OpenMP Tutorial

CScADs Workshop 2009

Lei Huang

Barbara Chapman

University of Houston

Acknowledgements: Tim Mattson (Intel), Rud van der Paas (Sun), OpenMP ARB

Talk Outline

- Parallel Computing, threads, and OpenMP
- The important constructs of OpenMP
- OpenMP Practices for optimizations
- Hybrid MPI/OpenMP Applications
- Case Studies and Examples


File Edit View History Bookmarks Tools Help
http://openmp.org/wp/
Google
Getting Started
Latest Headlines
Developer Guide
OpenMP.org
http://www.openmp.org
OpenMP
THE OPENMP API SPECIFICATION FOR PARALLEL PROGRAMMING
RSS
What's Here:
» API Specs
»About OpenMP.org
»OpenMP Compilers
»OpenMP Resources
»OpenMP Forum
Input Register
Alert the OpenMP.org webmaster about new products or updates and we'll post it here.
»webmaster@openmp.org
Search OpenMP.org
Google™ Custom Search
Search
Archives
June 2008
May 2008
April 2008
Admin
Log in
Copyright © 2008 OpenMP Architecture Review Board. All rights reserved.
OpenMP News
»Christian's First Experiments with Tasking in OpenMP 3.0
From Christian Terboven's blog:
OpenMP 3.0 is out, maybe a bit later than we hoped for, but I think that we got a solid standard document. At IWOMP 2008 a couple of weeks ago, there was an OpenMP tutorial which included a talk by Alex Duran (from UPC in Barcelona, Spain) on what is new in OpenMP 3.0 - which is really worth a look! My talk was on some OpenMP application experiences, including a case study on Windows, and I really think that many of our codes can profit from Tasks. Motivated by Alex' talk I tried the updated Nanos compiler and prepared a couple of examples for my lectures on Parallel Programming in Maastricht and Aachen. In this post I am walking through the simplest one: Computing the Fibonacci number in parallel.
Read more...
Posted on June 6, 2008
»New Forum Created
The OpenMP 3.0 API Specifications forum is now open for discussing the specs document itself.
Posted on May 31, 2008
»New Links
New links and information have been added to the OpenMP Compilers and the OpenMP Resources pages.
Posted on May 23, 2008
»Recent Forum Posts
strange behavior of C function strcmp() With OPENMP
virtual destructor not called with first private clause
OpenMP.org
The OpenMP Application Program Interface (API) supports multi-platform shared-memory parallel programming in C/C++ and Fortran. OpenMP is a portable, scalable model with a simple and flexible interface for developing parallel applications on platforms from the desktop to the supercomputer.
»Read about OpenMP
Get It
»OpenMP specs
Use It
»OpenMP Compilers
Learn It
Using OpenMP
Done

OpenMP Overview:

OpenMP: An API for Writing Multithreaded Applications

- **A set of compiler directives and library routines for parallel application programmers**
- **Greatly simplifies writing multi-threaded (MT) programs in Fortran, C and C++**
- **Standardizes last 20 years of SMP practice**

- ***Version 3.0 has been released May 2008***

When to consider OpenMP?

- The compiler may not be able to do the parallelization in the way you like to see it:
  - It can not find the parallelism
    - The data dependence analysis is not able to determine whether it is safe to parallelize or not
  - The granularity is not high enough
    - The compiler lacks information to parallelize at the highest possible level
- This is when explicit parallelization through OpenMP directives comes into the picture

Advantages of OpenMP

- *Good performance and scalability*
  - *If you do it right ....*
- *De-facto and mature standard*
- *An OpenMP program is portable*
  - *Supported by a large number of compilers*
- *Requires moderate programming effort*
- *Allows the program to be parallelized incrementally*

How Does OpenMP Enable Us to Exploit Threads?

- OpenMP provides thread programming model at a "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)

The OpenMP API

provides the means to:

- create and destroy threads
- assign / distribute work to threads
- specify which data is shared and which is private to a thread
- coordinate actions of threads on shared data

OpenMP Overview:
How do threads interact?

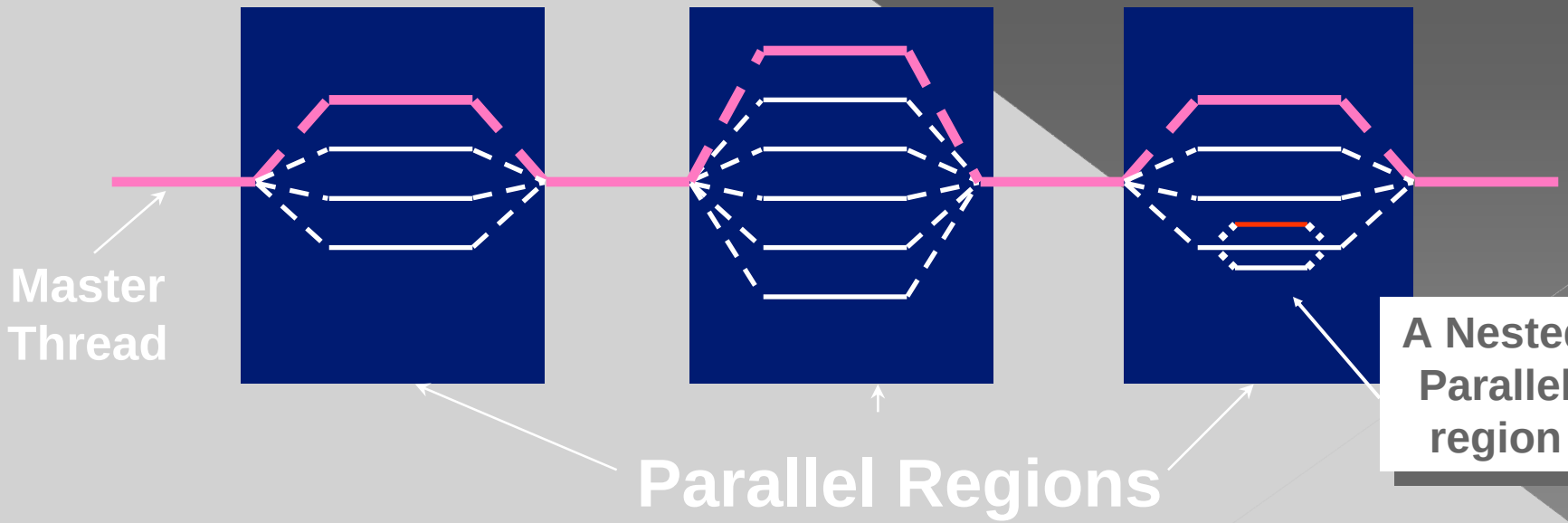
- OpenMP is a shared memory model.
  - Threads communicate by sharing variables.
- Synchronization protects data conflicts.
  - Synchronization is expensive.
    - Change how data is accessed to minimize the need for synchronization.

OpenMP Programming Model:

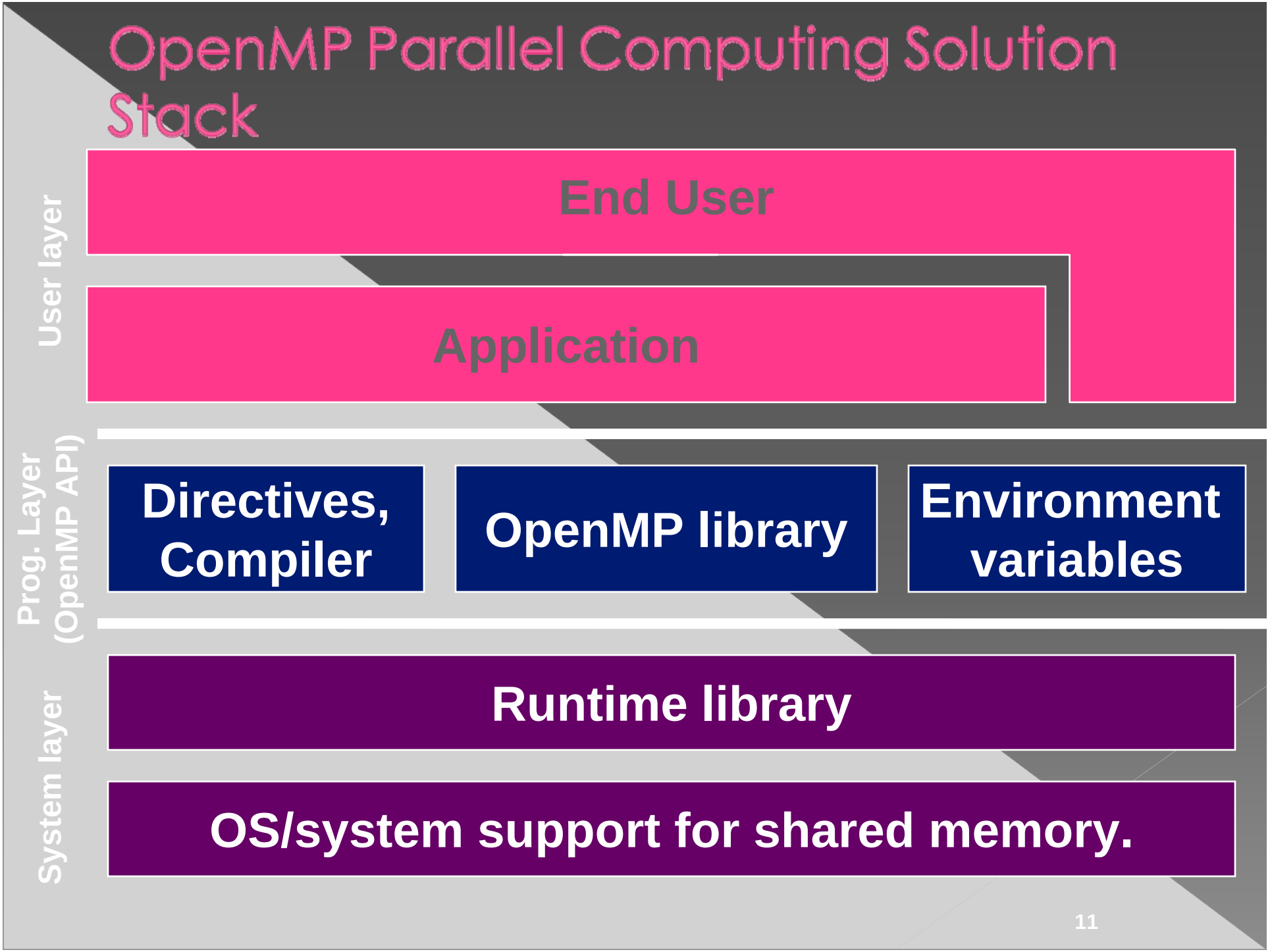
Fork-Join Parallelism:

Master thread spawns a team of threads as needed.

Parallelism is added incrementally until desired performance is achieved: i.e. the sequential program evolves into a parallel program.



OpenMP Parallel Computing Solution Stack



OpenMP Syntax

- Most of the constructs in OpenMP are compiler directives or 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 Parallel Regions:

Structured Block Boundaries

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

```
#pragma omp parallel
{
    id = omp_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];
  }
```

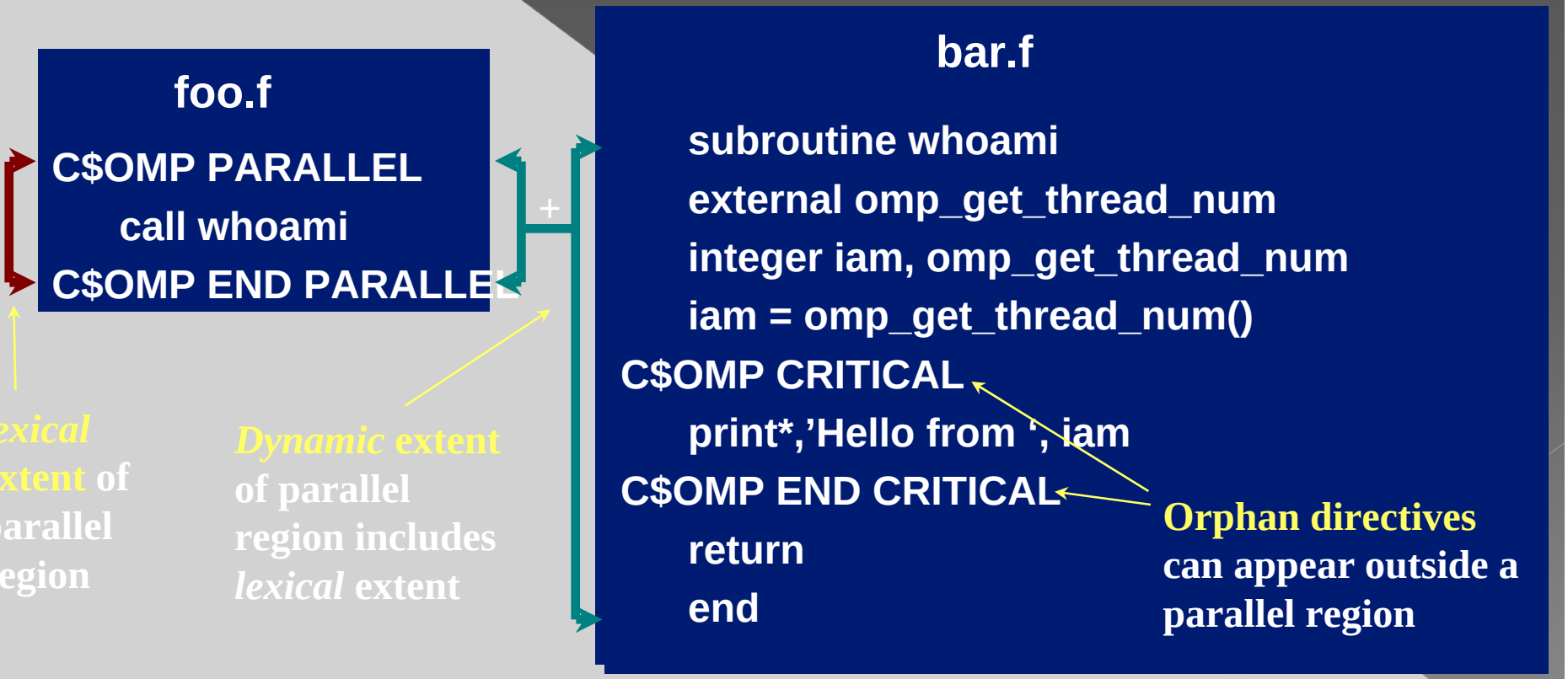
In Fortran: a block is a single statement or a group of statements between directive/end-directive pairs.

```
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 constructs:

OpenMP constructs can span multiple source files.



OpenMP: Work-Sharing Constructs

- The "for" Work-Sharing construct splits up loop iterations among the threads in a team

```
#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.

Work Sharing Constructs
A motivating example

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 work-sharing for-construct

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

OpenMP For/Do schedule clause

Schedule Clause	When To Use
STATIC	Pre-determined and predictable by the programmer
DYNAMIC	Unpredictable, highly variable work per iteration
GUIDED	Special case of dynamic to reduce scheduling overhead

Least work at runtime : scheduling done at compile-time

Most work at runtime : complex scheduling logic used at run-time

OpenMP For/Do construct:
The schedule clause

- The schedule clause affects how loop iterations are mapped onto threads
  - schedule(static [,chunk])

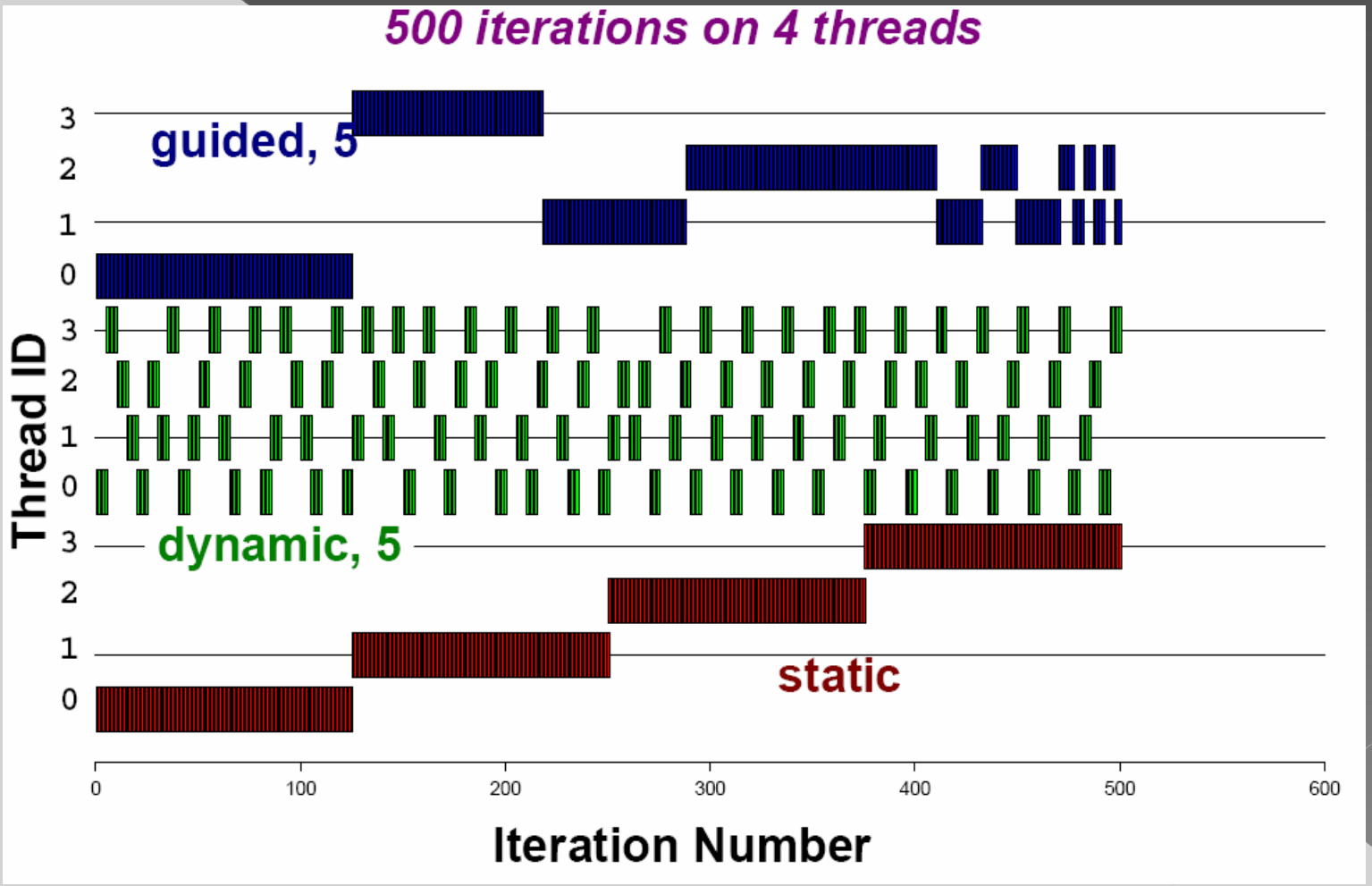
    Deal-out blocks of iterations of size "chunk" to each thread.
  - schedule(dynamic[,chunk])

    Each thread grabs "chunk" iterations off a queue until all iterations have been handled.
  - schedule(guided[,chunk])

    Threads dynamically grab blocks of iterations. The size of the block starts large and shrinks down to size "chunk" as the calculation proceeds.
  - schedule(runtime)

    Schedule and chunk size taken from the OMP_SCHEDULE environment variable.

Experiment


500 iterations on 4 threads
guided, 5
dynamic, 5
static
Thread ID
3
2
1
0
3
2
1
0
3
2
1
0
0
100
200
300
400
500
600
Iteration Number

Additional Schedule in OpenMP 3.0

- Auto
  - ***The compiler (or runtime system) decides what is best to use***
  - ***Choice could be implementation dependent***

Loop Collapsing (in OpenMP 3.0)

```
!$omp parallel do collapse(2)
do i=1,n
     do j=1,n
          .....
     end do
end do
```

OpenMP Sections
Work-Sharing Constructs

- The *Sections* work-sharing 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.

OpenMP Master
Work-Sharing Constructs

- The master construct 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
Work-Sharing Constructs

- The single construct 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 Task Construct (in 3.0)

- The **task construct defines an explicit task.**

```
#pragma omp task [clause[[,] clause] ...]
{
        do_a_task();
}
```

where *clause is one of the following:*

- **if(*scalar-expression*)**
- **untied**
- **default(shared | none)**
- **private(*list*)**
- **firstprivate(*list*)**
- **shared(*list*)**

When are Tasks Complete?

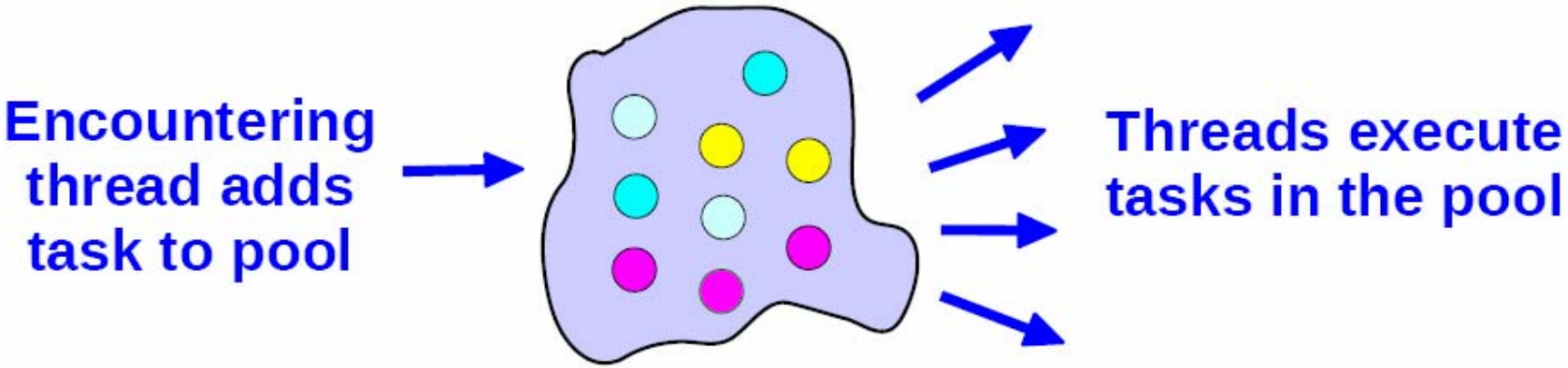
- *At implicit thread barrier*
- *At explicit thread barrier*
  - *C/C++: #pragma omp barrier*
  - *Fortran: !$omp barrier*
- *At task barrier*
  - *C/C++: #pragma omp taskwait*
  - *Fortran: !$omp taskwait*

Task Example: Linked list traverse

```
void increment_list_items(node * head)
{
#pragma omp parallel
  {
#pragma omp single
    {
      node * p = head;
      while (p) {
#pragma omp task       // p is firstprivate by default
                  process(p);
        p = p->next;
      }
    }
  }
}
```

Hard to do before OpenMP 3.0

The Tasking Example

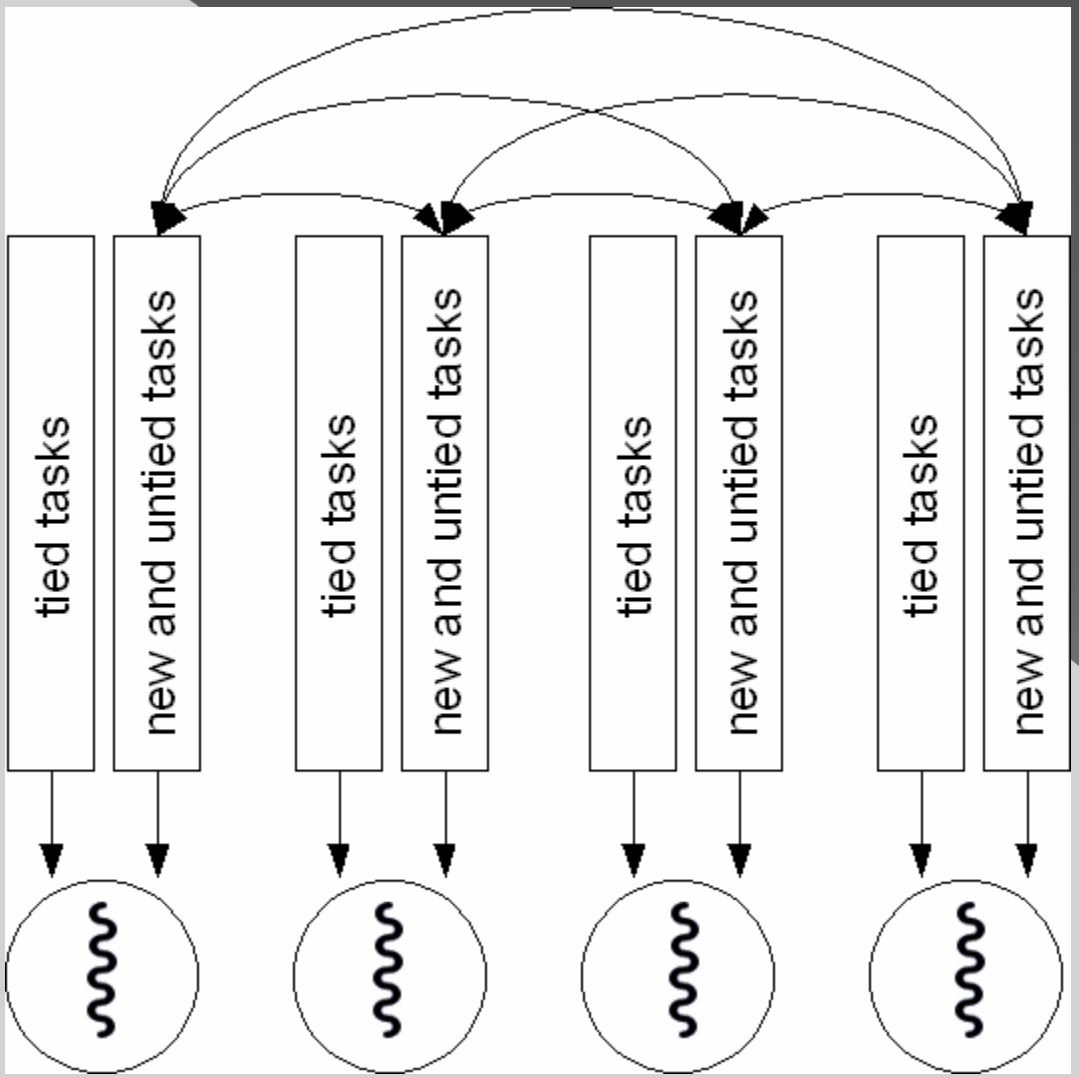


Developer specifies tasks in application
Run-time system executes tasks

Task Scheduling

- Tasks are tied by default
  - Tied tasks are executed always by the same thread
  - Tied tasks have scheduling restrictions
    - Deterministic scheduling points (creation, synchronization, ...)
    - Another constraint to avoid deadlock problems
  - Tied tasks may run into performance problems
- Programmer can use untied clause to lift all restrictions
  - Note: Mix very carefully with threadprivate, critical and thread-ids

Task Scheduling



Implemented in OpenUH

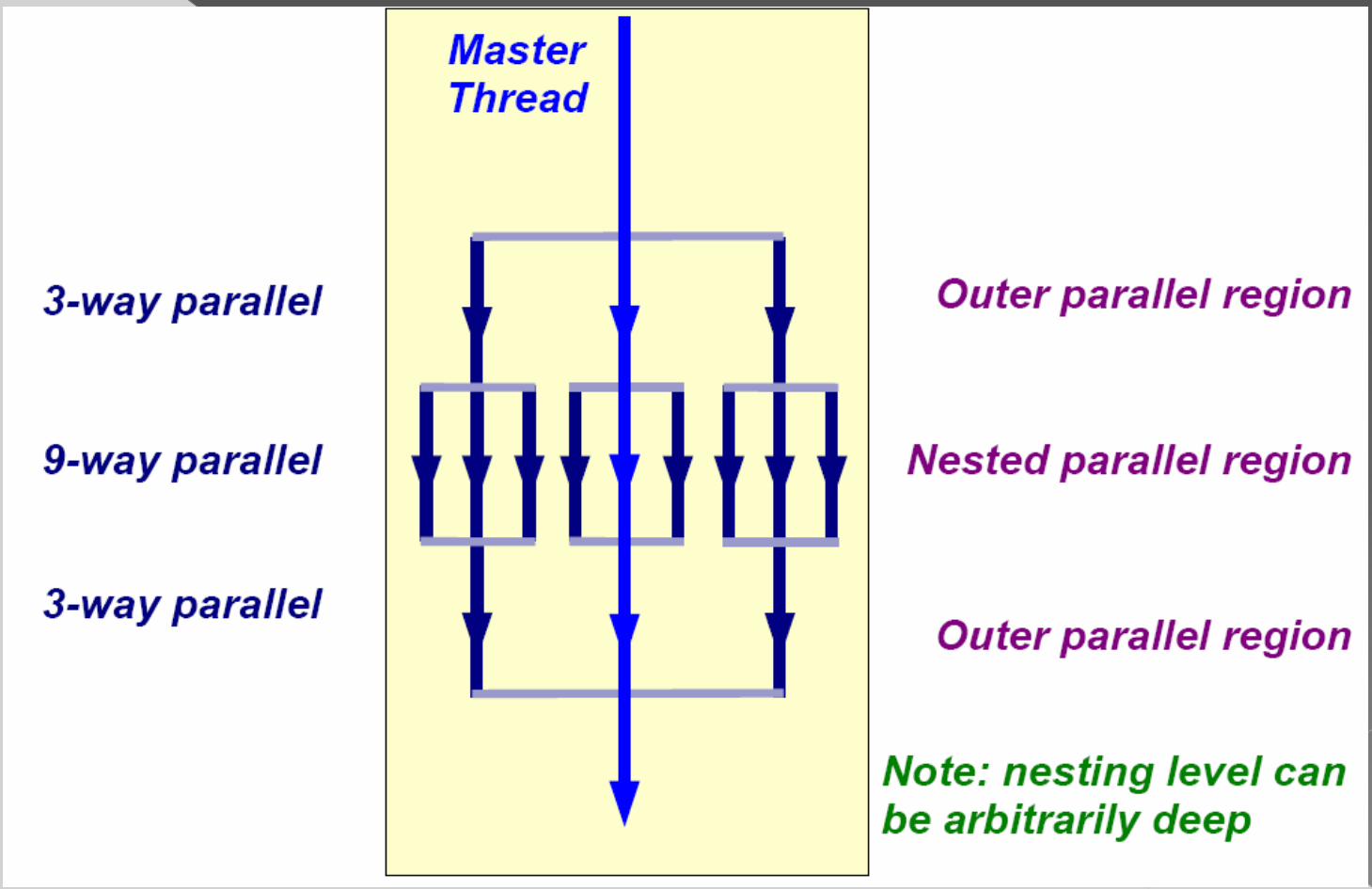
The IF clause

- If the expression of a IF clause evaluates to false
  - The encountering task is suspended
  - The generated task is executed immediately
    - with its own data environment
    - different task with respect to synchronization
  - The parent task resumes when the task finishes
  - Allows implementations to optimize task creation

Task Granularity and Control the number of outstanding tasks

- Granularity is a key performance factor
  - Tasks tend to be fine-grained
  - Try to "group" tasks together
  - Use if clause or manual transformations

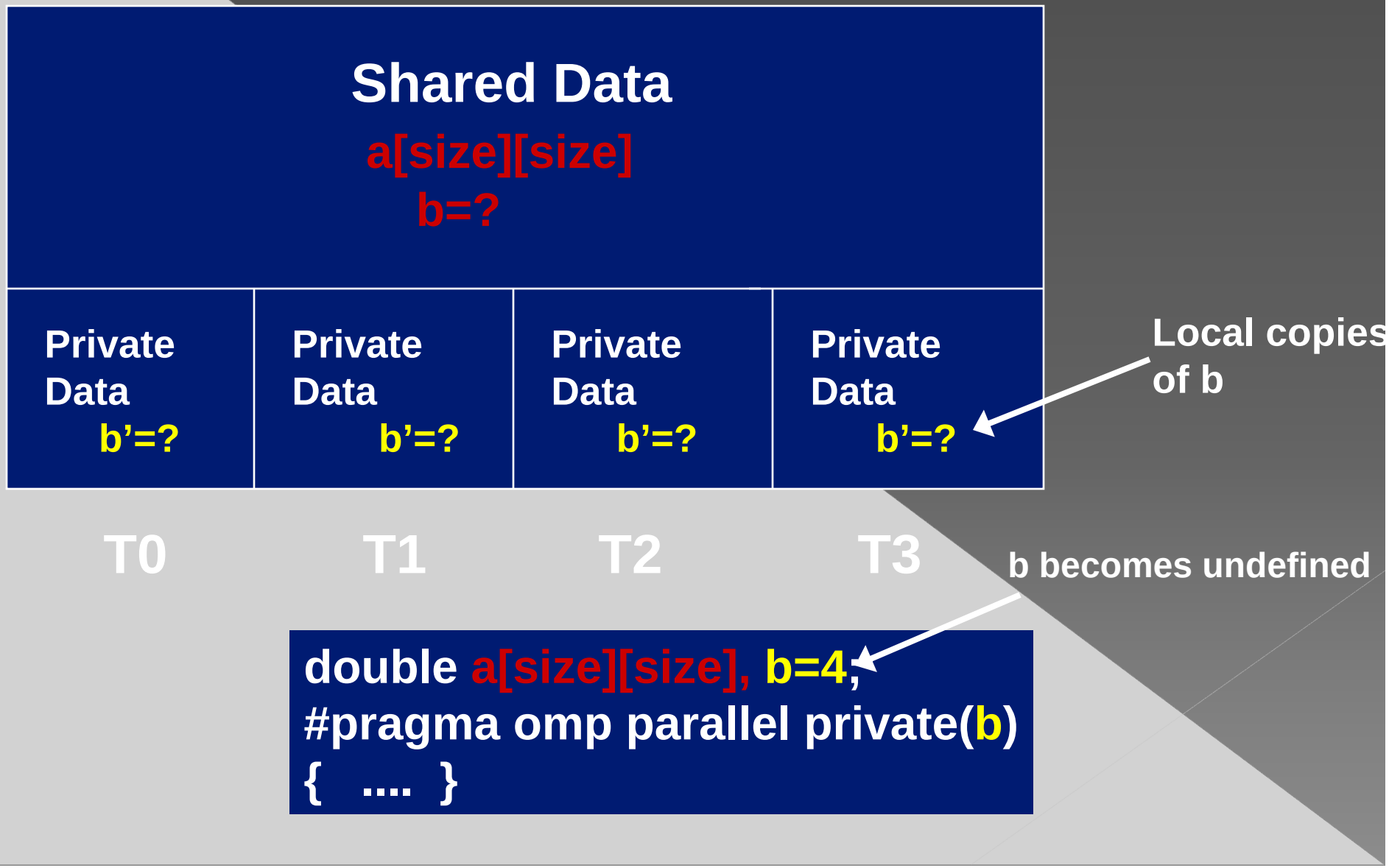
Nested Parallelism



Data Environment:
Default storage attributes

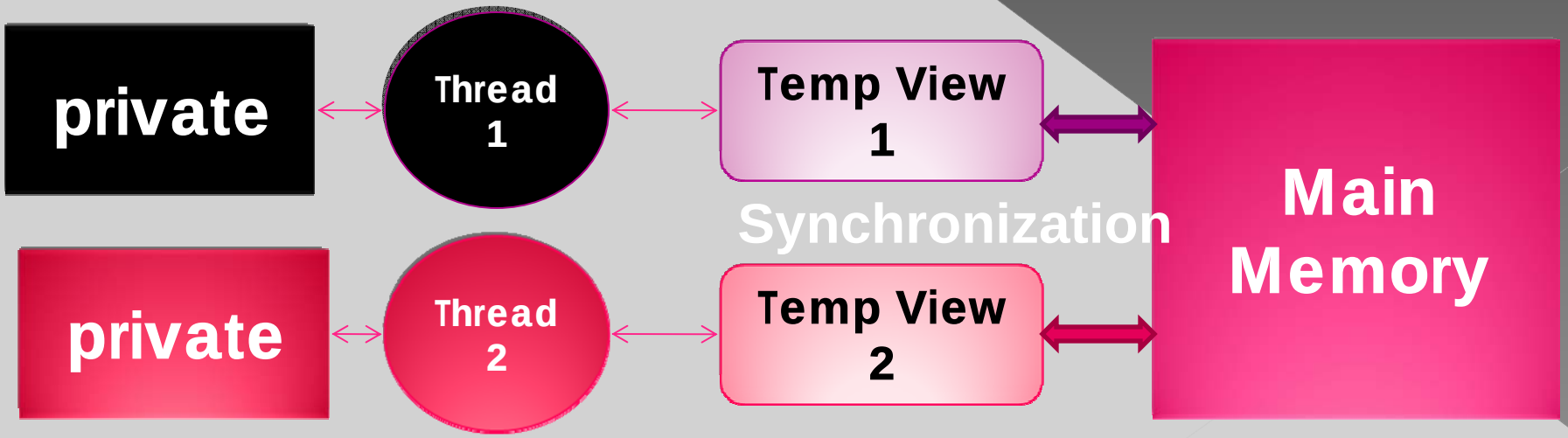
- Shared Memory programming model:
  - Most variables are shared by default
- Global variables are SHARED among threads
  - Fortran: COMMON blocks, SAVE variables, MODULE variables
  - C: File scope variables, static
- But not everything is shared...
  - Stack variables in sub-programs called from parallel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

OpenMP Data Enviroment



OpenMP Memory Model

- Relaxed-consistency, shared-memory model
- All threads have access to a "main memory" and its own "temporary" view of memory for shared data
  - Temporary view can be any intervening structure between threads and main memory, e.g. cache, registers, or other local storage
  - Synchronization between temporary view and main memory done through hardware, or specified by user



OpenMP Memory Model

- A variable reference can be shared or private with respect to a parallel region
- Key problem: When should the temporary view of a shared variable synchronize with main memory? Range of possibilities:
  - Always synchronized (i.e. no temp view)
  - Based on H/W coherence scheme
  - **Only synchronize when FLUSH is explicitly or implicitly specified in OpenMP**

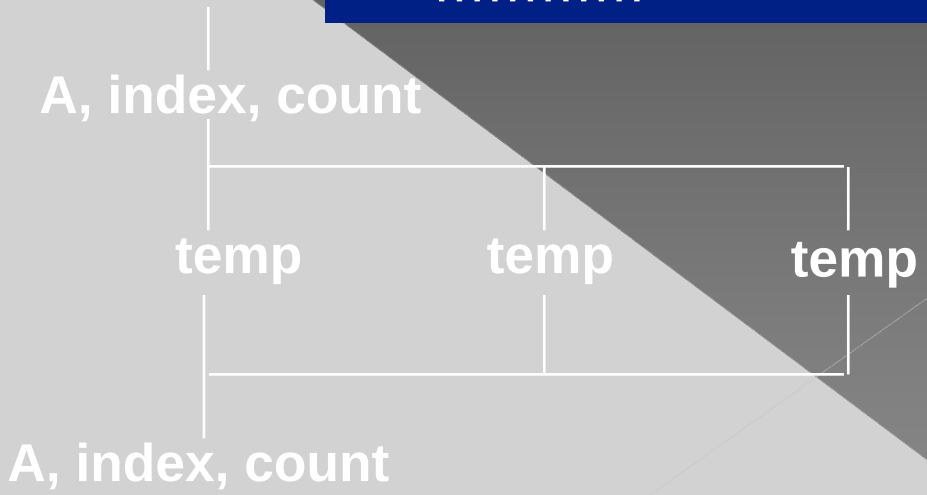
OpenMP Data Enviroment
across procedures

```
    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

- One can 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)

All the clauses on this page only apply to the *lexical extent* of the OpenMP construct.

All data clauses apply to parallel regions and worksharing constructs except "shared" which only applies to parallel regions.

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

```
      program wrong
        IS = 0
C$OMP PARALLEL DO PRIVATE(IS)
        DO J=1,1000
           IS = IS + J
        END 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.

```
      program almost_right
      IS = 0
C$OMP PARALLEL DO FIRSTPRIVATE(IS)
      DO J=1,1000
            IS = IS + J
1000  CONTINUE
      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 a global variable.

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

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.

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

OpenMP:
Reduction operands/initial-values

- A range of associative operands can be 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
  - 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.

OpenMP 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$OMP PARALLEL COPYIN(A)
 … Now each thread sees threadprivate array A initialied
 … to the global value set in the subroutine init_data()
C$OMP END PARALLEL
....
C$OMP PARALLEL
... Values of threadprivate are persistent across parallel regions
C$OMP END PARALLEL

end
```

Threadprivate Example for static pointers

```
Static int *tmp;
#pragma omp threadprivate(tmp)
#pragma omp parallel
{
   tmp = (int *)malloc(size);  /* tmp is a
thread private pointer, each thread has its
own memory allocation */
#pragma omp for
   for(i=0;i<N;i++)
        tmp[i]=…

}
```

OpenMP: Synchronization

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

OpenMP: Synchronization

- 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
```

OpenMP: Synchronization

- 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 (the update of X in the following example)

```
C$OMP PARALLEL PRIVATE(B)
        B =  DOIT(I)
tmp = big_ugly();

 C$OMP ATOMIC
        X = X + temp

C$OMP END PARALLEL
```

OpenMP: Synchronization

- 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

OpenMP: Synchronization

- 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 Synchronizations: Taskwait

- The **taskwait** construct specifies a wait on the completion of child tasks generated since the beginning of the current task.

#pragma omp taskwait
newline

Task Example: tree recursive traverse

```
void traverse( struct node *p ) {
    if (p->left)
#pragma omp task      // p is firstprivate by
default
        traverse(p->left);
    if (p->right)
#pragma omp task      // p is firstprivate by
default
        traverse(p->right);
    process(p);
}
```

- Note: no specific traverse order guaranteed

Task Example: postorder tree traverse

```
void traverse( struct node *p ) {
    if (p->left)
#pragma omp task      // p is firstprivate by
default
        traverse(p->left);
    if (p->right)
#pragma omp task      // p is firstprivate by
default
        traverse(p->right);
#pragma omp taskwait
    process(p);
}
```

- Note: post-order traverse guaranteed

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.

OpenMP:
A flush example

This example shows how flush is used to implement 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: OpenMP's flush is analogous to a fence in other shared memory API's.

OpenMP: Lock routines

- Simple Lock routines:
  - A simple lock is available if it is unset.
    - omp_init_lock(), omp_set_lock(), omp_unset_lock(), omp_test_lock(), omp_destroy_lock()
- Nested Locks
  - A nested lock is 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()

Note: a thread always accesses the most recent copy of the lock, so you don't need to use a flush on the lock variable.

OpenMP: Simple 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:

- Runtime environment 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]"

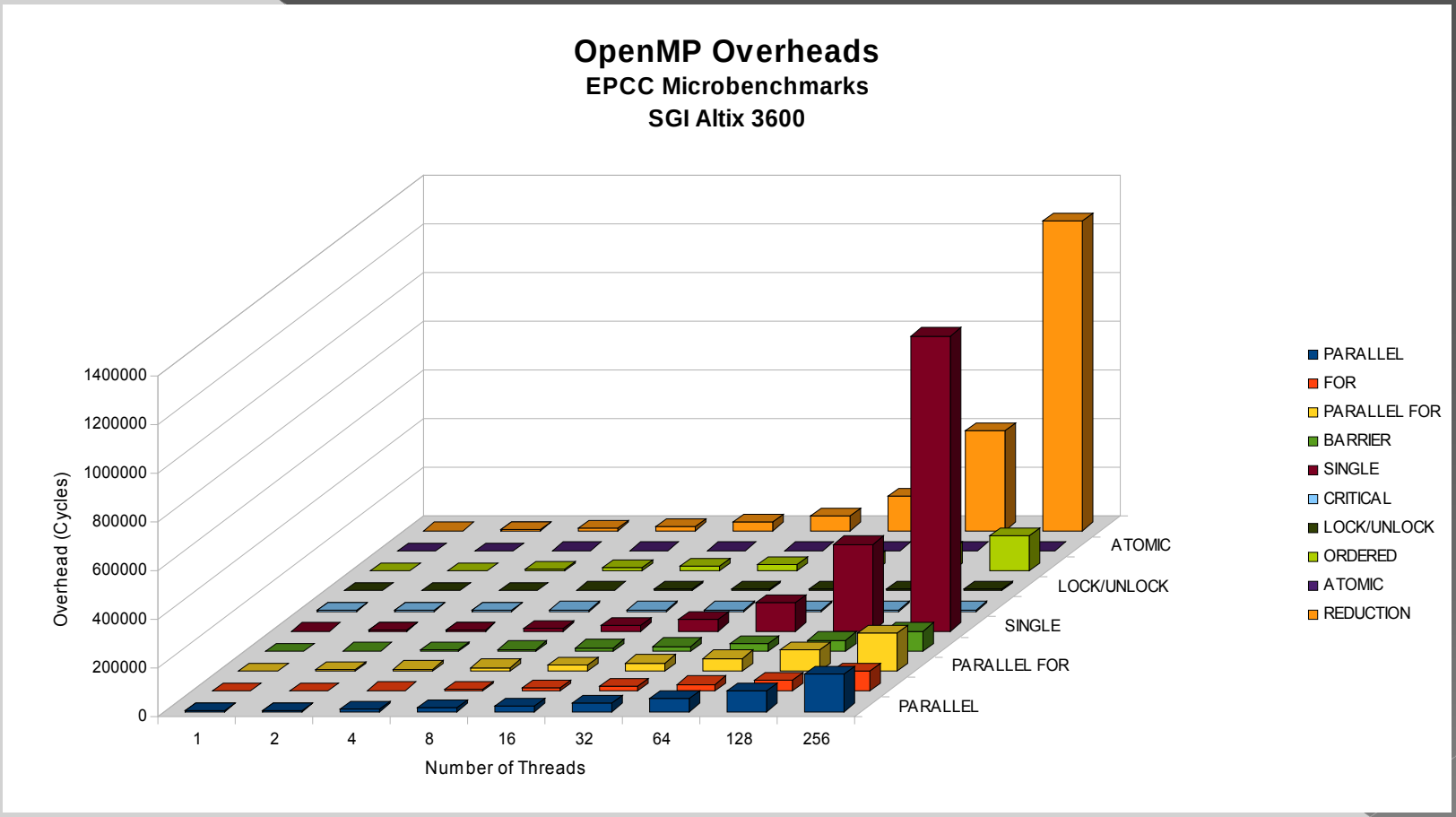
OpenMP Performance

- Relative easy of OpenMP is a mixed blessing
- We can quickly write a correct OpenMP but without the desired level of performance.
- There are certain "best practices" to avoid common performance problems.
- Extra work needed for 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
- Overheads of runtime library routines
  - Some are called frequently
- Load balance
- Cache utilization and false sharing
- Large parallel regions help reduce overheads, enable better cache usage and standard optimizations

Overheads of OpenMP Directives



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++)
}
```

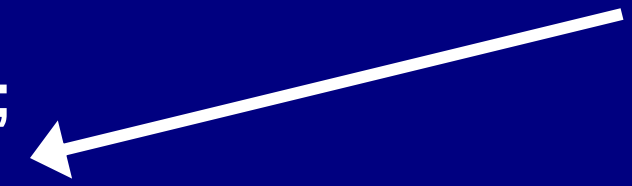
OpenMP: best practices

```
#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 the 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 */}
}
```

Avoid parallel region overheads

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++)
       { …….}
```

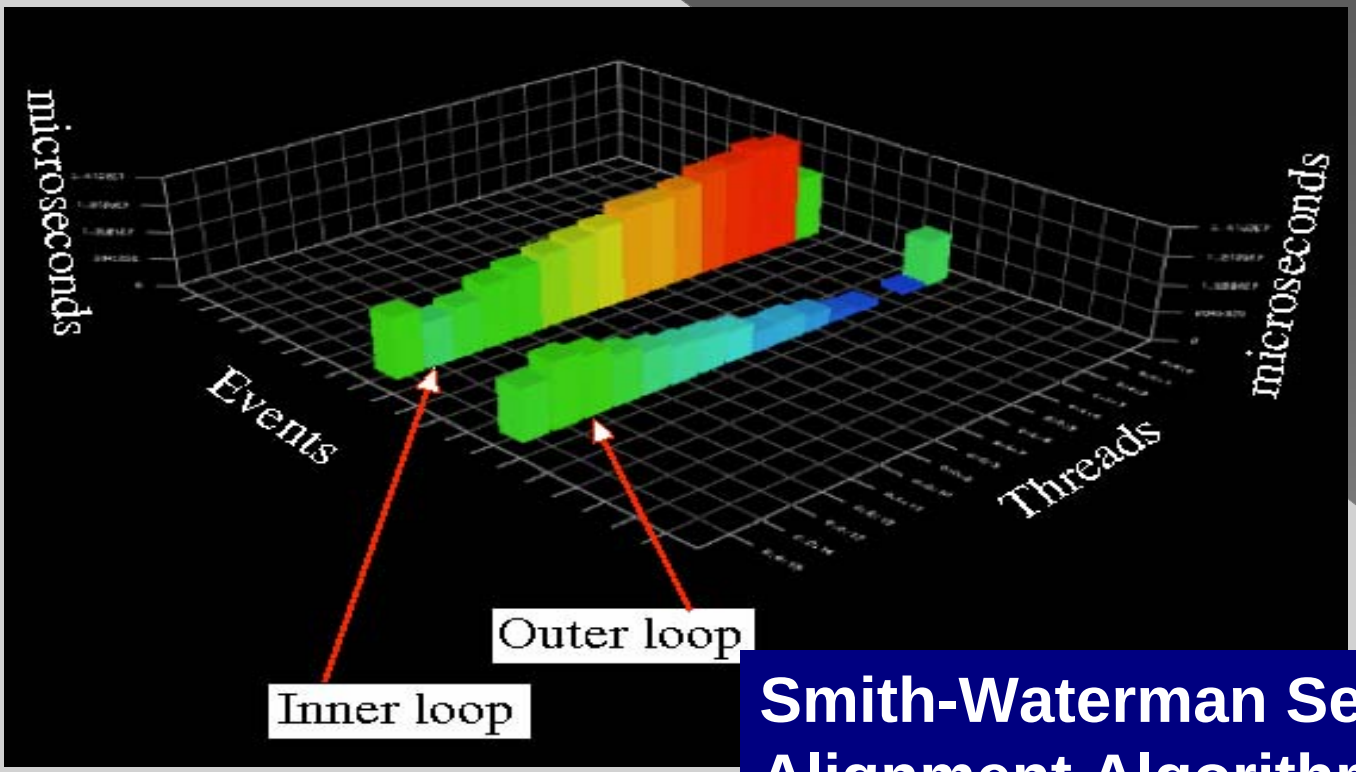
```
#pragma omp parallel private(i,j,k)
{
 for (i=0; i<n; i++)
   for (j=0; j<n; j++)
#pragma omp for
   for (k=0; k<n; k++)
        { …….}

}
```

Avoid parallel region overheads

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:

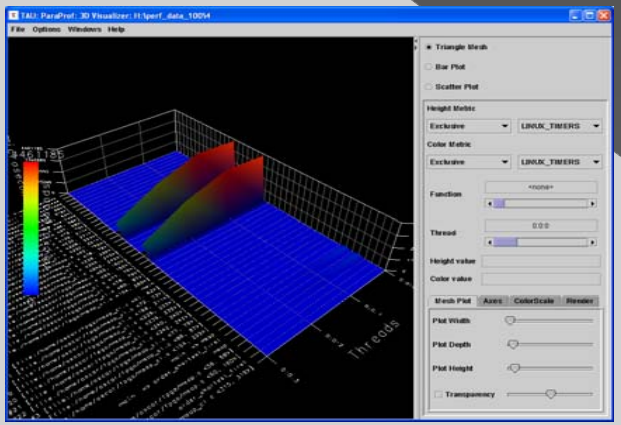
```
#pragma omp for
 for(...)
  for(...)
   for(...)
    for(...)
        { /* compute alignments */ }
#pragma omp critical
   {. /* compute scores */ }
```

Default scheduler is static even.

Not good for load imbalance.

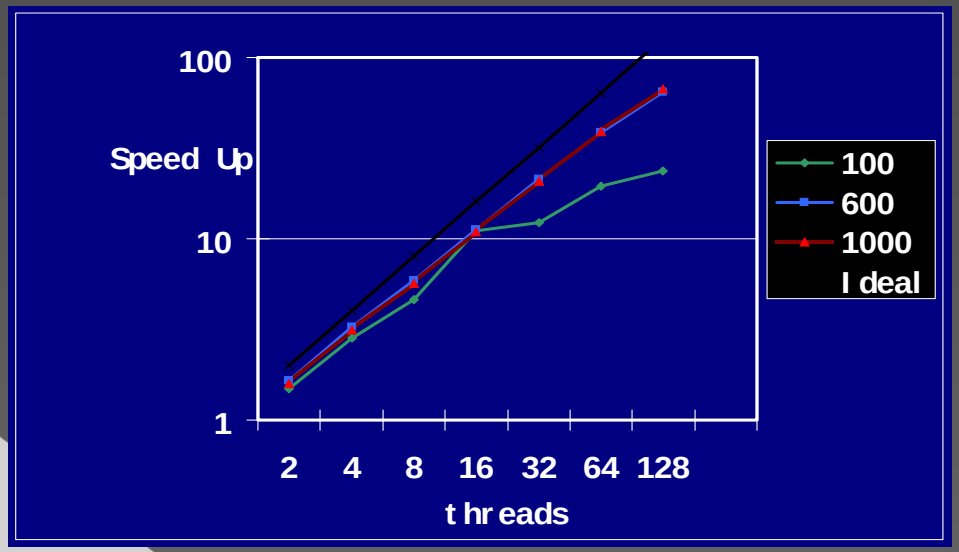
OpenMP: best practices

Smith-Waterman Sequence Alignment Algorithm

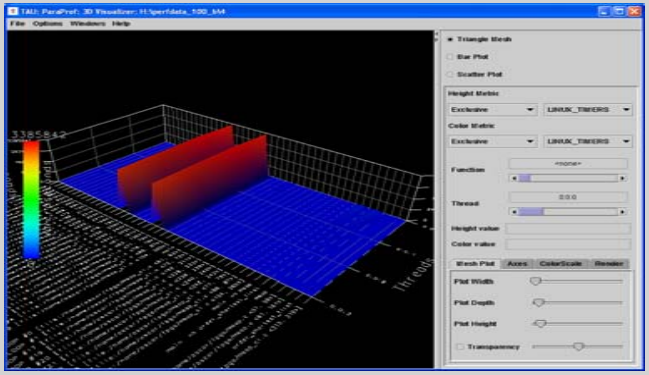


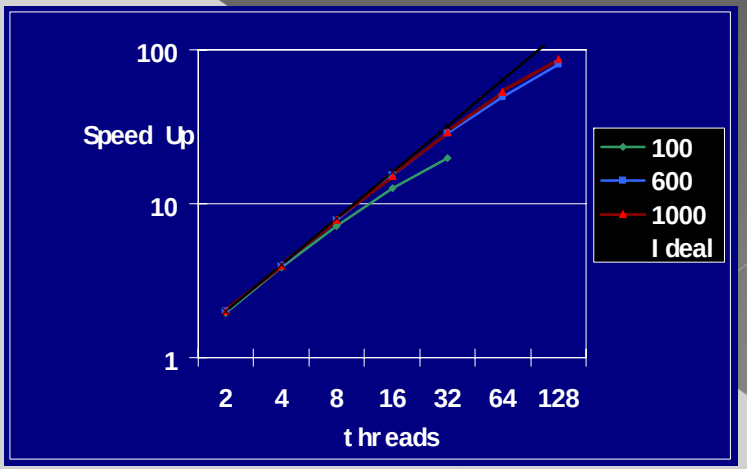
128 threads with 80% efficiency

#pragma omp for



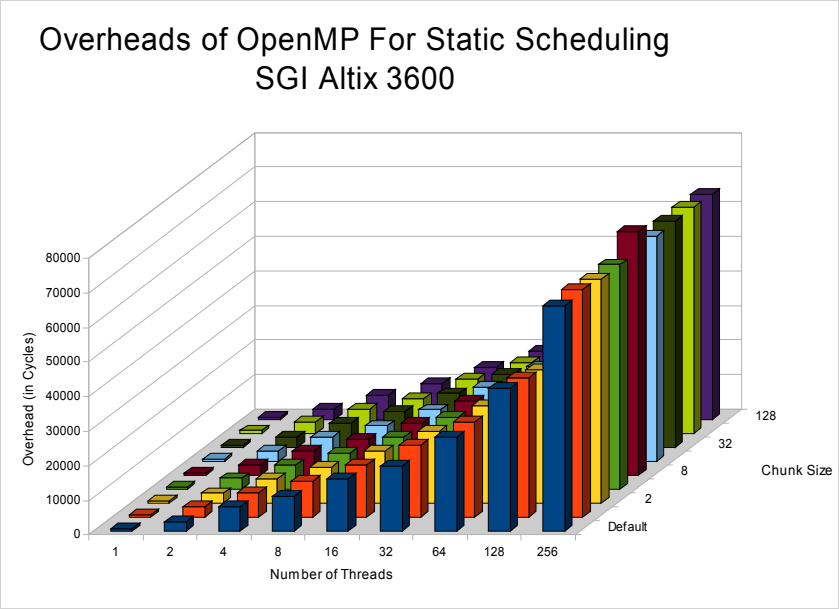
#pragma omp for schedule(dynamic, 1)

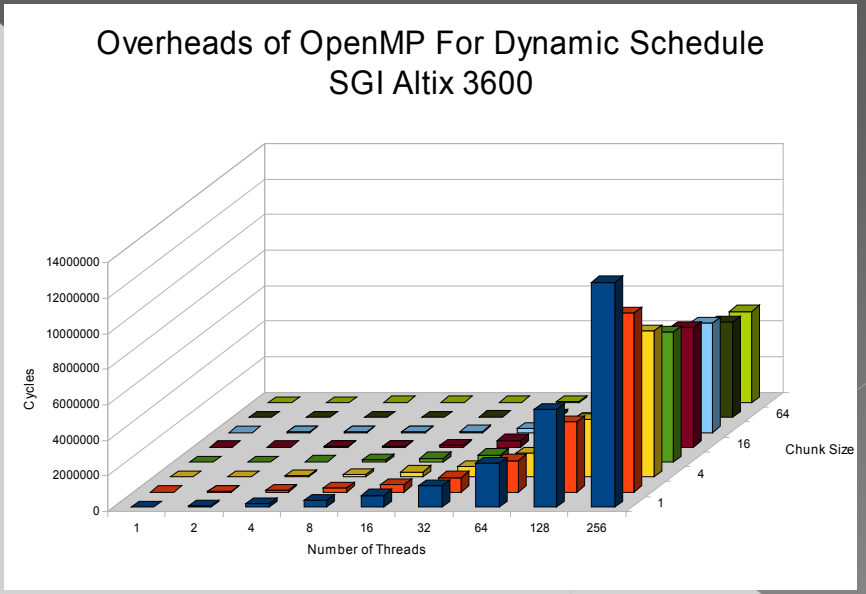




OpenMP: best practices

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





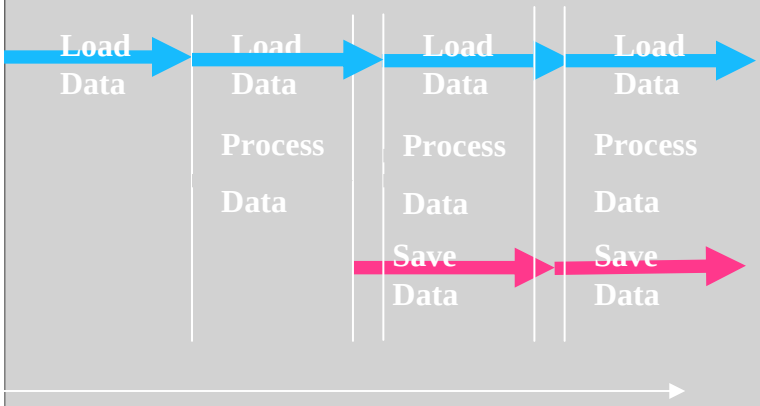
OpenMP: best practices

- OpenMP Pipeline Processing to overlap I/O and Computations

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

OpenMP: best practices

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

OpenMP: best practices

- Single vs. Master work-sharing.
  - Depends on the application
  - Master is more efficient but requires thread 0 to be available
  - Single more efficient if master thread not available but has implicit barrier.

OpenMP: best practices

- Avoid False Sharing
  - Problem when threads access same cache line
  - Use array padding/change schedule to fix the problem.

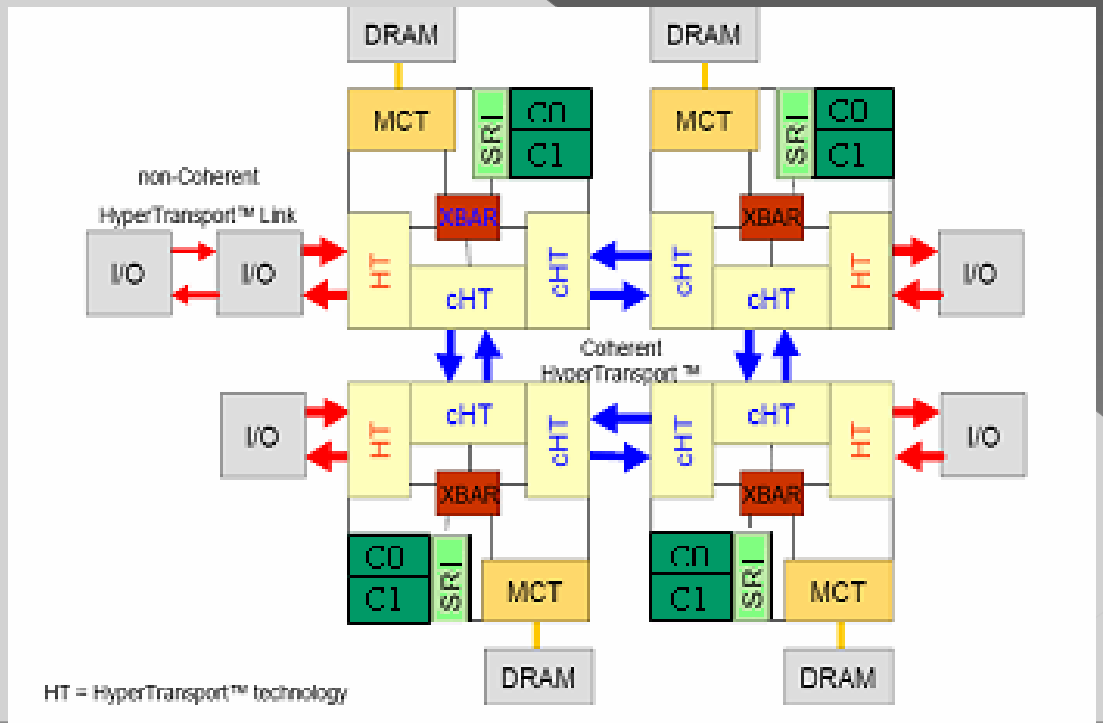
```
int a[max_threads];
#pragma omp parallel for schedule(static,1)
   for(int i=0; i<N; i++)
          a[i] +=i;
```

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

OpenMP: best practices

- Data placement on NUMA architectures
- Use First Touch Policy or system commands to place data.

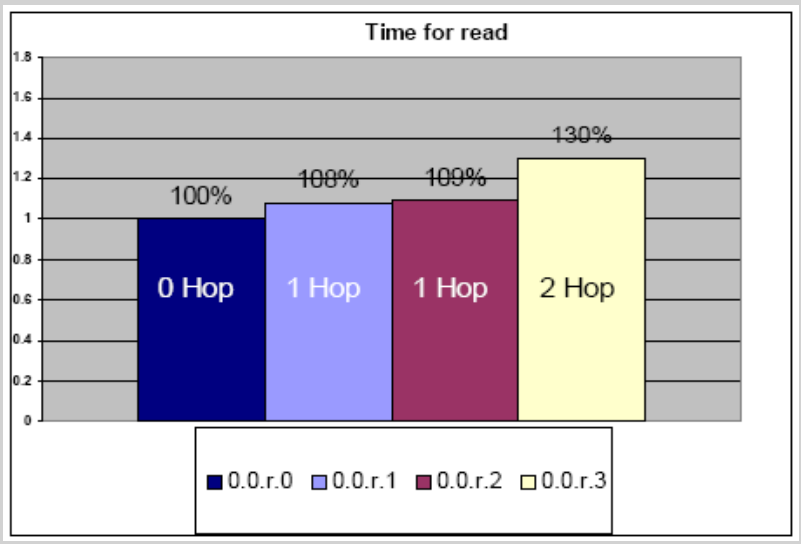
Quartet of four dual-core Opteron Processor

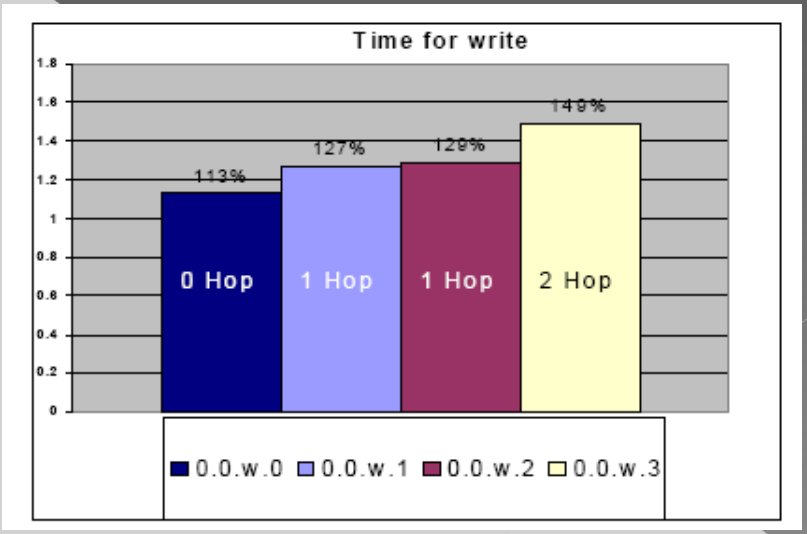


OpenMP: best practices

- NUMA architectures: remote vs. local memory accesses
- Excessive remote memory accesses saturates the interconnect

Quartet of four dual-core Opteron Processor





OpenMP: best practices

- NUMA architectures
- 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);
/* computations */
#pragma omp parallel for
for(i=0; i<N; i++) {
    a[i] = b[i] + c[i];
}
```

OpenMP: best practices

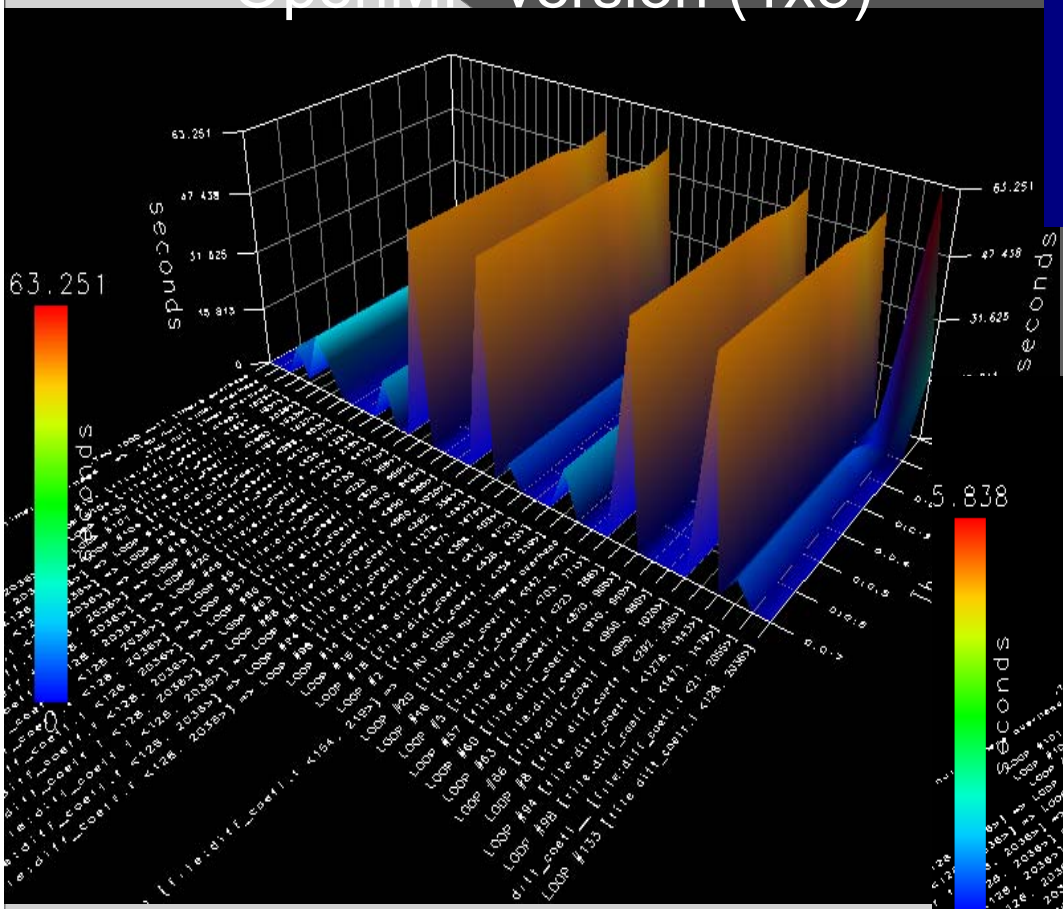
- Privatize variables as much as possible
- Private variables are stored in the local stack to the thread
- Private data close to cache

```
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 ] = …
}
```

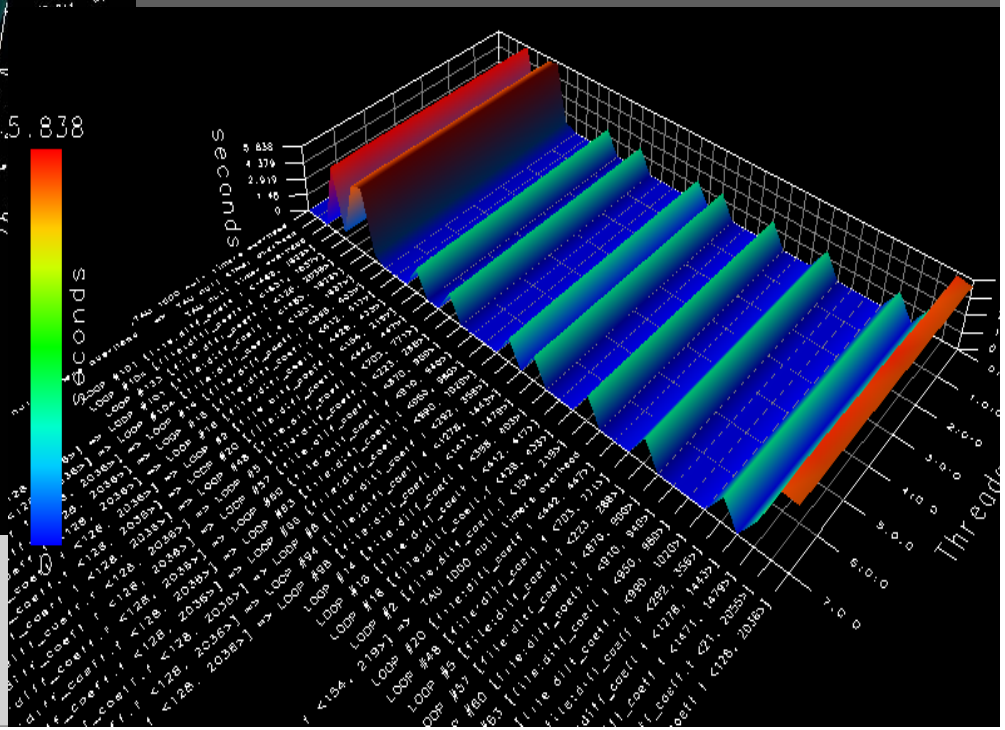
Example: Hybrid CFD code
MPIxOpenMP

OpenMP version (1x8)



We find that a single procedure is responsible for 20% of the total time the OpenMP version and is 9 times slower than the MPI version…. Why?

MPI version (8x1)

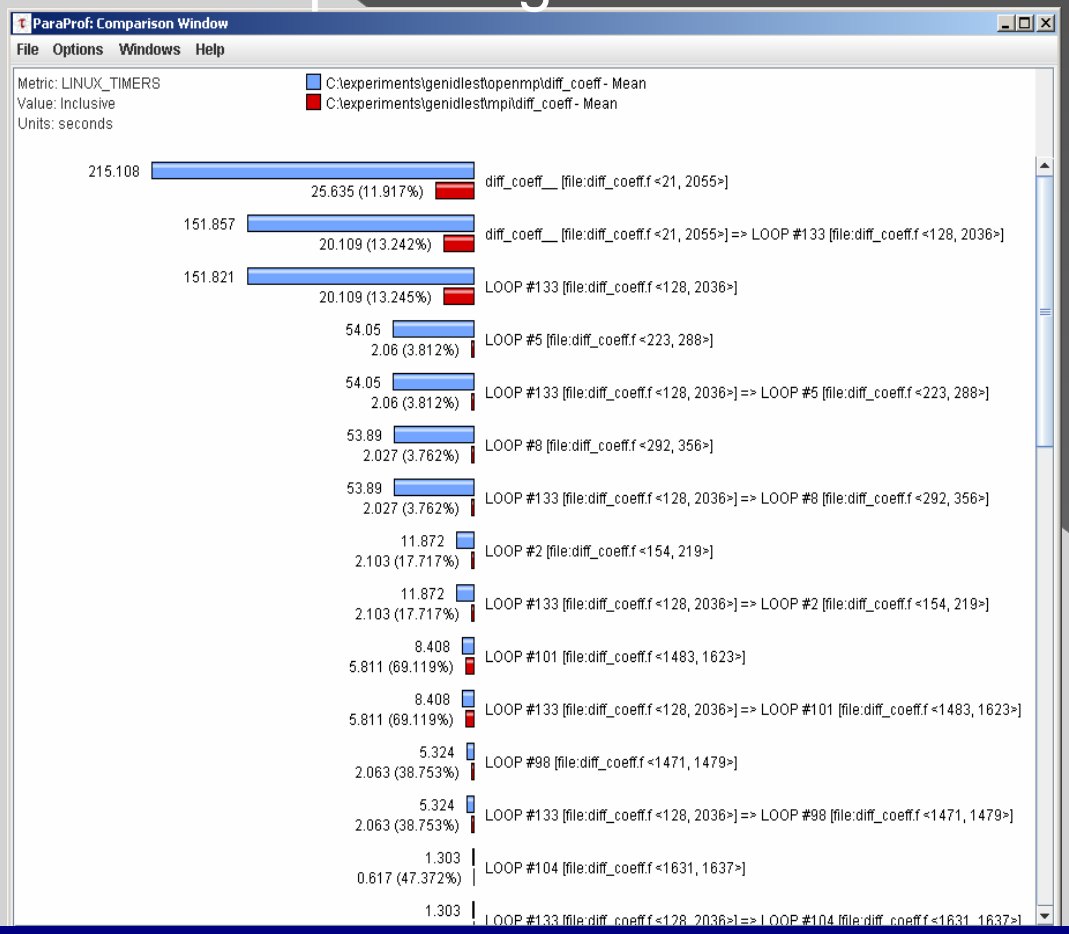


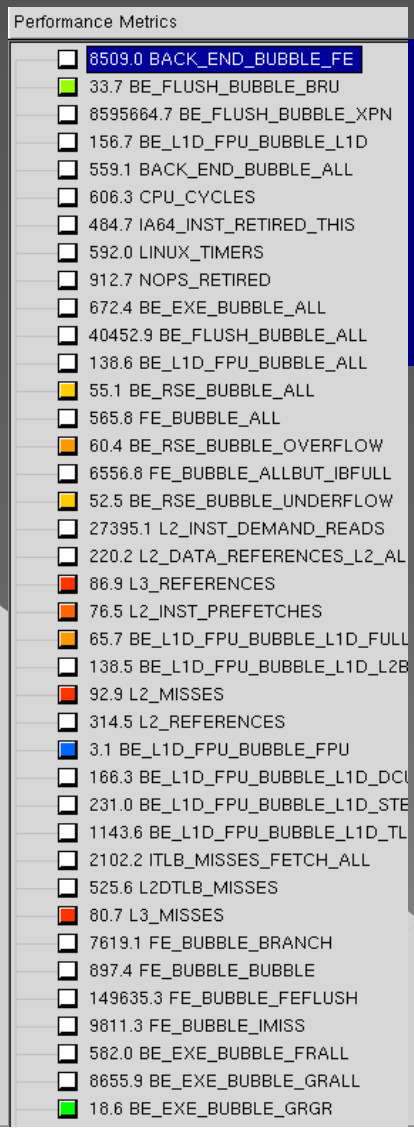
Example: Hybrid CFD code

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 psudo-code: Privatization & First Touch

```
procedure diff_coeff()
{
    allocation of arrays to heap by master thread
    initialization of shared arrays

    PARALLEL REGION
    {
      loop lower_bn [my thread id] , upper bound [my thread id]
      computation on shared arrays
      ....
    }
}
```

Shared Arrays

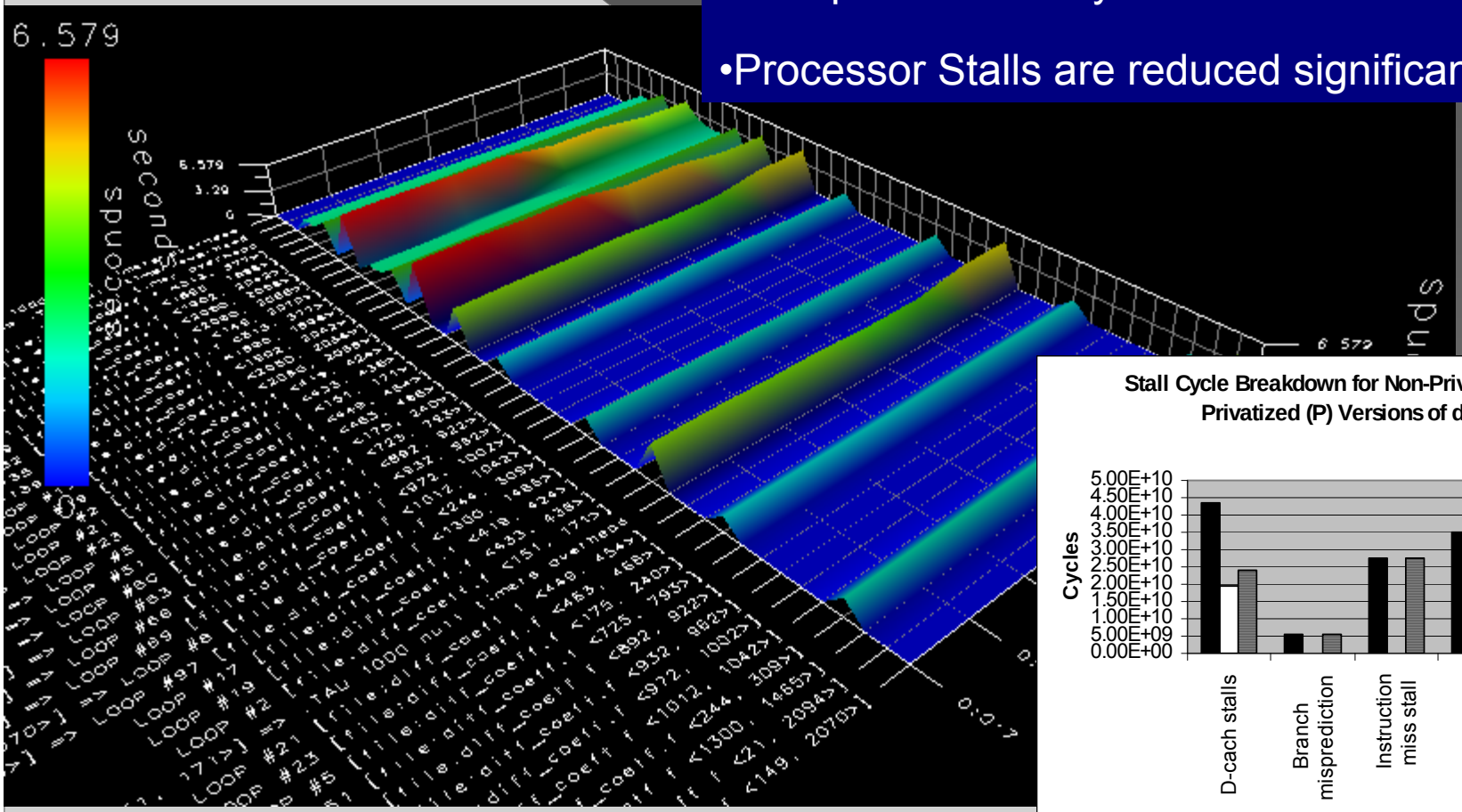
ISSUES:

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

OpenMP: best practices

- Privatizing the arrays improved the performance of the whole program by 30% and a speedup of 10 for the procedure.
- Now procedure only takes 5% of total time
- Processor Stalls are reduced significantly

OpenMP Privatized Version



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.

OpenMP: 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.

OpenMP: 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.

OpenMP: MPI 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.
- Thread support levels:
  - **MPI_THREAD_SINGLE**: Only one thread will execute.
  - **MPI_THREAD_FUNNELED**: Process may be multi-threaded, but only main thread will make MPI calls (all MPI calls are ”funneled" to main thread)..
  - MPI_THREAD_SERIALIZED: Process may be multi-threaded, multiple threads may make MPI calls, but only one at a time: MPI calls are not made concurrently from two distinct threads (all MPI calls are ”serialized").
  - MPI_THREAD_MULTIPLE: Multiple threads may call MPI, with no restrictions.

OpenMP: hybrid MPI/OpenMP

- MPI_THREAD_SERIALIZED is required.
- 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
```

OpenMP: hybrid MPI/OpenMP

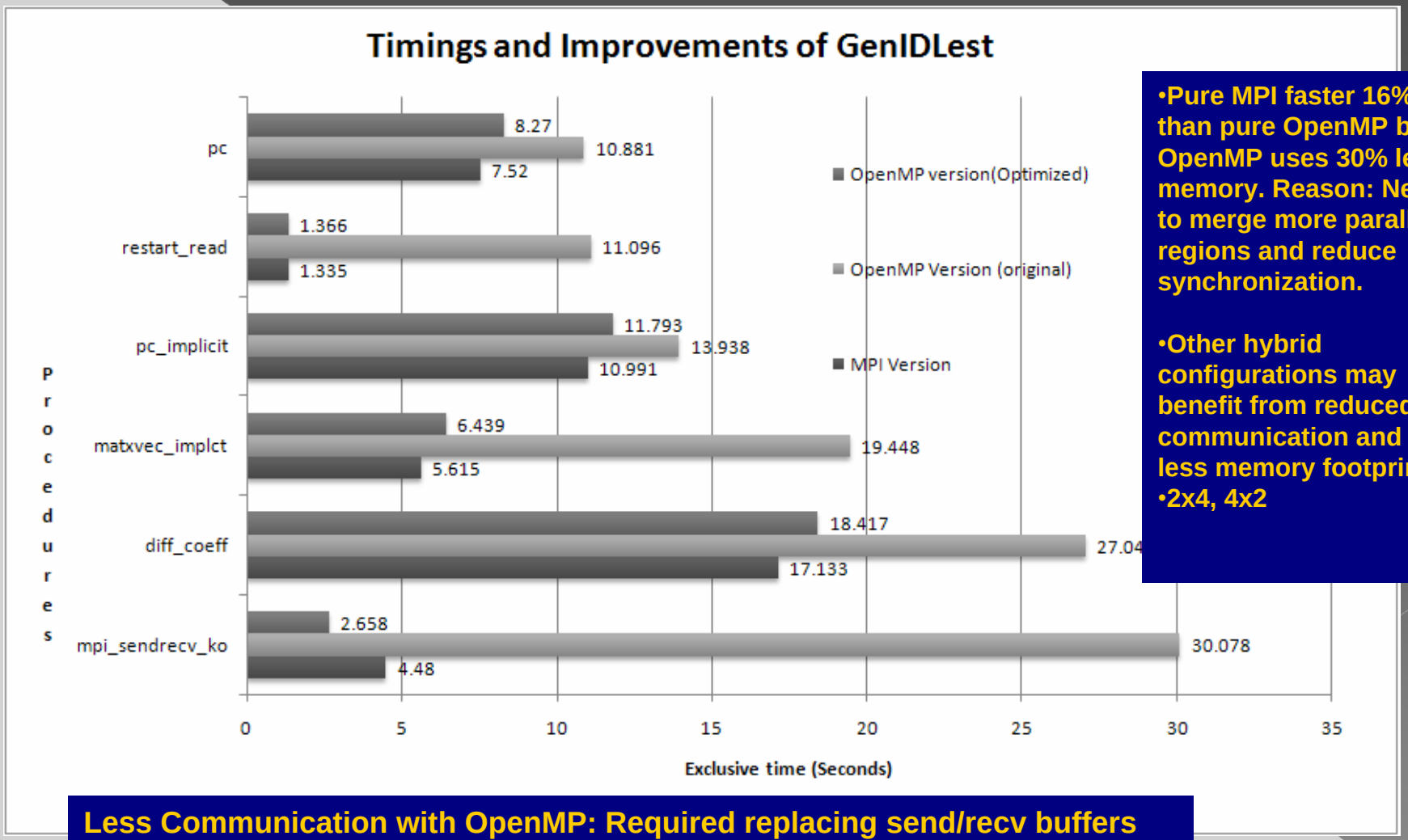
- Need at least MPI_THREAD_MULTIPLE
- Good to overlap computations and communication.

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

You may create OMP tasks to do the MPI communication

s

GenIDLest Hybrid 1x8 vs. 8x1



- 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 nodes, OpenMP:
  - Uses less memory
  - Reduces MPI communication overhead.

Reference Material on OpenMP

- mepage www.openmp.org:

  The primary source of information about OpenMP and its development.

- OMPunity) Homepage

- 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

- www.google.com: OpenMP