

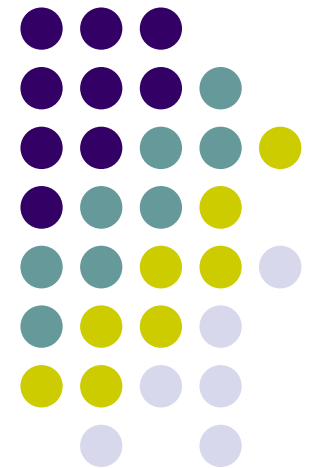
# ECE/ME/EMA/CS 759

## High Performance Computing for Engineering Applications

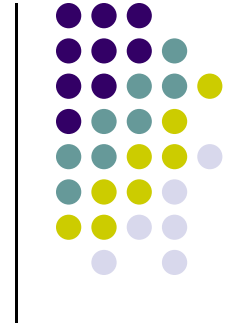
---

Work Sharing in OpenMP

November 2, 2015  
Lecture 21



# Quote of the Day



“Success consists of going from failure to failure without loss of enthusiasm”

-- Sir Winston Churchill

1874-1965

# Before We Get Started



- Issues covered last time:
  - CUDA libraries
  - Multi-core parallel computing w/ OpenMP – get started
    - Discussed the OpenMP execution model
    - Discussed concept of “parallel region”
- Today’s topics
  - Work sharing in OpenMP
    - parallel for constructs
    - parallel sections
    - parallel task constructs
- Other issues:
  - Assignment: HW07 - due on Wd, Nov. 4 at 11:59 PM
  - Final project proposal: 2 pages, due on 11/13 at 11:59 pm (Learn@UW dropbox)

# Work Plan



- What is OpenMP?

Parallel regions

**Work sharing** ←

Data environment

Synchronization

- Advanced topics

# Work Sharing



- **Work sharing** is the general term used in OpenMP to describe distribution of work across threads
- Three primary avenues for work sharing in OpenMP:
  - “omp for” construct
  - “omp sections” construct
  - “omp task” construct

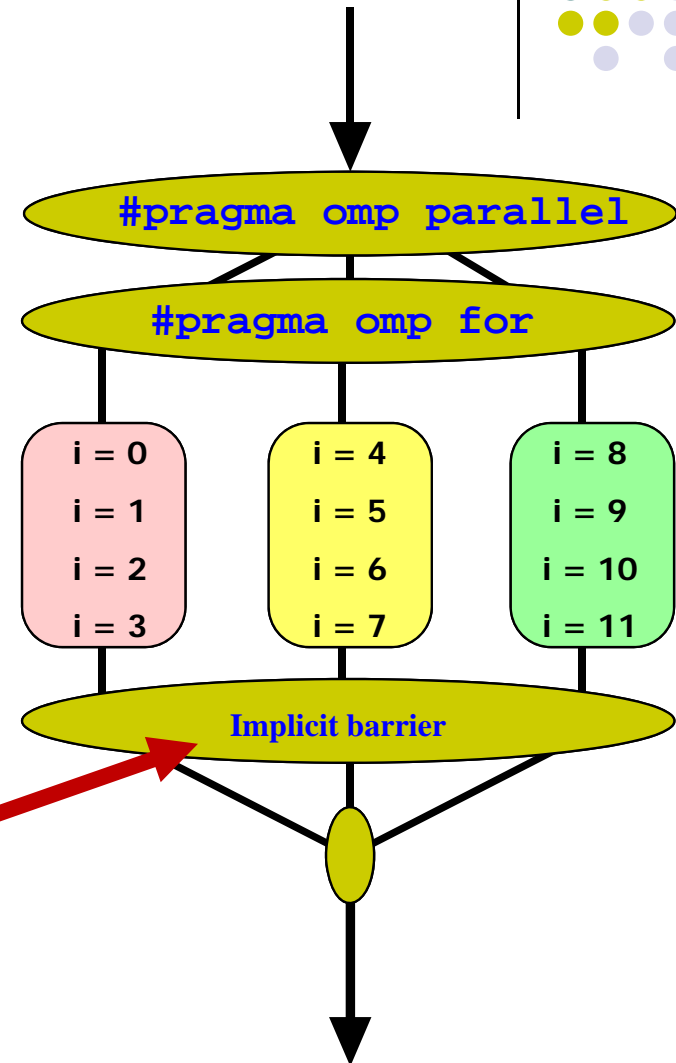
Each of them automatically divides work among threads

# “omp for” construct

```
// assume N=12  
#pragma omp parallel  
#pragma omp for  
    for(i = 0; i < N; i++)  
        c[i] = a[i] + b[i];
```

- Threads are assigned an independent set of iterations
- Threads must wait at the end of work-sharing construct

[example above assumes three threads are in the thread team]



# Combining Constructs

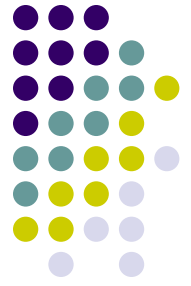


- These two code segments are equivalent

```
#pragma omp parallel
{
    #pragma omp for
    for ( int i=0;i< MAX; i++) {
        res[i] = huge();
    }
}
```

```
#pragma omp parallel for
    for (int i=0;i< MAX; i++) {
        res[i] = huge();
    }
```

# OpenMP: Important Remark



- One of the key tenets of OpenMP is that of **data independence** across parallel jobs
- Specifically, when distributing work among parallel threads it is assumed that there is no data dependency
- Since you place the **omp parallel** directive around some code, it is your responsibility to make sure that data dependency is ruled out
  - Compilers many times can't identify data dependency between what might look as independent parallel jobs



# The Private Clause



- Reproduces the variable for each task
  - By declaring a variable as being private it means that each thread will have a private copy of that variable
    - The value that Thread\_1 stores in x is different than value that Thread\_2 stores in variable x
  - Variables are un-initialized; C++ object is default constructed

```
void* work(float* c, int N) {  
    float x, y;  
    int i;  
    #pragma omp parallel for private(x,y)  
    for(i=0; i<N; i++) {  
        x = a[i]; y = b[i];  
        c[i] = x + y;  
    }  
}
```

# The `schedule` Clause



- The `schedule` clause affects how loop iterations are mapped onto threads

`schedule(static [,chunk])`

- Blocks of iterations of size “chunk” assigned to each thread
- Round robin distribution
- Low overhead, may cause load imbalance

`schedule(dynamic[,chunk])`

- Threads grab “chunk” iterations
- When done with iterations, thread requests next “chunk”
- Higher threading overhead, can reduce load imbalance

`schedule(guided[,chunk])`

- Dynamic schedule starting with large block
- Size of the blocks shrink; no smaller than “chunk”



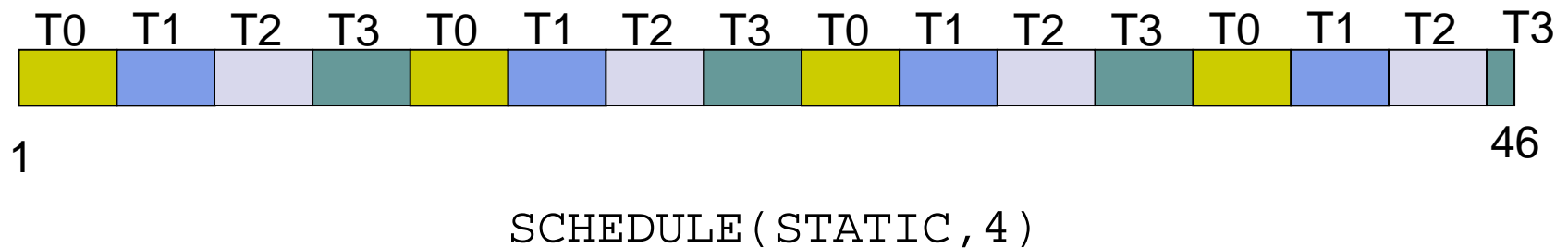
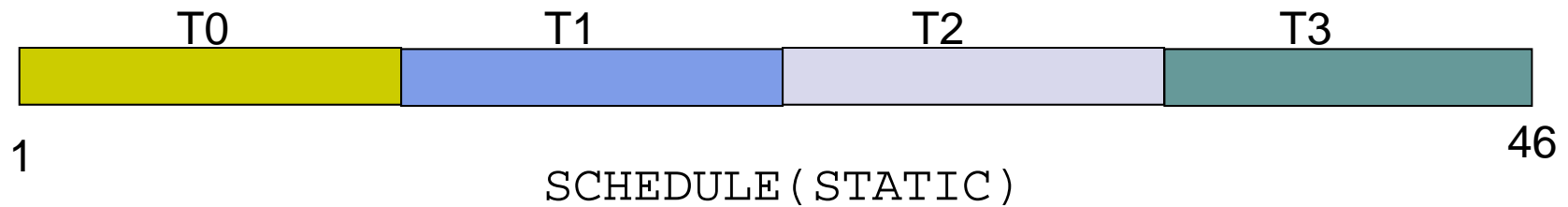
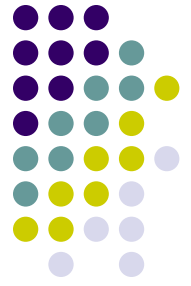
# schedule Clause Example

```
#pragma omp parallel for schedule (static, 8)
  for( int i = start; i <= end; i += 2 )
  {
    if ( TestForPrime(i) ) gPrimesFound++;
  }
```

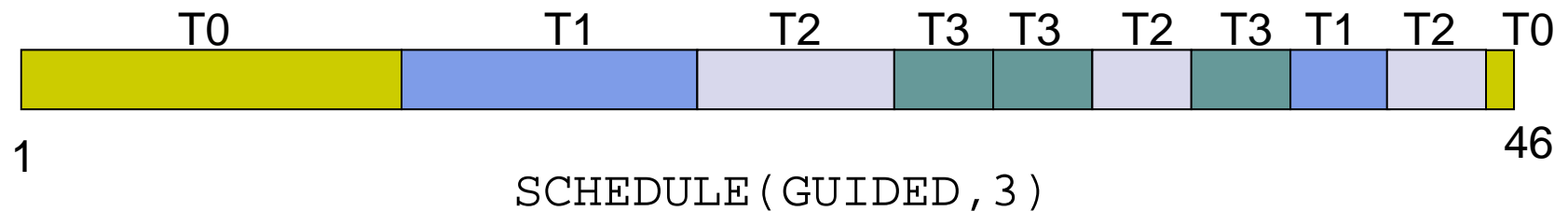
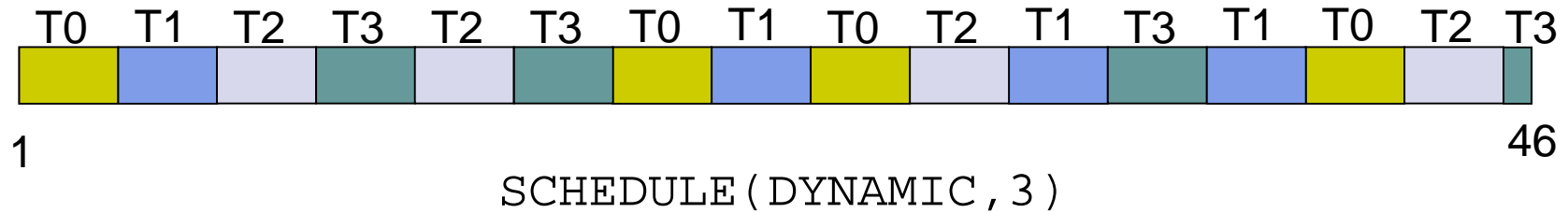
- Iterations are divided into chunks of 8
- If start = 3, then first chunk is

$\mathbf{i}=\{3,5,7,9,11,13,15,17\}$

# Example, STATIC Schematic [assume 4 cores/4 threads]



# Examples, DYNAMIC and GUIDED Schematics [assume 4 cores/4 threads]



# Parallel for Construct: Choosing a Schedule



- STATIC is best for balanced loops – least overhead.
- STATIC,n good for loops with mild or smooth load imbalance
  - Prone to introduce “false sharing” (discussed later)
- DYNAMIC useful if iterations have widely varying loads
  - Prove to adversely impact data locality (cache misses)
- GUIDED often less expensive than DYNAMIC
  - Beware of loops where first iterations are the most expensive



# Work Plan

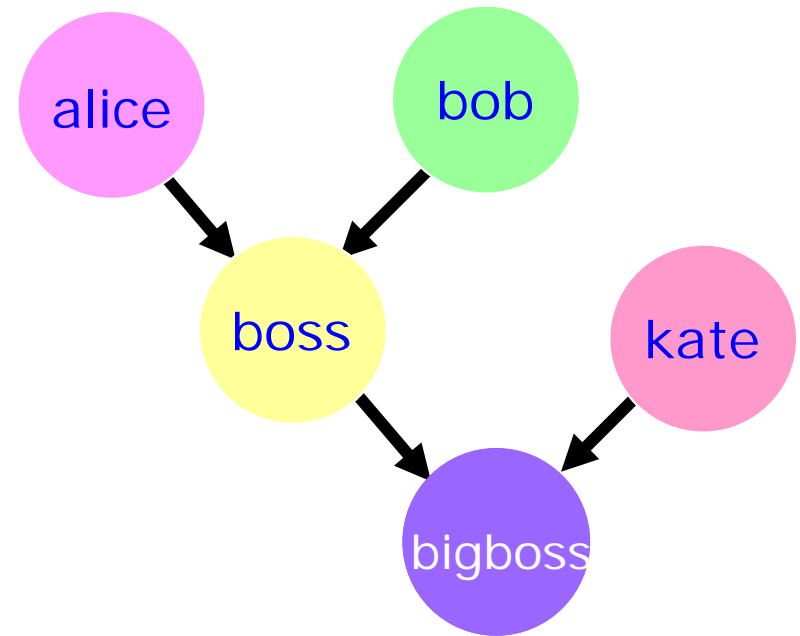
- What is OpenMP?
  - Parallel regions
  - Work sharing – Parallel Sections
  - Data environment
  - Synchronization
- Advanced topics

# Function Level Parallelism



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

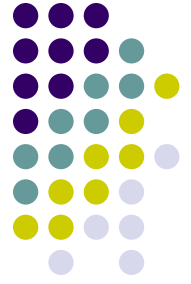
**alice**, **bob**, and **kate**  
can be computed  
in parallel





# omp sections

There is an “s” here



- `#pragma omp sections`
- Must be inside a parallel region
- Precedes a code block containing  $N$  sub-blocks of code that may be executed concurrently by  $N$  threads
- Encompasses each `omp section`, see below

There is no “s” here

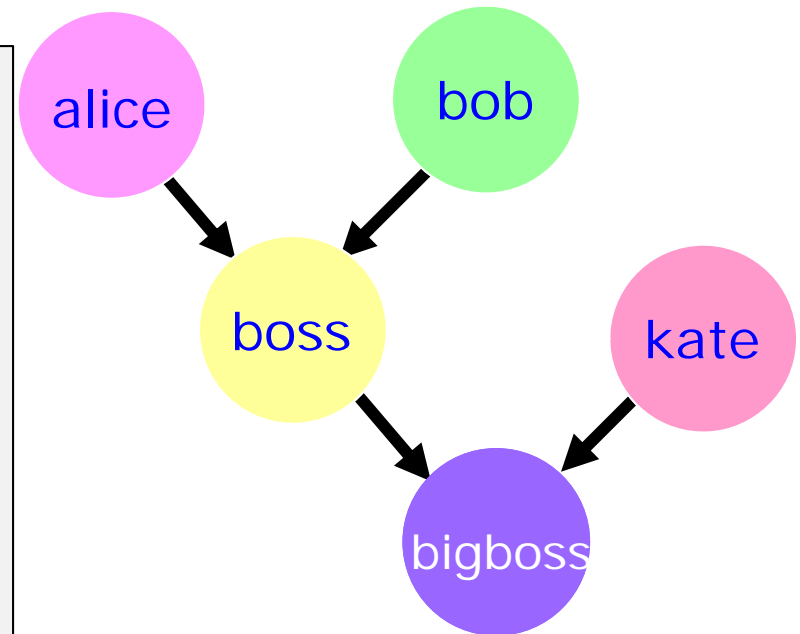
- `#pragma omp section`
- Precedes each sub-block of code within the encompassing block described above
- Enclosed program segments are distributed for parallel execution among available threads

# Functional Level Parallelism Using omp sections



```
#pragma omp parallel sections
{
#pragma omp section
    a = alice();
#pragma omp section
    b = bob();
#pragma omp section
    k = kate();
}

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

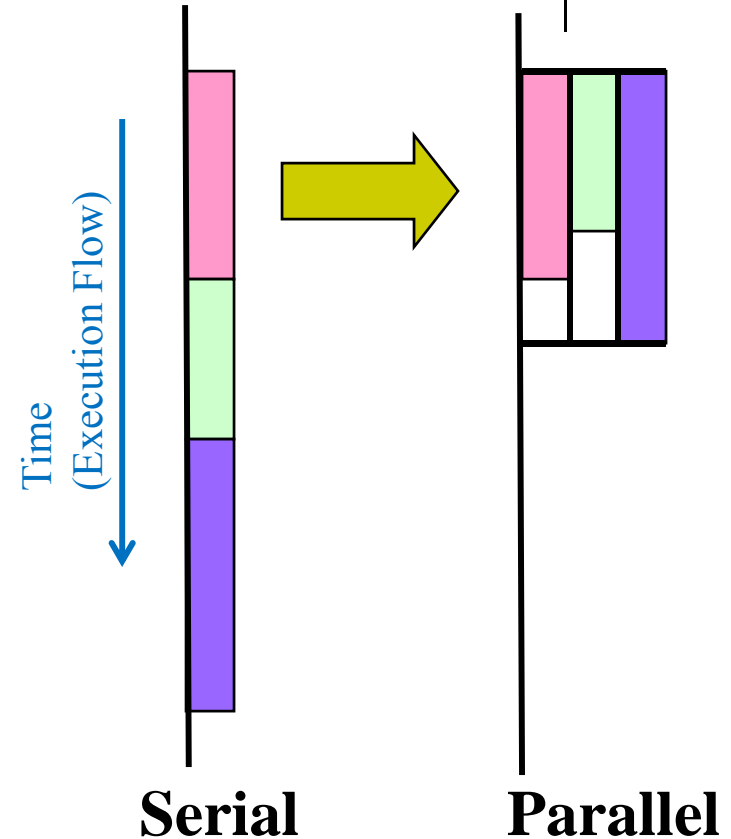


# Advantage of Parallel Sections



- Independent sections of code can execute concurrently → reduces execution time

```
#pragma omp parallel sections
{
  #pragma omp section
  phase1();
  #pragma omp section
  phase2();
  #pragma omp section
  phase3();
}
```



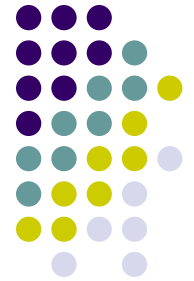
The pink and green tasks are executed at no additional time-penalty in the shadow of the blue task

# sections, Example

```
#include <stdio.h>
#include <omp.h>

int main() {
    printf("Start with 2 procs only.. \n\n");
    #pragma omp parallel sections num_threads(2)
    {
        #pragma omp section
        {
            printf("Start work 1\n");
            double startTime = omp_get_wtime();
            while( (omp_get_wtime() - startTime) < 2.0);
            printf("Finish work 1\n");
        }
        #pragma omp section
        {
            printf("Start work 2\n");
            double startTime = omp_get_wtime();
            while( (omp_get_wtime() - startTime) < 2.0);
            printf("Finish work 2\n");
        }
        #pragma omp section
        {
            printf("Start work 3\n");
            double startTime = omp_get_wtime();
            while( (omp_get_wtime() - startTime) < 2.0);
            printf("Finish work 3\n");
        }
    }
    return 0;
}
```

# sections, Example: 2 threads

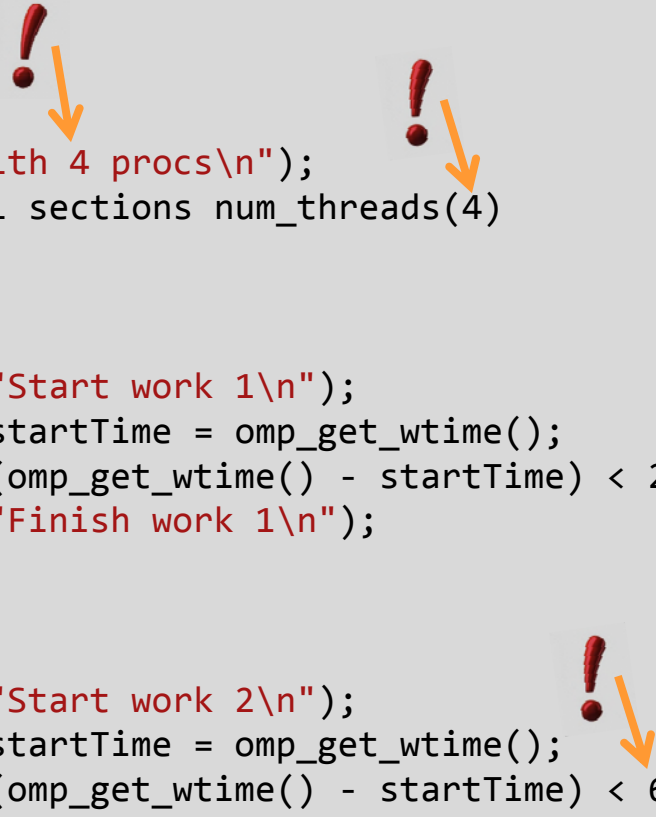


```
ca. C:\Windows\system32\cmd.exe
Start with 2 procs only...
Start work 1
Start work 2
Finish work 1
Start work 3
Finish work 2
Finish work 3
Press any key to continue . . . _
```

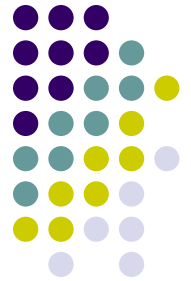
# sections, Example

```
#include <stdio.h>
#include <omp.h>

int main() {
    printf("Start with 4 procs\n");
    #pragma omp parallel sections num_threads(4)
    {
        #pragma omp section
        {
            printf("Start work 1\n");
            double startTime = omp_get_wtime();
            while( (omp_get_wtime() - startTime) < 2.0);
            printf("Finish work 1\n");
        }
        #pragma omp section
        {
            printf("Start work 2\n");
            double startTime = omp_get_wtime();
            while( (omp_get_wtime() - startTime) < 6.0);
            printf("Finish work 2\n");
        }
        #pragma omp section
        {
            printf("Start work 3\n");
            double startTime = omp_get_wtime();
            while( (omp_get_wtime() - startTime) < 2.0);
            printf("Finish work 3\n");
        }
    }
    return 0;
}
```



# sections, Example: 4 threads



```
C:\Windows\system32\cmd.exe
Start with 4 procs
Start work 1
Start work 2
Start work 3
Finish work 1
Finish work 3
Finish work 2
Press any key to continue . . .
```

# Work Plan



- What is OpenMP?
  - Parallel regions
  - Work sharing – Tasks
  - Data environment
  - Synchronization
- Advanced topics



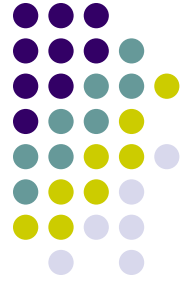
# OpenMP Tasks



- **Task** – Most important feature added as of OpenMP 3.0 version
- Allows parallelization of irregular problems
  - Unbounded loops (not clear how many iterations – see next example)
  - Recursive algorithms
  - Producer/consumer

[Preamble]

# Example, Static Scheduling



```
#include <stdio.h>
#include <omp.h>

int main() {
#pragma omp parallel for schedule (static)
  for (int i = 0; i<= 14; i++){
    printf("I'm thread %d working on iteration %d\n", omp_get_thread_num(), i);
  }
  printf("All done here...\n");
}
```

```
C:\Windows\system32\cmd.exe
I'm thread 0 working on iteration 0
I'm thread 0 working on iteration 1
I'm thread 0 working on iteration 2
I'm thread 0 working on iteration 3
I'm thread 2 working on iteration 4
I'm thread 2 working on iteration 5
I'm thread 2 working on iteration 6
I'm thread 2 working on iteration 7
I'm thread 1 working on iteration 8
I'm thread 1 working on iteration 9
I'm thread 1 working on iteration 10
I'm thread 1 working on iteration 11
I'm thread 3 working on iteration 12
I'm thread 3 working on iteration 13
I'm thread 3 working on iteration 14
All done here...
Press any key to continue . . . -
```

```

#include <stdio.h>
#include <omp.h>

int getUpperBound(int i, int N){
    if (i <= N)
        return N;
    else
        return 0;
}

```

```

int main() {
    int upperB = 14;

```

```

    for (int i = 0; i <= getUpperBound(i,upperB); i++){
        printf("I'm thread %d working on iteration %d\n", omp_get_thread_num(), i);
    }
    printf("All done here...\n");
}

```

```

C:\Windows\system32\cmd.exe
I'm thread 0 working on iteration 0
I'm thread 0 working on iteration 1
I'm thread 0 working on iteration 2
I'm thread 0 working on iteration 3
I'm thread 0 working on iteration 4
I'm thread 0 working on iteration 5
I'm thread 0 working on iteration 6
I'm thread 0 working on iteration 7
I'm thread 0 working on iteration 8
I'm thread 0 working on iteration 9
I'm thread 0 working on iteration 10
I'm thread 0 working on iteration 11
I'm thread 0 working on iteration 12
I'm thread 0 working on iteration 13
I'm thread 0 working on iteration 14
All done here...
Press any key to continue . . . _

```

Code run on one thread, sequential execution, no OpenMP



```
#include <stdio.h>
#include <omp.h>

int getUpperBound(int i, int N){
    if (i <= N)
        return N;
    else
        return 0;
}

int main() {
    int upperB = 14;

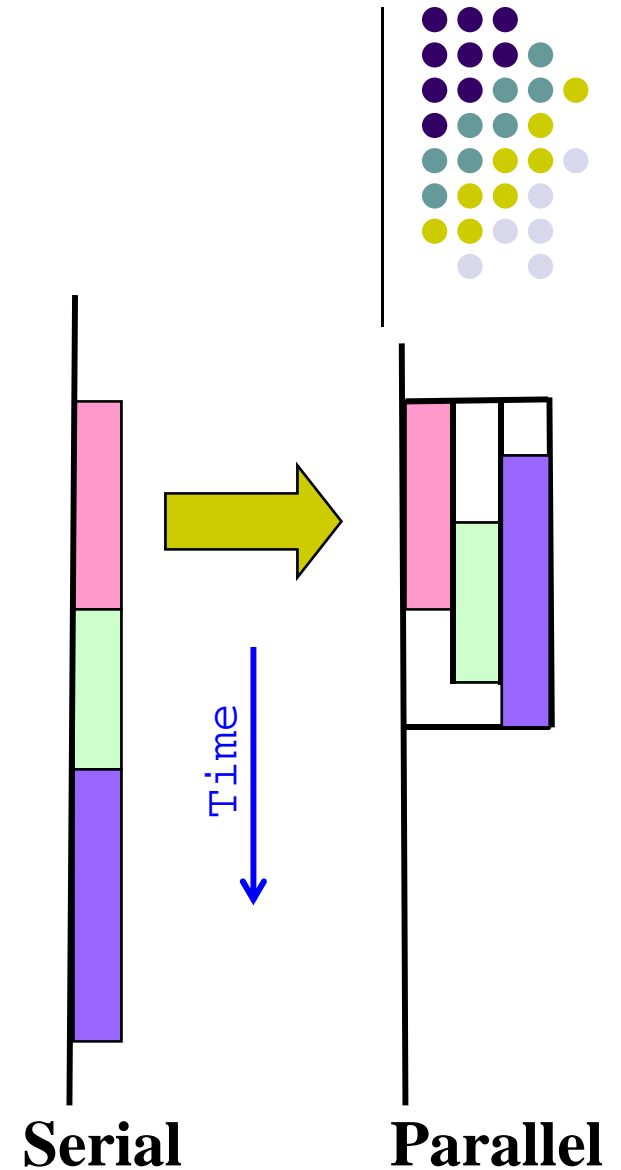
#pragma omp parallel for schedule (static)
    for (int i = 0; i <= getUpperBound(i,upperB); i++){
        printf("I'm thread %d working on iteration %d\n", omp_get_thread_num(), i);
    }
    printf("All done here...\n");
}
```

```
1>----- Build started: Project: TestOpenMP, Configuration: Debug Win32 -----
1> driverOpenMP.cpp
1>c:\users\negrut\bin\vs13projects\testopenmp\testopenmp\driveropenmp.cpp(15):
error C3017: termination test in OpenMP 'for' statement has improper form
===== Build: 0 succeeded, 1 failed, 0 up-to-date, 0 skipped =====
```

[Back to Usual Program]

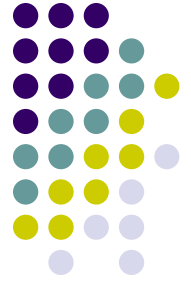
# Tasks: What Are They?

- Tasks are independent units of work
- A thread is assigned to perform a task
- Tasks might be executed immediately or might be deferred
  - The OS & runtime decide which of the above
- Tasks are composed of
  - **code** to execute
  - **data** environment
  - **internal control variables (ICV)**



# Tasks: What Are They?

[More specifics...]



- Code to execute
  - The literal code in your program enclosed by the task directive
- Data environment
  - The shared & private data manipulated by the task
- Internal control variables
  - Thread scheduling and environment variables
- More formal definition: A task is a specific instance of executable code and its data environment, generated when a thread encounters a **task** construct
- Two activities: (1) packaging, and (2) execution
  - A thread packages new instances of a task (code and data)
  - Some thread in the team executes the task at some later time

```

using namespace std ;
typedef list<double> LISTDBL;

void doSomething(LISTDBL::iterator& itrtr) {
    *itrtr *= 2.;
}

int main() {
    LISTDBL test; // default constructor
    LISTDBL::iterator it;

    for( int i=0;i<4;++i)
        for( int j=0;j<8;++j) test.insert(test.end(), pow(10.0,i+1)+j);
    for( it = test.begin(); it!= test.end(); it++ ) cout << *it << endl;

    it = test.begin();
    #pragma omp parallel num_threads(8)
    {
        #pragma omp single
        {
            #pragma omp task firstprivate(it)
            {
                {
                    doSomething(it);
                }
                it++;
            }
        }
    }
    for( it = test.begin(); it != test.end(); it++ ) cout << *it << endl;
    return 0;
}

```

```

#include <omp.h>
#include <list>
#include <iostream>
#include <math.h>

```

The image shows a screenshot of a Linux desktop environment with an IDE (emacs) and a terminal window. The IDE window displays the source code for `testOMP.cpp`, which includes headers for `omp.h`, `list`, `iostream`, and `math.h`. It defines a `LISTDBL` typedef and a `doSomething` function that doubles the value of an iterator. The `main` function creates a `LISTDBL` object, inserts values from  $10 \cdot 2^j$  for  $j$  from 0 to 8, and then uses OpenMP to parallelize the `doSomething` function over 8 threads. The terminal window shows the execution of `./testOMP.exe`, which outputs a list of 20 values. A yellow box highlights the first 10 values (10 to 10000), labeled "Initial values...". A red box highlights the last 10 values (20 to 20010), labeled "Final values...".

```
testOMP.cpp - emacs@euler.msvc.wisc.edu
File Edit Options Buffers Tools C++ Help
#include <omp.h>
#include <list>
#include <iostream>
#include <math.h>

using namespace std ;
typedef list<double> LISTDBL;

void doSomething(LISTDBL::iterator& itrtr) {
    *itrtr *= 2.;
}

int main() {
    LISTDBL test; // default constructor
    LISTDBL::iterator it;

    for( int i=0;i<4;++i)
        for( int j=0;j<8;++j) test.insert(test.end(), pow(10.0,i+1)+j);
    for( it = test.begin();it!= test.end(); it++ ) cout << *it << endl;

    it = test.begin();
    #pragma omp parallel num_threads(8)
    {
        #pragma omp single private(it)
        {
            while( it != test.end() ) {
                #pragma omp task
                {
                    doSomething(it);
                }
                it++;
            }
        }
        for( it = test.begin();it!= test.end(); it++ ) cout << *it << endl;
        return 0;
    }
}

negrut@euler:~/CodeBits
File Edit View Search Terminal Help
[negrut@euler22 CodeBits]$ ./testOMP.exe
10
11
12
13
14
15
16
17
100
101
102
103
104
105
106
107
1000
1001
1002
1003
1004
1005
1006
1007
10000
10001
10002
10003
10004
10005
10006
10007
20
22
24
26
28
30
32
34
200
202
204
206
208
210
212
214
2000
2002
2004
2006
2008
2010
2012
2014
20000
20002
20004
20006
20008
20010

-U:--- testOMP.cpp All L1 (C++/l Abbrev)---

Compile like:
$ g++ -o testOMP.exe testOMP.cpp
```





# More on the task Construct

- A team of threads is created at the `omp parallel` construct
- A single thread is chosen to execute the while loop – let's call this thread “L”
- Thread L runs the while loop, creates tasks, and fetches next pointers
- Each time L crosses the `omp task` construct it generates a new task and has a thread assigned to it
- Each task run by one thread
- All tasks complete at the barrier at the end of the parallel region's construct
- Each task has its own stack space that will be destroyed when the task is completed
  - See example in a little bit

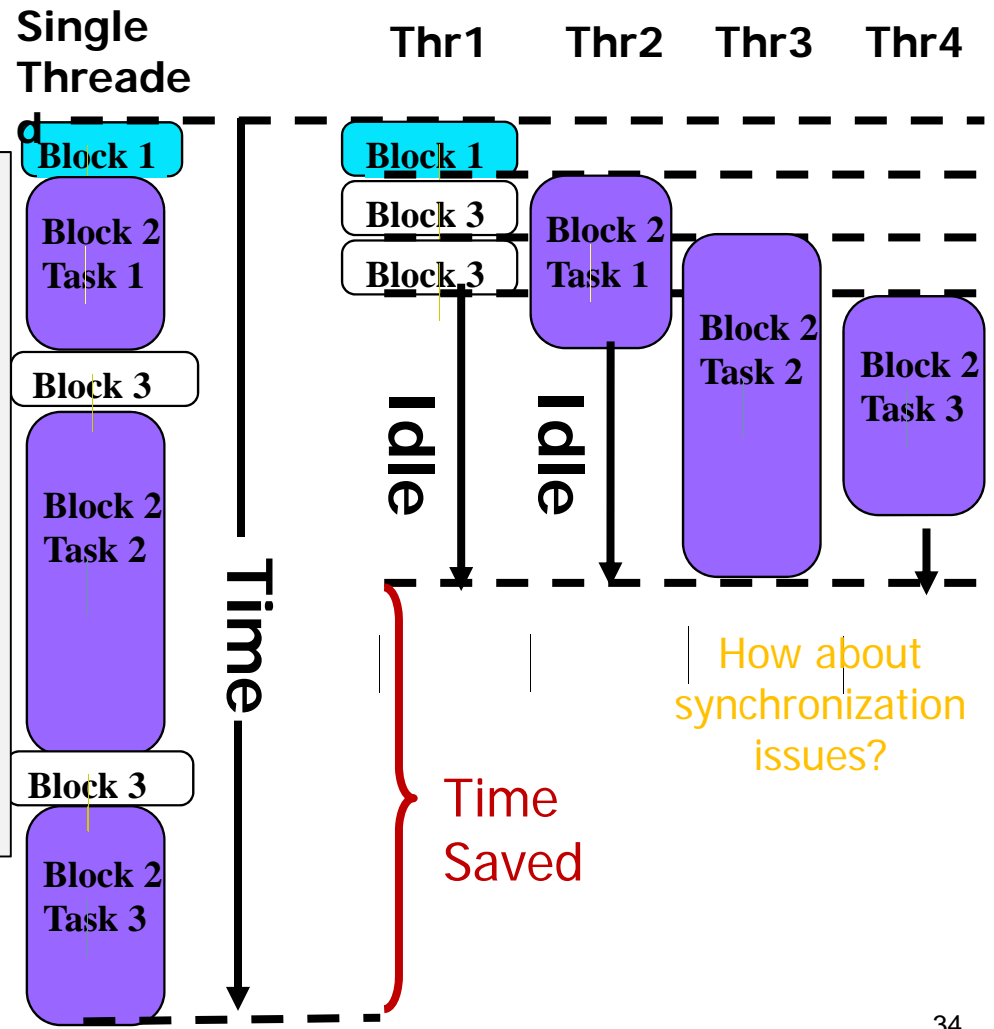
```
#pragma omp parallel
//threads are ready to go now
{
    #pragma omp single
    { // block 1
        node *p = head_of_list;
        while (p!=listEnd) { //block 2
            #pragma omp task firstprivate(p)
            process(p);
            p = p->next; //block 3
        }
    }
}
```

# Why are tasks useful?



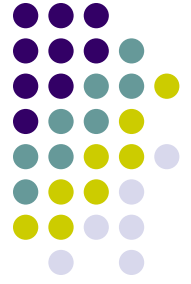
Have potential to parallelize irregular patterns and recursive function calls

```
#pragma omp parallel
//threads are ready to go now
{
  #pragma omp single
  { // block 1
    node *p = head_of_list;
    while (p) { //block 2
      #pragma omp task firstprivate(p)
      {
        process(p);
      }
      p = p->next; //block 3
    }
  }
}
```



# Tasks: Synchronization Issues

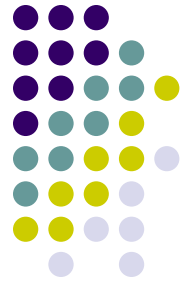
[1/2]



- Tasks are guaranteed to be complete:
  - At the end of a `parallel` region
  - At the directive: `#pragma omp barrier`
    - Threads wait there until all touch the barrier and then they move on (or disappear)
    - Example on next slide
  - At the directive: `#pragma omp taskwait`
    - Execution for a task is suspended until the child tasks spawned are finished
    - Example in a couple of slides

# Tasks: Synchronization Issues

[2/2]



- Showcase below use of
  - `#pragma omp barrier`
- Setup:
  - Assume Task B specifically relies on completion of Task A
  - You need to be in a position to guarantee completion of Task A before invoking the execution of Task B

# Task Completion Example



```
#pragma omp parallel
{
  #pragma omp task
  foo( omp_get_thread_num() );
  #pragma omp barrier
  #pragma omp single
  {
    #pragma omp task
    bar();
  }
}
```

Multiple foo tasks created here – one for each thread

All foo tasks guaranteed to be completed here

One bar task created here

bar task guaranteed to be completed here



# Comments: sections vs. tasks

- **sections** have a “*static*” attribute: many things settled at compile time
- The **tasks** construct is more recent and more sophisticated
  - They have a “*dynamic*” attribute: things are figured out at run time and the construct counts under the hood on the presence of a scheduling agent
  - They can encapsulate any block of code
    - Can handle nested loops and scenarios when the number of jobs is not clear
  - The runtime generates and executes the tasks, either at implicit synchronization points in the program or under explicit control of the programmer
- NOTE: It’s the developer’s responsibility to ensure that different tasks can be executed concurrently; i.e., there is no data dependency

# Work Plan



- What is OpenMP?
  - Parallel regions
  - Work sharing
  - Data scoping**
  - Synchronization**
- **Advanced topics**



# Data Scoping – What’s shared

- OpenMP uses a shared-memory programming model
- **Shared variable** - a variable that can be read or written by multiple threads
- **shared** clause can be used to make items explicitly shared
- Some variables being shared by default
  - Global variables
  - File scope variables
  - Namespace scope variables
  - Variables with heap allocated storage
  - Static variables which are declared in a scope inside the construct



# Data Scoping – What's Private



- Not everything is shared...
  - Examples of implicitly PRIVATE variables:
    - Stack (local) variables in functions called from parallel regions
    - Automatic variables within a statement block
    - Loop iteration variables
    - Implicitly declared private variables within `tasks` will be treated as firstprivate
- **firstprivate**
  - Specifies that each thread should have its own instance of a variable
  - Data initialized using the value of the variable of same name from the master thread

# Example: private vs. firstprivate



```
#include <stdio.h>
#include <omp.h>

int main(void) {
    int i = 10;

    #pragma omp parallel private(i)
    {
        int threadID = omp_get_thread_num();
        printf("thread %d: i = %d\n", threadID, i);
        i = 1000 + threadID;
    }

    printf("i = %d\n", i);

    return 0;
}
```

```
tux-112.cae.wisc.edu - PuTTY
negrut@tux-112:~/Software$
negrut@tux-112:~/Software$
negrut@tux-112:~/Software$ g++ driver.c
pp -fopenmp
negrut@tux-112:~/Software$ ./a.out
thread 1: i = -516449569
thread 3: i = 0
thread 0: i = 0
thread 2: i = 0
i = 10
negrut@tux-112:~/Software$ ./a.out
thread 1: i = -2001814817
thread 0: i = 0
thread 3: i = 0
thread 2: i = 0
i = 10
negrut@tux-112:~/Software$ ./a.out
thread 0: i = 0
thread 3: i = 0
thread 2: i = 0
thread 1: i = 1045736159
i = 10
negrut@tux-112:~/Software$ ./a.out
thread 0: i = 0
thread 3: i = 0
thread 1: i = -596108577
thread 2: i = 0
i = 10
negrut@tux-112:~/Software$ ./a.out
thread 1: i = 967064287
thread 0: i = 0
thread 2: i = 0
thread 3: i = 0
i = 10
negrut@tux-112:~/Software$
```

# Example: private vs. firstprivate



```
#include <stdio.h>
#include <omp.h>

int main(void) {
    int i = 10;

#pragma omp parallel firstprivate(i)
    {
        int threadID = omp_get_thread_num();
        printf("threadID + i = %d\n", threadID+i);
    }

    printf("i = %d\n", i);

    return 0;
}
```

```
C:\Windows\system32\cmd.exe
threadID + i = 10
threadID + i = 11
threadID + i = 12
threadID + i = 13
i = 10
Press any key to continue . . . _
```

# Other Tidbits



- There is a `lastprivate` flavor of `private` variable
  - The enclosing context's version of the variable is set equal to the private version of whichever thread executes the final iteration of the work-sharing construct (`for`, `section`, `task`)

# Data Scoping – The Basic Rule



- When in doubt, explicitly indicate who's what
  - Data scoping: common sources of errors in OpenMP