



Parallel Programming with OpenMP

CsCADs Summer Workshop, 2011

Yonghong Yan

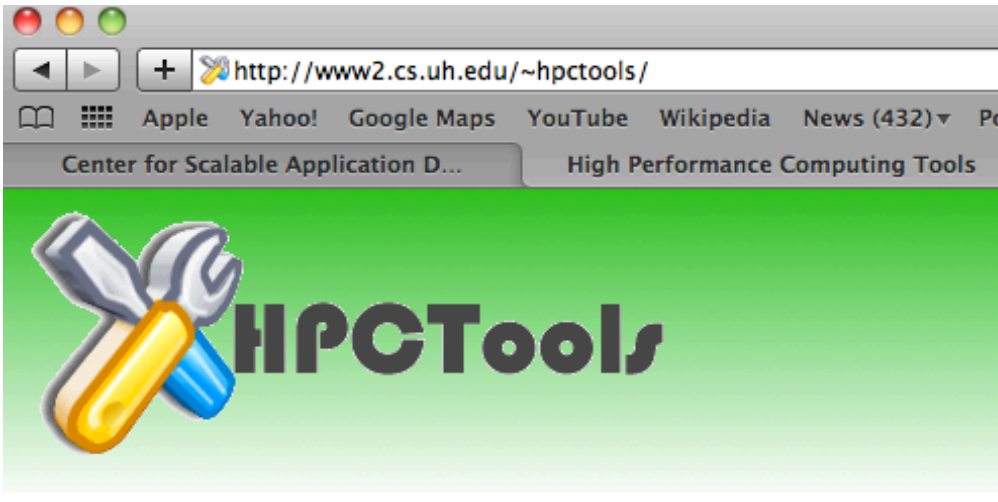
<http://www.cs.uh.edu/~hpctools>

University of Houston

Acknowledgements: [Tim Mattson](#) (Intel), [Ruud van der Pas](#) (Oracle)

UH HPCTools Research Group

<http://www.cs.uh.edu/~hpctools>

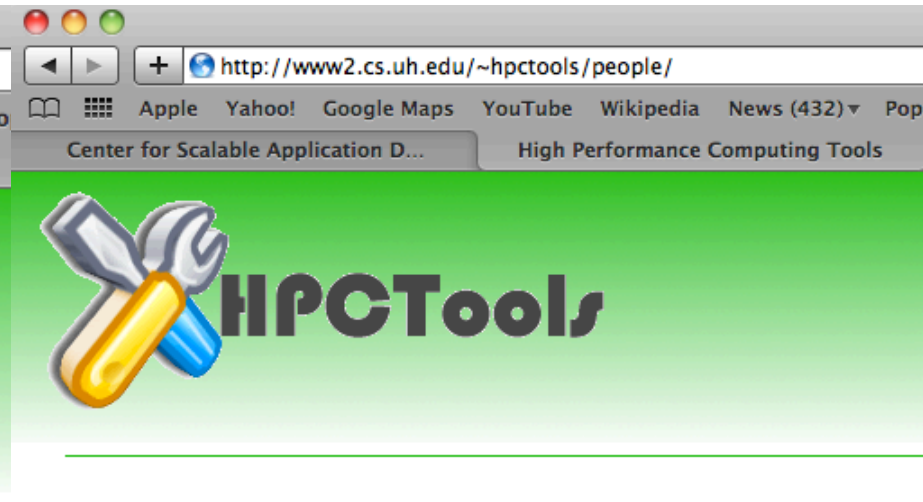


Welcome

The High Performance Computing Tools group (HPCTools) is a research group at the University of Houston. The group was created by Dr. Barbara Chapman who joined the group in 2004. The group is currently engaged in edge research in compiler and tools for High performance computing. Current research areas include:

- Compilers and Tools: [OpenUH](#) compiler, [Dragon](#) Analysis Tool
- Parallel Programming Models: Hybrid OpenMP/MPI, OpenMP, etc.
- [COPPER](#) Project
- Performance Tuning Environments: Selective Instrumentation and Optimization
- Compiler Optimization: Wide Area Privatization, Parallel Data Optimization
- Grid Computing, Web Services, Scheduling Policy

Current Projects



Name : Barbara Chapman
Position : Principle Investigator
Phone : 713-743-3354
Email : chapman AT cs.uh.edu
Office : PGH 215
Research Areas : Visit Website
Projects : Visit Website



Name : Yonghong Yan
Position : Research Assistant Professor
Phone : 713-743-3384 Office : PGH 204
Email : yanyh AT cs.uh.edu
Research Areas : Programming Languages and Compilers, Parallel and High Performance Computing, and Computer Architectures
Projects : Extreme OpenMP, CAF, OpenUH and others

Outline

- OpenMP Introduction
- Parallel Programming with OpenMP
 - Worksharing, tasks, data environment, synchronization
- OpenMP Performance and Best Practices
- Hybrid MPI/OpenMP
- Case Studies and Examples
- Reference Materials

What is OpenMP

- De-facto standard **API** to write shared memory parallel applications in C, C++, and Fortran
- Consists of:
 - Compiler directives
 - Runtime routines
 - Environment variables
- OpenMP Architecture Review Board (ARB)
 - Maintains OpenMP specification
 - Permanent members
 - AMD, Cray, Fujitsu, HP, IBM, Intel, NEC, PGI, Oracle, Microsoft, Texas Instruments, CAPS-Enterprise, NVIDIA
 - Auxiliary members
 - ANL, ASC/LLNL, cOMPunity, EPCC, LANL, NASA, TACC, etc
 - <http://www.openmp.org>
- **Version 3.1** released July 2011

OpenMP Components

Directives

- Parallel region
- Worksharing constructs
- Tasking
- Synchronization
- Data-sharing attributes

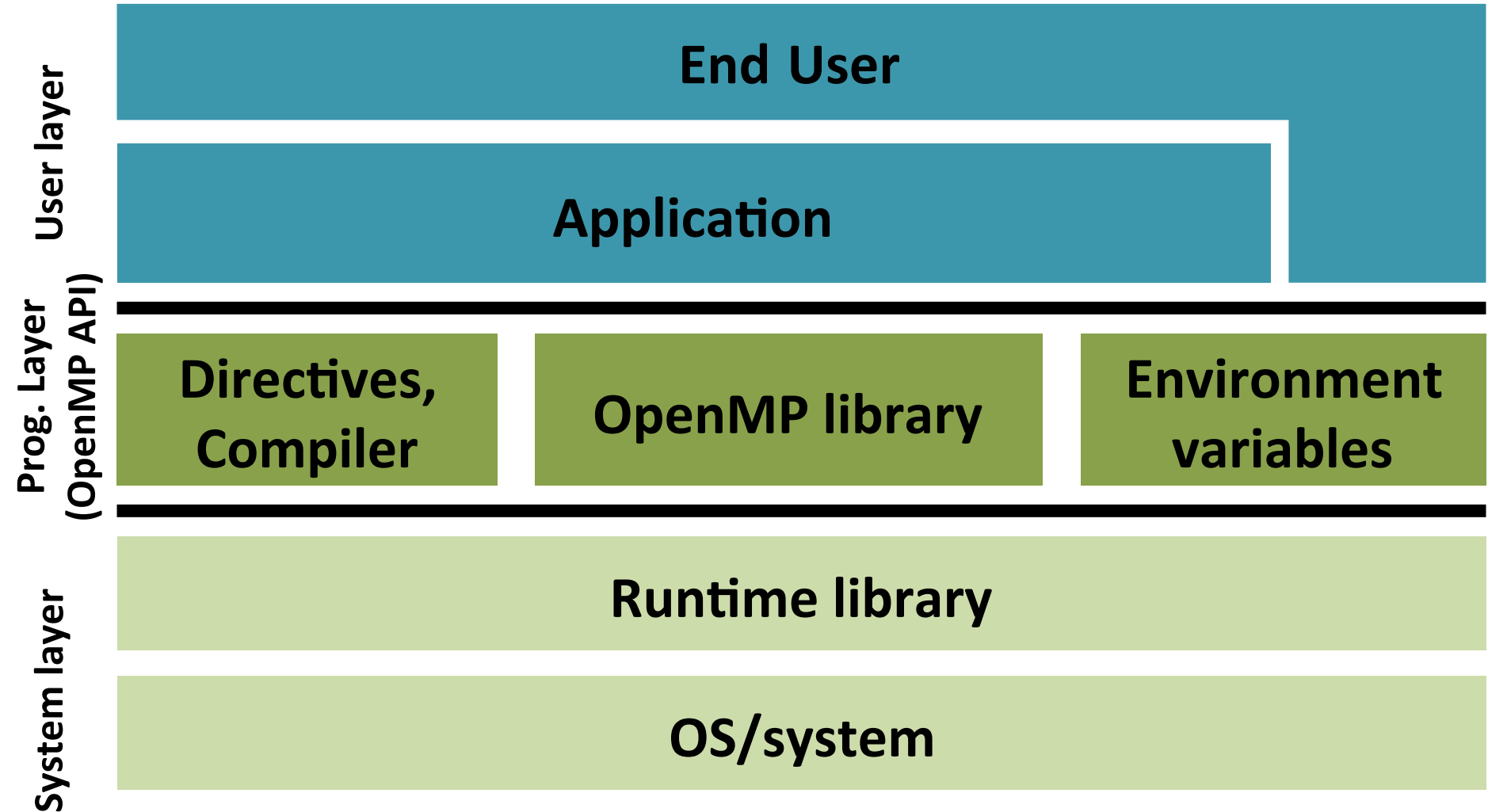
Runtime environment

- Number of threads
- Thread ID
- Dynamic thread adjustment
- Nested parallelism
- Schedule
- Active levels
- Thread limit
- Nesting level
- Ancestor thread
- Team size
- Locking
- Wallclock timer

Environment variables

- Number of threads
- Scheduling type
- Dynamic thread adjustment
- Nested parallelism
- Stacksize
- Idle threads
- Active levels
- Thread limit

OpenMP Parallel Computing Solution Stack



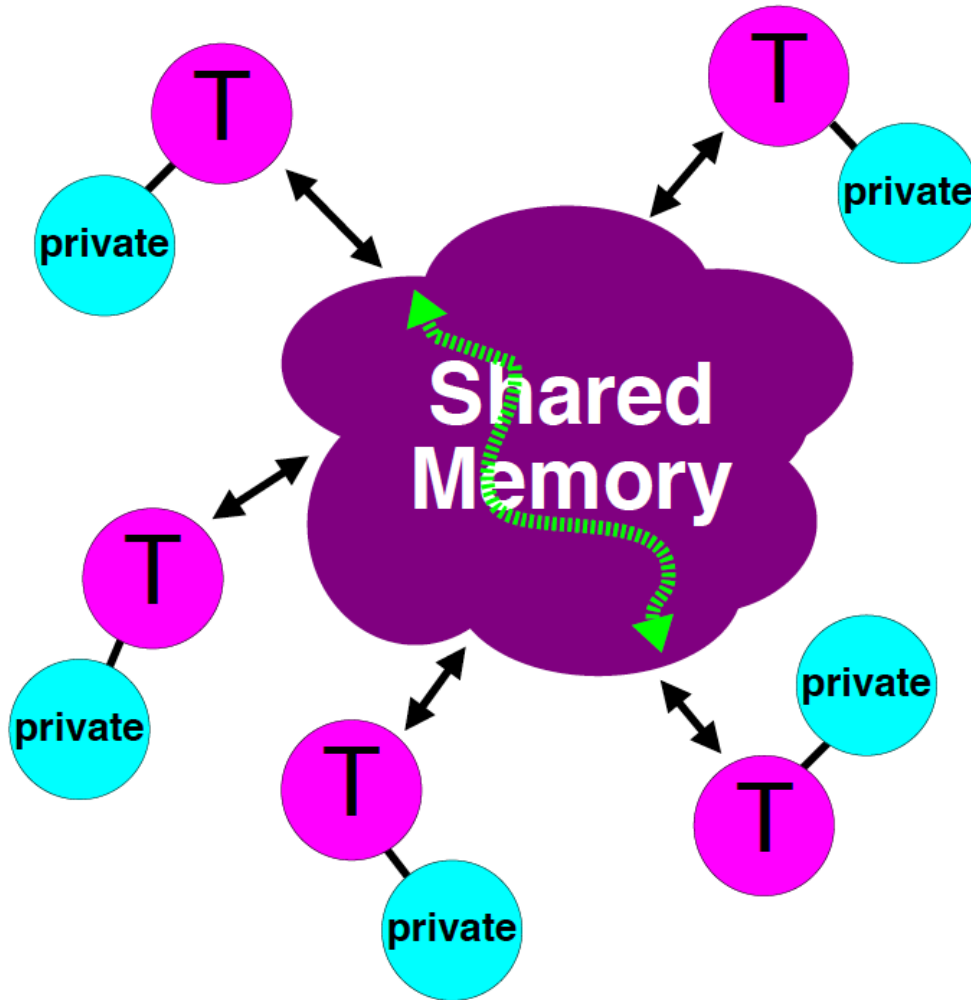
OpenMP Syntax

- Most OpenMP constructs are compiler directives using pragmas.
 - For C and C++, the pragmas take the form:
`#pragma omp construct [clause [clause]...]`
 - For Fortran, the directives take one of the forms:
 - Fixed form
`*$OMP construct [clause [clause]...]`
`C$OMP construct [clause [clause]...]`
 - Free form (but works for fixed form too)
`!$OMP construct [clause [clause]...]`
- Include file and the OpenMP lib module
`#include <omp.h>`
`use omp_lib`

OpenMP Compiler

- OpenMP: thread programming at “high level”.
 - The user does not need to specify all the details
 - Assignment of work to threads
 - Creation of threads
- User makes strategic decisions
- Compiler figures out details
 - Compiler flags enable OpenMP (e.g. `-openmp`, `-xopenmp`, `-fopenmp`, `-mp`)

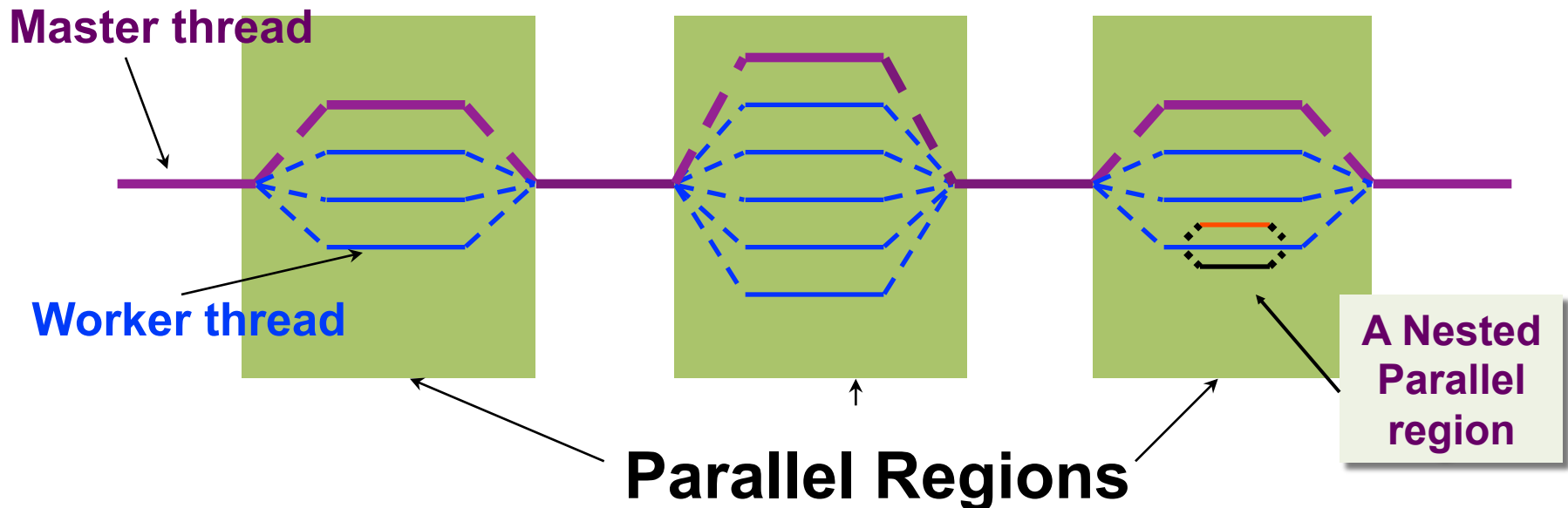
OpenMP Memory Model



- ✓ All threads have access to the same, globally shared, memory
- ✓ Data can be shared or private
- ✓ Shared data is accessible by all threads
- ✓ Private data can only be accessed by the thread that owns it
- ✓ Data transfer is transparent to the programmer
- ✓ Synchronization takes place, but it is mostly implicit

OpenMP Fork-Join Execution Model

- **Master thread** spawns multiple **worker threads** as needed, together form a **team**
- **Parallel region** is a block of code executed by all threads in a team simultaneously



OpenMP Parallel Regions

- In **C/C++**: a **block** is a single statement or a group of statement between { }

```
#pragma omp parallel
{
    id = omp_get_thread_num();
    res[id] = lots_of_work(id);
}
```

```
#pragma omp parallel for
for(i=0;i<N;i++) {
    res[i] = big_calc(i);
    A[i] = B[i] + res[i];
}
```

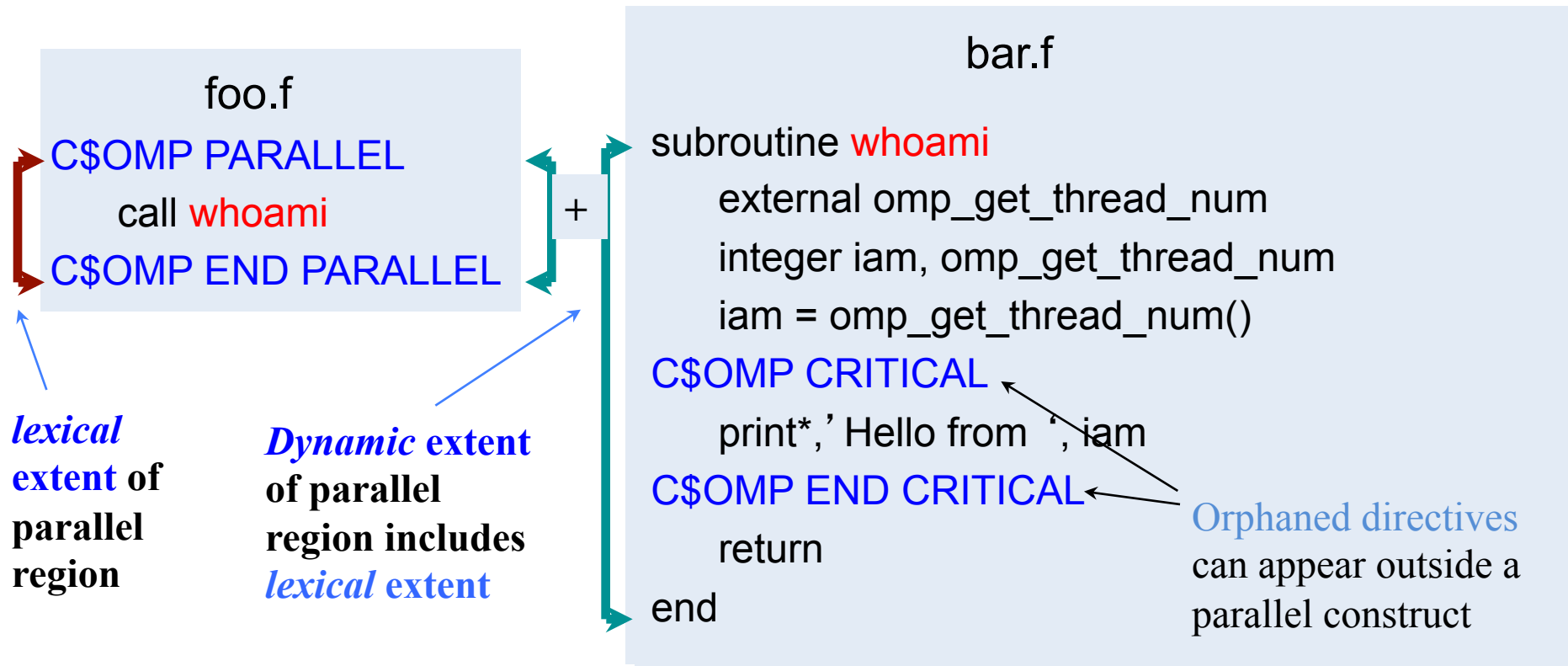
- In **Fortran**: a **block** is a single statement or a group of statements between directive/end-directive pairs.

```
C$OMP PARALLEL
10  wrk(id) = garbage(id)
    res(id) = wrk(id)**2
    if(.not.conv(res(id))) goto 10
C$OMP END PARALLEL
```

```
C$OMP PARALLEL DO
    do i=1,N
        res(i)=bigComp(i)
    end do
C$OMP END PARALLEL DO
```

Scope of OpenMP Region

A parallel region can span multiple source files.



OpenMP Worksharing Constructs

- Divides the execution of the enclosed code region among the members of the team
- The “for” worksharing construct splits up loop iterations among threads in a team
 - Each thread gets one or more “chunk”

```
#pragma omp parallel
#pragma omp for
for (i = 0; i < N; i++) {
    work(i);
}
```

By default, there is a barrier at the end of the “omp for”. Use the “**nowait**” clause to turn off the barrier.

#pragma omp for nowait

“**nowait**” is useful between two consecutive, independent omp for loops.

Worksharing Constructs

Sequential code

```
for(i=0;i<N;i++) { a[i] = a[i] + b[i]; }
```

OpenMP parallel region

```
#pragma omp parallel  
{  
    int id, i, Nthrds, istart, iend;  
    id = omp_get_thread_num();  
    Nthrds = omp_get_num_threads();  
    istart = id * N / Nthrds;  
    iend = (id+1) * N / Nthrds;  
    for(i=istart;i<iend;i++) { a[i] = a[i] + b[i]; }  
}
```

OpenMP parallel region and a worksharing for construct

```
#pragma omp parallel  
#pragma omp for schedule(static)  
    for(i=0;i<N;i++) { a[i] = a[i] + b[i]; }
```

OpenMP `schedule` Clause

`schedule (static | dynamic | guided [, chunk])`
`schedule (auto | runtime)`

<code>static</code>	Distribute iterations in blocks of size "chunk" over the threads in a round-robin fashion
<code>dynamic</code>	Fixed portions of work; size is controlled by the value of chunk; When a thread finishes, it starts on the next portion of work
<code>guided</code>	Same dynamic behavior as "dynamic", but size of the portion of work decreases exponentially
<code>auto</code>	The compiler (or runtime system) decides what is best to use; choice could be implementation dependent
<code>runtime</code>	Iteration scheduling scheme is set at runtime through environment variable <code>OMP_SCHEDULE</code>

OpenMP Sections

- Worksharing construct
- Gives a different structured block to each thread

```
#pragma omp parallel
#pragma omp sections
{
  #pragma omp section
    x_calculation();
  #pragma omp section
    y_calculation();
  #pragma omp section
    z_calculation();
}
```

By default, there is a barrier at the end of the “`omp sections`”. Use the “`nowait`” clause to turn off the barrier.

Loop Collapse

- Allows parallelization of perfectly nested loops without using nested parallelism
- The `collapse` clause on for/do loop indicates how many loops should be collapsed

```
!$omp parallel do collapse(2) ...  
do i = il, iu, is  
  do j = jl, ju, js  
    do k = kl, ku, ks  
      .....  
    end do  
  end do  
end do  
!$omp end parallel do
```

OpenMP Master

- Denotes a structured block executed by the master thread
- The other threads just skip it
 - no synchronization is implied

```
#pragma omp parallel private (tmp)
{
    do_many_things();
    #pragma omp master
        { exchange_boundaries(); }
    #pragma barrier
        do_many_other_things();
}
```

OpenMP Single

- Denotes a block of code that is executed by only one thread.
- A barrier is implied at the end of the single block.

```
#pragma omp parallel private (tmp)
{
    do_many_things();
    #pragma omp single
    {   exchange_boundaries();   }
    do_many_other_things();
}
```

OpenMP Tasks

Define a task:

- C/C++: **#pragma omp task**
- Fortran: **!\$omp task**
- A **task** is generated when a thread encounters a **task** construct or a *parallel construct*
 - Contains a task region and its data environment
 - Task can be nested
- A **task region** is a region consisting of all code encountered during the execution of a task.
- The **data environment** consists of all the variables associated with the execution of a given task.
 - constructed when the task is generated

Task completion and synchronization

- **Task completion** occurs when the task reaches the end of the task region code
- Multiple tasks joined to complete through the use of **task synchronization constructs**
 - **taskwait**
 - **barrier** construct
- **taskwait** constructs:
 - `#pragma omp taskwait`
 - `!$omp taskwait`

```
int fib(int n) {  
    int x, y;  
    if (n < 2) return n;  
    else {  
        #pragma omp task shared(x)  
        x = fib(n-1);  
        #pragma omp task shared(y)  
        y = fib(n-2);  
        #pragma omp taskwait  
        return x + y;  
    }  
}
```

Example: A Linked List

```
.....  
while(my_pointer) {  
  
    (void) do_independent_work (my_pointer);  
    my_pointer = my_pointer->next ;  
} // End of while loop  
  
.....
```

***Hard to do before OpenMP 3.0:
First count number of iterations, then
convert while loop to for loop***

Example: A Linked List with Tasking

```
my_pointer = listhead;
#pragma omp parallel
{
    #pragma omp single nowait
    {
        while(my_pointer) {
            #pragma omp task firstprivate(my_pointer)
            {
                (void) do_independent_work (my_pointer);
            }
            my_pointer = my_pointer->next ;
        } // End of single - no implied barrier (nowait)
    } // End of parallel region - implied barrier
}
```

OpenMP Task is specified here
(executed in parallel)

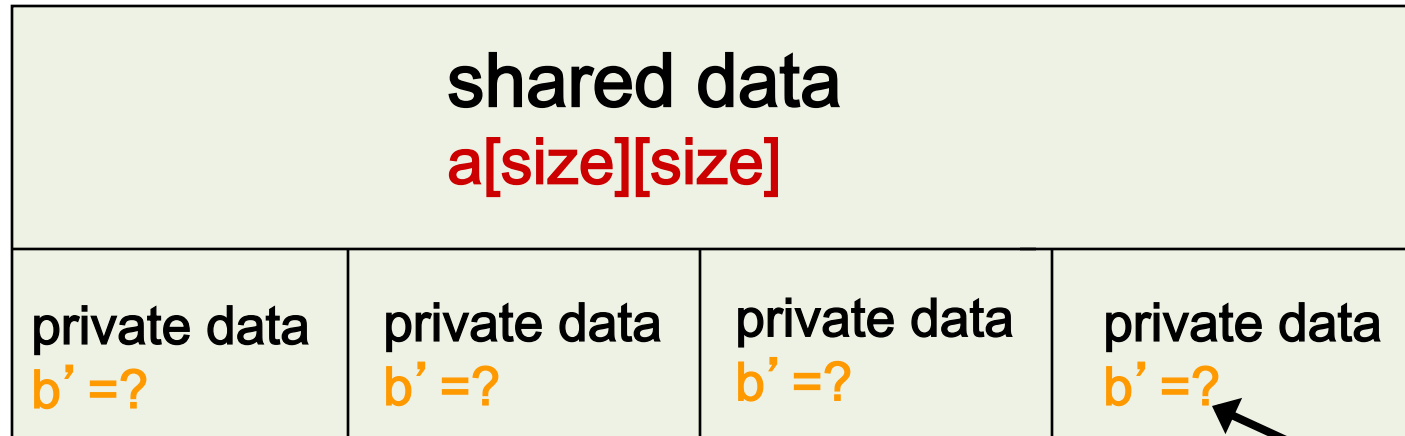


Data Environment

- Most variables are shared by default
- Global variables are SHARED among threads
 - Fortran: COMMON blocks, SAVE variables, MODULE variables
 - C: File scope variables, static
- But not everything is shared...
 - Stack variables in sub-programs called from parallel regions are PRIVATE
 - Automatic variables defined inside the parallel region are PRIVATE.

OpenMP Data Environment

```
double a[size][size], b=4;  
#pragma omp parallel private (b)  
{ .... }
```



T0

T1

T2

T3

b becomes undefined

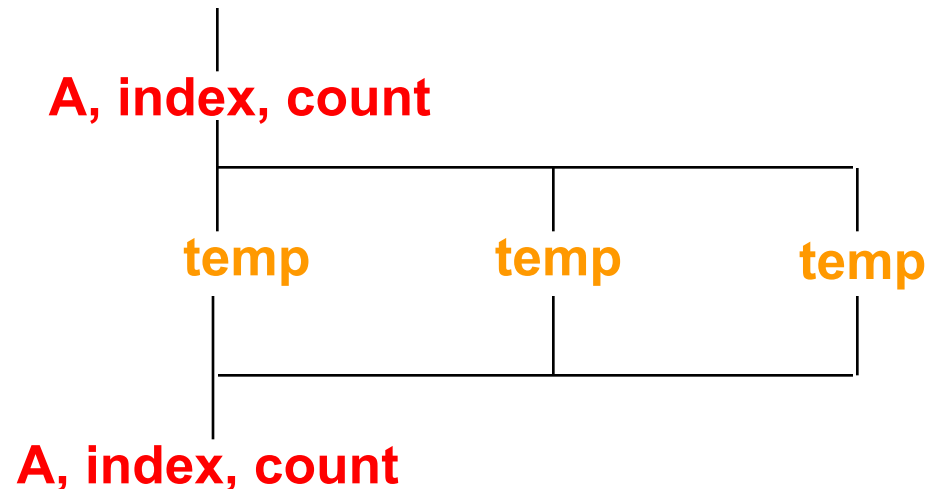
OpenMP Data Environment

```
program sort
common /input/ A(10)
integer index(10)
C$OMP PARALLEL
  call work (index)
C$OMP END PARALLEL
print*, index(1)
```

```
subroutine work (index)
common /input/ A(10)
integer index(*)
real temp(10)
integer count
  save count
  .....
```

A, **index** and **count** are shared by all threads.

temp is local to each thread



Data Environment:

Changing storage attributes

- Selectively change storage attributes constructs using the following clauses
 - SHARED
 - PRIVATE
 - FIRSTPRIVATE
 - THREADPRIVATE
- The value of a private inside a parallel loop can be transmitted to a global value outside the loop with
 - LASTPRIVATE
- The default status can be modified with:
 - DEFAULT (PRIVATE | SHARED | NONE)

OpenMP Private Clause

- `private(var)` creates a local copy of `var` for each thread.
 - The value is *uninitialized*
 - Private copy is *not storage-associated* with the original
 - The original is *undefined* at the end

```
IS = 0
C$OMP PARALLEL DO PRIVATE(IS)
  DO J=1,1000
    IS = IS + J
  END DO
C$OMP END PARALLEL DO
print *, IS
```

IS was not initialized

IS is undefined here

Firstprivate Clause

- **firstprivate** is a special case of private.
 - Initializes each private copy with the corresponding value from the master thread.

```
IS = 0
C$OMP PARALLEL DO FIRSTPRIVATE(IS)
  DO 20 J=1,1000
    IS = IS + J
  20 CONTINUE
C$OMP END PARALLEL DO
print *, IS
```



Each thread gets its own IS with an initial value of 0

Regardless of initialization, IS is undefined at this point

Lastprivate Clause

- **Lastprivate** passes the value of a private from the last iteration to the variable of the master thread

```
IS = 0
C$OMP PARALLEL DO FIRSTPRIVATE(IS)
C$OMP& LASTPRIVATE(IS)
  DO 20 J=1,1000
    IS = IS + J
20 CONTINUE
C$OMP END PARALLEL DO
print *, IS
```

Are you sure ?

Each thread gets its own IS with an initial value of 0

IS is defined as its value at the last iteration (i.e. for J=1000)

OpenMP Reduction

- Here is the correct way to parallelize this code.

```
IS = 0
C$OMP PARALLEL DO REDUCTION(+:IS)
  DO 1000 J=1,1000
    IS = IS + J
  1000 CONTINUE
  print *, IS
```

Reduction implies firstprivate (?)

Reduction operands/initial-values

- Associative operands used with reduction
- Initial values are the ones that make sense mathematically

Operand	Initial value
+	0
*	1
-	0
.AND.	All 1's

Operand	Initial value
.OR.	0
MAX	1
MIN	0
//	All 1's

OpenMP Threadprivate

- Makes global data private to a thread, *thus crossing parallel region boundary*
 - Fortran: COMMON blocks
 - C: File scope and static variables
- Different from making them PRIVATE
 - With PRIVATE, global variables are masked.
 - **THREADPRIVATE** preserves global scope within each thread
- Threadprivate variables can be initialized using COPYIN or by using DATA statements.

Threadprivate/copyin

- You initialize **threadprivate** data using a **copyin** clause.

```
parameter (N=1000)
common/buf/A(N)
C$OMP THREADPRIVATE(/buf/)

C Initialize the A array
  call init_data(N,A)

C$OMP PARALLEL COPYIN(A)
... Now each thread sees threadprivate array A initialized
... to the global value set in the subroutine init_data()
C$OMP END PARALLEL

....
C$OMP PARALLEL
... Values of threadprivate are persistent across parallel regions
C$OMP END PARALLEL
```

OpenMP Synchronization

- High level synchronization:
 - critical section
 - atomic
 - barrier
 - ordered
- Low level synchronization
 - flush
 - locks (both simple and nested)

Critical section

- Only one thread at a time can enter a **critical** section.

```
C$OMP PARALLEL DO PRIVATE(B)
C$OMP& SHARED(RES)
    DO 100 I=1,NITERS
        B = DOIT(I)
C$OMP CRITICAL
        CALL CONSUME (B, RES)
C$OMP END CRITICAL
100    CONTINUE
C$OMP END PARALLEL DO
```

Atomic

- **Atomic** is a special case of a critical section that can be used for certain simple statements
- It applies only to the update of a memory location

```
C$OMP PARALLEL PRIVATE(B)
```

```
  B = DOIT(I)
```

```
  tmp = big_ugly();
```

```
C$OMP ATOMIC
```

```
  X = X + temp
```

```
C$OMP END PARALLEL
```

Barrier

- **Barrier:** Each thread waits until all threads arrive.

```
#pragma omp parallel shared (A, B, C) 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_calc3(id);
```

```
} ←
```

implicit barrier at the end of a **for** work-sharing construct

implicit barrier at the end of a parallel region

no implicit barrier due to **nowait**

Ordered

- The **ordered** construct enforces the sequential order for a block.

```
#pragma omp parallel private (tmp)
#pragma omp for ordered
for (i=0;i<N;i++){
    tmp = NEAT_STUFF(i);
#pragma ordered
    res += consum(tmp);
}
```

OpenMP Synchronization

- The **flush** construct denotes a sequence point where a thread tries to create a consistent view of memory.
 - All memory operations (both reads and writes) defined prior to the sequence point must complete.
 - All memory operations (both reads and writes) defined after the sequence point must follow the flush.
 - Variables in registers or write buffers must be updated in memory.
- Arguments to flush specify which variables are flushed. No arguments specifies that all thread visible variables are flushed.

A flush example

- pair-wise synchronization.

```
integer ISYNC(NUM_THREADS)
C$OMP PARALLEL DEFAULT (PRIVATE) SHARED (ISYNC)
  IAM = OMP_GET_THREAD_NUM()
  ISYNC(IAM) = 0
C$OMP BARRIER
  CALL WORK()
  ISYNC(IAM) = 1 ! I'm all done; signal this to other threads
C$OMP FLUSH(ISYNC)
  DO WHILE (ISYNC(NEIGH) .EQ. 0)
C$OMP FLUSH(ISYNC)
  END DO
C$OMP END PARALLEL
```

Make sure other threads can see my write.

Make sure the read picks up a good copy from memory.

Note: **flush** is analogous to a fence in other shared memory APIs.

OpenMP Lock routines

- Simple Lock routines: available if it is unset.
 - `omp_init_lock()`, `omp_set_lock()`,
`omp_unset_lock()`, `omp_test_lock()`,
`omp_destroy_lock()`
- Nested Locks: available if it is unset or if it is set but owned by the thread executing the nested lock function
 - `omp_init_nest_lock()`, `omp_set_nest_lock()`,
`omp_unset_nest_lock()`, `omp_test_nest_lock()`,
`omp_destroy_nest_lock()`

OpenMP Locks

- Protect resources with locks.

```
omp_lock_t lck;  
omp_init_lock(&lck);  
#pragma omp parallel private (tmp, id)  
{  
    id = omp_get_thread_num();  
    tmp = do_lots_of_work(id);  
    omp_set_lock(&lck);  
    printf(“%d %d”, id, tmp);  
    omp_unset_lock(&lck);  
}  
omp_destroy_lock(&lck);
```

Wait here for
your turn.

Release the lock so
the next thread gets
a turn.

Free-up storage when done.

OpenMP Library Routines

- Modify/Check the number of threads
 - `omp_set_num_threads()`, `omp_get_num_threads()`,
`omp_get_thread_num()`, `omp_get_max_threads()`
- Are we in a parallel region?
 - `omp_in_parallel()`
- How many processors in the system?
 - `omp_num_procs()`

OpenMP Environment Variables

- Set the default number of threads to use.
 - `OMP_NUM_THREADS` *int_literal*
- Control how “omp for schedule(RUNTIME)” loop iterations are scheduled.
 - `OMP_SCHEDULE` “schedule[, chunk_size]”

Outline

- OpenMP Introduction
- Parallel Programming with OpenMP
 - Worksharing, tasks, data environment, synchronization
- OpenMP Performance and Best Practices
- Hybrid MPI/OpenMP
- Case Studies and Examples
- Reference Materials

OpenMP Performance

- Relative ease of using OpenMP is a mixed blessing
- We can quickly write a correct OpenMP program, but without the desired level of performance.
- There are certain “best practices” to avoid common performance problems.
- Extra work needed to program with large thread count

Typical OpenMP Performance Issues

- Overheads of OpenMP constructs, thread management. E.g.
 - dynamic loop schedules have much higher overheads than static schedules
 - Synchronization is expensive, use NOWAIT if possible
 - Large parallel regions help reduce overheads, enable better cache usage and standard optimizations
- Overheads of runtime library routines
 - Some are called frequently
- Load balance
- Cache utilization and false sharing

OpenMP Best Practices

- Reduce usage of barrier with `nowait` clause

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

OpenMP Best Practices

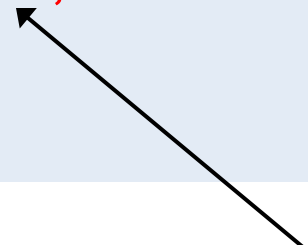
```
#pragma omp parallel private(i)
{
    #pragma omp for nowait
    for(i=0;i<n;i++)
        a[i] +=b[i];
    #pragma omp for nowait
    for(i=0;i<n;i++)
        c[i] +=d[i];
    #pragma omp barrier
    #pragma omp for nowait reduction(+:sum)
    for(i=0;i<n;i++)
        sum += a[i] + c[i];
}
```

OpenMP Best Practices

- Avoid large **ordered** construct
- Avoid large **critical** regions

```
#pragma omp parallel shared(a,b) private(c,d)
{
    ....
    #pragma omp critical
    {
        a += 2*c;
        c = d*d;
    }
}
```

Move out this
Statement



OpenMP Best Practices

- Maximize Parallel Regions

```
#pragma omp parallel
{
    #pragma omp for
    for (...) { /* Work-sharing loop 1 */ }
}
opt = opt + N; //sequential
#pragma omp parallel
{
    #pragma omp for
    for(...) { /* Work-sharing loop 2 */ }

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

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

    #pragma omp single nowait
    opt = opt + N; //sequential

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

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

OpenMP Best Practices

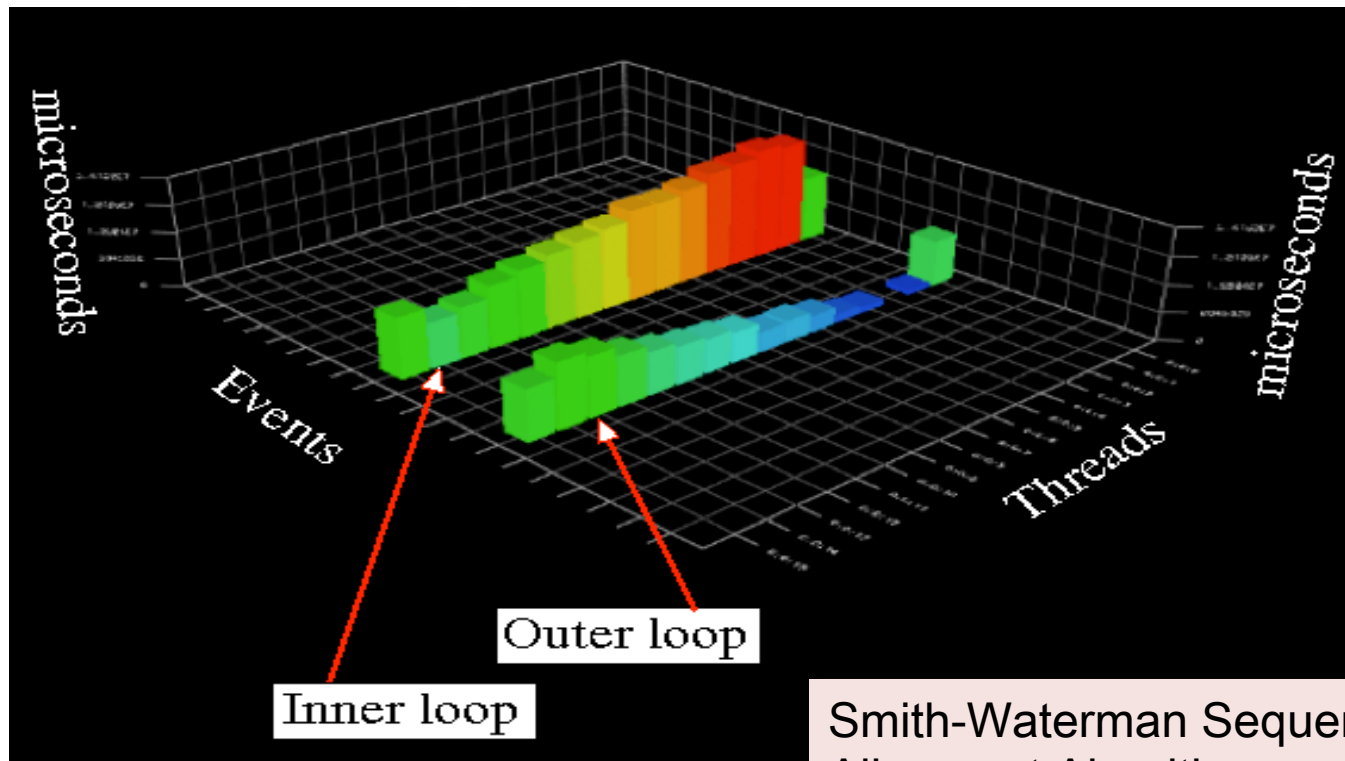
- Single parallel region enclosing all work-sharing loops.

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

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

OpenMP Best Practices

- Address load imbalances
- Use parallel for **dynamic** schedules and different chunk sizes



Smith-Waterman Sequence Alignment Algorithm

OpenMP Best Practices

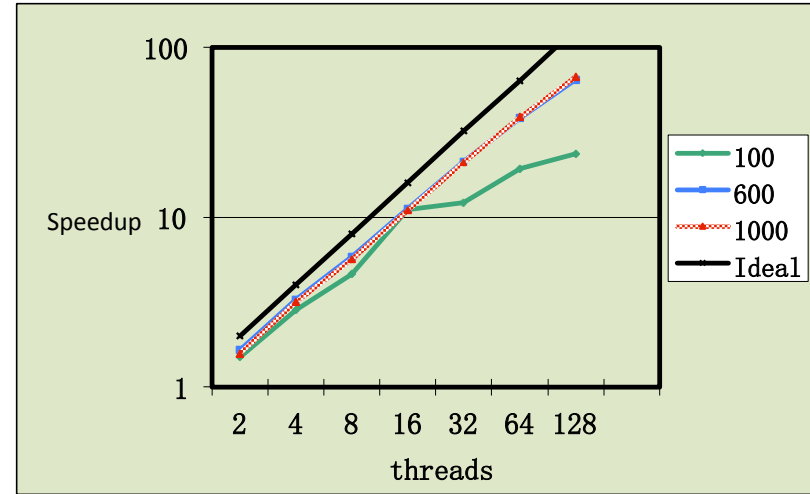
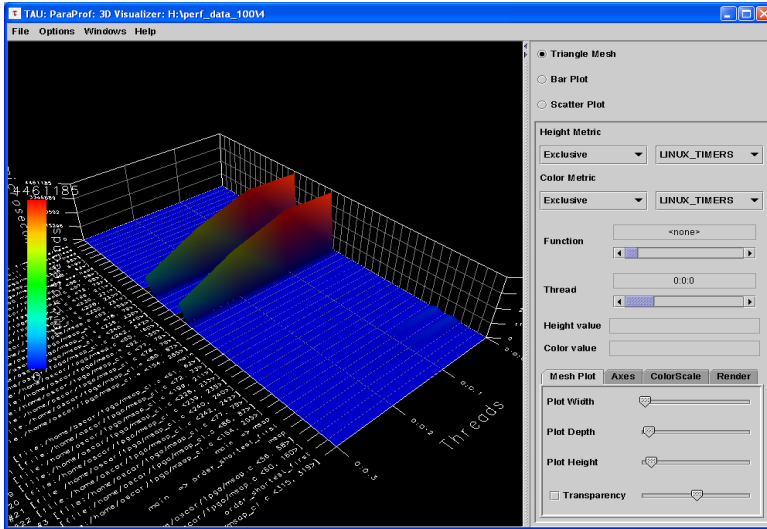
- Smith-Waterman Algorithm
 - Default `schedule` is for `static` even → load imbalance

```
#pragma omp for
for(...)
  for(...)
    for(...)
      for(...)
        { /* compute alignments */ }
#pragma omp critical
{. /* compute scores */ }
```

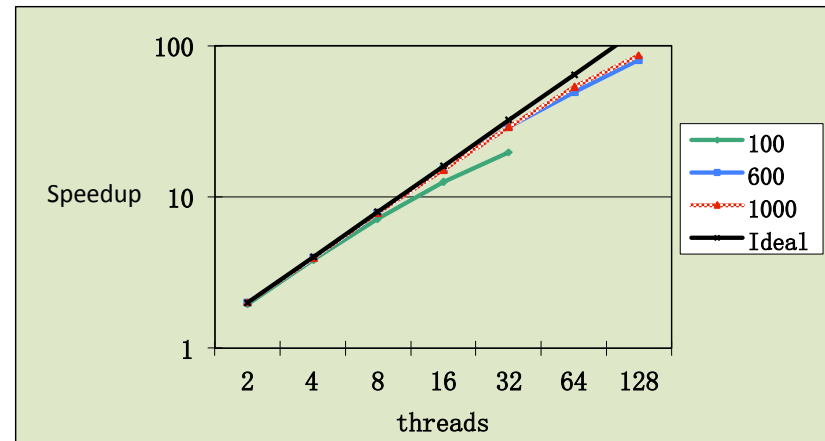
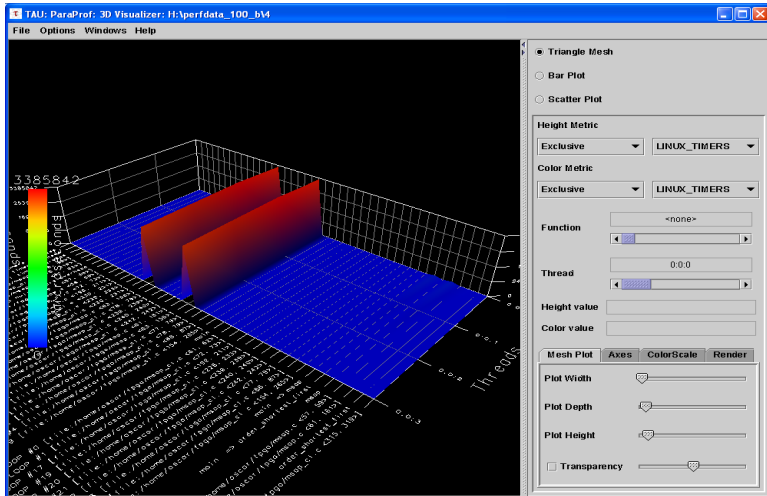

OpenMP Best Practices

Smith-Waterman Sequence Alignment Algorithm

#pragma omp for



#pragma omp for dynamic(schedule, 1)

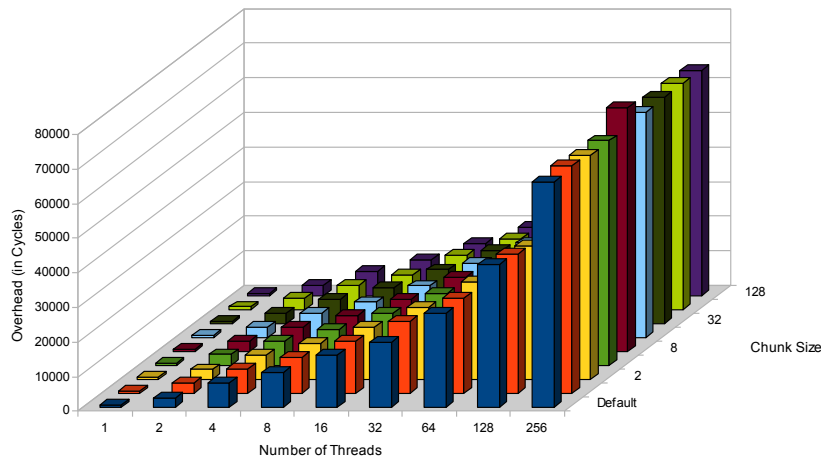


128 threads with 80% efficiency

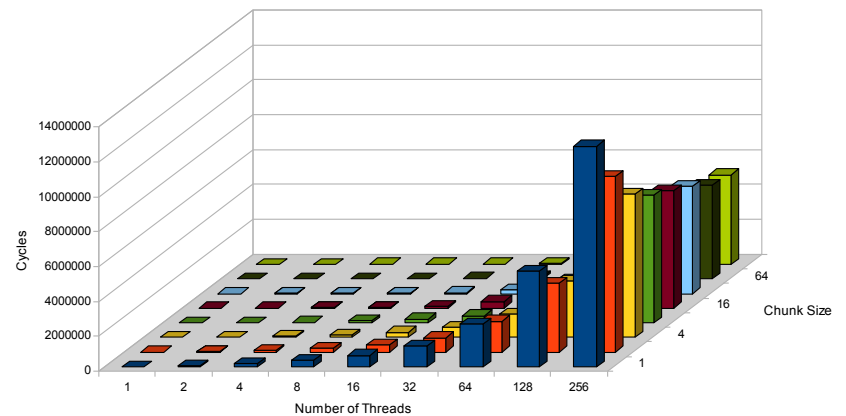
OpenMP Best Practices

- Address load imbalances by selecting the best schedule and chunk size
- Avoid selecting small chunk size when work in chunk is small.

Overheads of OpenMP For Static Scheduling
SGI Altix 3600



Overheads of OpenMP For Dynamic Schedule
SGI Altix 3600



OpenMP Best Practices

- Pipeline processing to overlap I/O and computations

```
for (i=0; i<N; i++) {  
    ReadFromFile(i,...);  
  
    for(j=0; j<ProcessingNum; j++)  
        ProcessData(i, j);  
  
    WriteResultsToFile(i)  
}
```

OpenMP Best Practices

- Pipeline Processing
- Pre-fetches I/O
- Threads reading or writing files joins the computations

```
#pragma omp parallel
{
    #pragma omp single
    { ReadFromFile(0,...); }

    for (i=0; i<N; i++) {
        #pragma omp single nowait
        { ReadFromFile(i+1,...); }

        #pragma omp for schedule(dynamic)
        for (j=0; j<ProcessingNum; j++)
            ProcessChunkOfData(i, j);

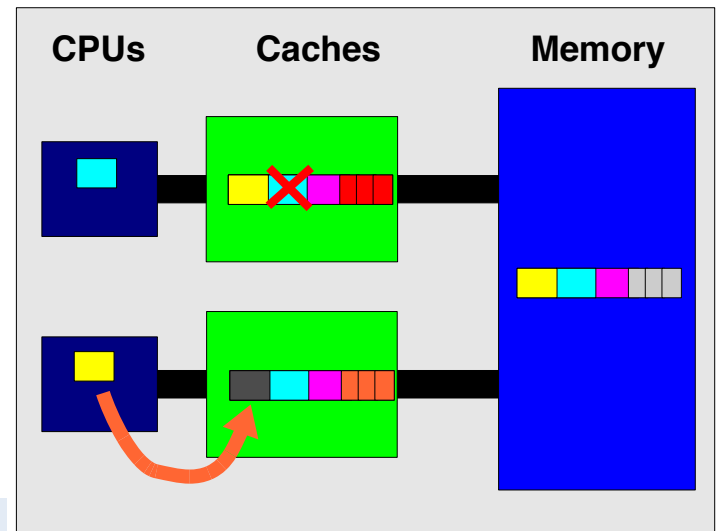
        #pragma omp single nowait
        { WriteResultsToFile(i); }
    }
}
```

OpenMP Best Practices

- **single** vs. **master** work-sharing
 - **master** is more efficient but requires thread 0 to be available
 - **single** is more efficient if **master** thread not available
 - **single** has **implicit barrier**

OpenMP Best Practices

- Avoid false sharing
 - When at least one thread write to a cache line while others access it
 - Use array padding

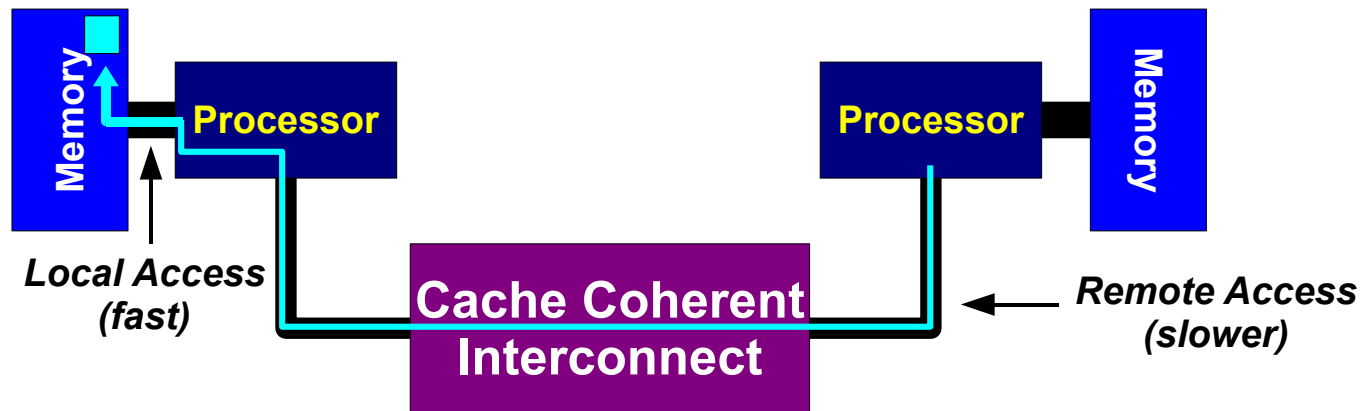


```
int a[max_threads];  
#pragma omp parallel for schedule(static,1)  
for(int i=0; i<max_threads; i++)  
    a[i] +=i;
```

```
int a[max_threads][cache_line_size];  
#pragma omp parallel for schedule(static,1)  
for(int i=0; i<max_threads; i++)  
    a[i][0] +=i;
```

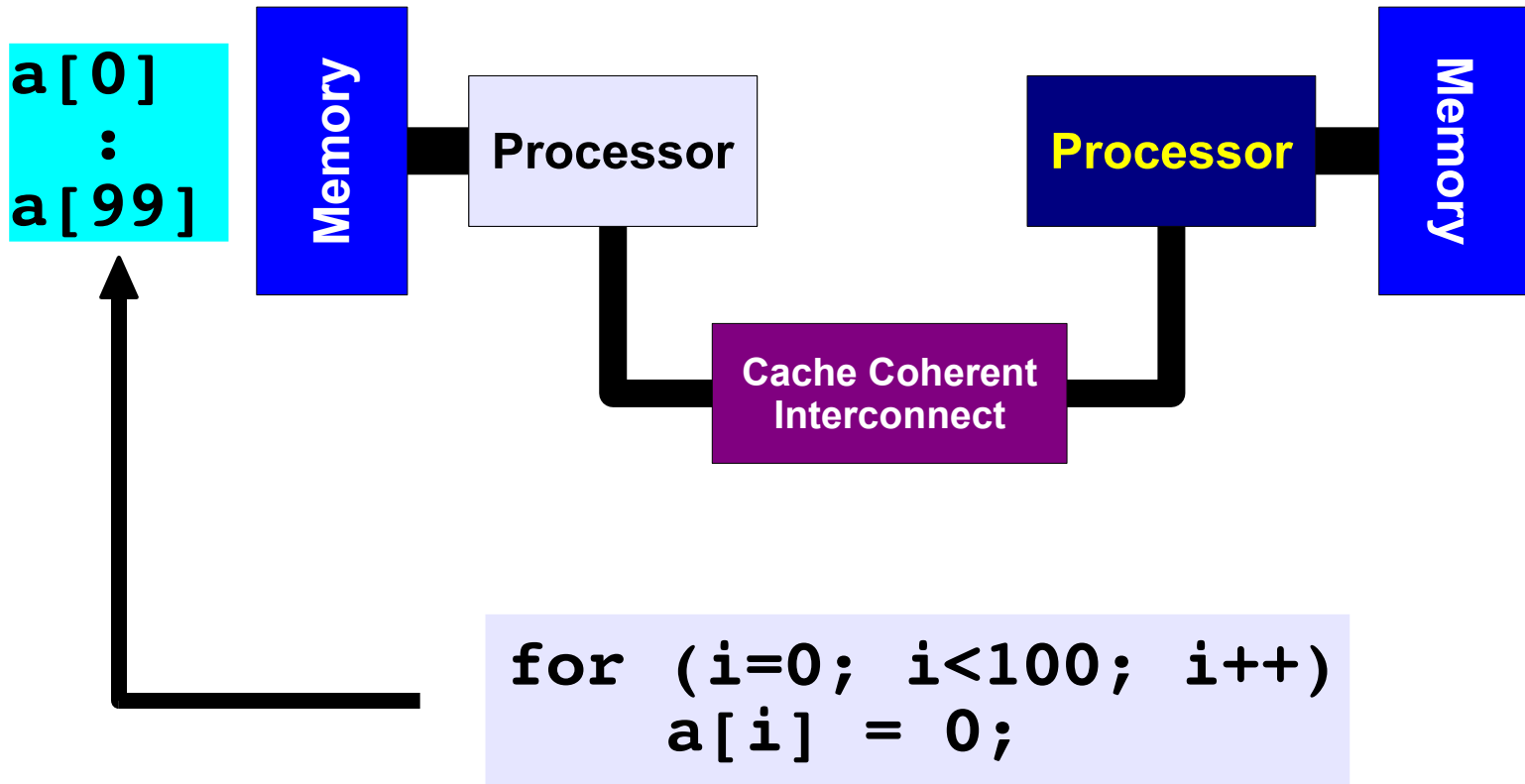
OpenMP Best Practices

- Data placement policy on NUMA architectures



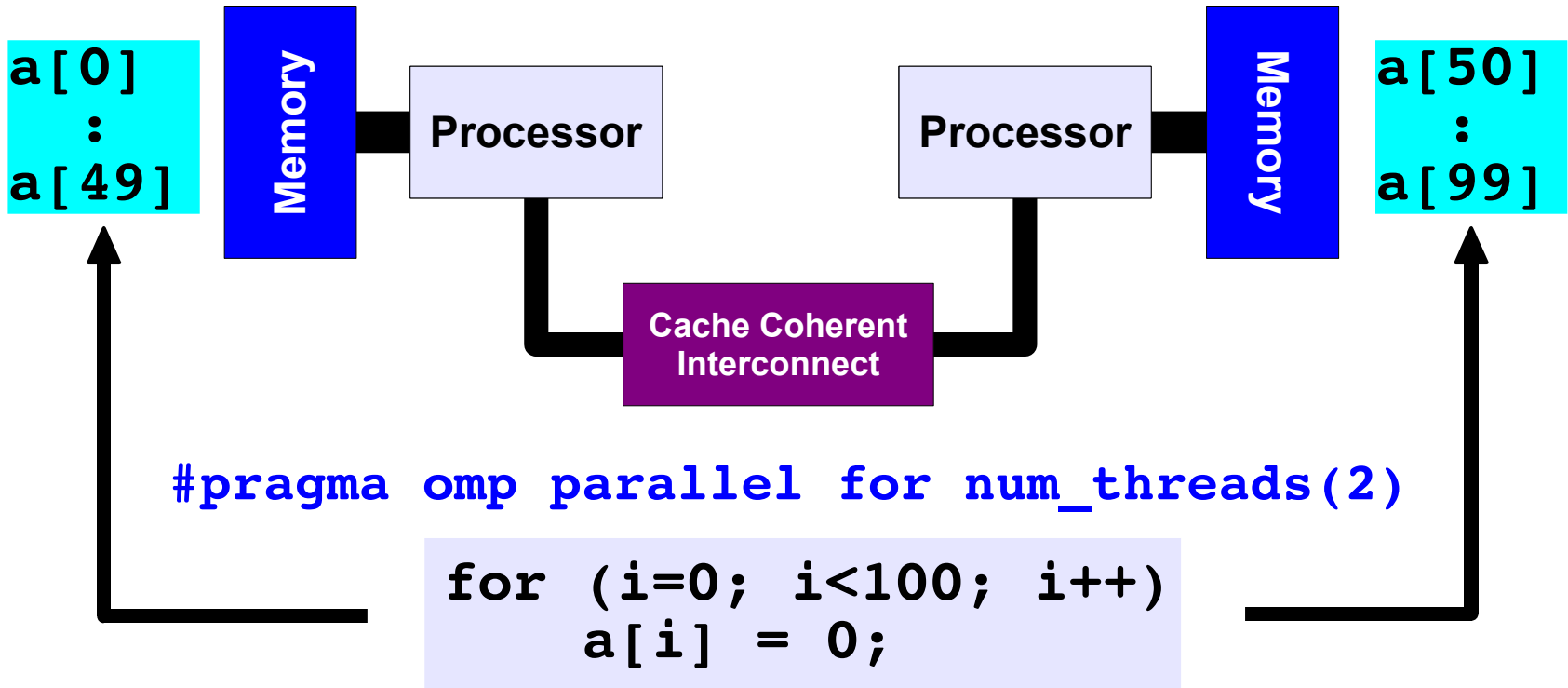
- First Touch Policy
 - The process that first touches a page of memory causes that page to be allocated in the node on which the process is running

NUMA First-touch placement/1



First Touch
All array elements are in the memory of the processor executing this thread

NUMA First-touch placement/2



First Touch
Both memories each have "their half" of the array

OpenMP Best Practices

- First-touch in practice
 - Initialize data consistently with the computations

```
#pragma omp parallel for
for(i=0; i<N; i++) {
    a[i] = 0.0; b[i] = 0.0 ; c[i] = 0.0;
}
readfile(a,b,c);
```

```
#pragma omp parallel for
for(i=0; i<N; i++) {
    a[i] = b[i] + c[i];
}
```

OpenMP Best Practices

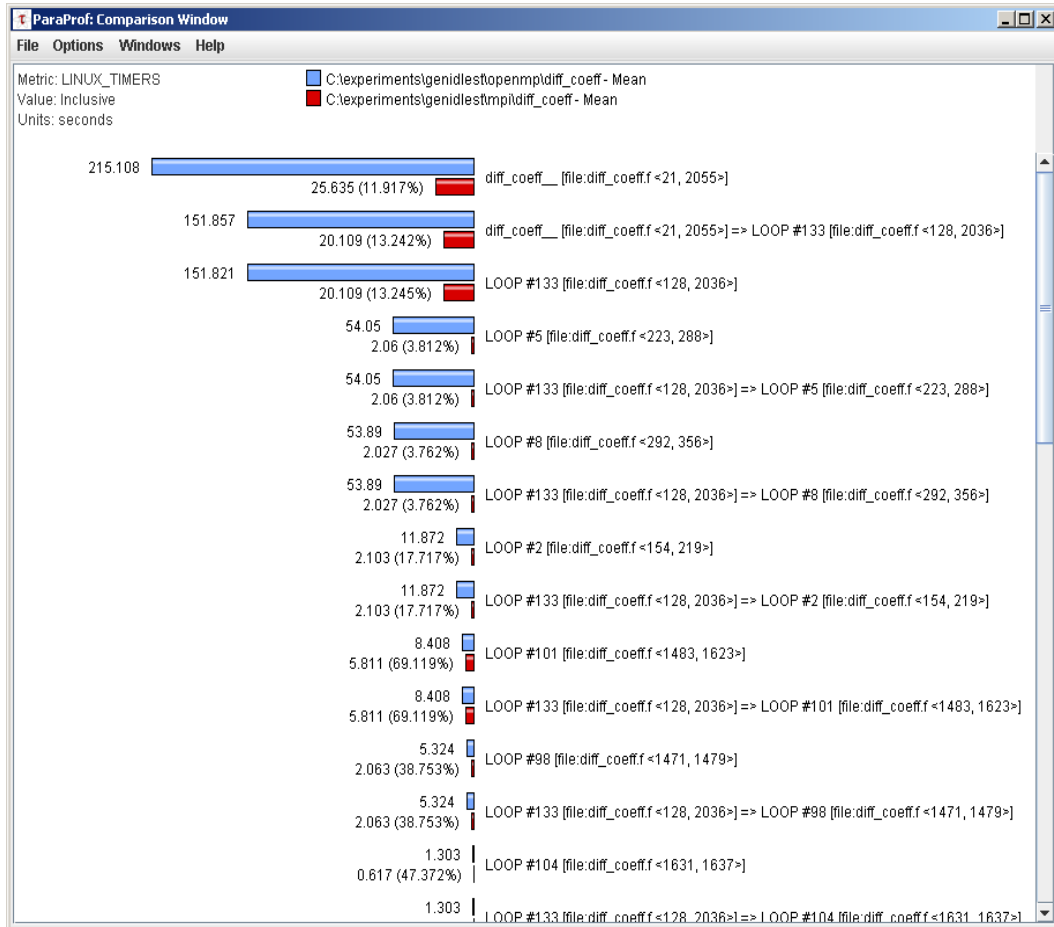
- Privatize variables as much as possible
 - Private variables are stored in the local stack to the thread
- Private data close to cache

```
double a[MaxThreads][N][N]
#pragma omp parallel for
for(i=0; i<MaxThreads; i++) {
    for(int j...)
        for(int k...)
            a[i][j][k] = ...
}
```

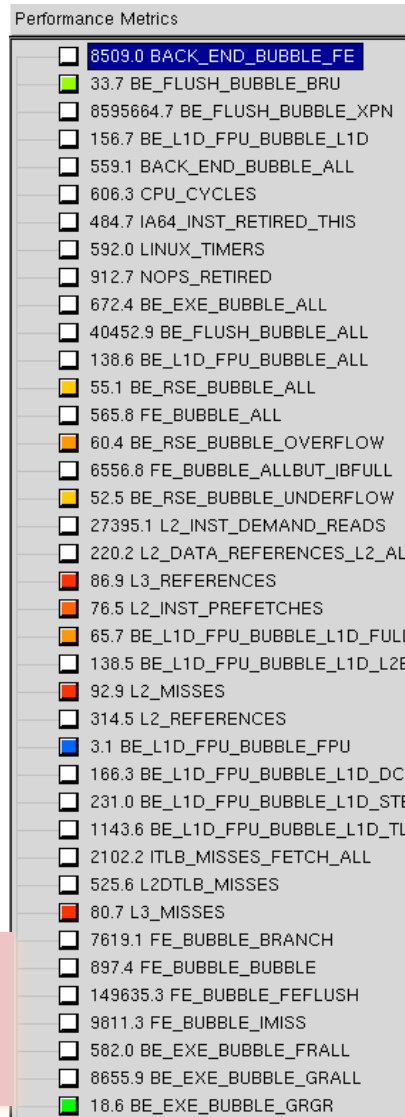
```
double a[N][N]
#pragma omp parallel private(a)
{
    for(int j...)
        for(int k...)
            a[j][k] = ...
}
```


Example: Hybrid CFD code, MPIxOpenMP

Loop Timings



When comparing the metrics between OpenMP and MPI using KOJAK performance algebra.



We found:

Large # of:

- Exceptions
- Flushes
- Cache Misses
- Pipeline stalls

Some loops are 27 times slower in OpenMP (1x8) than MPI (8x1). These loops contains large amounts of stalling due to remote memory accesses to the shared heap.

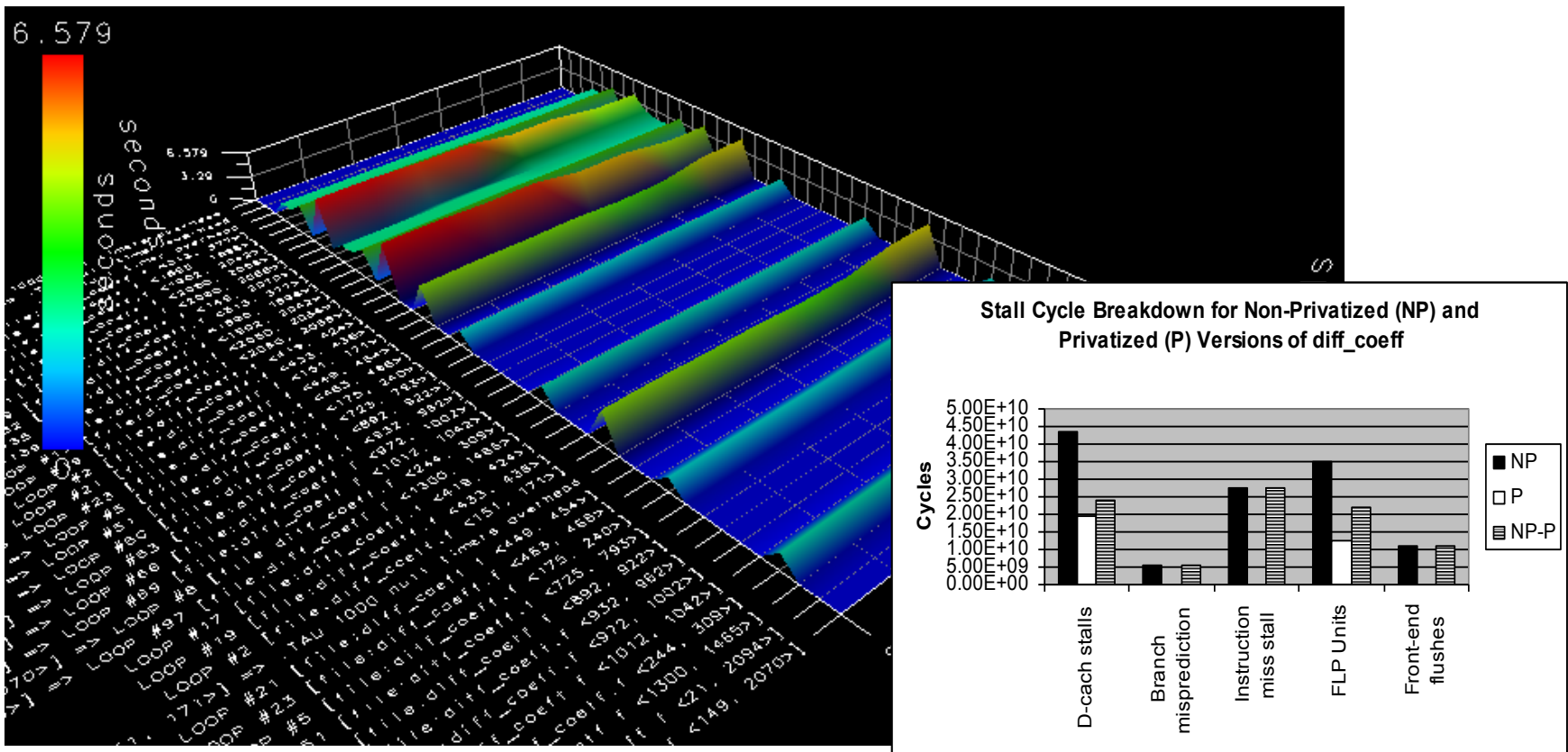
OpenMP Best Practices

- CFD application pseudo-code
 - Shared arrays initialized incorrectly (first touch policy)
 - Delays in remote memory accesses are probable causes by saturation of interconnect

```
procedure diff_coeff() {  
    array allocation by master thread  
    initialization of shared arrays  
  
    PARALLEL REGION  
    {  
        loop lower_bn [id] , upper_bn [id]  
        computation on shared arrays  
        .....    }  
}
```

OpenMP Best Practices

- Array privatization
 - Improved the performance of the whole program by 30%
 - Speedup of 10 for the procedure, now only 5% of total time
- Processor stalls are reduced significantly



OpenMP Best Practices

- Avoid Thread Migration
 - Affects data locality
- Bind threads to cores.
- Linux:
 - numactl –cpubind=0 foobar
 - taskset –c 0,1 foobar
- SGI Altix
 - dplace –x2 foobar

OpenMP Source of Errors

- Incorrect use of synchronization constructs
 - Less likely if user sticks to directives
 - Erroneous use of NOWAIT
- Race conditions (true sharing)
 - Can be very hard to find
- Wrong “spelling” of sentinel
- Use tools to check for data races.

Outline

- OpenMP Introduction
- Parallel Programming with OpenMP
 - Worksharing, tasks, data environment, synchronization
- OpenMP Performance and Best Practices
- Hybrid MPI/OpenMP
- Case Studies and Examples
- Reference Materials

Hybrid MPI/OpenMP

- Good for:
 - MPI communication overhead can be reduced by using OpenMP within the node, exploiting shared data
 - Application with two levels of parallelism
 - Application with unbalanced work load at the MPI level.
 - Application with limited # of MPI processes.

Hybrid MPI/OpenMP

- Not Good for:
 - When MPI library implementation doesn't support threads.
 - Application with one level of parallelism, no need for hierarchical parallelism.
 - OpenMP is not written correctly, introducing its drawbacks.
 - Implementation of OpenMP is not scalable.
 - Compiler dependent.

MPI Thread Support

- `MPI_INIT_THREAD` (required, provided, ierr)
 - IN: required, desired level of thread support (integer).
 - OUT: provided, provided level of thread support (integer).
 - Returned provided maybe less than required.
- `MPI_THREAD_SINGLE`: Only one thread will execute.
- `MPI_THREAD_FUNNELED`: Only main thread makes MPI calls
 - all MPI calls are "funneled" to main thread
- `MPI_THREAD_SERIALIZED`: multiple threads may make MPI calls, but only one at a time
 - MPI calls are not made concurrently from two distinct threads
- `MPI_THREAD_MULTIPLE`: Multiple threads may call MPI, with no restrictions.

Hybrid MPI/OpenMP

- If MPI_THREAD_SERIALIZED is supported
- OMP_BARRIER is needed since OMP_SINGLE only guarantees synchronization at the end.
- It also implies all other threads are sleeping!

```
!$OMP BARRIER  
!$OMP SINGLE  
    call MPI_xxx(...)  
!$OMP END SINGLE
```

Overlap COMM and COMP

- If MPI_THREAD_FUNNELED is supported
- While master or single thread is making MPI calls, other threads are performing work.
- Must be able to separate codes that can run before or after halo info is received.

```
!$OMP PARALLEL
  if (my_thread_rank < 1) then
    call MPI_xxx(...)
  else
    do some computation
  endif
!$OMP END PARALLEL
```

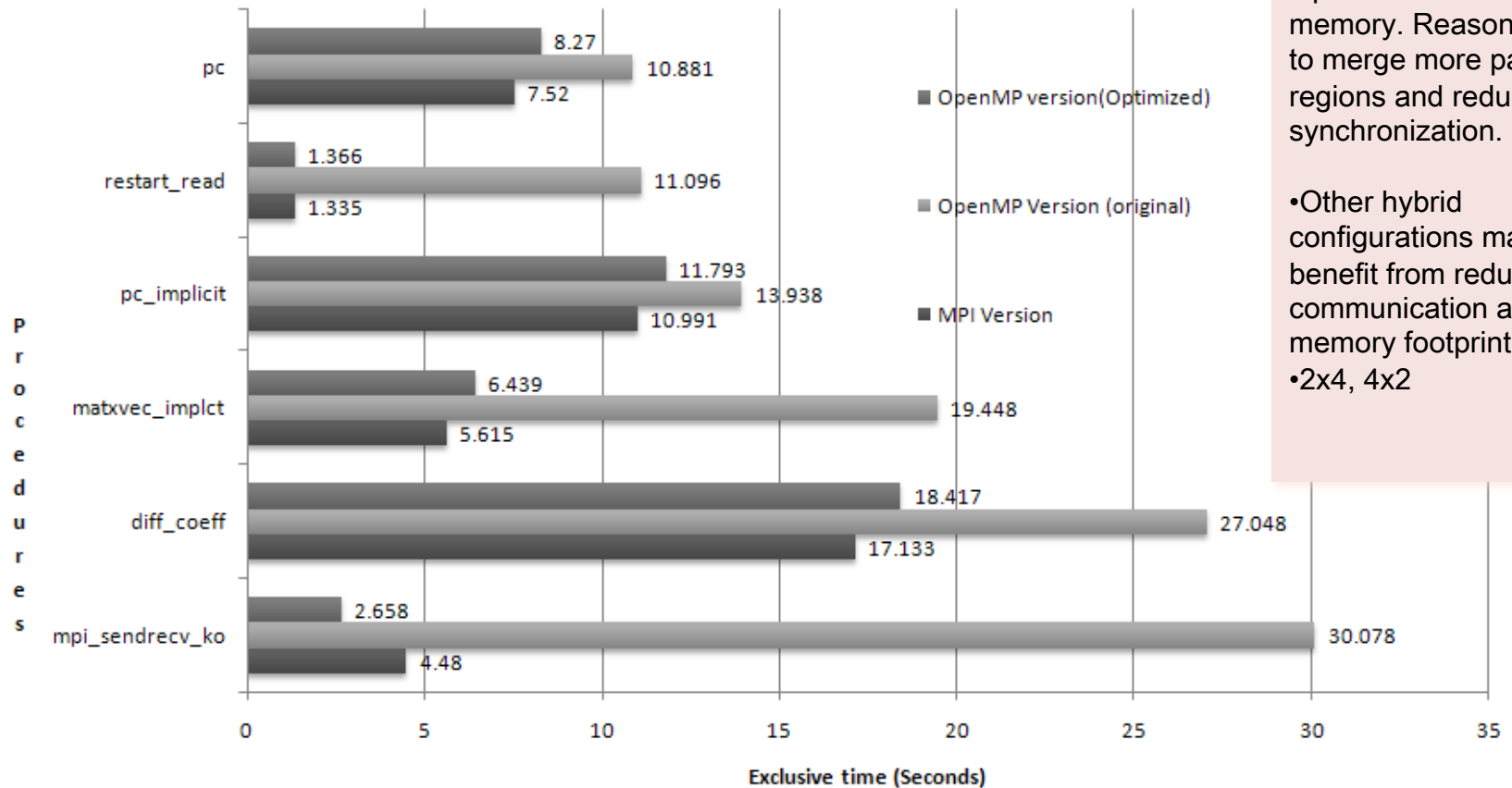
Hybrid MPI/OpenMP

- If MPI_THREAD_MULTIPLE is supported
- Good to overlap computations and communication.

```
!$OMP PARALLEL
  if (thread_id .eq. id1) then
    call mpi_routine1()
  else if (thread_id .e.q. id2) then
    call mpi_routine2()
  else
    do_compute()
  endif
!$OMP END PARALLEL
```


GenIDLest Hybrid 1x8 vs. 8x1

Timings and Improvements of GenIDLest



•Pure MPI faster 16% than pure OpenMP but OpenMP uses 30% less memory. Reason: Need to merge more parallel regions and reduce synchronization.

•Other hybrid configurations may benefit from reduced communication and less memory footprint.
•2x4, 4x2

Less Communication with OpenMP: Required replacing send/recv buffers with direct memory copies

Remarks

- Important to use OpenMP Best Practices strategy to achieve good performance
- Data locality is extremely important for OpenMP
 - Privatization or Implicit Data Placement.
- Important to reduce synchronizations
- Hybrid MPI/OpenMP
 - Uses less memory
 - Reduces MPI communication overhead.

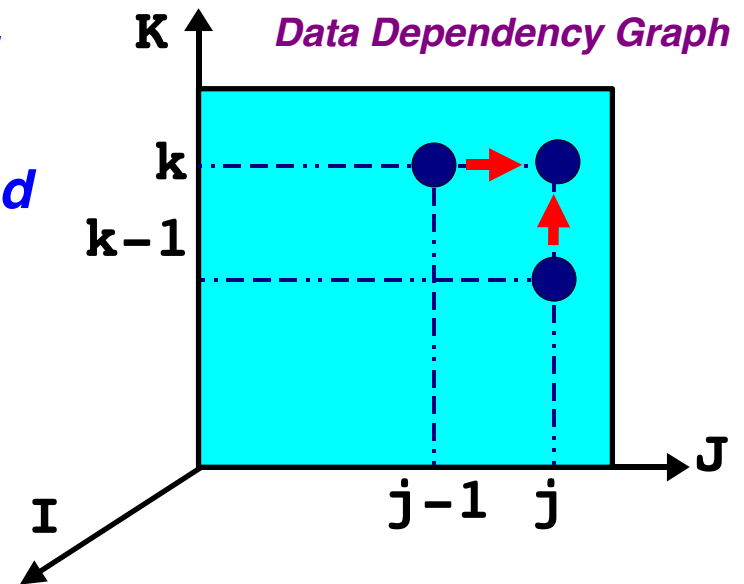
Outline

- OpenMP Introduction
- Parallel Programming with OpenMP
 - Worksharing, tasks, data environment, synchronization
- OpenMP Performance and Best Practices
- Hybrid MPI/OpenMP
- Case Studies and Examples
- Reference Materials

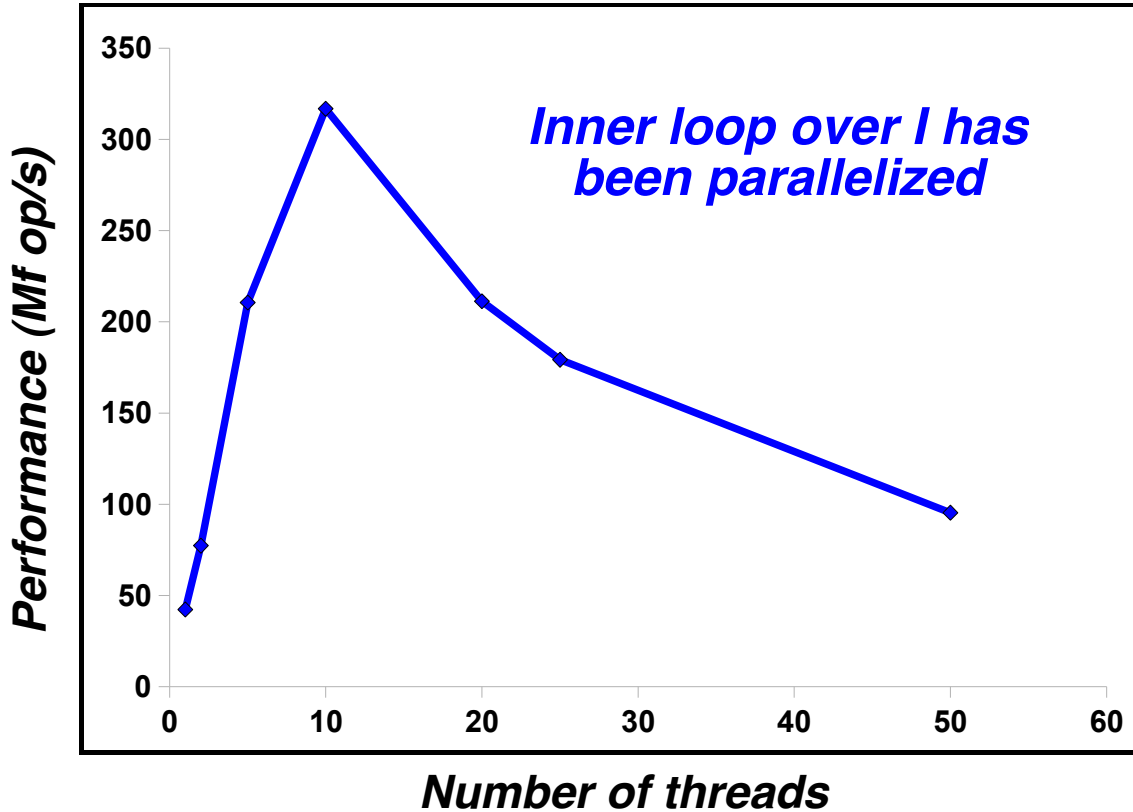
A 3D matrix update

```
do k = 2, n
  do j = 2, n
    !$omp parallel do default(shared) private(i) &
    !$omp schedule(static)
      do i = 1, m
        x(i,j,k) = x(i,j,k-1) + x(i,j-1,k)*scale
      end do
    !$omp end parallel do
  end do
end do
```

- ❑ *The loops are correctly nested for serial performance*
- ❑ *Due to a data dependency on J and K, only the inner loop can be parallelized*
- ❑ *This will cause the barrier to be executed $(N-1)^2$ times*



The performance



*Scaling is very poor
(as to be expected)*

*Dimensions : M=7,500 N=20
Footprint : ~24 MByte*

Performance analyzer data

Name	Excl. CPU	User %	Incl. User CPU	Excl. Wall
	sec.	%	sec.	sec.
Using 10 threads				
<Total>	10.590	100.0	10.590	1.550
__mt_EndOfTask_Barrier_	5.740	54.2	5.740	0.240
__mt_WaitForWork_	3.860	36.4	3.860	0.
__mt_MasterFunction_	0.480	4.5	0.680	0.480
MAIN_	0.230	2.2	1.200	0.470
block_3d_ -- MP doall from line 14 [_\$d1A14	0.170	1.6	5.910	0.170
block_3d_	0.040	0.4	6.460	0.040
memset	0.030	0.3	0.030	0.080
Using 20 threads				
<Total>	47.120	100.0	47.120	2.900
__mt_EndOfTask_Barrier_	25.700	54.5	25.700	0.980
__mt_WaitForWork_	19.880	42.2	19.880	0.
__mt_MasterFunction_	1.100	2.3	1.320	1.100
MAIN_	0.190	0.4	2.520	0.470
block_3d_ -- MP doall from line 14 [_\$d1A14.block_3d_]	0.100	0.2	25.800	0.100
__mt_setup_doJob_int_	0.080	0.2	0.080	0.080
__mt_setup_job_	0.020	0.0	0.020	0.020
block_3d_	0.010	0.0	27.020	0.010

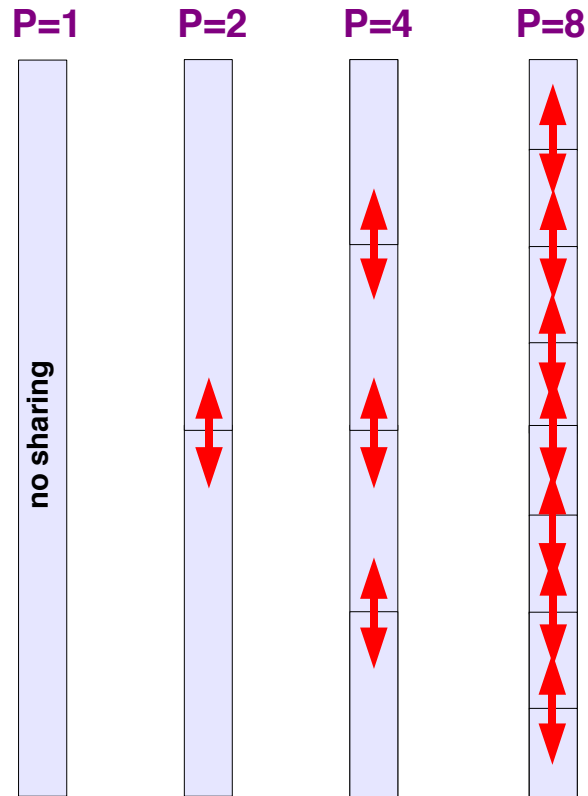
do not scale at all

scales somewhat

Question: Why is __mt_WaitForWork so high in the profile ?

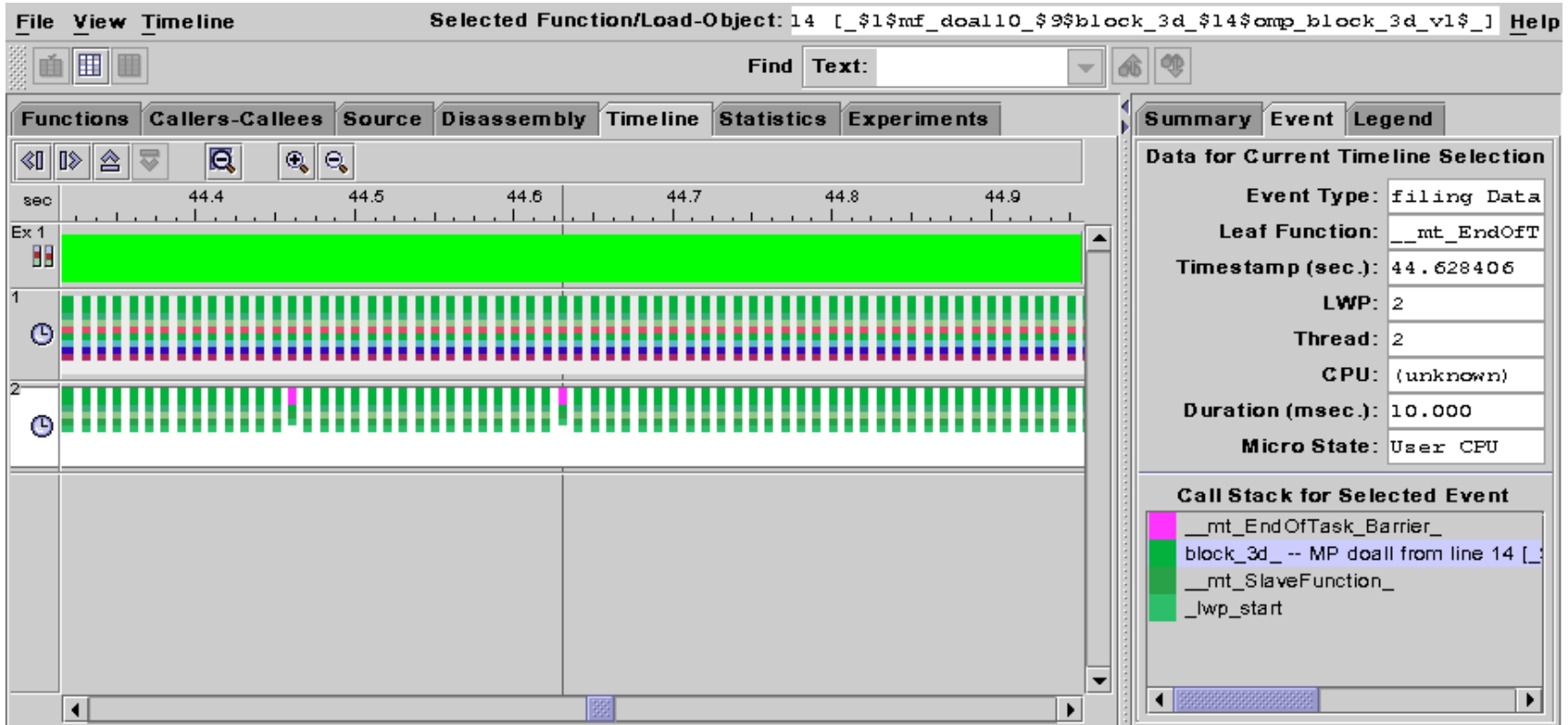
False sharing at work

```
!$omp parallel do default(shared) private(i) &  
!$omp schedule(static)  
  do i = 1, m  
    x(i,j,k) = x(i,j,k-1) + x(i,j-1,k)*scale  
  end do  
!$omp end parallel do
```



False sharing increases as we increase the number of threads

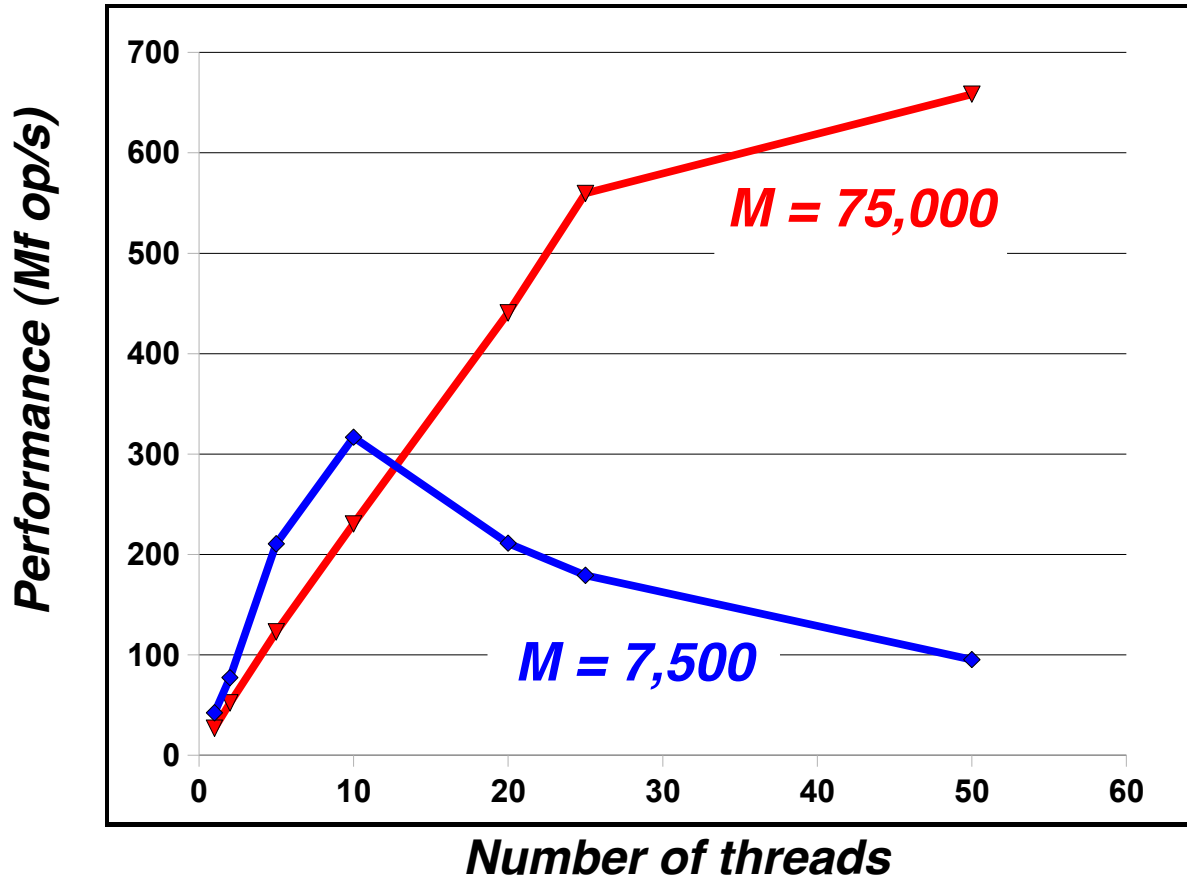
Sanity check: set M=75000*



Only a very few barrier calls now

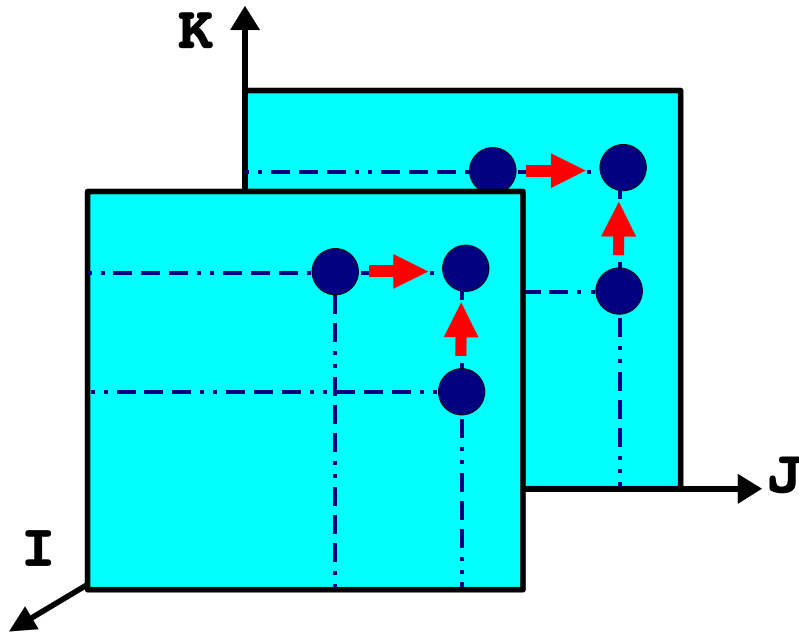
**) Increasing the length of the loop should decrease false sharing*

Performance compared



For a higher value of M , the program scales better

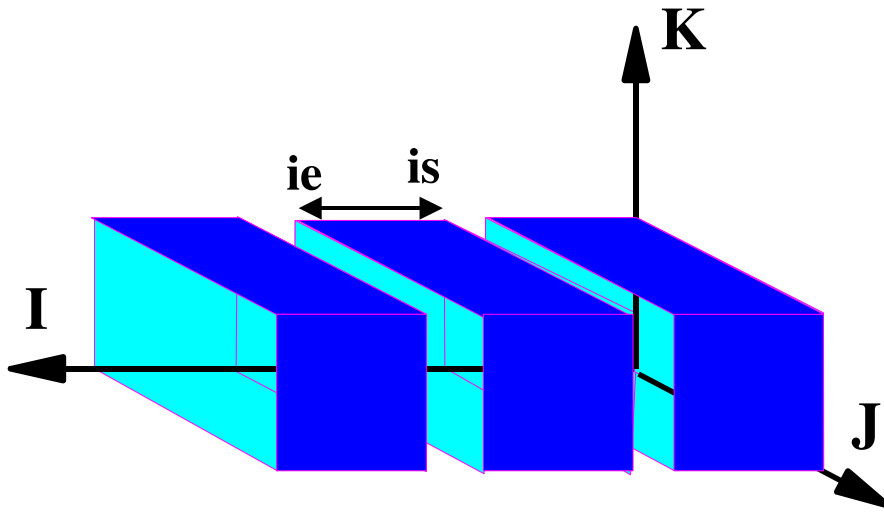
Observation



- *No data dependency on 'I'*
- *Therefore we can split the 3D matrix in larger blocks and process these in parallel*

```
do k = 2, n
  do j = 2, n
    do i = 1, m
      x(i,j,k) = x(i,j,k-1) + x(i,j-1,k)*scale
    end do
  end do
end do
```

The idea



- *We need to distribute the M iterations over the number of processors*
- *We do this by controlling the start (IS) and end (IE) value of the inner loop*
- *Each thread will calculate these values for it's portion of the work*

```
do k = 2, n
  do j = 2, n
    do i = is, ie
      x(i,j,k) = x(i,j,k-1) + x(i,j-1,k)*scale
    end do
  end do
end do
```

The first implementation

```
use omp_lib
.....
nrem    = mod(m,nthreads)
nchunk  = (m-nrem)/nthreads

!$omp parallel default (none) &
!$omp private (P,is,ie)      &
!$omp shared  (nrem,nchunk,m,n,x,scale)

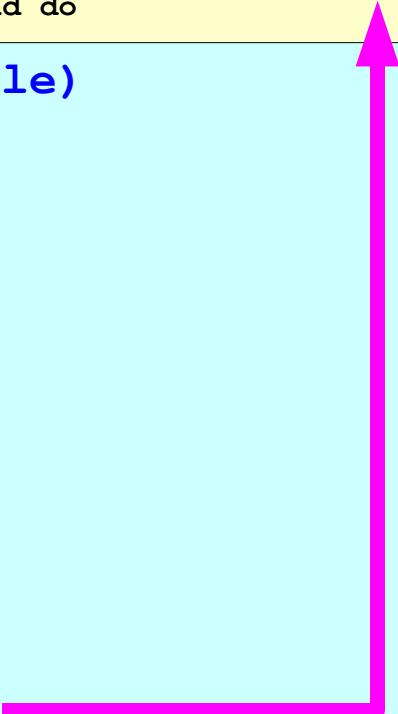
    P = omp_get_thread_num()

    if ( P < nrem ) then
        is = 1 + P*(nchunk + 1)
        ie = is + nchunk
    else
        is = 1 + P*nchunk+ nrem
        ie = is + nchunk - 1
    end if

    call kernel(is,ie,m,n,x,scale)

!$omp end parallel
```

```
subroutine kernel(is,ie,m,n,x,scale)
.....
do k = 2, n
do j = 2, n
do i = is, ie
x(i,j,k)=x(i,j,k-1)+x(i,j-1,k)*scale
end do
end do
end do
```



OpenMP version

```
use omp_lib

implicit none
integer      :: is, ie, m, n
real(kind=8) :: x(m,n,n), scale
integer      :: i, j, k

!$omp parallel default(none) &
!$omp private(i,j,k) shared(m,n,scale,x)
  do k = 2, n
    do j = 2, n
!$omp do schedule(static)
      do i = 1, m
        x(i,j,k) = x(i,j,k-1) + x(i,j-1,k)*scale
      end do
!$omp end do nowait
    end do
  end do
!$omp end parallel
```

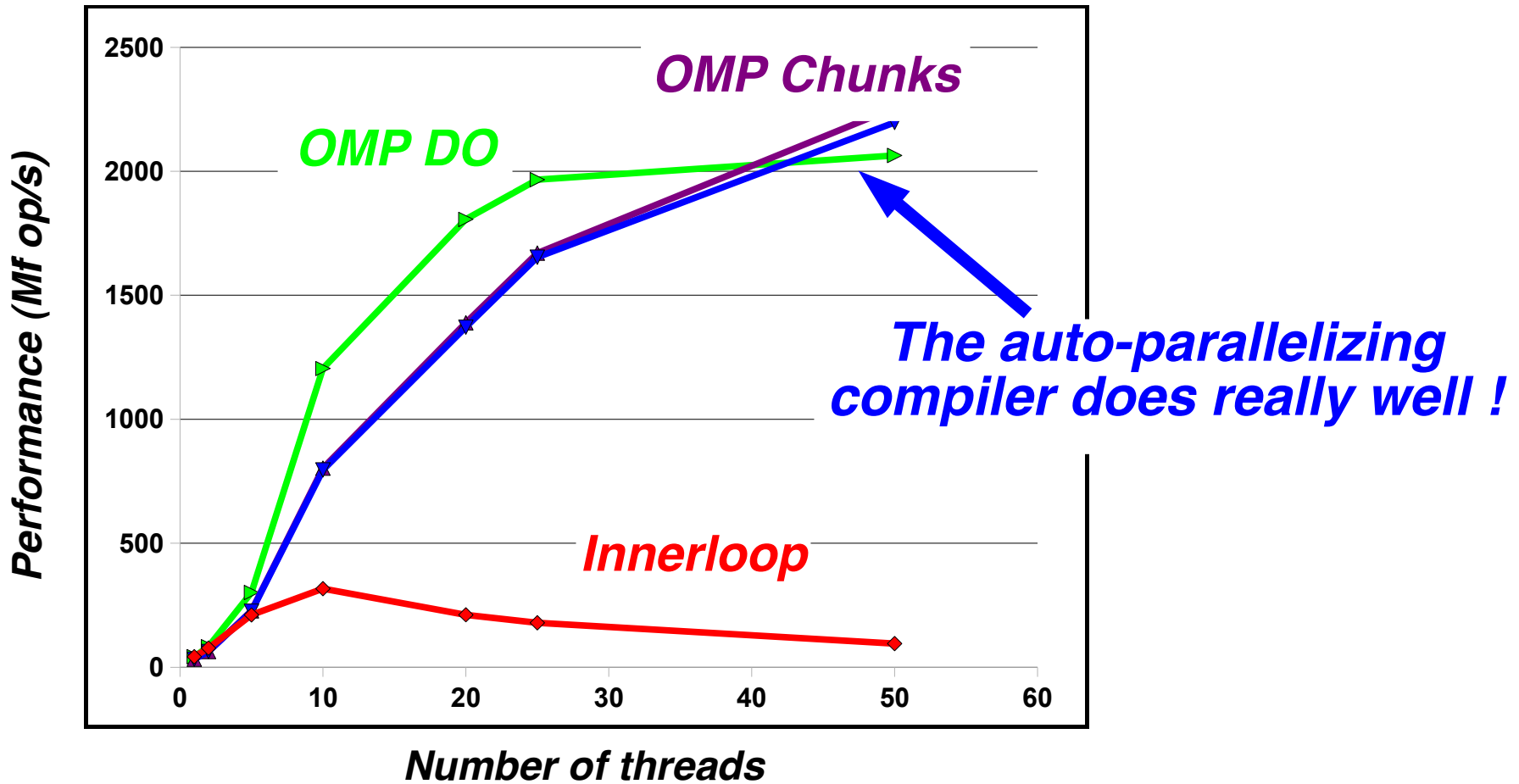
How this works

Thread 0 Executes:		Thread 1 Executes:
k=2 j=2	parallel region	k=2 j=2
do i = 1,m/2 x(i,2,2) = ... end do	work sharing	do i = m/2+1,m x(i,2,2) = ... end do
k=2 j=3	parallel region	k=2 j=3
do i = 1,m/2 x(i,3,2) = ... end do	work sharing	do i = m/2+1,m x(i,3,2) = ... end do
... etc etc ...

Performance

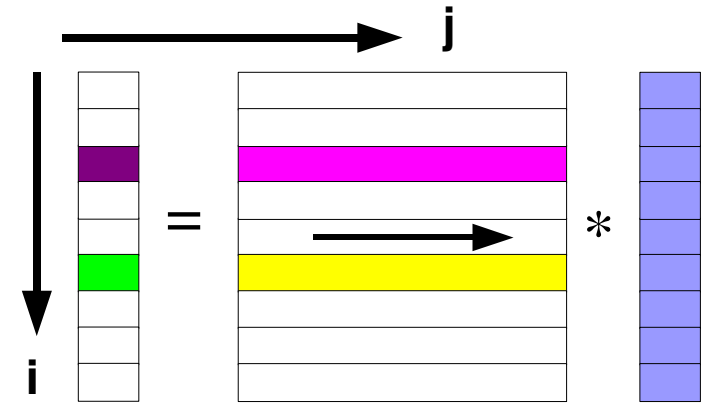
- ***We have set $M=7500$ $N=20$***
 - ***This problem size does not scale at all when we explicitly parallelized the inner loop over 'l'***
- ***We have have tested 4 versions of this program***
 - ***Inner Loop Over 'l' - Our first OpenMP version***
 - ***AutoPar - The automatically parallelized version of 'kernel'***
 - ***OMP_Chunks - The manually parallelized version with our explicit calculation of the chunks***
 - ***OMP_DO - The version with the OpenMP parallel region and work-sharing DO***

Performance

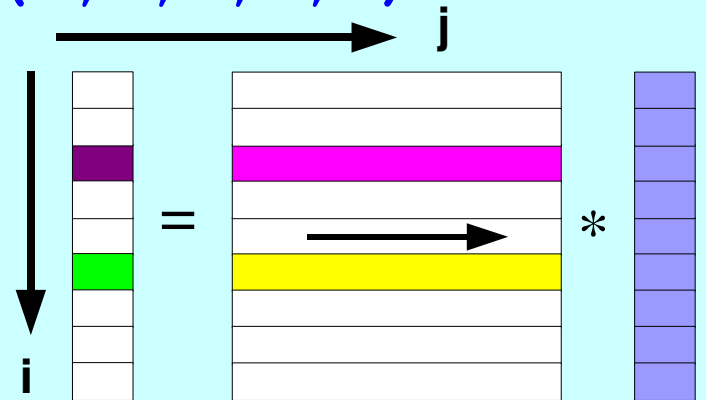


Matrix times vector

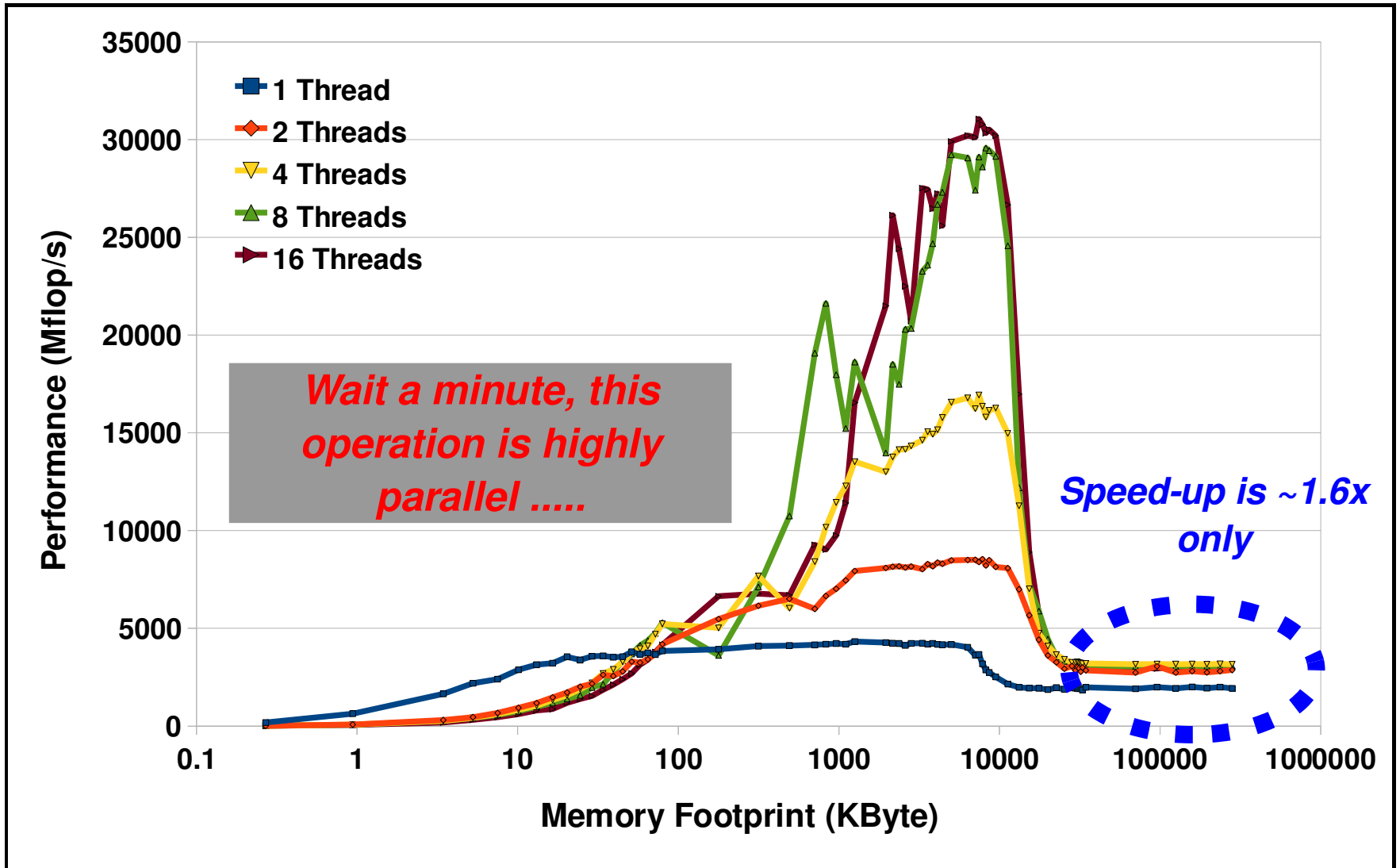
```
for (i=0; i<m; i++)
{
    a[i] = 0.0;
    for (j=0; j<n; j++)
        a[i] += b[i][j]*c[j];
}
```



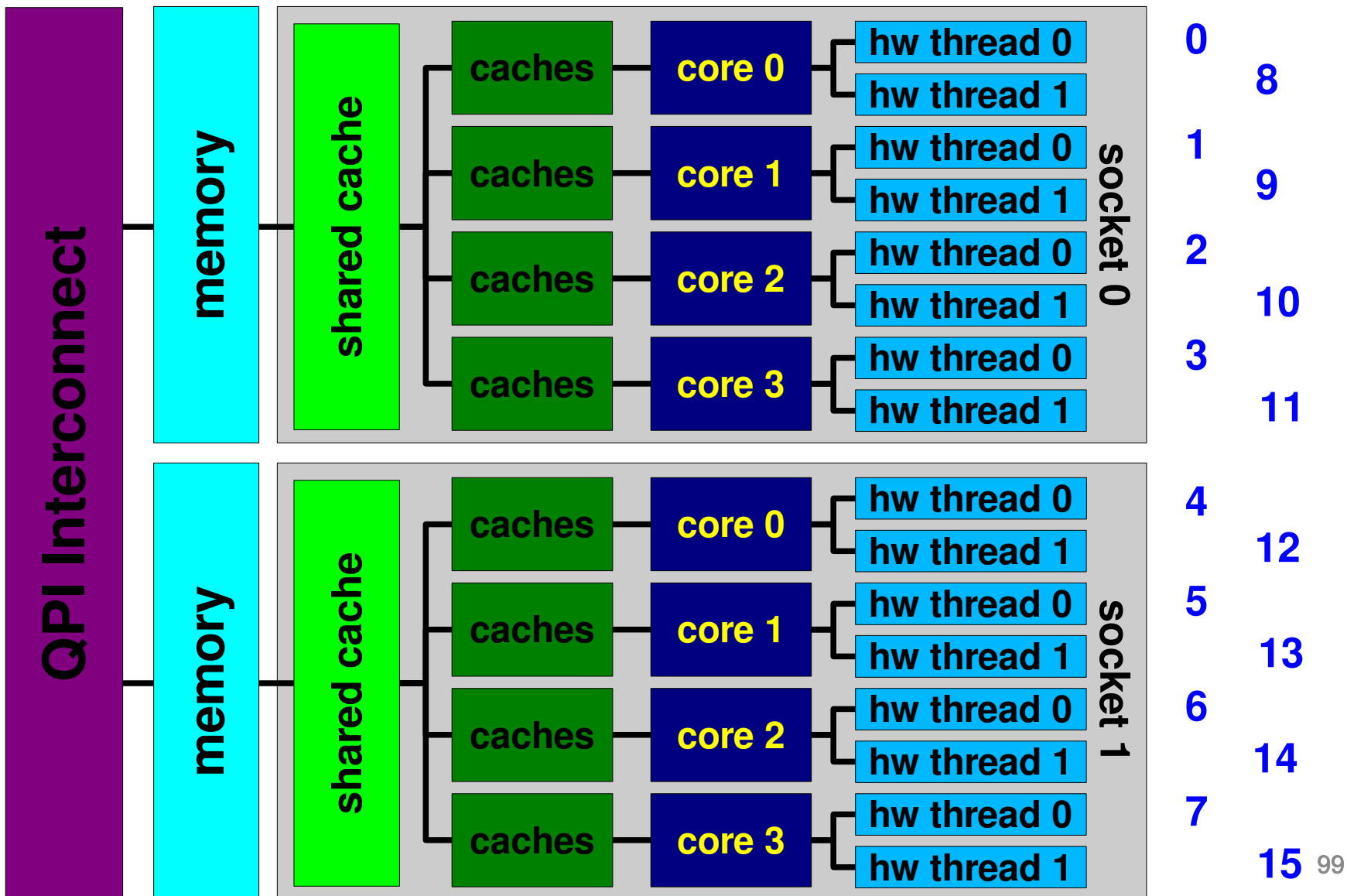
```
#pragma omp parallel for default(none) \
    private(i,j) shared(m,n,a,b,c)
for (i=0; i<m; i++)
{
    a[i] = 0.0;
    for (j=0; j<n; j++)
        a[i] += b[i][j]*c[j];
}
```



Performance – 2-socket Nehalem

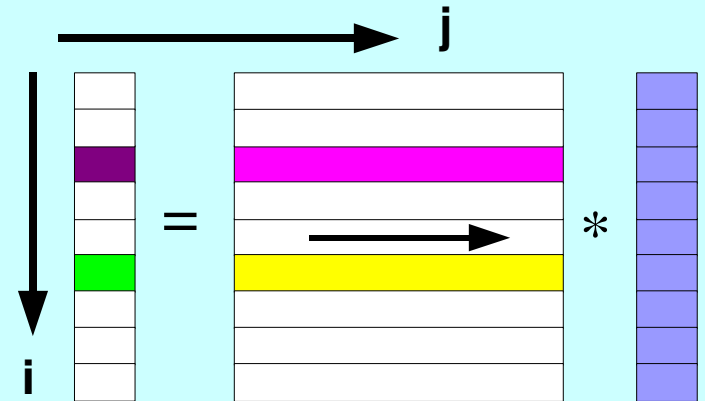


2-socket Nehalem

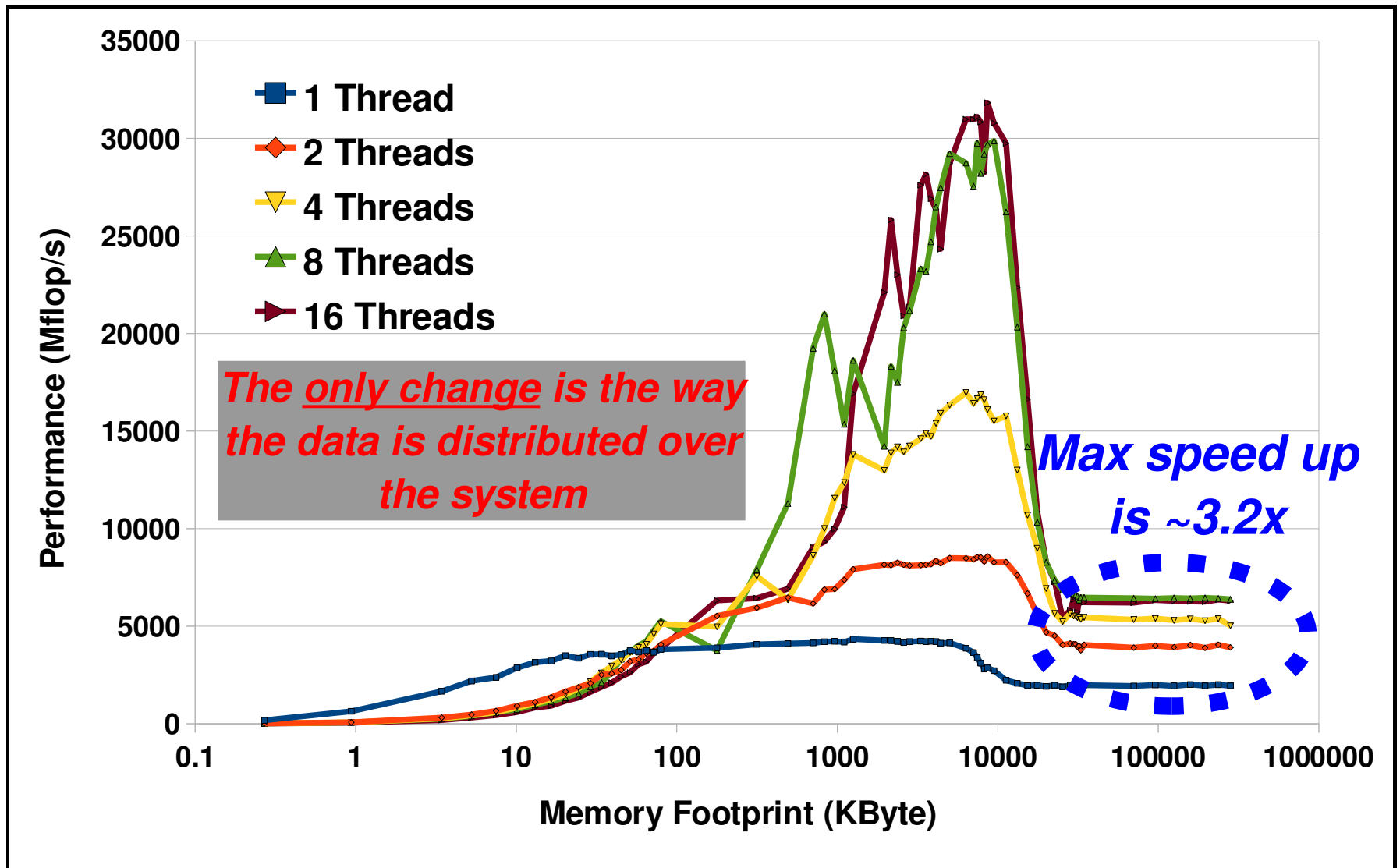


Data initialization

```
#pragma omp parallel default(none) \  
    shared(m,n,a,b,c) private(i,j)  
{  
#pragma omp for  
    for (j=0; j<n; j++)  
        c[j] = 1.0;  
  
#pragma omp for  
    for (i=0; i<m; i++)  
    {  
        a[i] = -1957.0;  
        for (j=0; j<n; j++)  
            b[i][j] = i;  
    } /*-- End of omp for --*/  
  
} /*-- End of parallel region --*/
```



Exploit First Touch



Reference Material on OpenMP

- OpenMP Homepage www.openmp.org:
 - The primary source of information about OpenMP and its development.
- OpenMP User's Group (cOMPunity) Homepage
 - www.compunity.org:
- Books:
 - Using OpenMP, Barbara Chapman, Gabriele Jost, Ruud Van Der Pas, Cambridge, MA : The MIT Press 2007, ISBN: 978-0-262-53302-7
 - Parallel programming in OpenMP, Chandra, Rohit, San Francisco, Calif. : Morgan Kaufmann ; London : Harcourt, 2000, ISBN: 1558606718

OpenMP™

<http://www.openmp.org>

OMP
community

<http://www.compunity.org>