP?
- ✓ Fork/Join Programming model
- ✓ OPENMP Core Elements
- ✓ #pragma omp parallel OR Parallel construct
- ✓ run time variables
- ✓ environment variables
- ✓ data scoping (private, shared…)
- ✓ work sharing constructs
  - #pragma omp for
- ✓ compile and run openmp program in c++ and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

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)

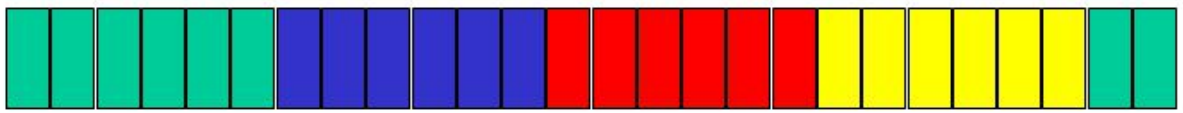
DYNAMIC: Iterations of a loop are divided into chunks of size 1.
Chunks are assigned to threads on a first-come, first-serve basis as threads become available. This continues until all work is completed.
DYNAMIC, N: **Same as above,** all chunks are set to size **N**

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

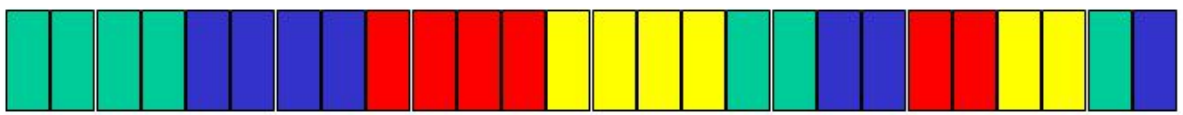
Schedule Clause

- **STATIC** iterations are divided into pieces at compile time (default)



`SCHEDULE (STATIC, 6)`
26 iter on 4 processors

- **DYNAMIC** iterations assigned to processors as they finish, dynamically.
  This requires synchronization after each chunk iterations.
- **GUIDED** pieces reduce exponentially in size with each dispatched piece



`SCHEDULE (GUIDED, 4)`
26 iter on 4 processors

- **RUNTIME** schedule determined by an environment variable `OMP_SCHEDULE`
  With `RUNTIME` it is illegal to specify `chunk`. Example:

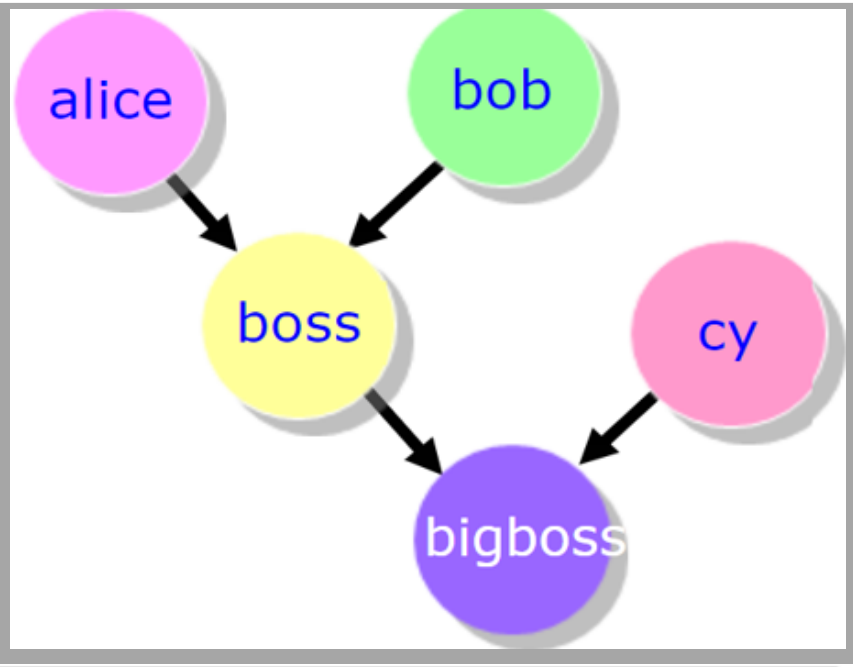
```
setenv OMP_SCHEDULE "dynamic, 4"
```

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.

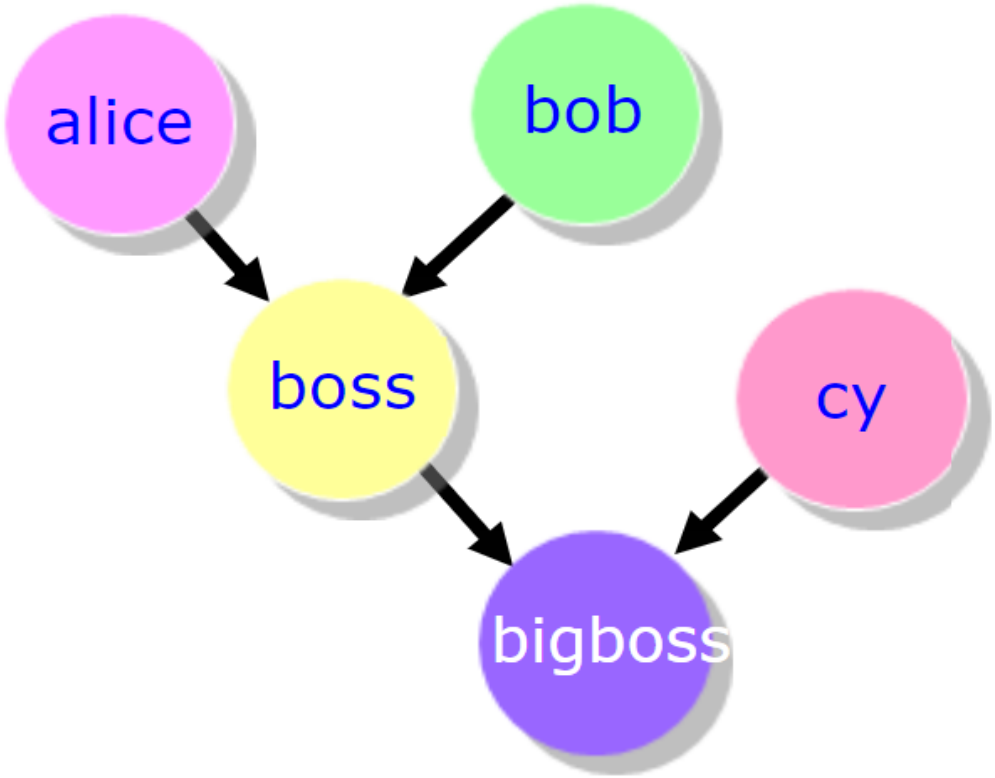
Each SECTION is executed ONCE by a thread in the team.




```
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));
```

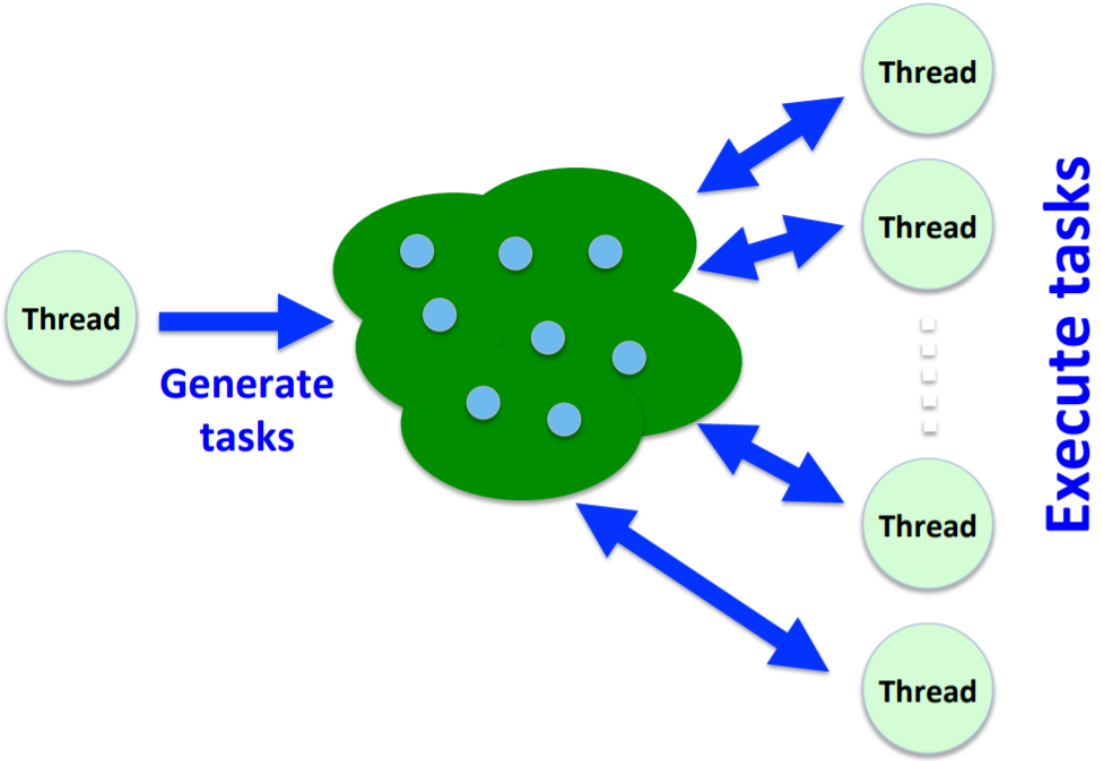


OpenMP: lastprivate Clause - review

```
!$OMP DO PRIVATE(I)
LASTPRIVATE(B)
DO i = 1, 1000
B = i
ENDDO
!$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.




```
#pragma omp task
```

```
!$omp task
```

On encountering a **task** construct, a new task is generated.
Unlike sections, the moment of execution of any task is upto the runtime system and is non-deterministic.

Work Sharing: tasks

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
- All tasks within a team have to be independent and there is no implicit barrier.

Work Sharing: tasks

Fibonacci series:

f(1) = 1

f(2) = 1

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

Task1

Task2

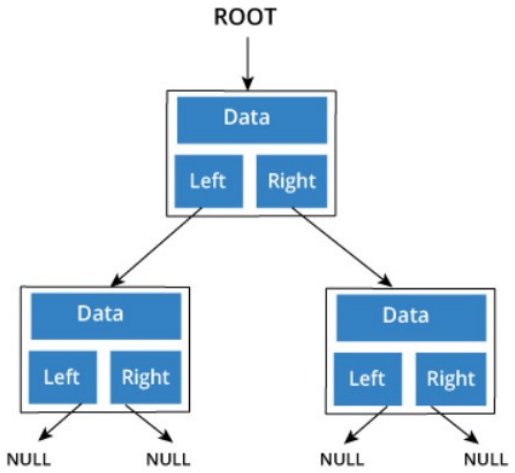
```
/* serial code to compute Fibonacci */
int fib(int n)
{
   int i, j;
   if(n < 2) return n;
   i = fib(n-1);
   j = fib(n-2);
   return (i+j);
}
int main(){
   int n = 8;
   printf("fib(%d) = %d\n", n, fib(n);
}
```

```
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);
}
```

No implicit barrier
USE #pragma omp taskwait

Binary Tree Representation

A node of a binary tree is represented by a structure containing a data part and two pointers to other structures of the same type.



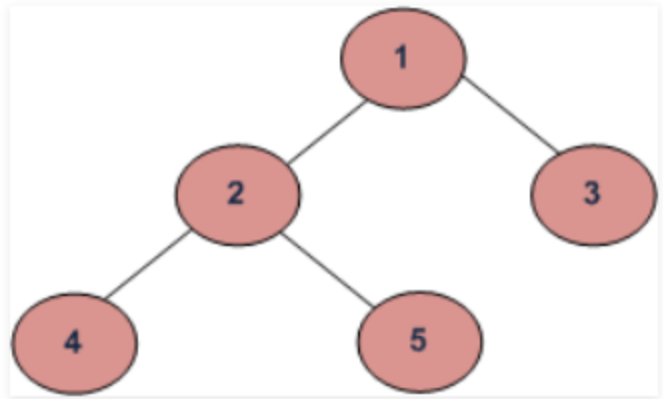

```c
struct node
{
  int data;
  struct node *left;
  struct node *right;
};
```

Applications of Trees

•Binary Search Trees(BSTs) are used to quickly check whether an element is present in a set or not.
•Heap is a kind of tree that is used for heap sort.
•A modified version of tree called Tries is used in modern routers to store routing information.
•Most popular databases use B-Trees and T-Trees, which are variants of the tree structure.
•Compilers use a syntax tree to validate the syntax of every program you write.

Tree Traversals (Inorder, Preorder and Postorder)

Unlike linear data structures (Array, Linked List, Queues, Stacks, etc) which have only one logical way to traverse them, trees can be traversed in different ways. Following are the generally used ways for traversing trees.



Example Tree

Depth First Traversals:

(a) Inorder (Left, Root, Right) : 4 2 5 1 3

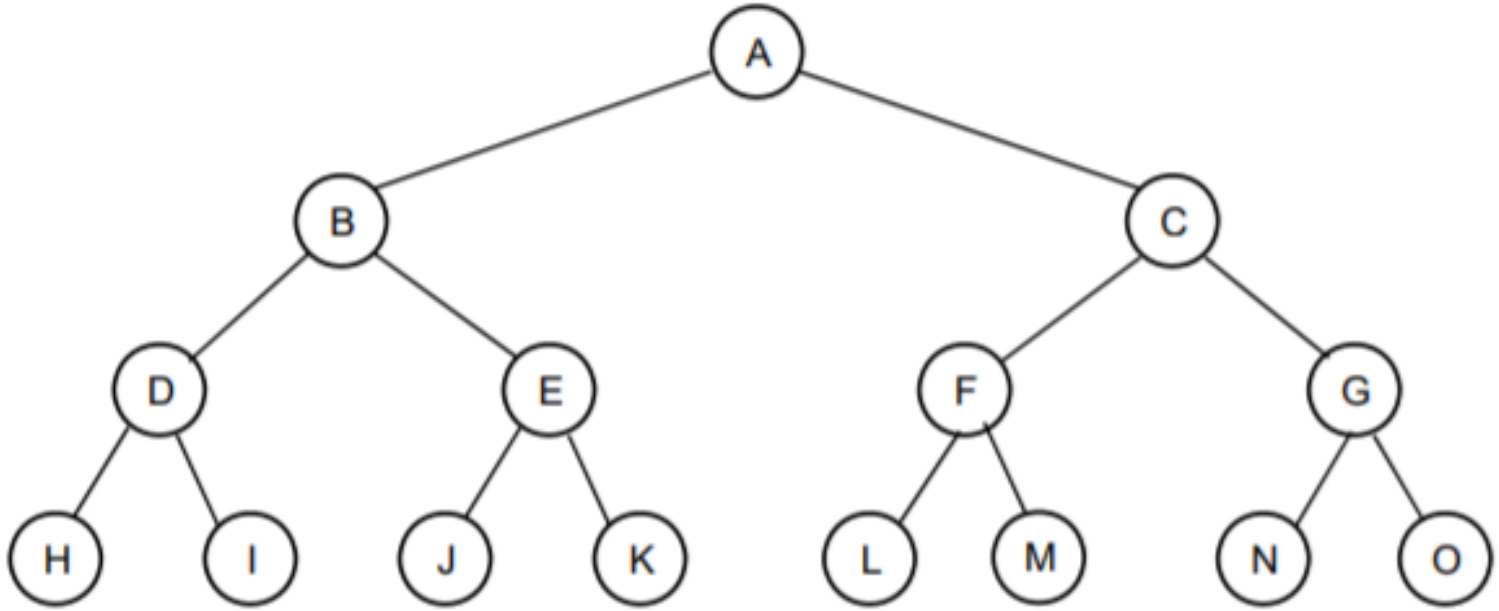
(b) Preorder (Root, Left, Right) : 1 2 4 5 3

(c) Postorder (Left, Right, Root) : 4 5 2 3 1

Breadth First or Level Order Traversal : 1 2 3 4 5

Work Sharing: tasks

Given a tree:



- We would like to traverse it as quickly as possible.
- We are assuming that we do not need to traverse it in order.
- We just need to visit all nodes.

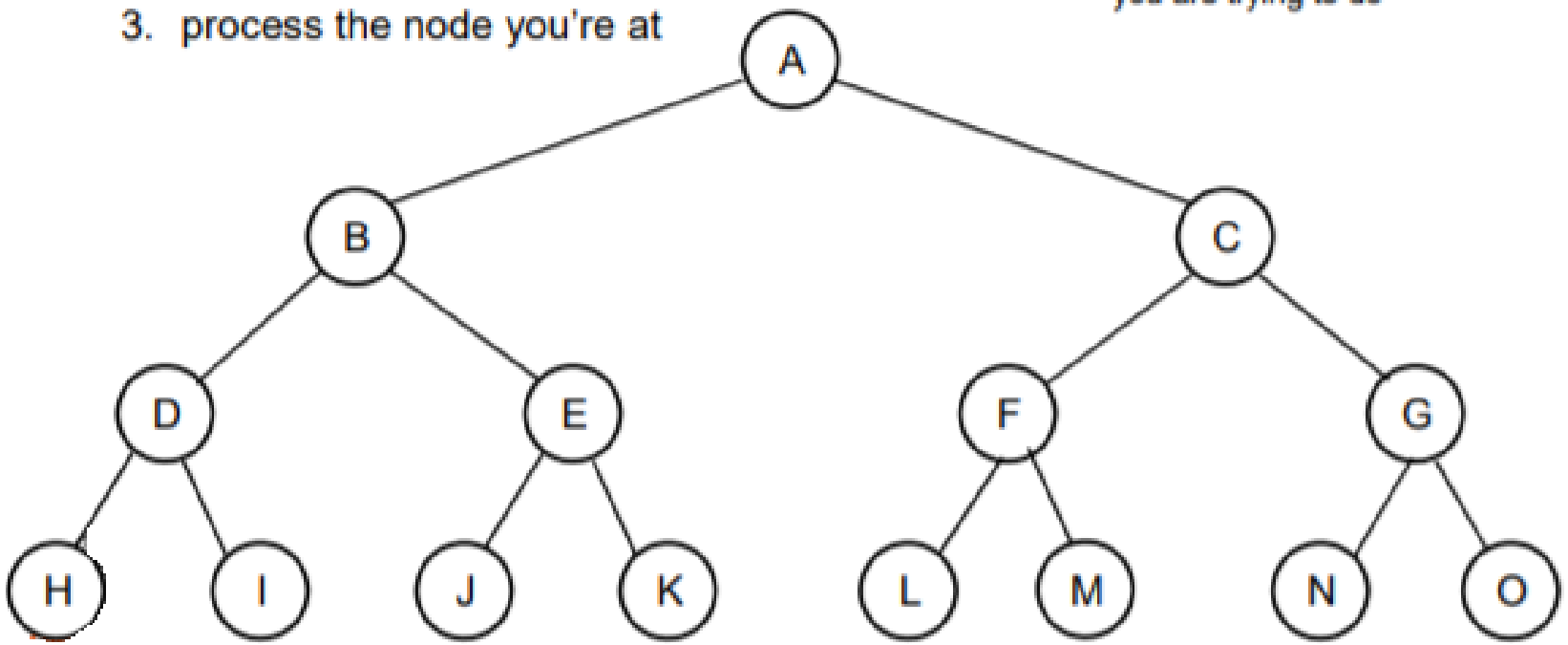
Work Sharing: tasks

- This is common in graph algorithms, such as searching.

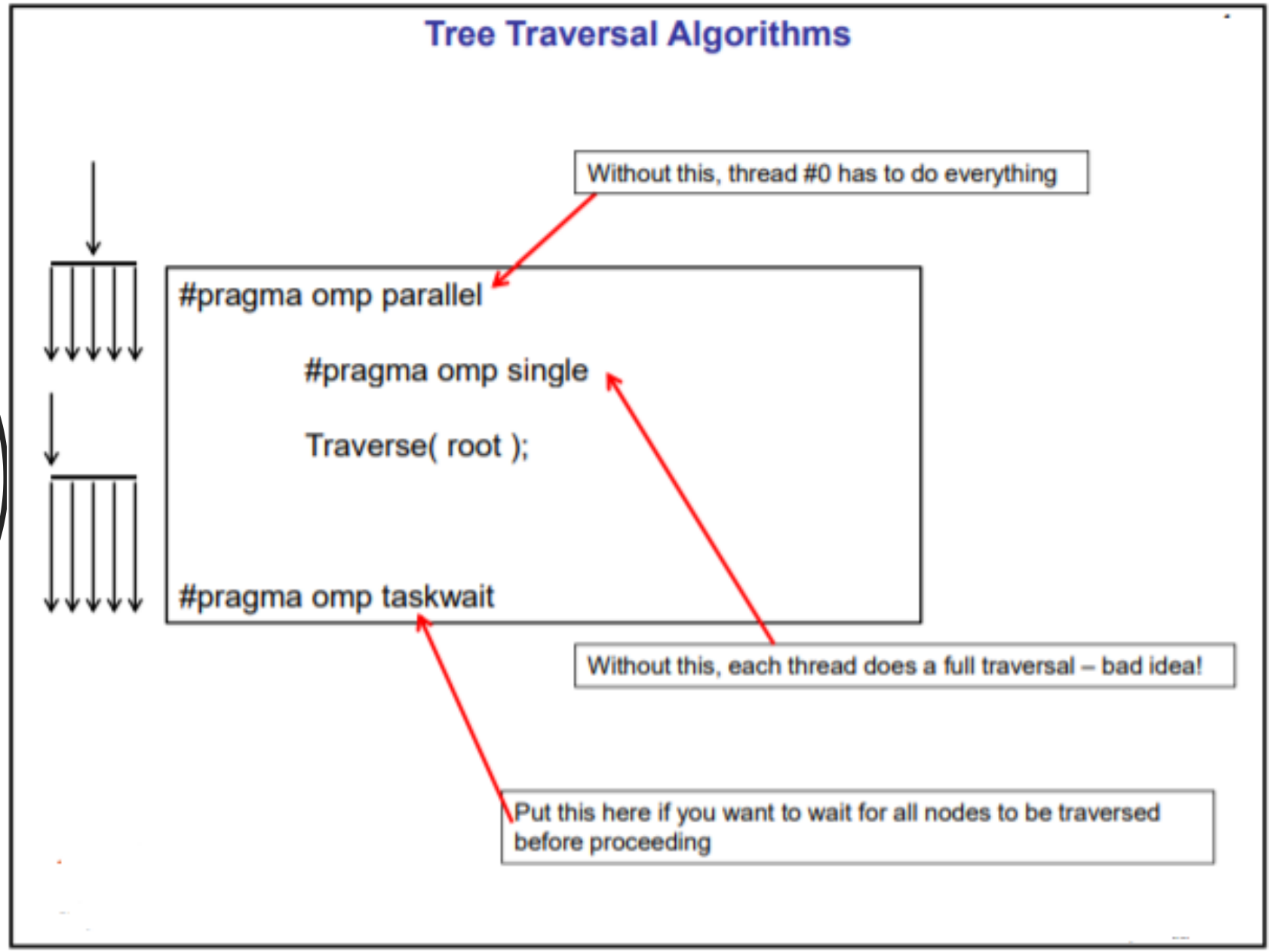
- If the tree is binary and is balanced, then the maximum depth of the tree is $\log_2$(# of Nodes)

- Strategy at a node:
1. follow one descendent node
2. follow the other descendent node
3. process the node you're at

This order could be re-arranged, depending on what you are trying to do



Work Sharing: tasks



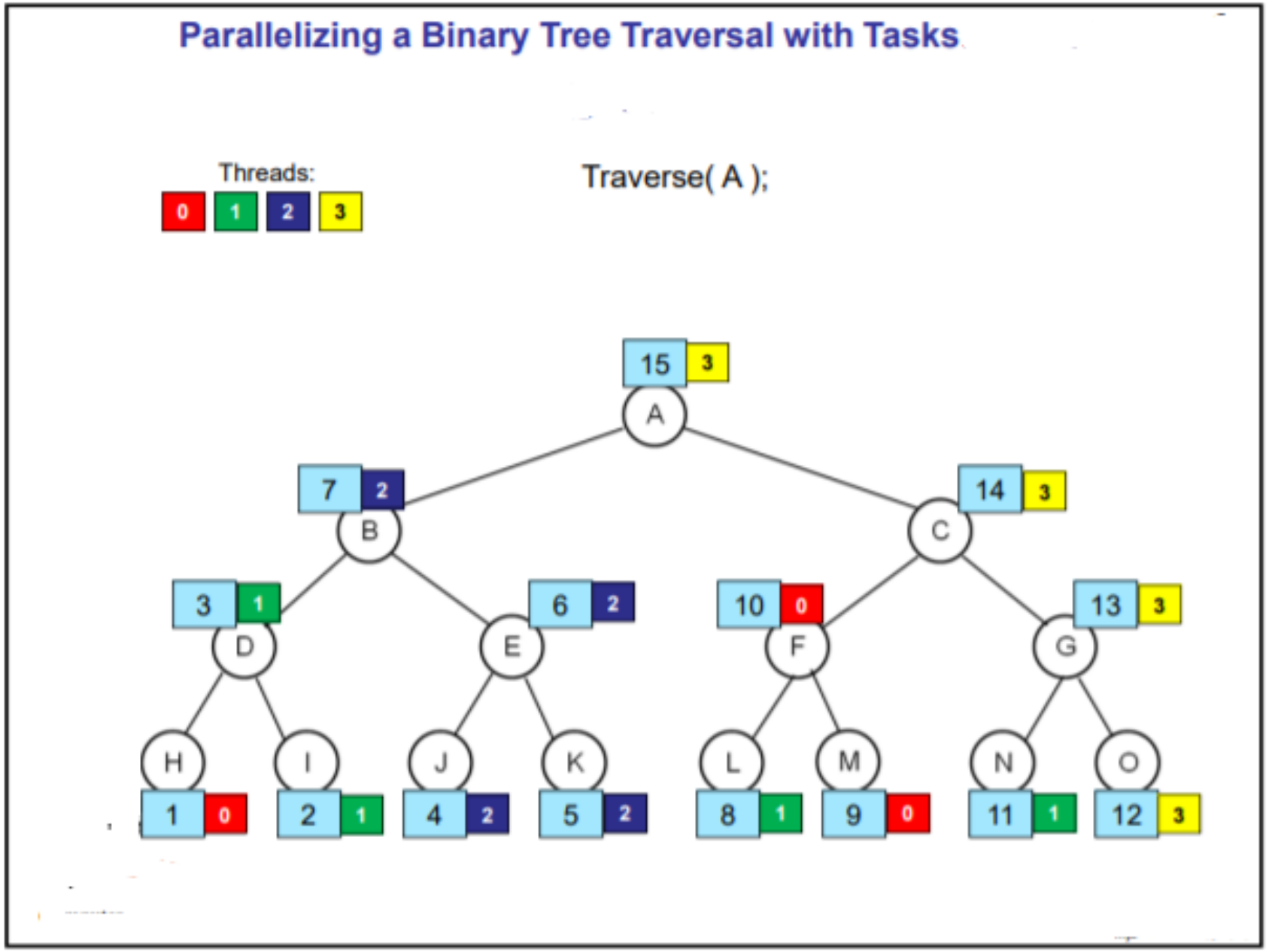
Work Sharing: tasks

Parallelizing a Binary Tree Traversal with Tasks

```
void
Traverse( Node *n )
{
        if( n->left  !=  NULL )
        {
                #pragma omp task private(n) untied
                Traverse( n->left );
        }


        if( n->right  !=  NULL )
        {
                #pragma omp task private(n) untied
                Traverse( n->right );
        }

        #pragma omp taskwait

        Process( n );
}
```

Put this here if you want to wait for both branches to be taken before processing the parent

Work Sharing: tasks



OpenMP: Synchronization

- Data dependencies and Task Dependencies
- Need for finer control over how variables are shared.
- Ensure that threads do not interfere with each other. (the output should 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.

- 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

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
```

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.

Computing Efficiency of Parallel Code

$$Speedup(P) = \frac{T_{Serial}(P)}{T_{Elapsed}(P)}$$

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

Performance in OPENMP programs

Easy to write OpenMP but hard to write an efficient program!

6 main causes of poor performance:

- Sequential code
- Communication
- Load imbalance
- Synchronization
- Parallel Overhead
- Caching issues in multicore performance (False Sharing)

Sequential code

In OpenMP, all code outside of parallel regions and inside MASTER, SINGLE and CRITICAL directives is sequential.

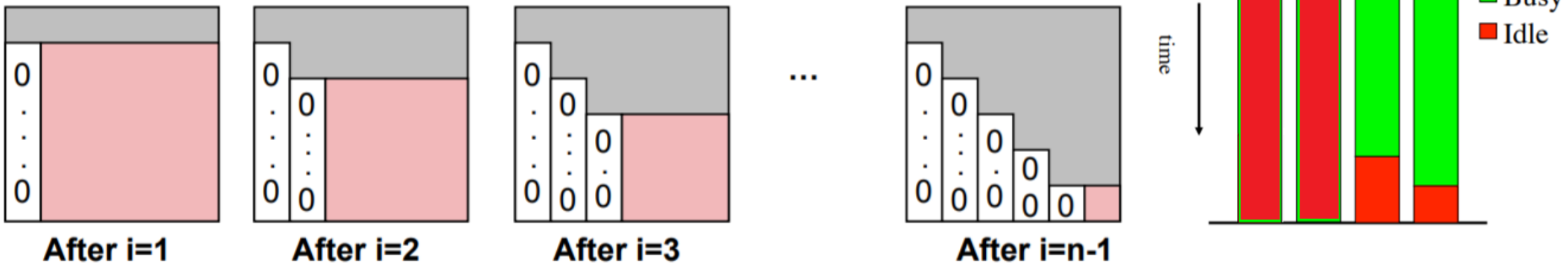
- This code should be as small as possible.

Communication

- On Shared memory machines, communication = increased memory access costs.
- It takes longer to access data in main memory or another processor's cache than it does from local cache.
- Memory accesses are expensive!
- Unlike message passing, communication is spread throughout the program. – Much harder to analyse and monitor.

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

Load Balancing

Important aspect of performance
For regular workloads (e.g. vector addition), load balancing is not an issue
For less regular workloads, care needs to be taken in distributing the work over the threads
Examples of irregular workloads:
- multiplication of triangular matrices
- parallel searches in a linked list or trees
- recursive algorithms that involve the use of tasks

Schedule clause could be used for various iteration scheduling algorithms
Can be addressed by pipelined processing.

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

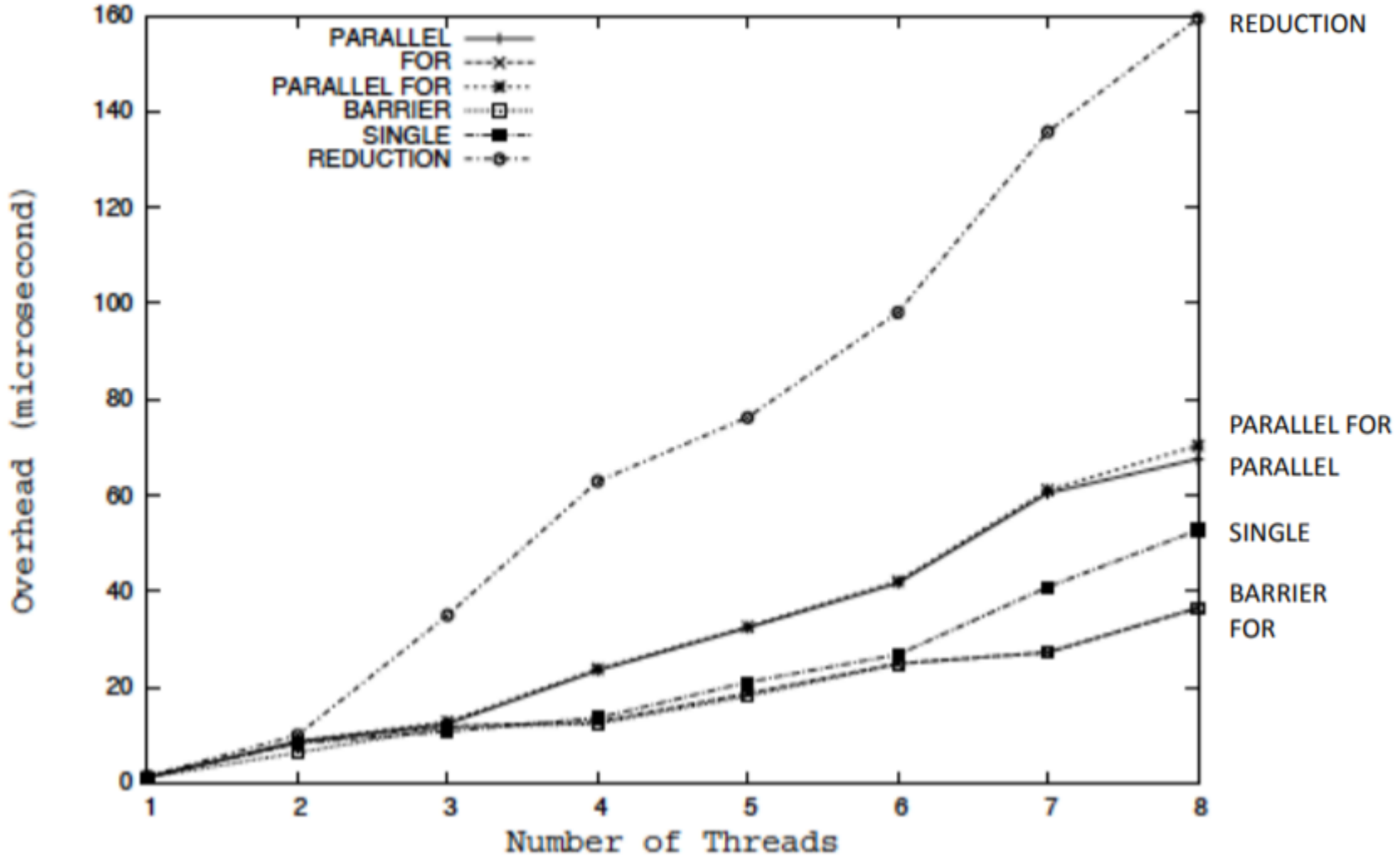
Some simple techniques to avoid overheads.

Parallel Overhead

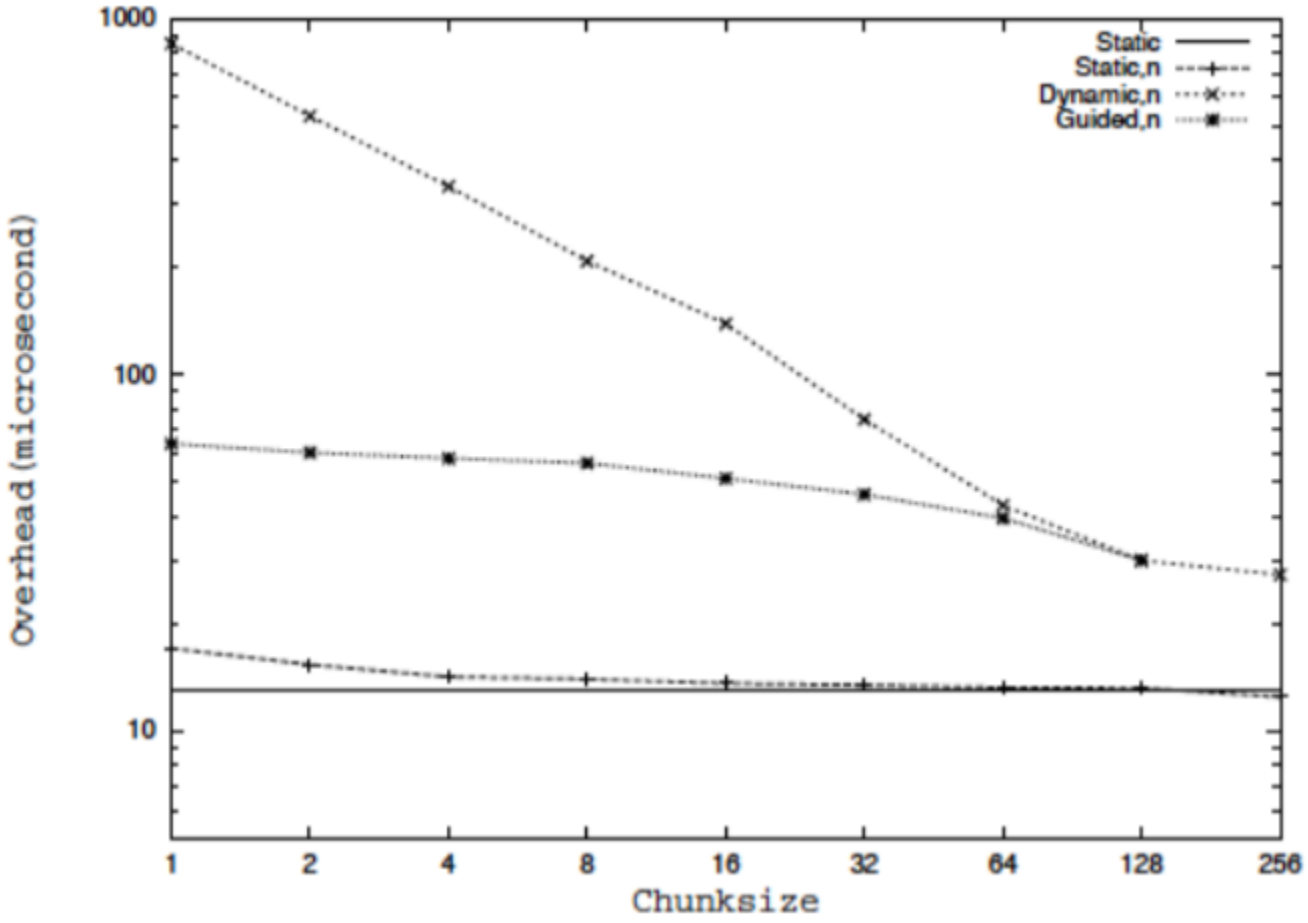
The amount of time required to coordinate parallel threads, as opposed to doing useful work.

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

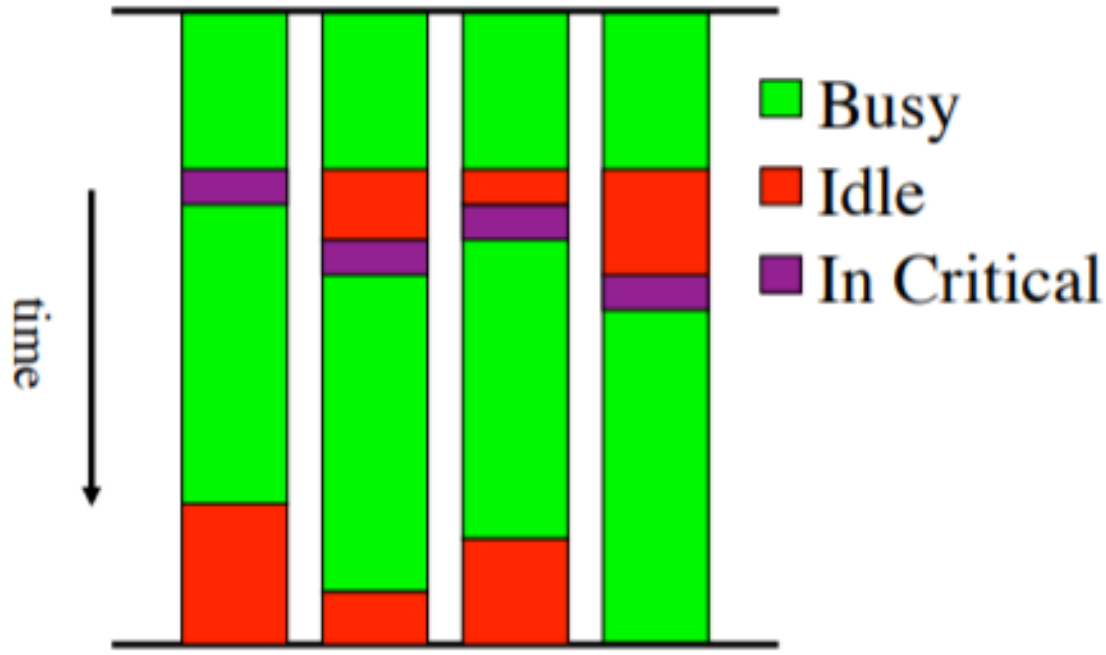


Optimize the use of barriers

```
#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++)
            { .........}
```

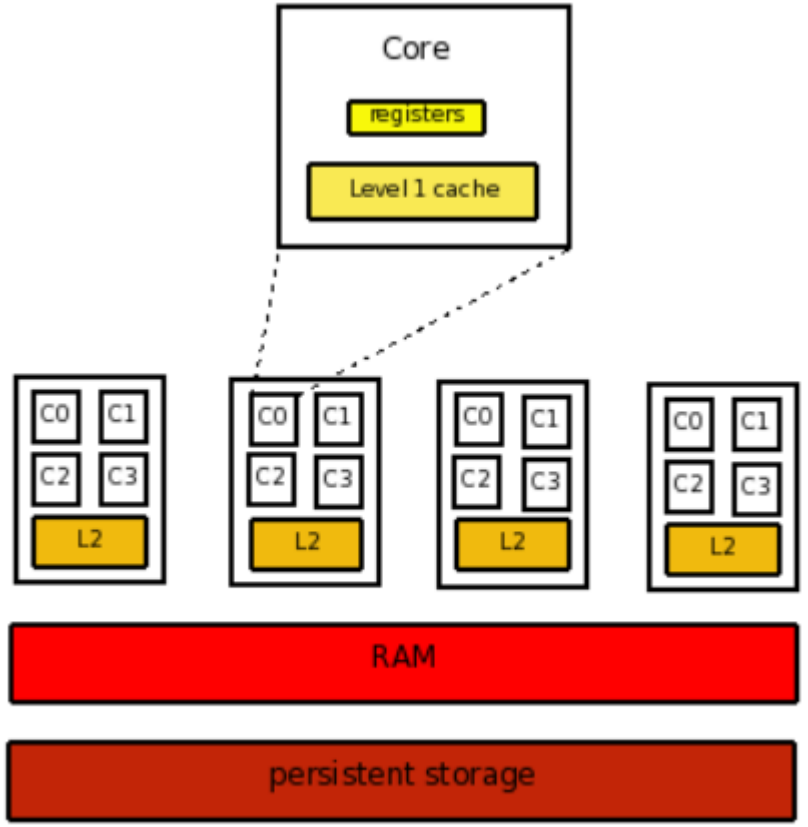
Figure 5.25: **Parallel region embedded in a loop nest** – The overheads of the parallel region are incurred n^2 times.

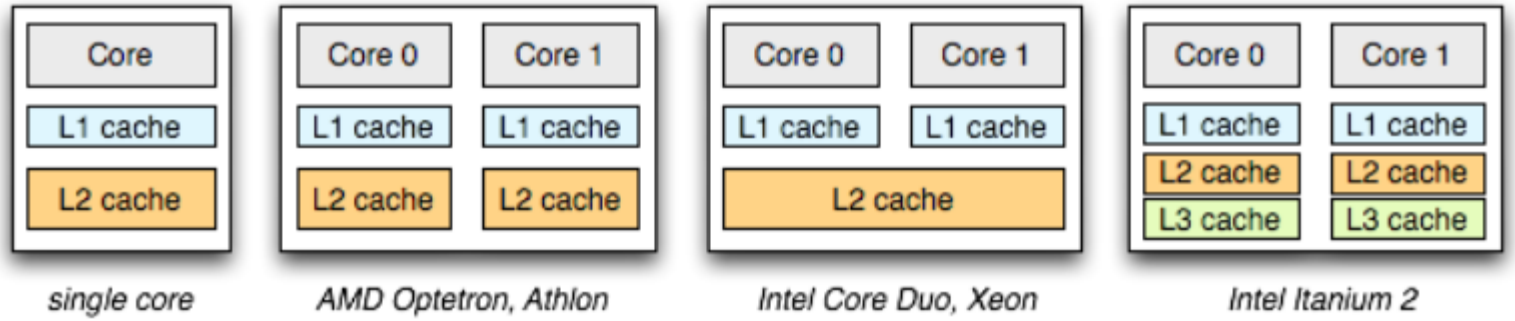
```
#pragma omp parallel
   for (i=0; i<n; i++)
       for (j=0; j<n; j++)
           #pragma omp for
           for (k=0; k<n; k++)
               { .........}
```

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

Caching issues in multicore performance

Memory Architecture





single core *AMD Optetron, Athlon* *Intel Core Duo, Xeon* *Intel Itanium 2*

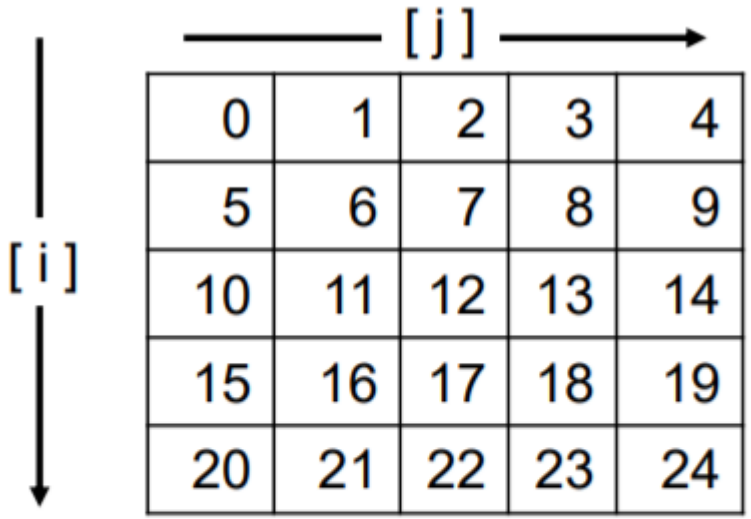
Cache hits and misses

Cache could be KB or MB, but bytes are transferred in much smaller sizes. Typical size of cache line is 64Bytes
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

Sequential: Cache hits and misses

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



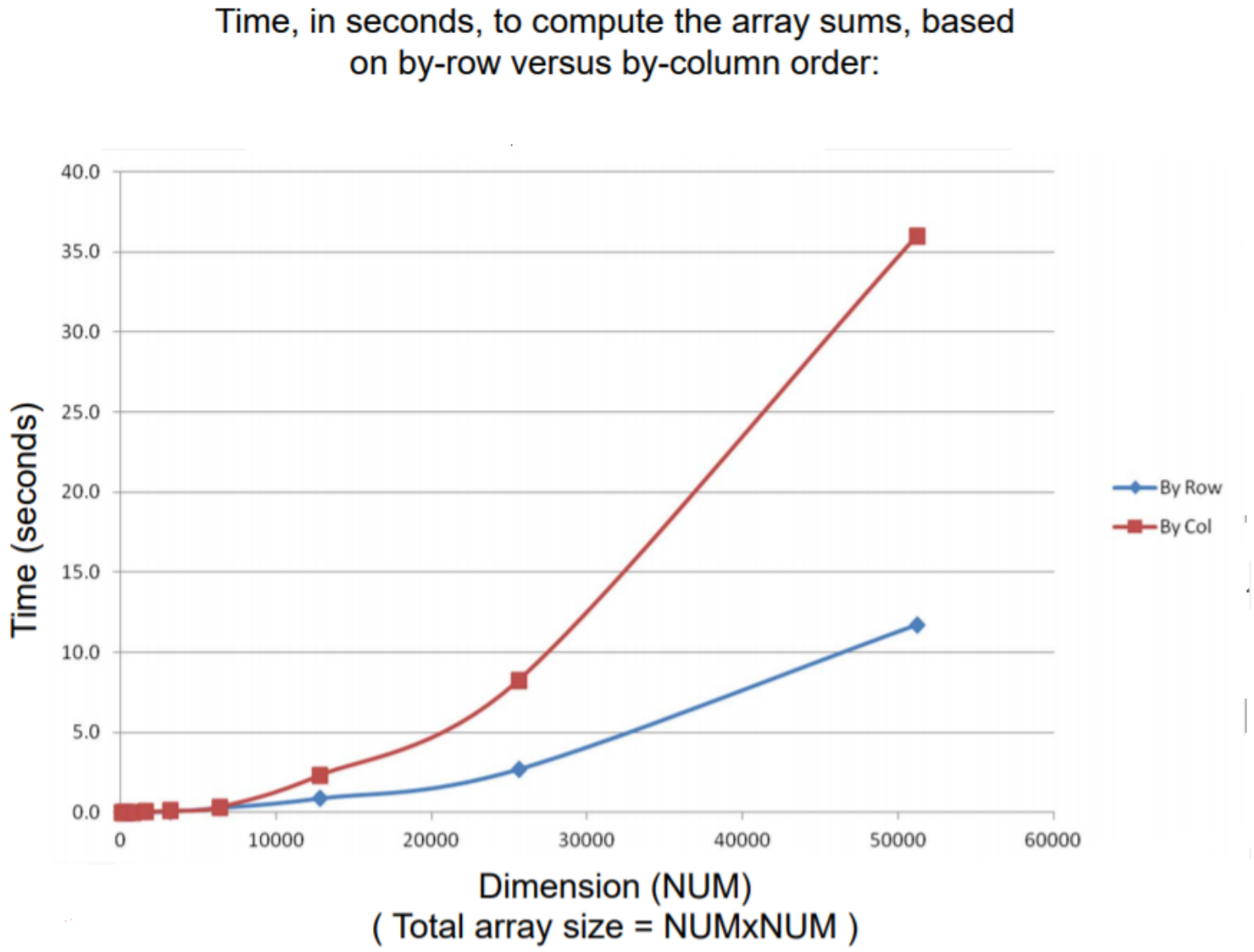

```
sum = 0.;
for( int i = 0; i < NUM; i++ )
{
        for( int j = 0; j < NUM; j++ )
        {
                float  f = ???
                sum += f;
        }
}
```

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

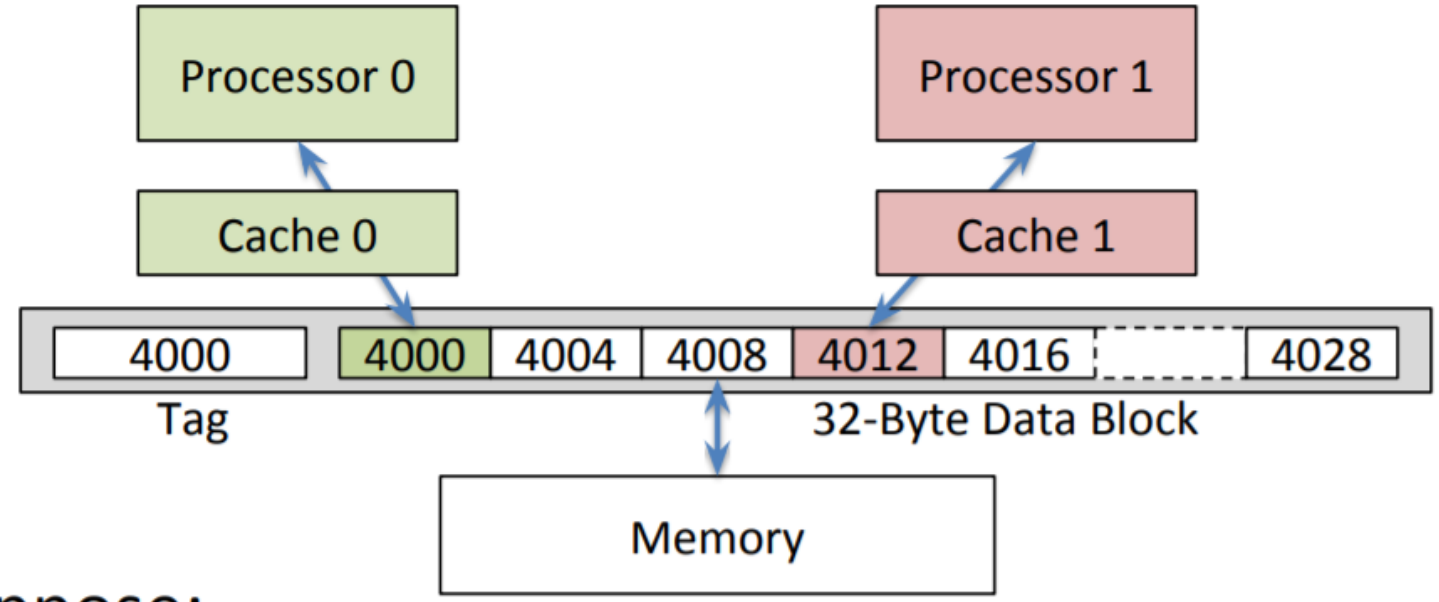
```
float  f  =  Array[ i ][ j ] ;
```
→ Sequential memory order

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

Sequential: Cache hits and misses



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?

Caching issues in multicore performance

Cache lines consist of several words of data.

When two processors are both writing to different words on the same cache line

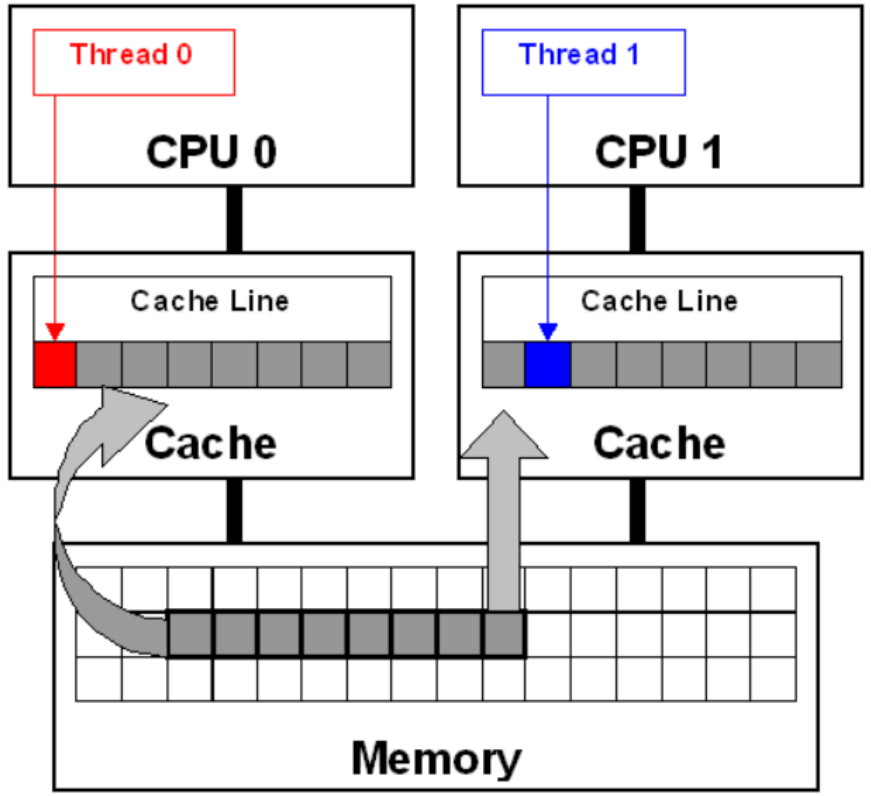
- Each write will invalidate the other processors copy.
- Lots of remote memory accesses.

Symptoms:

- Poor speedup
- High, non-deterministic numbers of cache misses.
- Mild, non-deterministic, unexpected load imbalance.

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];
}
```

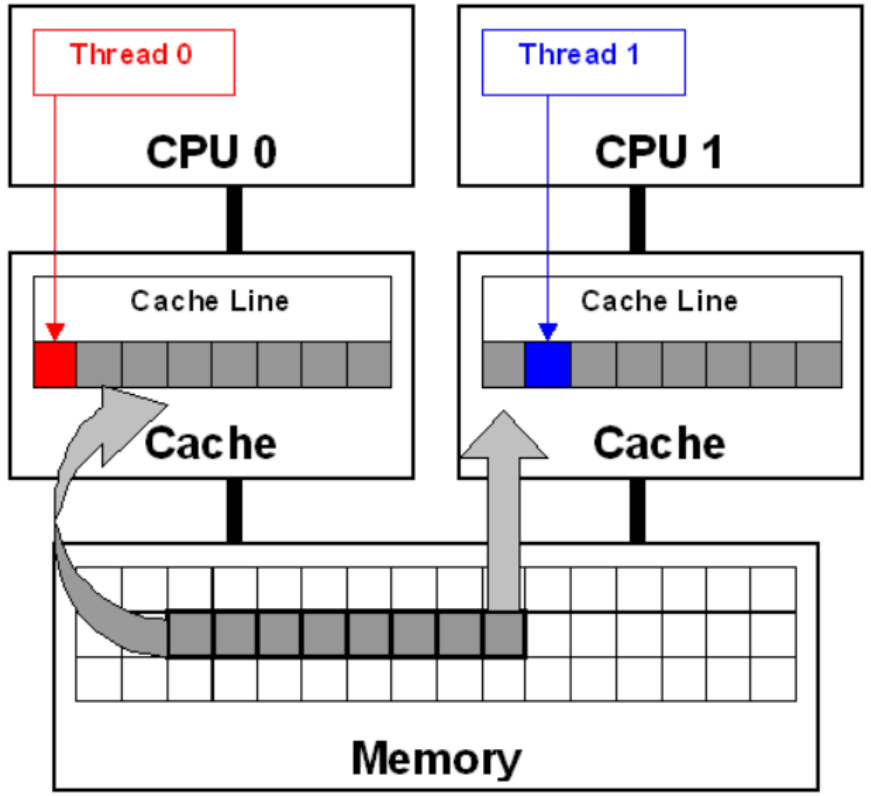


When threads on different processors modify variables that reside on the same cache line.
This invalidates the cache line and forces a memory update to maintain cache coherency.
Potential false sharing on the array **sum_local**.

"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.

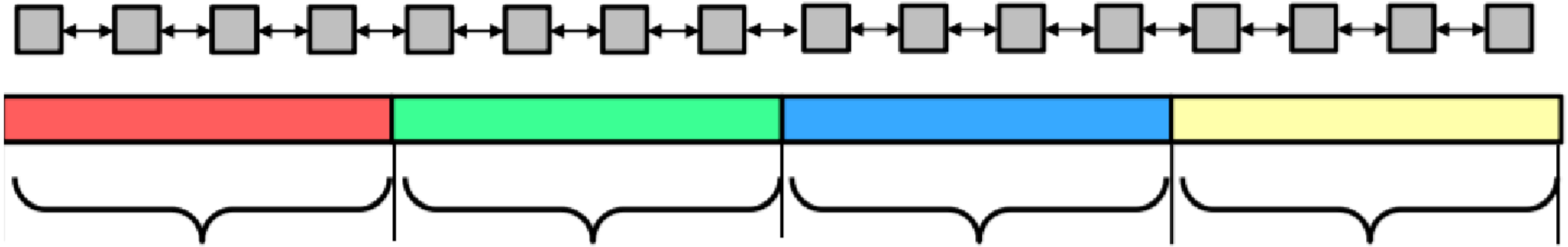
False Sharing: Solution 3

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

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."

Matrix-vector multiplication

$$y_i = a_{i0}x_0 + a_{i1}x_1 + \cdots + a_{i,n-1}x_{n-1}$$

a_{00}	a_{01}	$\cdots$	$a_{0,n-1}$
a_{10}	a_{11}	$\cdots$	$a_{1,n-1}$
$\vdots$	$\vdots$		$\vdots$
a_{i0}	a_{i1}	$\cdots$	$a_{i,n-1}$
$\vdots$	$\vdots$		$\vdots$
$a_{m-1,0}$	$a_{m-1,1}$	$\cdots$	$a_{m-1,n-1}$

x_0
x_1
$\vdots$
x_{n-1}

$=$

y_0
y_1
$\vdots$
$y_i = a_{i0}x_0 + a_{i1}x_1 + \cdots a_{i,n-1}x_{n-1}$
$\vdots$
y_{m-1}

```
for (i = 0; i < m; i++) {
    y[i] = 0.0;
    for (j = 0; j < n; j++)
        y[i] += A[i][j]*x[j];
}
```

Matrix-vector multiplication

```
#  pragma omp parallel for num_threads(thread_count)  \
      default(none) private(i, j) shared(A, x, y, m, n)
   for (i = 0; i < m; i++) {
      y[i] = 0.0;
      for (j = 0; j < n; j++)
         y[i] += A[i][j]*x[j];
   }
```

Far more write-misses than the other two.

Threads	Matrix Dimension					
	$8,000,000 \times 8$		8000×8000		$8 \times 8,000,000$	
	Time	Eff.	Time	Eff.	Time	Eff.
1	0.322	1.000	0.264	1.000	0.333	1.000
2	0.219	0.735	0.189	0.698	0.300	0.555
4	0.141	0.571	0.119	0.555	0.303	0.275

Matrix-vector multiplication

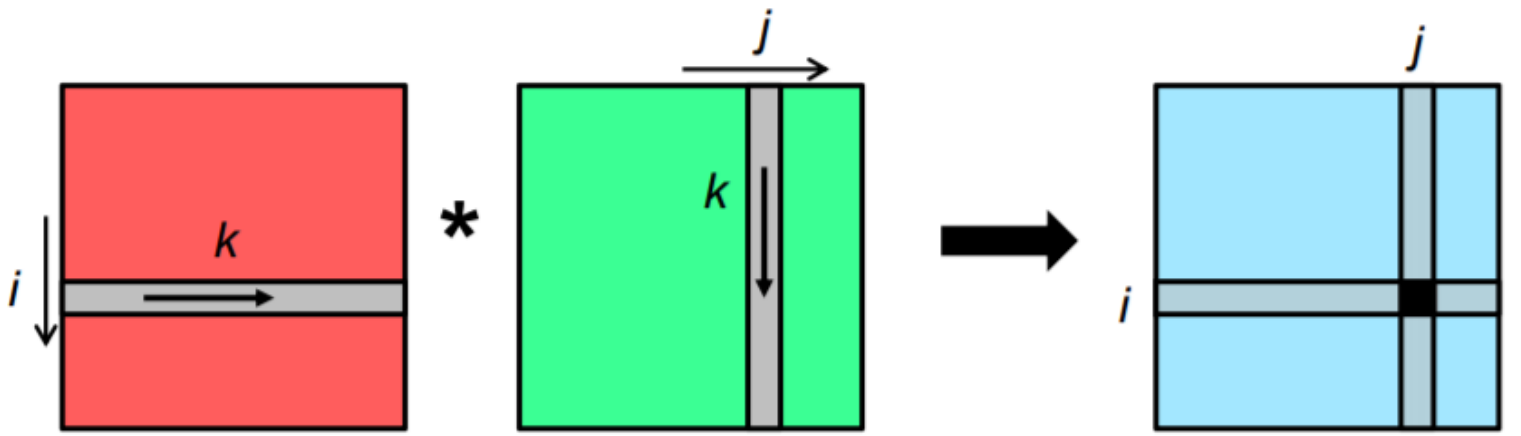
```
#  pragma omp parallel for num_threads(thread_count)  \
      default(none) private(i, j) shared(A, x, y, m, n)
   for (i = 0; i < m; i++) {
      y[i] = 0.0;
      for (j = 0; j < n; j++)
         y[i] += A[i][j]*x[j];
   }
```

Far more read-misses than the other two.

Threads	Matrix Dimension					
	$8,000,000 \times 8$		8000×8000		$8 \times 8,000,000$	
	Time	Eff.	Time	Eff.	Time	Eff.
1	0.322	1.000	0.264	1.000	0.333	1.000
2	0.219	0.735	0.189	0.698	0.300	0.555
4	0.141	0.571	0.119	0.555	0.303	0.275

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

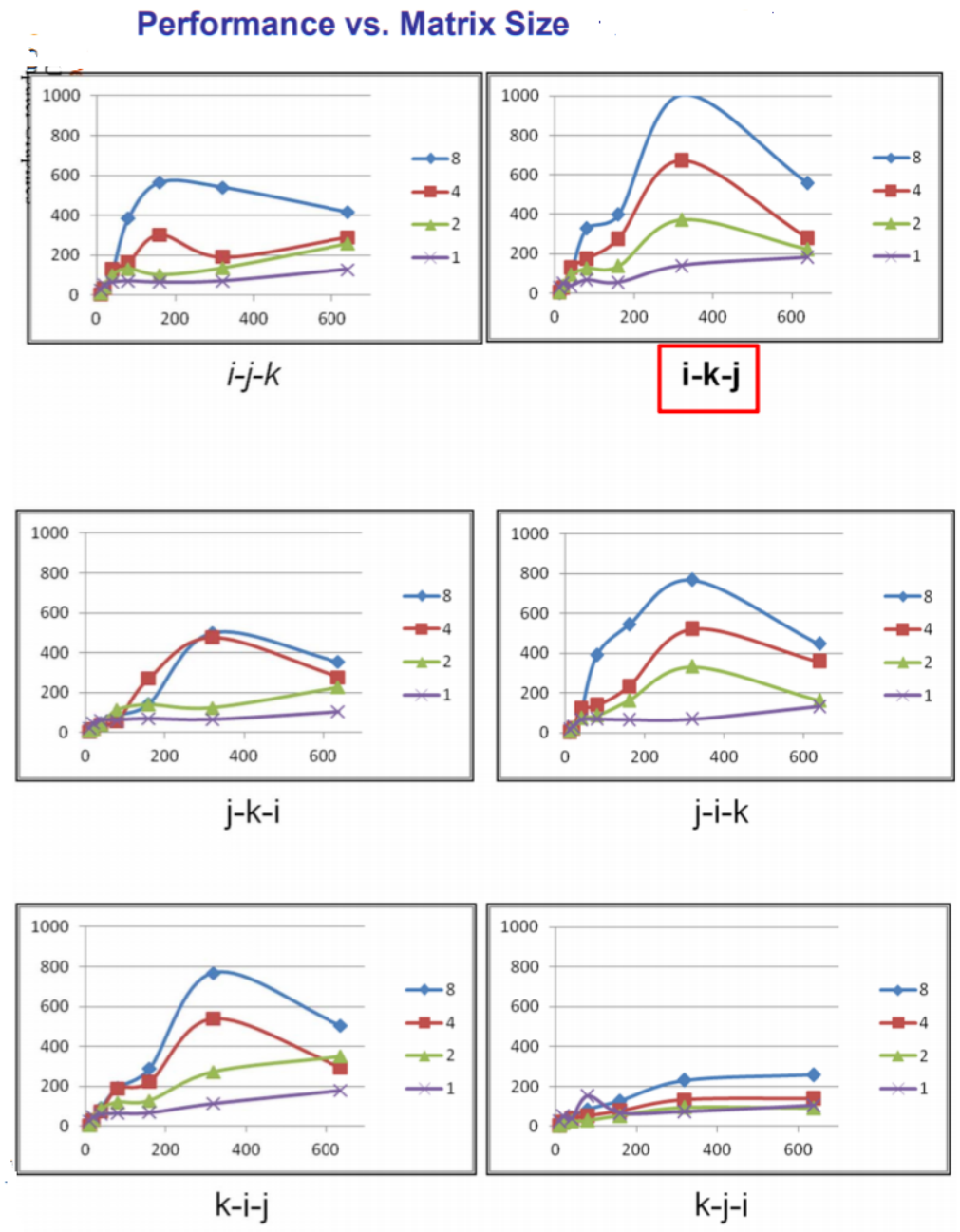



```
for( i = 0; i < SIZE; i++ )
    for( j = 0; j < SIZE; j++ )
        for( k = 0; k < SIZE; k++ )
```

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

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

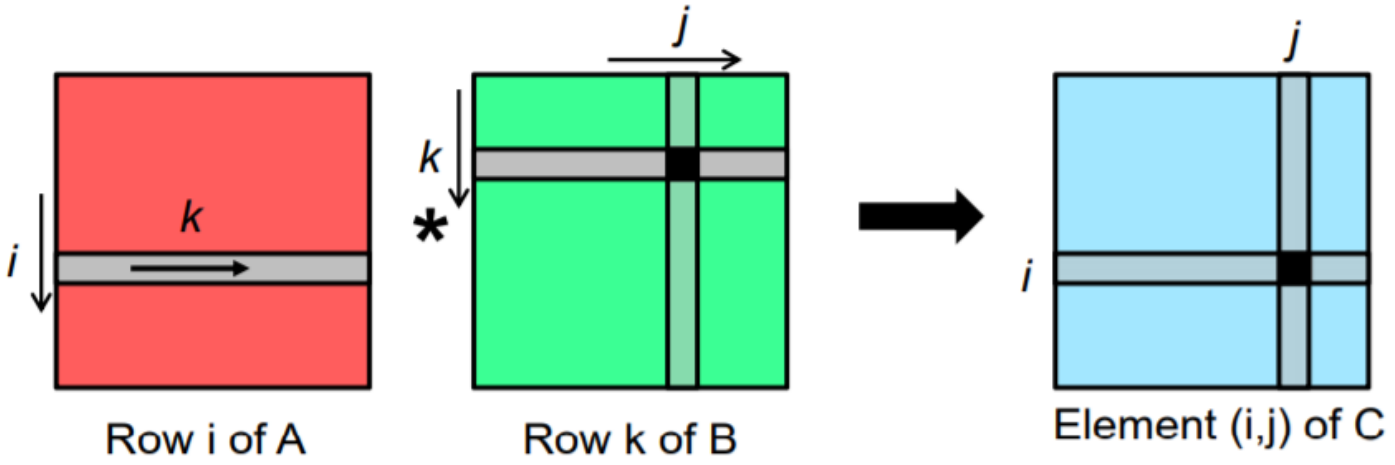


Where Cache Coherence Really Matters: Matrix Multiply

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}

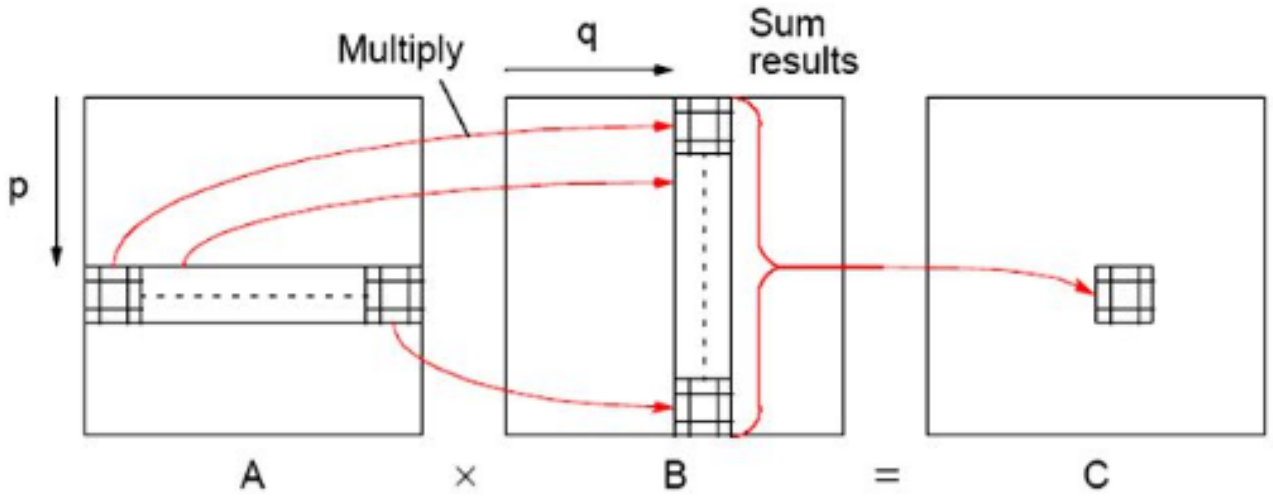
B_{11}	B_{12}	B_{13}	B_{14}
B_{21}	B_{22}	B_{23}	B_{24}
B_{32}	B_{32}	B_{33}	B_{34}
B_{41}	B_{42}	B_{43}	B_{44}

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

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.

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)