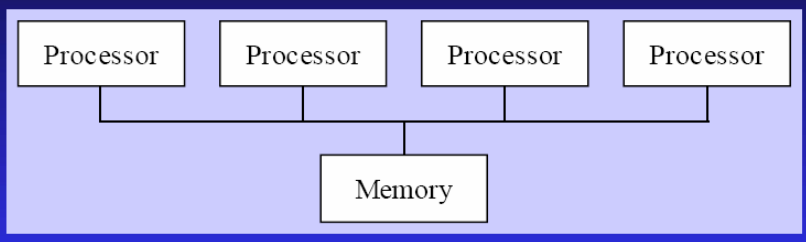
Parallel Programming
in C with MPI and OpenMP

OpenMP

- OpenMP: An application programming interface (API) for parallel programming on multiprocessors
  - Compiler directives
  - Library of support functions
- OpenMP works in conjunction with Fortran, C, or C++

Shared-memory Model

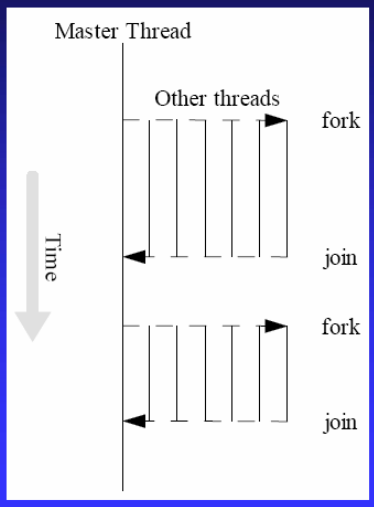


Processors interact and synchronize with each other through shared variables.

Fork/Join Parallelism

- Initially only master thread is active
- Master thread executes sequential code
- Fork: Master thread creates or awakens additional threads to execute parallel code
- Join: At end of parallel code created threads die or are suspended

Fork/Join Parallelism



Shared-memory Model vs. Message-passing Model (#1)

- Shared-memory model
  - Number active threads 1 at start and finish of program, changes dynamically during execution
- Message-passing model
  - All processes active throughout execution of program

Incremental Parallelization

- Sequential program a special case of a shared-memory parallel program
- Parallel shared-memory programs may only have a single parallel loop
- Incremental parallelization: process of converting a sequential program to a parallel program a little bit at a time

Shared-memory Model vs. Message-passing Model (#2)

- Shared-memory model
  - Execute and profile sequential program
  - Incrementally make it parallel
  - Stop when further effort not warranted
- Message-passing model
  - Sequential-to-parallel transformation requires major effort
  - Transformation done in one giant step rather than many tiny steps

Parallel for Loops

- C programs often express data-parallel operations as **`for`** loops

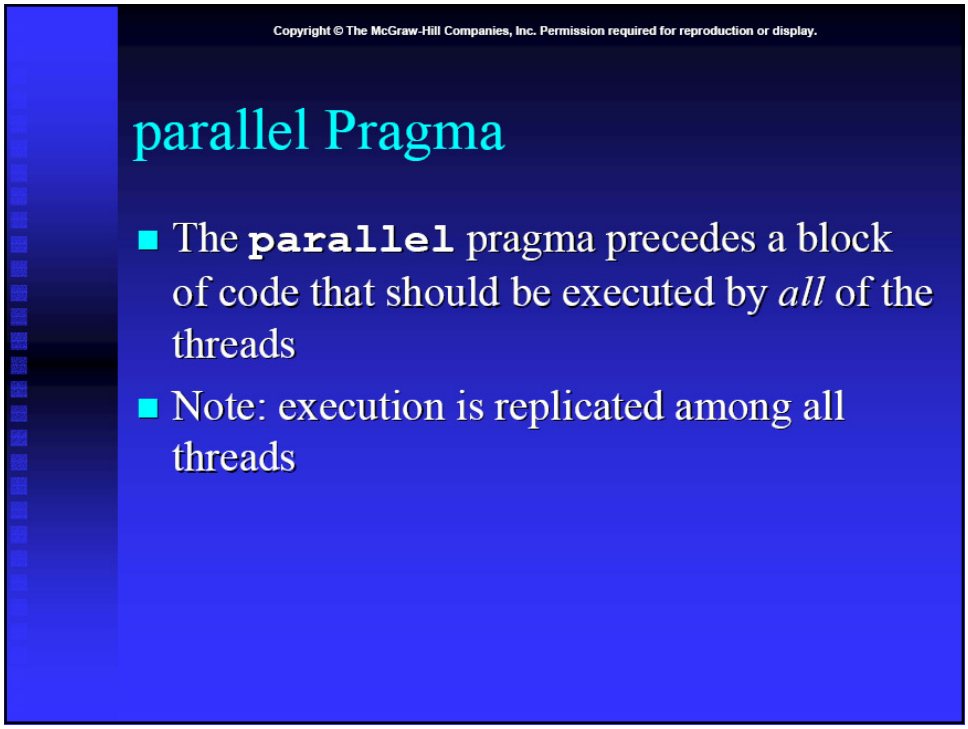
```
for (i = first; i < size; i += prime)
        marked[i] = 1;
```

- OpenMP makes it easy to indicate when the iterations of a loop may execute in parallel
- Compiler takes care of generating code that forks/joins threads and allocates the iterations to threads

Pragmas

- Pragma: a compiler directive in C or C++
- Stands for "pragmatic information"
- A way for the programmer to communicate with the compiler
- Compiler free to ignore pragmas
- Syntax:

`#pragma omp` *<rest of pragma>*



Code transformation

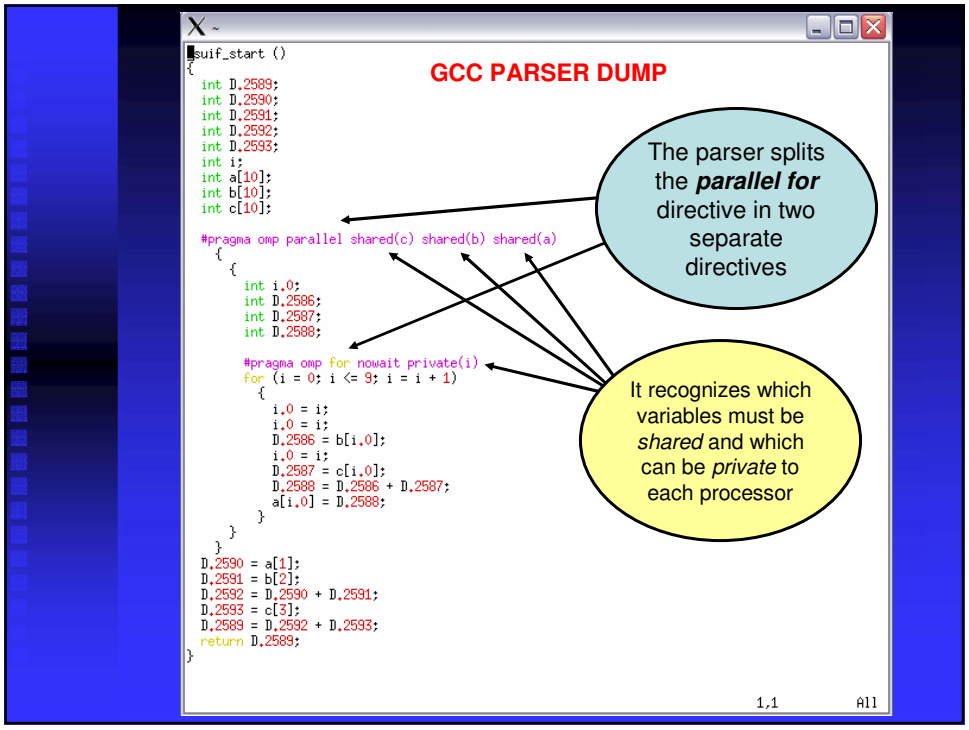
- Every time the compiler finds a **#pragma omp parallel** directive creates a new function in which the code belonging to the scope of the pragma itself is moved
- The directive is replaced with a call to a runtime function that is responsible for forking new threads (in a thread-based implementation) or for loading parallel code onto the slave processors
- Once the parallel region has been executed threads/processors need to synchronize.

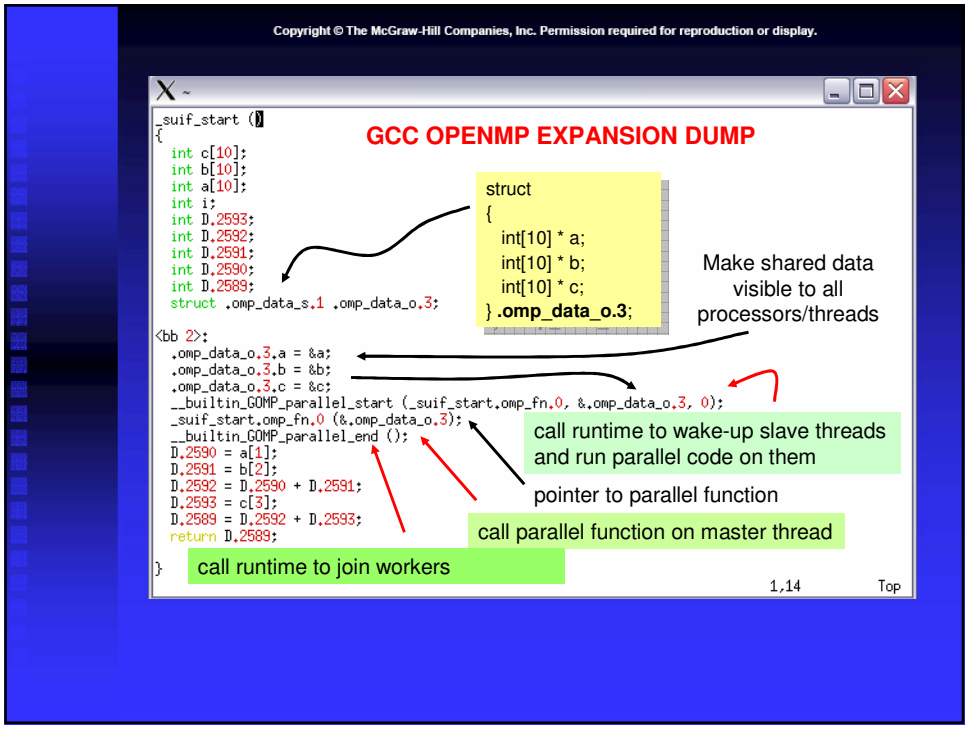
Parallel for Pragma

- Format:

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

- Compiler must be able to verify the run-time system will have information it needs to schedule loop iterations





Es #1 - pthreads

- Current implementation of GCC OpenMP runtime environment (libgomp) is basically a wrapper around the pthreads library.
- Master forks new worker threads with a call to the runtime function GOMP_parallel_start
- After parallel region master joins workers with a call to the runtime function GOMP_parallel_end

- Cu
  en
  pt
- Ma
  ru
- Af
  th

```
void
GOMP_parallel_start (void (*fn) (void *), void *data, unsigned num_threads)
{
  num_threads = gomp_resolve_num_threads (num_threads);
  ...

  /* Launch new threads.  */
  for (; i < nthreads; ++i, ++start_data)
    {
      pthread_t pt;
      int err;

      start_data->ts.team = team;
      start_data->ts.work_share = work_share;
      start_data->ts.team_id = i;
      start_data->ts.work_share_generation = 0;
      start_data->ts.static_trip = 0;
      start_data->fn = fn;
      start_data->fn_data = data;
      start_data->nested = nested;

      err = pthread_create (&pt, &gomp_thread_attr,
                            gomp_thread_start, start_data);
      if (err != 0)
        gomp_fatal ("Thread creation failed: %s", strerror (err));
    }

  gomp_barrier_wait (nested ? &team->barrier : &gomp_threads_dock);
  ...
}
```

If num_threads = 0 determine number of worker threads

Fork worker threads

Wait for all threads to be ready before starting parallel region

276,0-1 87%


```
void
GOMP_parallel_start (void (*fn) (void *), void *data, unsigned num_threads)
{
  num_threads = gomp_resolve_num_threads (num_threads);
```


```
/* Terminate the current team.  This is only to be called by the master
   thread.  We assume that we must wait for the other threads.  */

void
gomp_team_end (void)
{
  struct gomp_thread *thr = gomp_thread ();
  struct gomp_team *team = thr->ts.team;

  gomp_barrier_wait (&team->barrier);

  thr->ts = team->prev_ts;

  free_team (team);
}
```

We need to synchronize threads with a barrier at the end of a parallel region

Join worker threads and suspend them

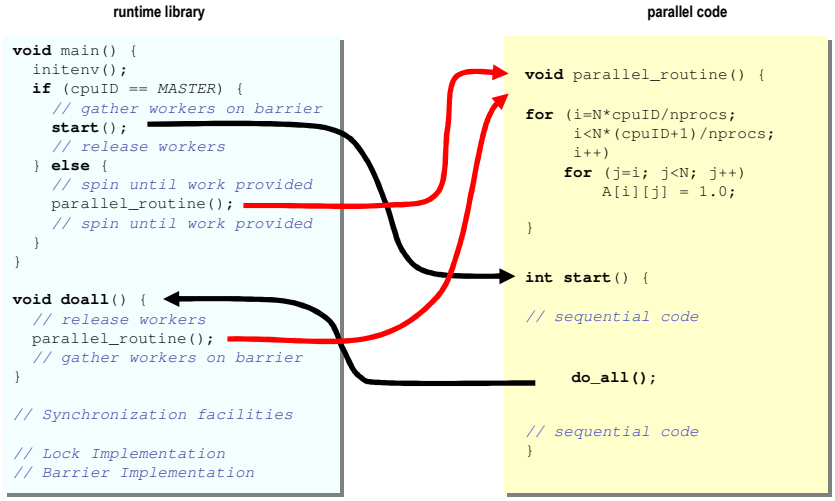
302,0-1 93%


```
  gomp_barrier_wait (nested ? &team->barrier : &gomp_threads_dock);
  ...
}
```

Wait for all threads to be ready before starting parallel region

276,0-1 87%

Es #2 - MPARM



Functions for SPMD-style Programming

- The parallel pragma allows us to write SPMD-style programs
- In these programs we often need to know number of threads and thread ID number
- OpenMP provides functions to retrieve this information

Function omp_get_thread_num

- This function returns the thread identification number
- If there are *t* threads, the ID numbers range from 0 to *t*-1
- The master thread has ID number 0

```
int omp_get_thread_num (void)
```


Function omp_get_num_threads

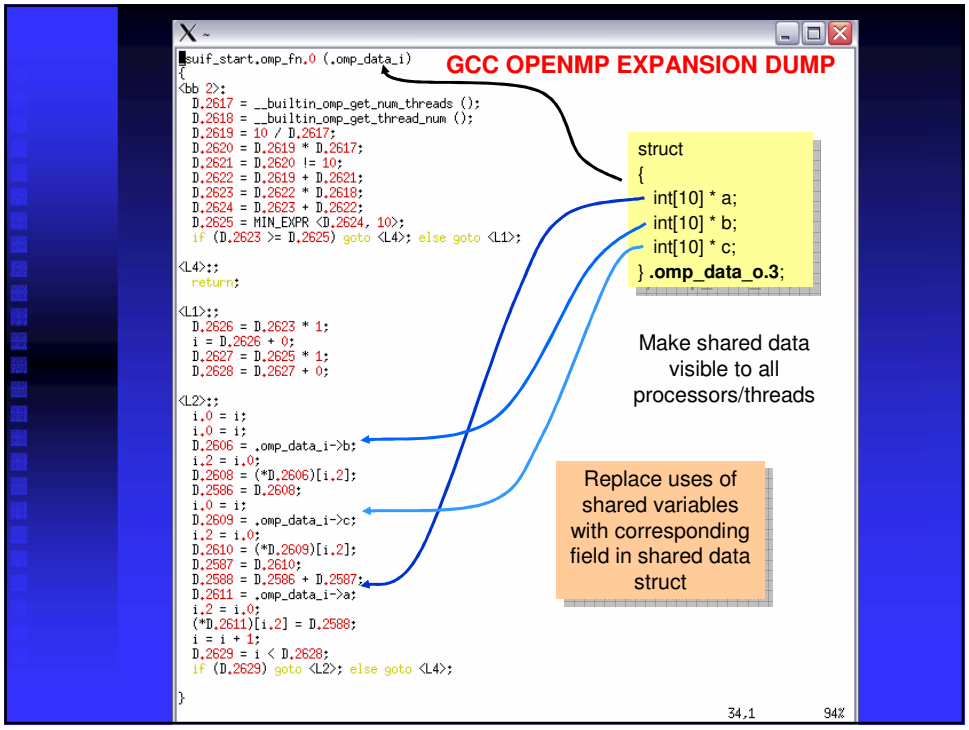
- Function omp_get_num_threads returns the number of active threads
- If call this function from sequential portion of program, it will return 1

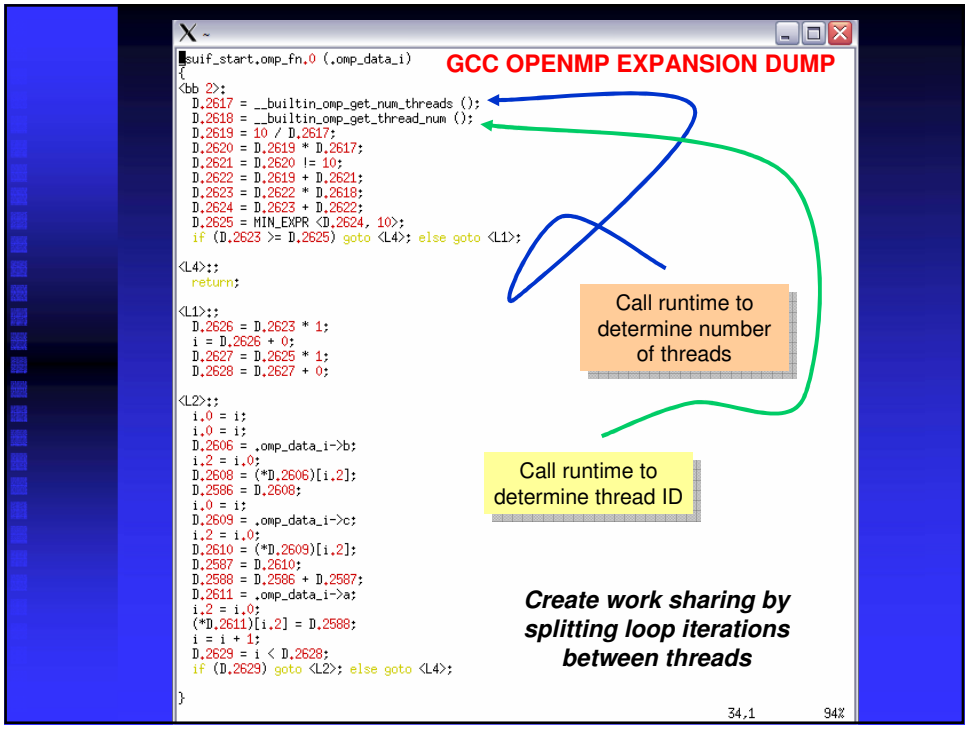
```
int omp_get_num_threads (void)
```

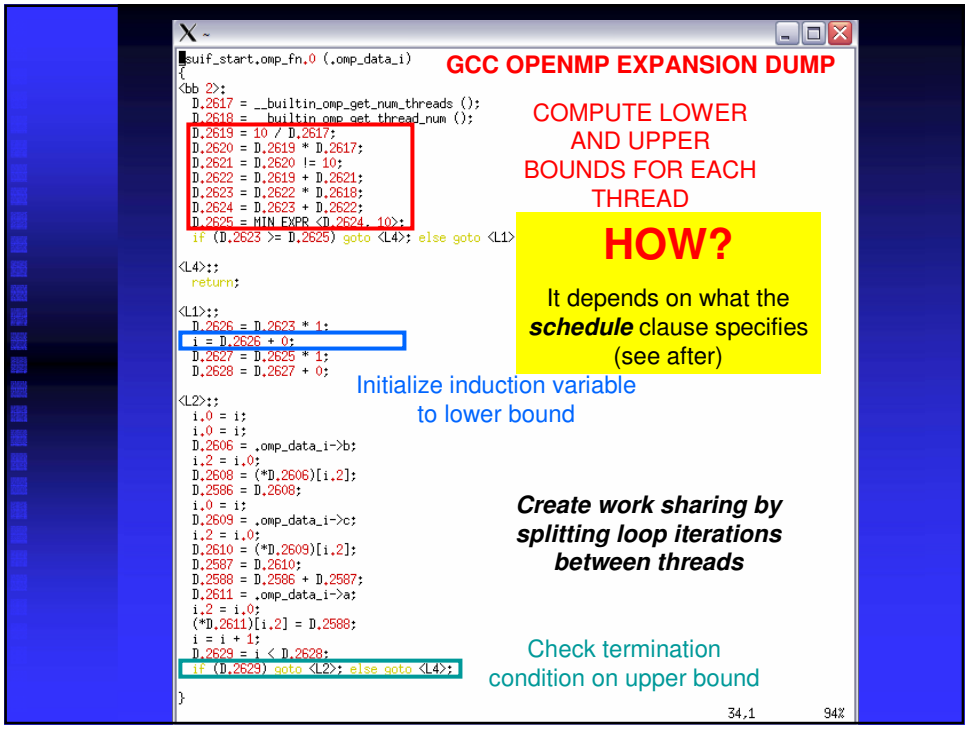
for Pragma

- The **`parallel`** pragma instructs every thread to execute all of the code inside the block
- If we encounter a **`for`** loop that we want to divide among threads, we use the **`for`** pragma

```
#pragma omp for
```




GCC OPENMP EXPANSION DUMP
Call runtime to determine number of threads
Call runtime to determine thread ID
Create work sharing by splitting loop iterations between threads


GCC OPENMP EXPANSION DUMP
COMPUTE LOWER AND UPPER BOUNDS FOR EACH THREAD
HOW?
It depends on what the schedule clause specifies (see after)
Initialize induction variable to lower bound
Create work sharing by splitting loop iterations between threads
Check termination condition on upper bound

Canonical Shape of for Loop Control Clause

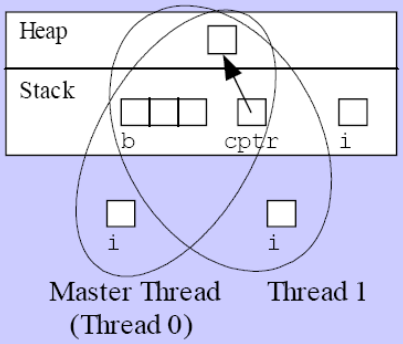
$$\text{for}(\text{index} = \textit{start};\ \text{index} \geq \left\{ \begin{array}{l} < \\ <= \\ >= \\ > \end{array} \right\} \textit{end};\ \left\{ \begin{array}{c} \text{index}++ \\ ++\text{index} \\ \text{index}-- \\ --\text{index} \\ \text{index} \mathrel{+}= \textit{inc} \\ \text{index} \mathrel{-}= \textit{inc} \\ \text{index} = \text{index} + \textit{inc} \\ \text{index} = \textit{inc} + \text{index} \\ \text{index} = \text{index} - \textit{inc} \end{array} \right\})$$

Shared and Private Variables

- Shared variable: has same address in execution context of every thread
- Private variable: has different address in execution context of every thread
- A thread cannot access the private variables of another thread

Shared and Private Variables

```
int main (int argc, char *argv[])
{
   int b[3];
   char *cptr;
   int i;

   cptr = malloc(1);
#pragma omp parallel for
   for (i = 0; i < 3; i++)
      b[i] = i;
```



Declaring Private Variables

```
for (i = 0; i < BLOCK_SIZE(id,p,n); i++)
   for (j = 0; j < n; j++)
      a[i][j] = MIN(a[i][j],a[i][k]+tmp);
```

- Either loop could be executed in parallel
- We prefer to make outer loop parallel, to reduce number of forks/joins
- We then must give each thread its own private copy of variable `j`

private Clause

- Clause: an optional, additional component to a pragma
- Private clause: directs compiler to make one or more variables private

```
private ( <variable list> )
```


Example Use of private Clause

```
#pragma omp parallel for private(j)
for (i = 0; i < BLOCK_SIZE(id,p,n); i++)
   for (j = 0; j < n; j++)
      a[i][j] = MIN(a[i][j],a[i][k]+tmp);
```

firstprivate Clause

- Used to create private variables having initial values identical to the variable controlled by the master thread as the loop is entered
- Variables are initialized once per thread, not once per loop iteration
- If a thread modifies a variable's value in an iteration, subsequent iterations will get the modified value

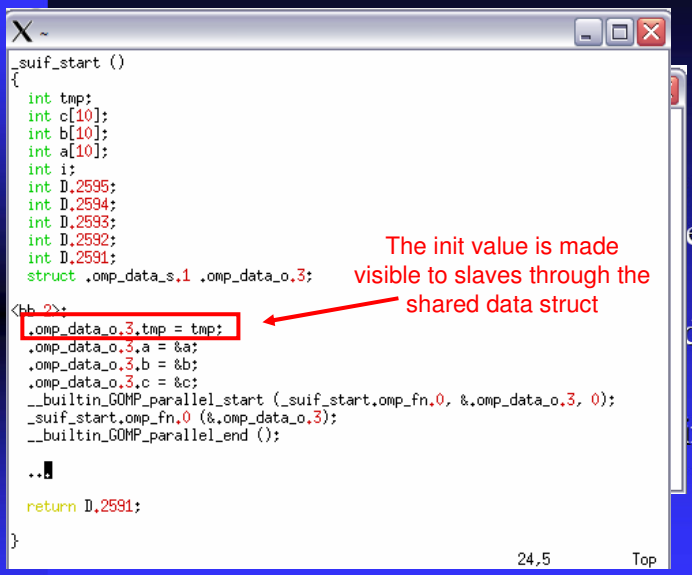
Copyright © The McGraw-Hill Companies, Inc. Permission required for reproduction or display.

```
#include <stdio.h>
#include "appsupport.h"

#define N       10

extern int
_suif_start () {
  int i;
  int a[N], b[N], c[N];
  int tmp = 5;

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

return a[1] + b[2] + c[3];
}
:syn on                                  1,1           Top
```

Variable **tmp** is only accessible by the master thread. It has an initial value that slaves need to know

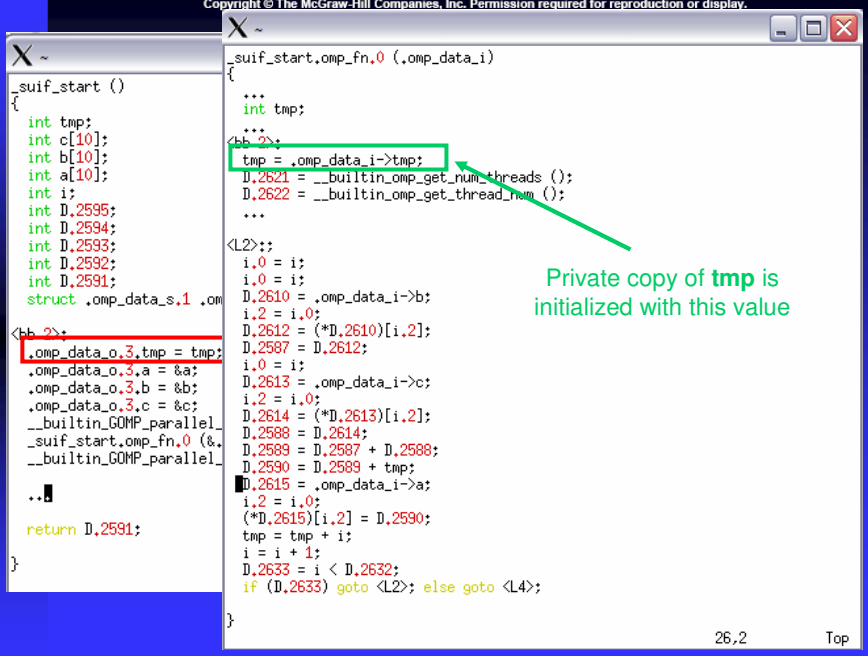
Slave threads will have a private copy of **tmp** initialized with this value

initial
ed by the
d, not once
n an
the
modified value



The init value is made visible to slaves through the shared data struct

initial
ed by the
d, not once
n an
the



Private copy of **tmp** is initialized with this value

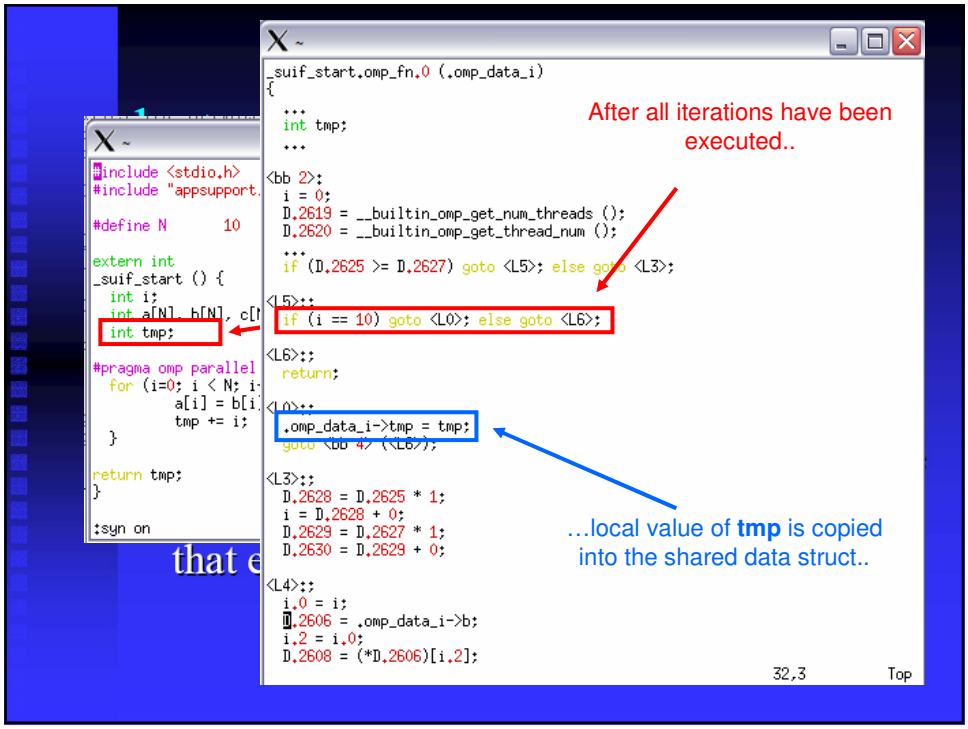
lastprivate Clause

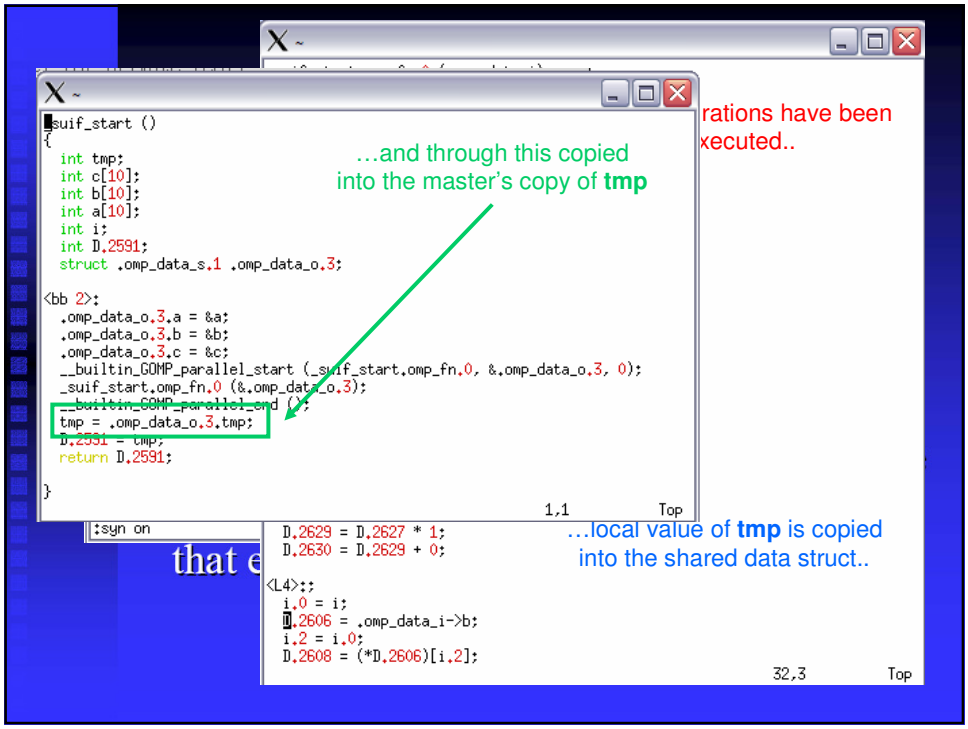
- Sequentially last iteration: iteration that occurs last when the loop is executed sequentially
- **`lastprivate`** clause: used to copy back to the master thread's copy of a variable the private copy of the variable from the thread that executed the sequentially last iteration


```
#include <stdio.h>
#include "appsupport.h"

#define N       10

extern int
_suif_start () {
  int i;
  int a[N], b[N], c[N];
  int tmp;

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

return tmp;
}
```

Variable **tmp** is only accessible by the master thread.

Its value will be written by the last thread that works on its private copy


#include <stdio.h>
#include "appsupport.
#define N 10
extern int
_suif_start () {
int i;
int tmp;
#pragma omp parallel
for (i=0; i < N; i-
a[i] = b[i
tmp += i;
}
return tmp;
}
:syn on
_suif_start.omp_fn.0 (.omp_data_i)
{
int tmp;
<bb 2>:
i = 0;
D.2619 = __builtin_omp_get_num_threads ();
D.2620 = __builtin_omp_get_thread_num ();
if (D.2625 >= D.2627) goto <L5>; else goto <L3>;
<L5>:;
if (i == 10) goto <L0>; else goto <L6>;
<L6>:;
return;
<L0>:;
.omp_data_i->tmp = tmp;
<L3>:;
D.2628 = D.2625 * 1;
i = D.2628 + 0;
D.2629 = D.2627 * 1;
D.2630 = D.2629 + 0;
<L4>:;
i.0 = i;
D.2606 = .omp_data_i->b;
i.2 = i.0;
D.2608 = (*D.2606)[i.2];
32,3 Top
After all iterations have been executed..
…local value of tmp is copied into the shared data struct..
that e


_suif_start ()
{
int tmp;
int c[10];
int b[10];
int a[10];
int i;
int D.2591;
struct .omp_data_s.1 .omp_data_o.3;
<bb 2>:
.omp_data_o.3.a = &a;
.omp_data_o.3.b = &b;
.omp_data_o.3.c = &c;
__builtin_GOMP_parallel_start (_suif_start.omp_fn.0, &.omp_data_o.3, 0);
_suif_start.omp_fn.0 (&.omp_data_o.3);
__builtin_GOMP_parallel_end ();
tmp = .omp_data_o.3.tmp;
D.2591 = tmp;
return D.2591;
}
1,1 Top
:syn on
D.2629 = D.2627 * 1;
D.2630 = D.2629 + 0;
<L4>:;
i.0 = i;
D.2606 = .omp_data_i->b;
i.2 = i.0;
D.2608 = (*D.2606)[i.2];
32,3 Top
…and through this copied into the master's copy of tmp
…local value of tmp is copied into the shared data struct..
that e

Critical Sections

```
double area, pi, x;
int i, n;
...
area = 0.0;
for (i = 0; i < n; i++) {
   x += (i+0.5)/n;
   area += 4.0/(1.0 + x*x);
}
pi = area / n;
```


Race Condition

- Consider this C program segment to compute π using the rectangle rule:

```
double area, pi, x;
int i, n;
...
area = 0.0;
for (i = 0; i < n; i++) {
   x = (i+0.5)/n;
   area += 4.0/(1.0 + x*x);
}
pi = area / n;
```

Race Condition (cont.)

- If we simply parallelize the loop...

```
double area, pi, x;
int i, n;
...
area = 0.0;
#pragma omp parallel for private(x)
for (i = 0; i < n; i++) {
   x = (i+0.5)/n;
   area += 4.0/(1.0 + x*x);
}
pi = area / n;
```


Race Condition (cont.)

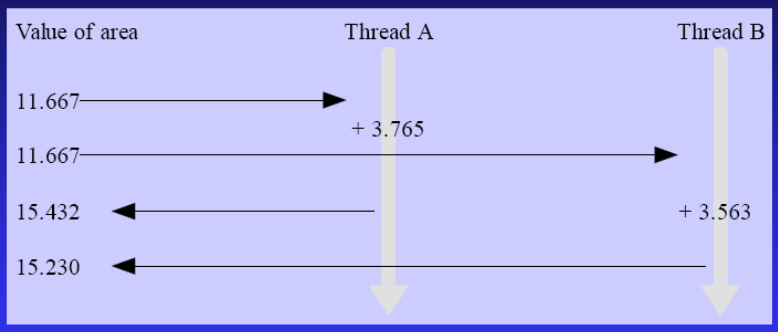
- ... we set up a race condition in which one process may "race ahead" of another and not see its change to shared variable **area**

area 15.230 **Answer should be 18.995**

Thread A 15.432 Thread B 15.230

```
area += 4.0/(1.0 + x*x)
```

Race Condition Time Line



critical Pragma

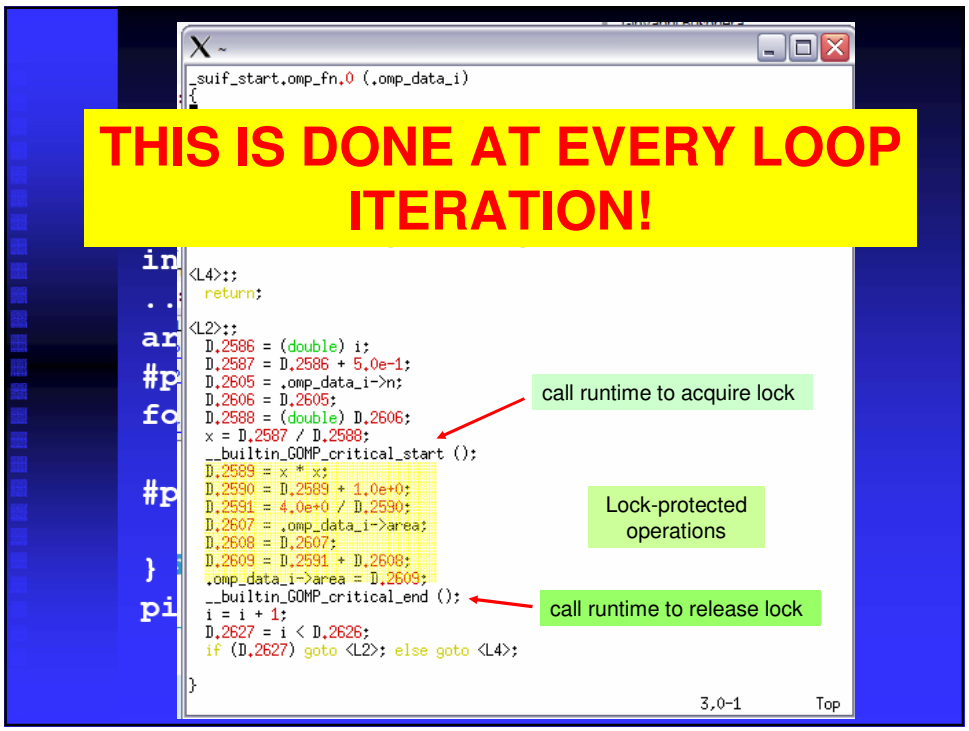
- Critical section: a portion of code that only thread at a time may execute
- We denote a critical section by putting the pragma

```
#pragma omp critical
```

in front of a block of C code

Correct, But Inefficient, Code

```
double area, pi, x;
int i, n;
...
area = 0.0;
#pragma omp parallel for private(x)
for (i = 0; i < n; i++) {
   x = (i+0.5)/n;
#pragma omp critical
   area += 4.0/(1.0 + x*x);
}
pi = area / n;
```



Source of Inefficiency

- Update to **`area`** inside a critical section
- Only one thread at a time may execute the statement; i.e., it is sequential code
- Time to execute statement significant part of loop
- By Amdahl's Law we know speedup will be severely constrained

Reductions

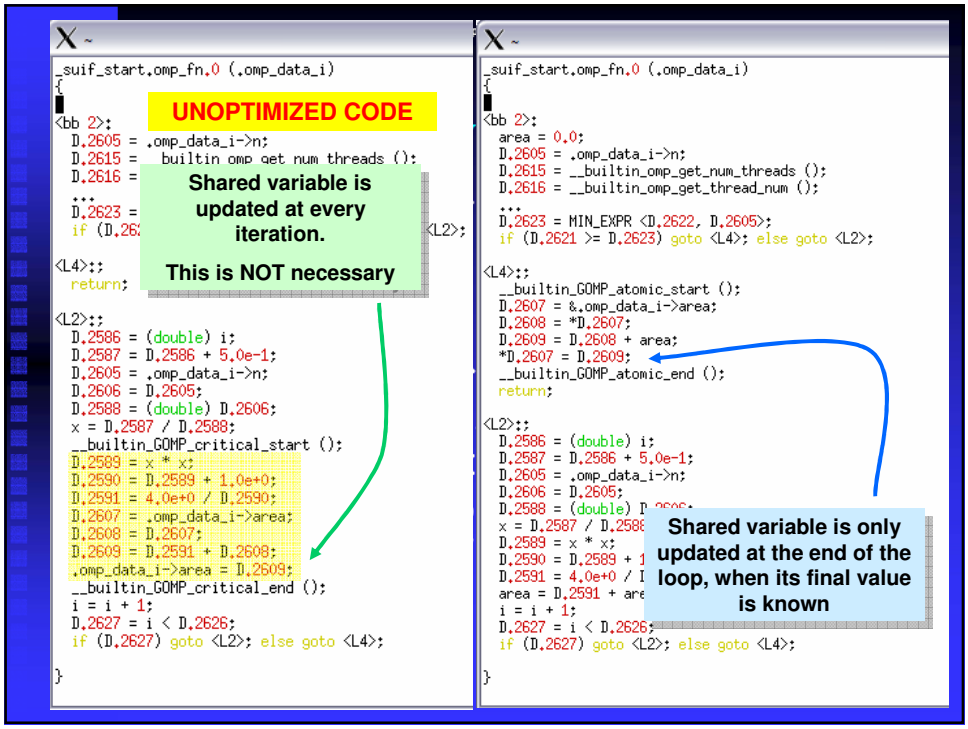
- Reductions are so common that OpenMP provides support for them
- May add reduction clause to **`parallel for`** pragma
- Specify reduction operation and reduction variable
- OpenMP takes care of storing partial results in private variables and combining partial results after the loop

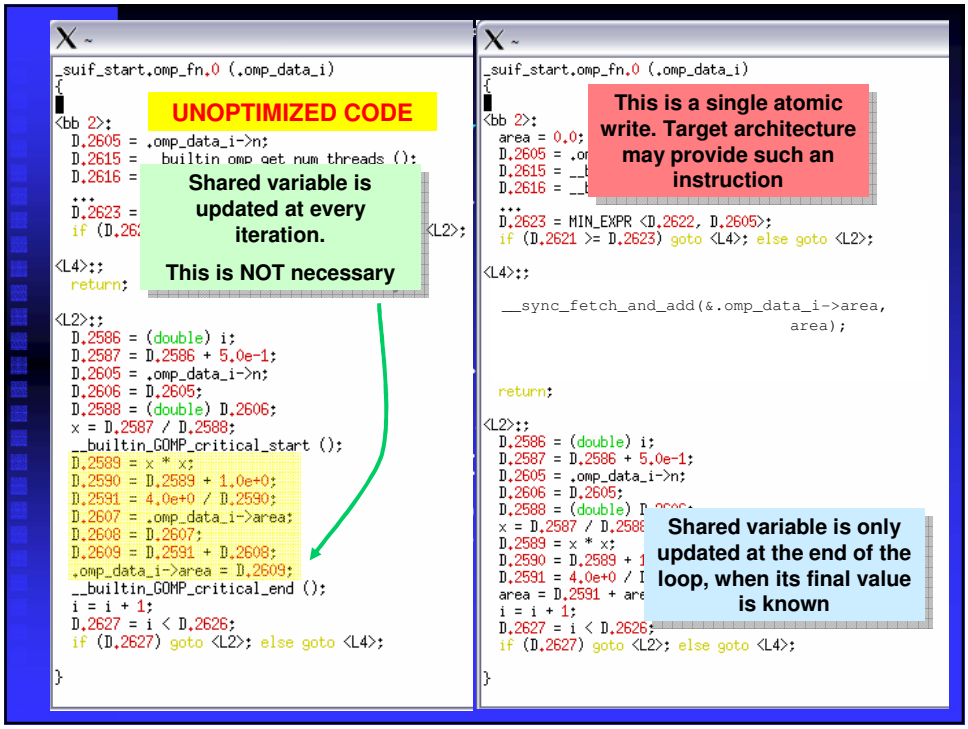
reduction Clause

- The reduction clause has this syntax:
  reduction (*<op>* :*<variable>*)
- Operators
  - \+ Sum
  - * Product
  - & Bitwise and
  - | Bitwise or
  - ^ Bitwise exclusive or
  - && Logical and
  - || Logical or

π-finding Code with Reduction Clause

```
double area, pi, x;
int i, n;
...
area = 0.0;
#pragma omp parallel for \
        private(x) reduction(+:area)
for (i = 0; i < n; i++) {
   x = (i + 0.5)/n;
   area += 4.0/(1.0 + x*x);
}
pi = area / n;
```


UNOPTIMIZED CODE
Shared variable is updated at every iteration.
This is NOT necessary
Shared variable is only updated at the end of the loop, when its final value is known


UNOPTIMIZED CODE
Shared variable is updated at every iteration.
This is NOT necessary
This is a single atomic write. Target architecture may provide such an instruction
__sync_fetch_and_add(&.omp_data_i->area, area);
Shared variable is only updated at the end of the loop, when its final value is known

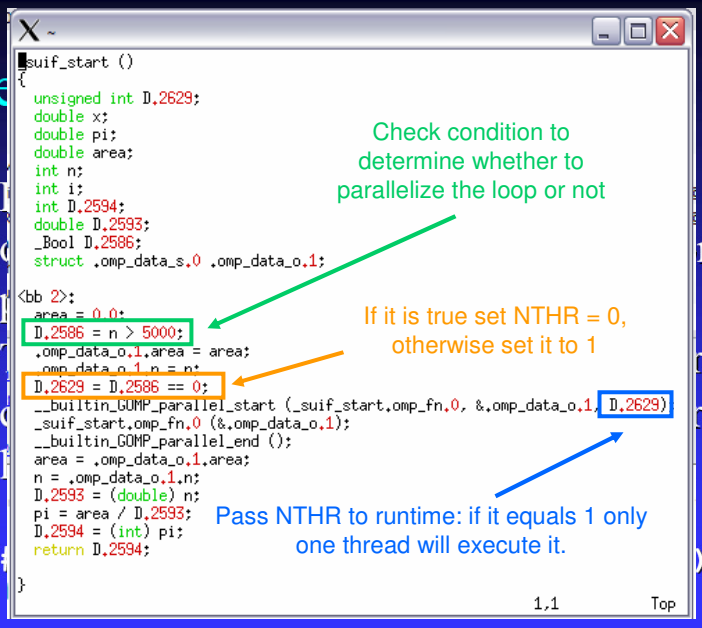
Performance Improvement #1

- Too many fork/joins can lower performance
- Inverting loops may help performance if
  - Parallelism is in inner loop
  - After inversion, the outer loop can be made parallel
  - Inversion does not significantly lower cache hit rate

Performance Improvement #2

- If loop has too few iterations, fork/join overhead is greater than time savings from parallel execution
- The **`if`** clause instructs compiler to insert code that determines at run-time whether loop should be executed in parallel; e.g.,

```
#pragma omp parallel for if(n > 5000)
```



Performance Improvement #3

- We can use **`schedule`** clause to specify how iterations of a loop should be allocated to threads
- Static schedule: all iterations allocated to threads before any iterations executed
- Dynamic schedule: only some iterations allocated to threads at beginning of loop's execution. Remaining iterations allocated to threads that complete their assigned iterations.

Static vs. Dynamic Scheduling

- Static scheduling
  - Low overhead
  - May exhibit high workload imbalance
- Dynamic scheduling
  - Higher overhead
  - Can reduce workload imbalance

Chunks

- A chunk is a contiguous range of iterations
- Increasing chunk size reduces overhead and may increase cache hit rate
- Decreasing chunk size allows finer balancing of workloads

USING THE schedule CLAUSE

- A parallel region has at least one barrier at its end, and may have additional barriers within it
- At each barrier the other members of the team must wait for the last thread to arrive
- To minimize this wait time shared work should be distributed so that all threads arrive at the barrier at about the same time
- The choice of a schedule for a **for** construct is also determined by characteristics of the memory system (presence of caches, uniform access times, etc.)

USING THE schedule CLAUSE

- A parallel r d, and may have a
- At each barrie m must wait for the last thre
- To minimize th d be distributed so t rrier at about the same time
- The choice of s also determined by system (presence of c etc.)

ASSIGNING SAME ITERATIONS TO SAME THREADS MAY IMPROVE DATA REUSE

```
#pragma omp parallel
{
#pragma omp for schedule (static)
  for (i=0; i<n; i++)
    a[i] = work1(i);
#pragma omp for schedule (static)
  for (i=0; i<n; i++)
    if (i>=k) a[i] = work2(i);
}
```

schedule Clause

- Syntax of schedule clause
  `schedule (<type>[,<chunk> ])`
- Schedule type required, chunk size optional
- Allowable schedule types
  - static: static allocation
  - dynamic: dynamic allocation
  - guided: guided self-scheduling
  - runtime: type chosen at run-time based on value of environment variable OMP_SCHEDULE

Scheduling Options

- schedule(static): block allocation of about n/t contiguous iterations to each thread
- schedule(static,C): interleaved allocation of chunks of size C to threads
- schedule(dynamic): dynamic one-at-a-time allocation of iterations to threads
- schedule(dynamic,C): dynamic allocation of C iterations at a time to threads

SPLITTING LOOP ITERATIONS – schedule (static)

- The **static** schedule is appropriate for a parallel region containing a single **for** construct, with each iteration requiring the same amount of work
- Loop boundaries are statically determined at compile time. No interaction with the runtime is required.

Es. N=10, Nthr=4.

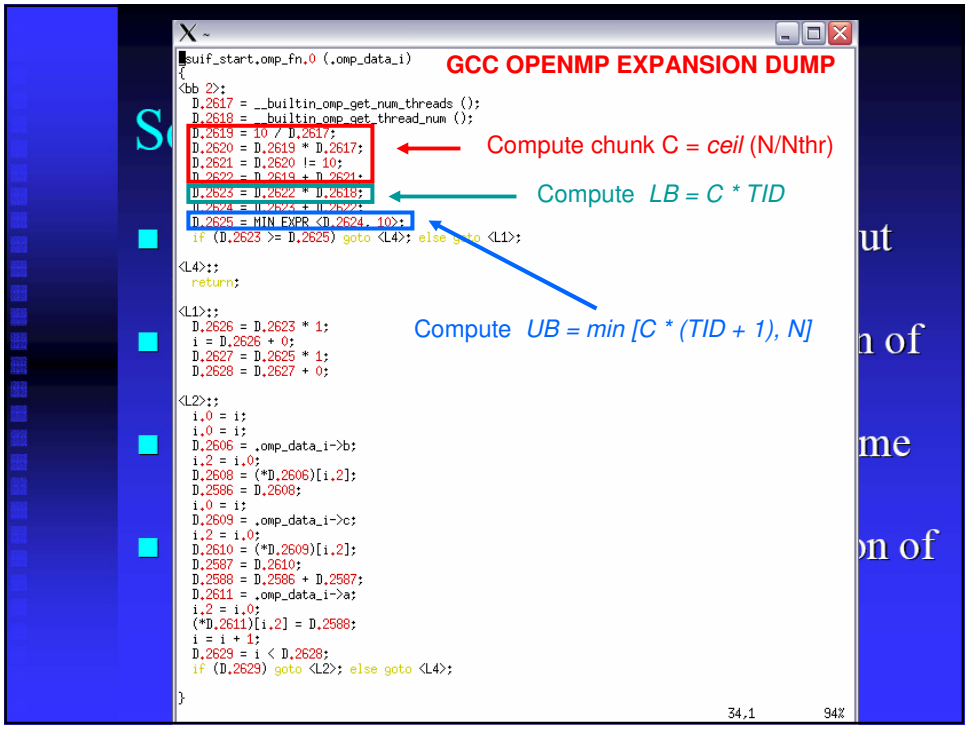
DATA CHUNK

$$C = ceil\left(\frac{N}{Nthr}\right)$$

3 vector elements

TID	**0**	**1**	**2**	**3**
LB = C * TID	0	3	6	9
UB = *MIN* { [C * (TID + 1)], N}	3	6	9	10

NOTE: There may be **LOAD IMBALANCE** between threads (i.e. they operate on data chunks of different sizes)



Scheduling Options

- schedule(static): block allocation of about n/t contiguous iterations to each thread
- schedule(static,C): interleaved allocation of chunks of size C to threads
- schedule(dynamic): dynamic one-at-a-time allocation of iterations to threads
- schedule(dynamic,C): dynamic allocation of C iterations at a time to threads

schedule (static, C)

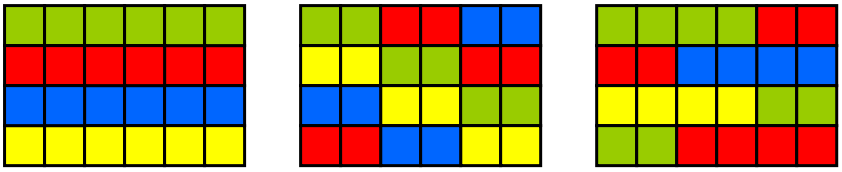
IN GENERAL: Small chunks allow finer grained control on workload

- Specifying a size for data chunks the loop is statically split between threads in an interleaved fashion

```
int main () {
  int a[24];
  int i;

#pragma omp parallel
  ...
#pragma omp for schedule (static)
  for (i = 0; i < 24; i++)
    a[i] = i;

  ...
}
```

```
int main () {
  int a[24];
  int i;

#pragma omp parallel
  ...
#pragma omp for schedule (static, 2)
  for (i = 0; i < 24; i++)
    a[i] = i;

  ...
}
```

```
int main () {
  int a[24];
  int i;

#pragma omp parallel
  ...
#pragma omp for schedule (static, 4)
  for (i = 0; i < 24; i++)
    a[i] = i;

  ...
}
```



Scheduling Options

- schedule(static): block allocation of about n/t contiguous iterations to each thread
- schedule(static,C): interleaved allocation of chunks of size C to threads
- schedule(dynamic): dynamic one-at-a-time allocation of iterations to threads
- schedule(dynamic,C): dynamic allocation of C iterations at a time to threads

SPLITTING LOOP ITERATIONS – schedule (dynamic, C)

- The **dynamic** schedule is appropriate for the case of a **for** construct with the iterations requiring varying, or even unpredictable, amounts of work
- Iterations are assigned one at a time to threads as they become available. This requires a strict cooperation with the runtime
- Runtime overhead can be reduced by specifying a chunk size *k* greater than 1, so that threads are assigned *k* at a time until fewer than k remain

```
int _suif_start ()
{
  int i;
  int n;
  int x[10000];

#pragma omp parallel
        #pragma omp for schedule (dynamic, 400) nowait
        for (i = 0; i < 10000; i++)
            x[i] = i;

        return x[10];
}
                                                    1,18          Top
```

SPL C)

- The
the
- Iter
Thi
- Ru
tha

```
_suif_start.omp_fn.0 (.omp_data_i)
{
  _Bool D.2607;
  _Bool D.2606;
  long int .iend0.5;
  long int .istart0.4;

<bb 2>:
  D.2605 = __builtin_GOMP_loop_dynamic_start (0, 10000, 1, 400, &.istart0.4, &.iend0.5);
  if (D.2605) goto <L1>; else goto <L4>;

<L4>:;
  __builtin_GOMP_loop_end_nowait ();
  return;

<L1>:;
  i = .istart0.4;
  D.2600 = .iend0.5;

<L2>:;
  i.0 = i;
  D.2594 = .omp_data_i->x;
  i.2 = i.0;
  (*D.2594)[i.2] = i;
  i = i + 1;
  D.2606 = i < D.2600;
  if (D.2606) goto <L2>; else goto <L3>;

<L3>:;
  D.2607 = __builtin_GOMP_loop_dynamic_next (&.istart0.4, &.iend0.5);
  if (D.2607) goto <L1>; else goto <L4>;

}
                                                        7,0-1         All
```

Call runtime

Iteration step

Chunk size

Absolute lower and upper bounds

Retrieve lower and upper bounds for current thread's first iteration

SPL C)

- The
the
- Iter
Thi
- Ru
tha

```
_suif_start.omp_fn.0 (.omp_data_i)
{
  _Bool D.2607;
  _Bool D.2606;
  long int .iend0.5;
  long int .istart0.4;

<bb 2>:
  D.2605 = __builtin_GOMP_loop_dynamic_start (0, 10000, 1, 400, &.istart0.4, &.iend0.5);
  if (D.2605) goto <L1>; else goto <L4>;

<L4>:;
  __builtin_GOMP_loop_end_nowait ();
  return;

<L1>:;
  i = .istart0.4;
  D.2600 = .iend0.5;

<L2>:;
  i.0 = i;
  D.2594 = .omp_data_i->x;
  i.2 = i.0;
  (*D.2594)[i.2] = i;
  i = i + 1;
  D.2606 = i < D.2600;
  if (D.2606) goto <L2>; else goto <L3>;

<L3>:;
  D.2607 = __builtin_GOMP_loop_dynamic_next (&.istart0.4, &.iend0.5);
  if (D.2607) goto <L1>; else goto <L4>;

}
                                                        7,0-1         All
```

Execute loop body over this iteration space..

Retrieve lower and upper bounds for current thread's first iteration

..then compute next chunk's iteration space

Scheduling Options (cont.)

- schedule(guided, C): dynamic allocation of chunks to tasks using guided self-scheduling heuristic. Initial chunks are bigger, later chunks are smaller, minimum chunk size is C.
- schedule(guided): guided self-scheduling with minimum chunk size 1
- schedule(runtime): schedule chosen at run-time based on value of OMP_SCHEDULE; Unix example:

```
setenv OMP_SCHEDULE "static,1"
```

SPLITTING LOOP ITERATIONS – schedule (guided, C)

- The **guided** schedule is appropriate for the case in which the threads may arrive at varying times at a **for** construct with each iterations requiring about the same amounts of work
- This can happen if, for example, the **for** construct is preceded by one or more **for** constructs with nowait clauses
- The interaction with the runtime works much like the dynamic schedule, but the size of chunks is computed dividing remaining iterations among threads, and considering C as a minimum size for the chunk

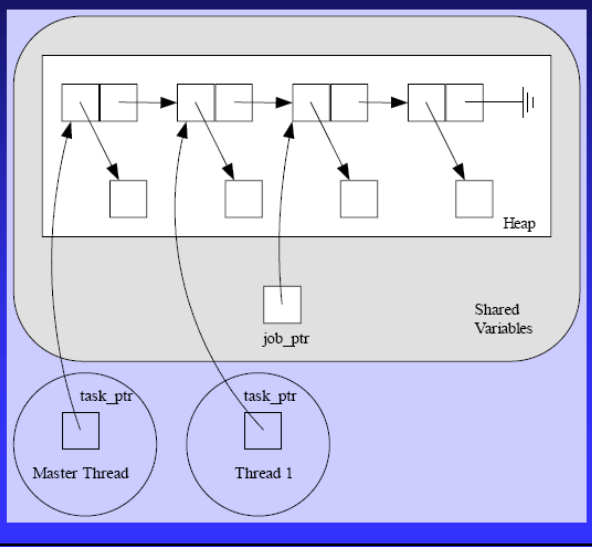
Scheduling Options (cont.)

- schedule(guided, C): dynamic allocation of chunks to tasks using guided self-scheduling heuristic. Initial chunks are bigger, later chunks are smaller, minimum chunk size is C.
- schedule(guided): guided self-scheduling with minimum chunk size 1
- schedule(runtime): schedule chosen at run-time based on value of OMP_SCHEDULE; Unix example:
  `setenv OMP_SCHEDULE "static,1"`

More General Data Parallelism

- Our focus has been on the parallelization of **`for`** loops
- Other opportunities for data parallelism
  - processing items on a "to do" list
  - **`for`** loop + additional code outside of loop

Processing a "To Do" List



Sequential Code (1/2)

```
int main (int argc, char *argv[])
{
   struct job_struct  *job_ptr;
   struct task_struct *task_ptr;

   ...
   task_ptr = get_next_task (&job_ptr);
   while (task_ptr != NULL) {
      complete_task (task_ptr);
      task_ptr = get_next_task (&job_ptr);
   }
   ...
}
```

Sequential Code (2/2)

```
char *get_next_task(struct job_struct
                                **job_ptr) {
   struct task_struct *answer;

   if (*job_ptr == NULL) answer = NULL;
   else {
      answer = (*job_ptr)->task;
      *job_ptr = (*job_ptr)->next;
   }
   return answer;
}
```


Parallelization Strategy

- Every thread should repeatedly take next task from list and complete it, until there are no more tasks
- We must ensure no two threads take same take from the list; i.e., must declare a critical section

Use of `parallel` Pragma

```
#pragma omp parallel private(task_ptr)
{
   task_ptr = get_next_task (&job_ptr);
   while (task_ptr != NULL) {
      complete_task (task_ptr);
      task_ptr = get_next_task (&job_ptr);
   }
}
```


Critical Section for `get_next_task`

```
char *get_next_task(struct job_struct
                              **job_ptr) {
   struct task_struct *answer;
#pragma omp critical
   {
   if (*job_ptr == NULL) answer = NULL;
   else {
      answer = (*job_ptr)->task;
      *job_ptr = (*job_ptr)->next;
   }
   }
   return answer;
}
```

Example Use of for Pragma

```
#pragma omp parallel private(i,j)
for (i = 0; i < m; i++) {
   low = a[i];
   high = b[i];
   if (low > high) {
      printf ("Exiting (%d)\n", i);
      break;
   }
#pragma omp for
   for (j = low; j < high; j++)
      c[j] = (c[j] - a[i])/b[i];
}
```


single Pragma

- Suppose we only want to see the output once
- The **`single`** pragma directs compiler that only a single thread should execute the block of code the pragma precedes
- Syntax:

```
#pragma omp single
```

Use of single Pragma

```
#pragma omp parallel private(i,j)
for (i = 0; i < m; i++) {
   low = a[i];
   high = b[i];
   if (low > high) {
#pragma omp single
      printf ("Exiting (%d)\n", i);
      break;
   }
#pragma omp for
   for (j = low; j < high; j++)
      c[j] = (c[j] - a[i])/b[i];
}
```


nowait Clause

- Compiler puts a barrier synchronization at end of every parallel for statement
- In our example, this is necessary: if a thread leaves loop and changes **`low`** or **`high`**, it may affect behavior of another thread
- If we make these private variables, then it would be okay to let threads move ahead, which could reduce execution time

Use of nowait Clause

```
#pragma omp parallel private(i,j,low,high)
for (i = 0; i < m; i++) {
   low = a[i];
   high = b[i];
   if (low > high) {
#pragma omp single
      printf ("Exiting (%d)\n", i);
      break;
   }
#pragma omp for nowait
   for (j = low; j < high; j++)
      c[j] = (c[j] - a[i])/b[i];
}
```

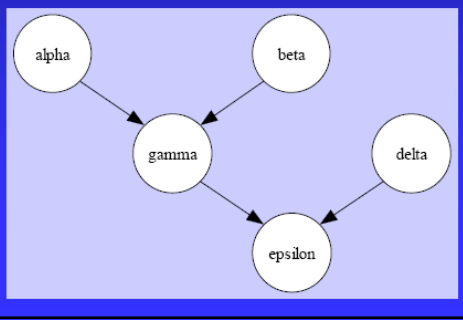

Functional Parallelism

- To this point all of our focus has been on exploiting data parallelism
- OpenMP allows us to assign different threads to different portions of code (functional parallelism)

Functional Parallelism Example

```
v = alpha();
w = beta();
x = gamma(v, w);
y = delta();
printf ("%6.2f\n", epsilon(x,y));
```

May execute alpha, beta, and delta in parallel



parallel sections Pragma

- Precedes a block of *k* blocks of code that may be executed concurrently by *k* threads
- Syntax:

```
#pragma omp parallel sections
```

section Pragma

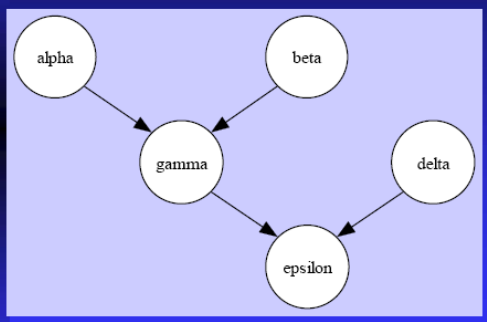
- Precedes each block of code within the encompassing block preceded by the parallel sections pragma
- May be omitted for first parallel section after the parallel sections pragma
- Syntax:

```
#pragma omp section
```


Example of `parallel sections`

```
#pragma omp parallel sections
   {
#pragma omp section  /* Optional */
      v = alpha();
#pragma omp section
      w = beta();
#pragma omp section
      y = delta();
   }
   x = gamma(v, w);
   printf ("%6.2f\n", epsilon(x,y));
```

Another Approach



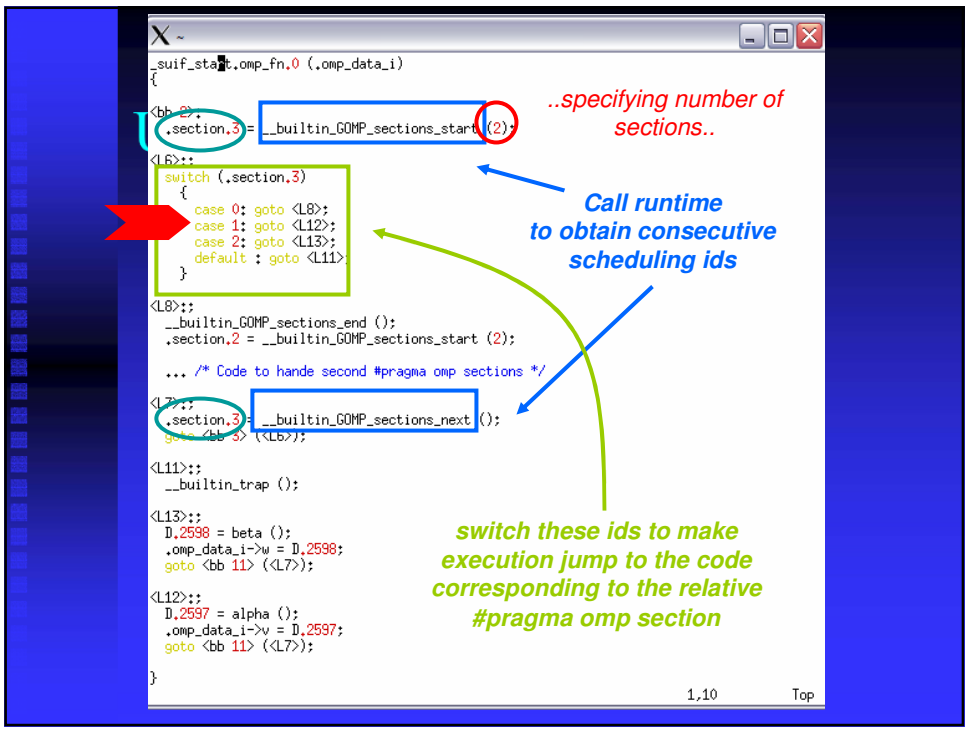
Execute alpha and beta in parallel. Execute gamma and delta in parallel.

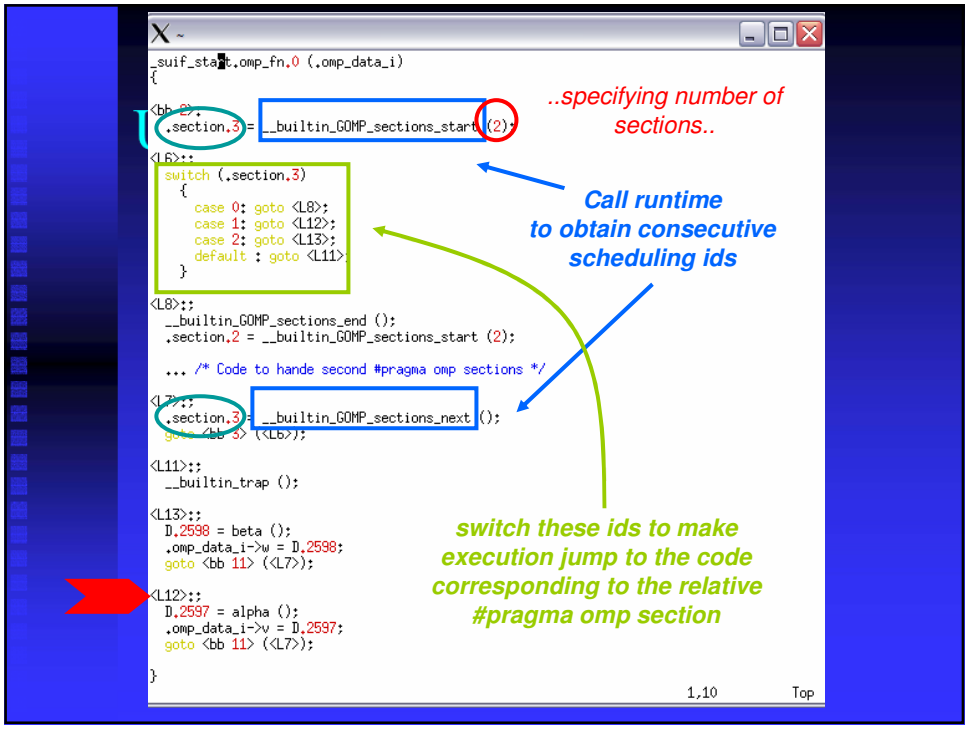
sections Pragma

- Appears inside a parallel block of code
- Has same meaning as the **`parallel sections`** pragma
- If multiple **`sections`** pragmas inside one parallel block, may reduce fork/join costs

Use of sections Pragma

```
#pragma omp parallel
   {
   #pragma omp sections
      {
         v = alpha();
      #pragma omp section
         w = beta();
      }
   #pragma omp sections
      {
         x = gamma(v, w);
      #pragma omp section
         y = delta();
      }
   }
   printf ("%6.2f\n", epsilon(x,y));
```





Summary (1/3)

- OpenMP an API for shared-memory parallel programming
- Shared-memory model based on fork/join parallelism
- Data parallelism
  - parallel for pragma
  - reduction clause

Summary (2/3)

- Functional parallelism (parallel sections pragma)
- SPMD-style programming (parallel pragma)
- Critical sections (critical pragma)
- Enhancing performance of parallel for loops
  - Inverting loops
  - Conditionally parallelizing loops
  - Changing loop scheduling

Summary (3/3)

Characteristic	*OpenMP*	*MPI*
Suitable for multiprocessors	Yes	Yes
Suitable for multicomputers	No	Yes
Supports incremental parallelization	Yes	No
Minimal extra code	Yes	No
Explicit control of memory hierarchy	No	Yes