LAU
الجامعة اللبنانية الأميركية
Lebanese American University

CSC 447: Parallel Programming for Multi-Core and Cluster Systems

Shared Parallel Programming Using OpenMP

Instructor: Haidar M. Harmanani

Spring 2017

Why OpenMP?

- Thread libraries are hard to use
  - Pthreads have many library calls for initialization, synchronization, thread creation, condition variables, etc.
  - Programmer must code with multiple threads in mind
- Synchronization between threads introduces a new dimension of program correctness
- Wouldn't be nice to write serial programs and somehow parallelize them "automatically" through simple directives?

Why OpenMP?

- OpenMP is a parallel programming model for Shared-Memory machines
  - All threads have access to a shared main memory
  - Each thread may have private data.
- Parallelism has to be expressed explicitly by the programmer.
  - Base construct is a Parallel Region which is a team of threads that is provided by the runtime system.
- Using the Worksharing constructs, the work can be distributed among the threads of a team.
  - The Task construct defines an explicit task along with it's data environment. Execution may be deferred.
- To control the parallelization, mutual exclusion as well as thread and task synchronization constructs are available.

Why OpenMP?

```
int main() {

    // Do this part in parallel

    printf( "Hello, World!\n" );

    return 0;
}
```

Why OpenMP? Pthread Version

```
int main() {
  pthread_attr_t attr;
  pthread_t threads[16];
  int tn;

  pthread_attr_init(&attr);
  pthread_attr_setscope(&attr, PTHREAD_SCOPE_SYSTEM);
  for(tn=0; tn<16; tn++) {
    pthread_create(&threads[tn], &attr, SayHello, NULL);
  }
```

```
for (tn=0; tn<16 ; tn++) {
    pthread_join(threads[tn], NULL);
  }
  return 0;
}

void* SayHello(void *foo) {
  printf( "Hello, world!\n" );
  return NULL;
}
```


Why OpenMP?

```
int main()
{
    omp_set_num_threads(16);

    // Do this part in parallel
    #pragma omp parallel
    {
      printf( "Hello, World!\n" );
    }

    return 0;
  }
```

Why OpenMP?

1. Start with *some* algorithm
   - Embarrassing parallelism is helpful, but not necessary
2. Implement serially, *ignoring*:
   - Data Races
   - Synchronization
   - Threading Syntax
3. Test and Debug
4. Automatically (*magically?*) parallelize
   - Expect linear speedup

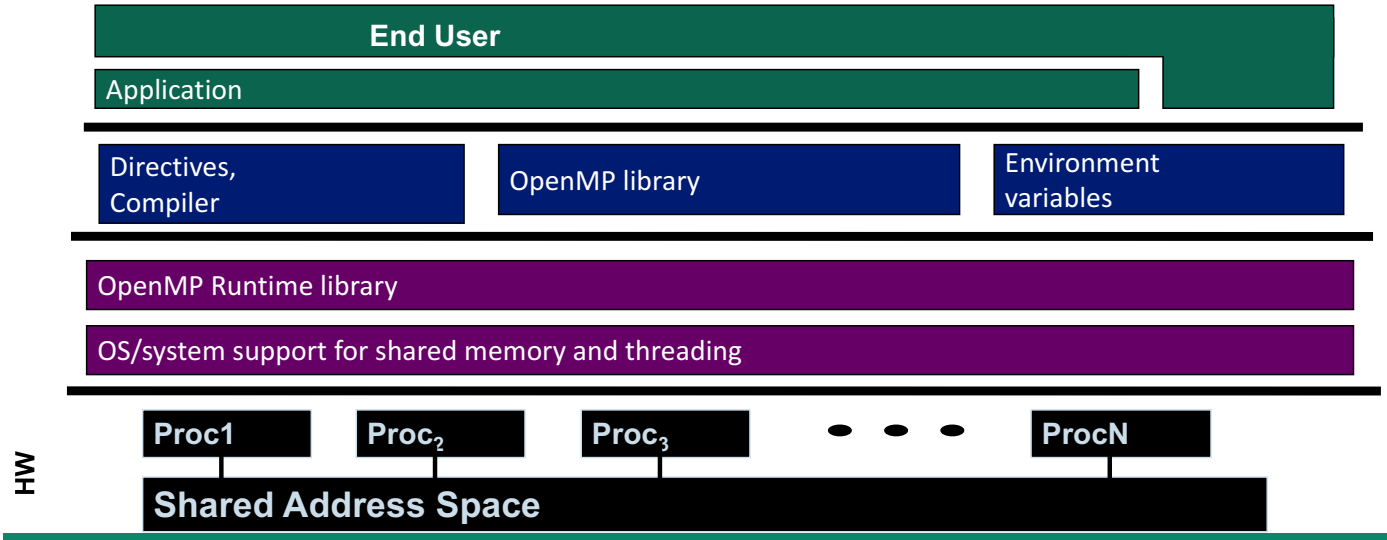
What Is OpenMP?

- Portable, shared-memory threading API
  - Fortran, C, and C++
  - Multi-vendor support for both Linux and Windows
- Standardizes task & loop-level parallelism
- Supports coarse-grained parallelism
- Combines serial and parallel code in single source
- Standardizes ~ 20 years of compiler-directed threading experience

Current spec is OpenMP 4.5 (http://www.openmp.org), 355 Pages

(combined C/C++ and Fortran)

OpenMP: Solution Stack

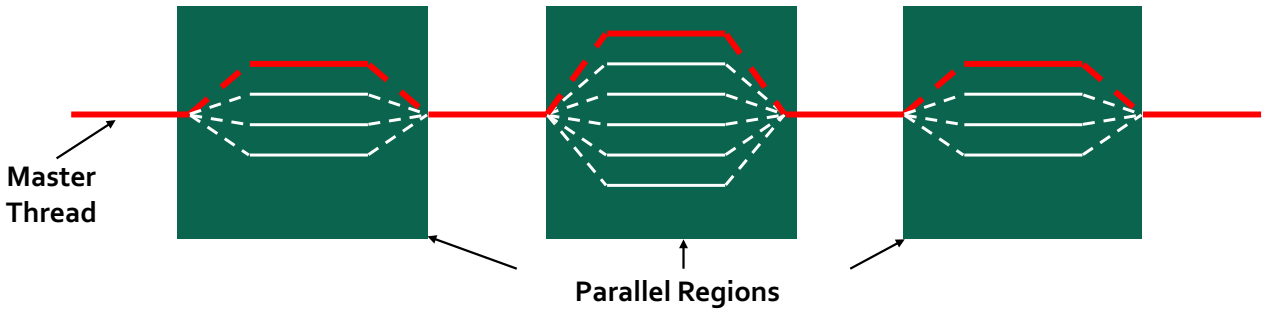


Execution Model

- A thread-based "fork-join" model
  - Initially, a single thread is executed by a master thread.
  - Parallel regions (sections of code) can be executed by multiple threads (a team of threads).
- Parallel directive creates a team of threads with a specified block of code executed by the multiple threads in parallel.
  - The exact number of threads in the team determined by one of several ways.
- Other directives used within a parallel construct to specify parallel for loops and different blocks of code for threads.

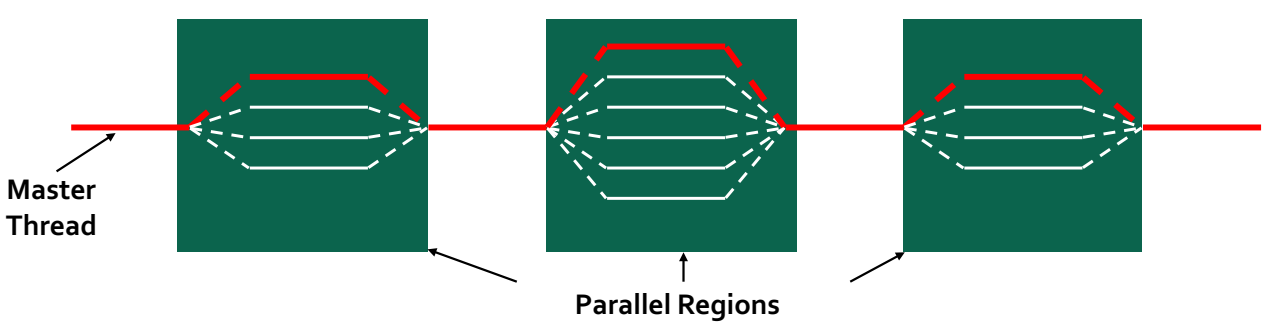
Execution Model

- Fork-Join Parallelism:
  - Master thread spawns a team of threads as needed
  - Parallelism is added incrementally: that is, the sequential program evolves into a parallel program



Execution Model

- Worker threads are spawned at Parallel Regions, together with the Master they form the Team of threads.
- In between Parallel Regions the Worker threads are put to sleep.
- The `OpenMP` Runtime takes care of all thread management work



OpenMP: How is OpenMP typically used?

- OpenMP is usually used to parallelize loops:
  - Find your most time consuming loops.
  - Split them up between threads.

Split-up this loop among multiple threads

```
void main()
{
    double Res[1000];

    for(int i=0;i<1000;i++) {
   do_huge_comp(Res[i]);
    }
}
```

Sequential Program

```
void main()
{
    double Res[1000];
   #pragma omp parallel for
    for(int i=0;i<1000;i++) {
   do_huge_comp(Res[i]);
    }
}
```

Parallel Program

OpenMP: How do Threads Interact?

- OpenMP is a shared memory model.
  - Threads communicate by sharing variables.
- Unintended sharing of data can lead to race conditions:
  - race condition: when the program's outcome changes as the threads are scheduled differently.
- To control race conditions:
  - Use synchronization to protect data conflicts.
- Synchronization is expensive so:
  - Change how data is stored to minimize the need for synchronization.

A Few Syntax Details to Get Started

- Most of the constructs in OpenMP are compiler directives or pragmas
  - For C and C++, the pragmas take the form:

```
#pragma omp directive-name [clause[ [,] clause] ... ] new-line
```


Structured Blocks

- Exactly one entry point at the top
- Exactly one exit point at the bottom
- Branching in or out is not allowed
- Terminating the program is allowed (abort / exit)

Parallel Region & Structured Blocks

- Most OpenMP constructs apply to structured blocks
  - The only "branches" allowed are exit() in C/C++

```
#pragma omp parallel
{
  int id = omp_get_thread_num();

more: res[id] = do_big_job (id);

  if (conv (res[id]) goto more;
}
printf ("All done\n");
```

A structured block

```
if (go_now()) goto more;
#pragma omp parallel
{
  int id = omp_get_thread_num();
more:  res[id] = do_big_job(id);
  if (conv (res[id]) goto done;
  goto more;
}
done: if (!really_done()) goto more;
```

Not a structured block

OpenMP: Parallel Regions

- Create threads in OpenMP using the "omp parallel" pragma.
- For example, To create a 4 thread Parallel region:

Each thread redundantly executes the code within the structured block

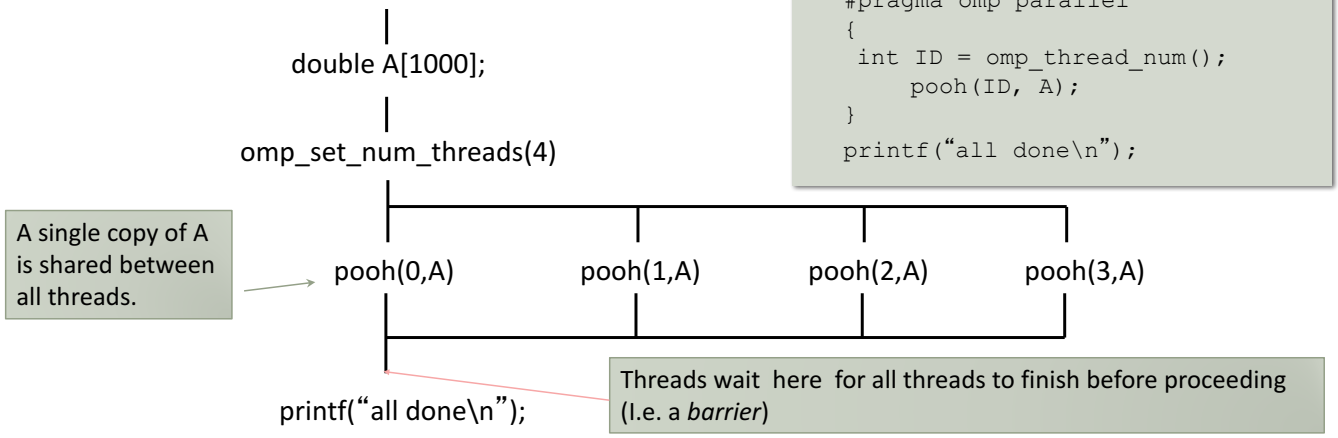
```
double A[1000];
omp_set_num_threads(4);
#pragma omp parallel
{
    int ID = omp_thread_num();
       pooh(ID,A);
}
```

Each thread calls pooh(ID) for ID = 0 to 3

OpenMP: Parallel Regions

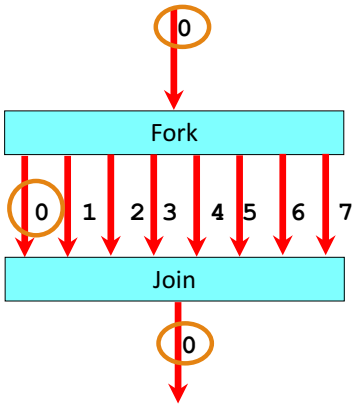
Each thread executes the same code redundantly.

```
double A[1000];
omp_set_num_threads(4);
#pragma omp parallel
{
 int ID = omp_thread_num();
     pooh(ID, A);
}
printf("all done\n");
```



OpenMP: Thread Identification

- Master Thread
  - Thread with ID=0
  - Only thread that exists in sequential regions
  - Depending on implementation, may have special purpose inside parallel regions
  - Some special directives affect only the master thread (like master)



Hello Worlds

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

int main() {
    #pragma omp parallel
    {
       int i;
       int ID = omp_get_thread_num();

        printf("Hello World\n");
        for(i=0;i<6;i++)
            printf("Iter:%d, %d\n",i, ID);
    }
    printf("GoodBye World\n");
}
```

Switches for compiling and linking:

gcc –fopenmp filename

Begin Parallel region

Runtime library function to return a thread ID.

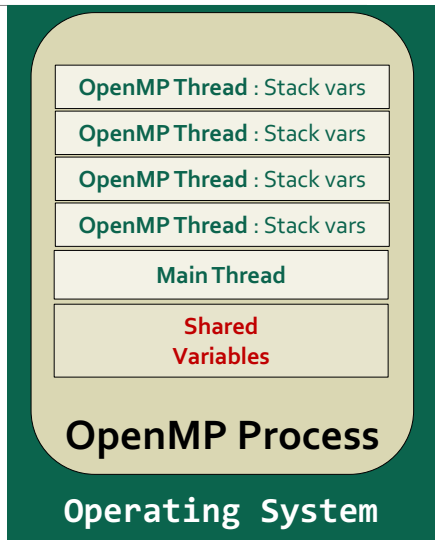
End Parallel region

LAU
الجامعة اللبنانية الأميركية
Lebanese American University

Sharing Variables in OpenMP

OpenMP shared-memory model

- OpenMP worker threads and the master thread share the same process and some variables.
- If variable scope includes the parallel region, it is shared by default : all the threads will read and write to the same memory location.



Scoping Rules

- OpenMP uses a shared-memory programming model
  - A shared variable is a variable that can be read or written by multiple threads
- Shared clause can be used to make items explicitly shared
  - Global variables are shared by default among tasks
    - File scope variables, namespace scope variables, static variables, Variables with const-qualified type having no mutable member are shared, Static variables which are declared in a scope inside the construct are shared

Scoping Rules

- But, not everything is shared...
  - Examples of implicitly determined private variables:
    - Stack (local) variables in functions called from parallel regions are PRIVATE
    - Automatic variables within a statement block are PRIVATE
    - Loop iteration variables are private
    - Implicitly declared private variables within tasks will be treated as `firstprivate`

Global Data

- Global data is shared and requires special care
- A problem may arise in case multiple threads access the same memory section simultaneously:
  - Read-only data is no problem
  - Updates have to be checked for race conditions
  - It is the programmer's responsibility to deal with this situation
- In general one can do the following:
  - Split the global data into a part that is accessed in serial parts only and a part that is accessed in parallel
    - Manually create thread private copies of the latter
    - Use the thread ID to access these private copies

Shared by default

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

- This code is executing correctly in serial, but may give a different results in parallel. Why?

Problem 1 : Race condition

- A race condition is nondeterministic behavior caused by the times at which two or more threads access a shared variable.
- Let's suppose we have 2 threads executing :
  `x = a[i]; y = b[i];`
  `c[i] = x + y;`
- If a thread can execute the two lines without having the other thread changing variables x and y, good
  - Not guaranteed.
- If the two threads have a mixed execution, the result c will be wrong.

Problem 1 : Race condition

- Race conditions may or may not be visible depending on various experimental conditions (number of cores, other software running, luck, ...)

Problem 2 : Corruption

- Independently from race conditions, writing to the same object or memory location from different threads without protection is risky.
  - Example : Different threads write to the serial output (console) at the same time.
  - If you are lucky, messages will intercalate nicely.
  - If you are not, the output may become garbled as bits of information representing the output text will be mixed together.

Problem 3 : Initialization

- If you use local copies instead of global variables to prevent race conditions and corruption, the last problem is initialization.
- Local variables created by the OpenMP layer may or may not be initialized, or initialized differently depending on the directive used :
  - PRIVATE
  - THREADPRIVATE
  - FIRSTPRIVATE
  - …

Solutions

- The solutions exist :
  - Large : force the execution of a block of code in serial with a critical section. Only one thread will execute this block at a given time.
  - Finer : protect a variable (force serial execution of write operations).
  - Finest : protect a variable with detailed information about the kind of operation.
    - Example : reduction.
- Ideal : recode to prevent sharing.

Scalability

- Thanks to Amdahl's law, we know serial parts will rapidly decrease the scalability of your parallel software.
- But forcing the serial execution of a block of code or serial access to a variable is a serial part, even if it's inside a parallel region.
- Try to rewrite or rethink your algorithm to prevent variable sharing and synchronization.
- If you have to synchronize, select the finest granularity to minimize the serial part.

Solution 1: Explicitly Change the Scope

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

Before : variables defined with global scope from the master thread, shared between threads.

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

After : local variables defined locally. Nothing shared.

Efficient and safe.

Solution 2: forcing serial execution of the critical block

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

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

***Before** : variables defined with global scope from the master thread, shared between threads.*

***After** : same thing, but serial execution forced.*

Safe but not scaling.

Solution 3: `atomic`

- Instead of protecting an entire block of code, is it enough to protect write accesses to a single shared variable only ?
- If yes, use atomic it will be a lot faster than critical :
  - atomic is like a mini critical section for a variable.

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

Solution 4: Changing the Scope Using the `private` Clause

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

Before : *global variables shared between threads.*

```
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;
}
```

After : *local copies of global variables.*
Nothing shared.

Efficient and safe.

Solution 4: Changing the Scope Using the `private` Clause

- The private clause reproduces the variable for each task
  - Variables are un-initialized;
  - C++ object is default constructed
  - Any value external to the parallel region is undefined

```
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;
     }
}
```

Other Data Scope Clauses

- `shared`
  - Declares variables in its list to be shared among all threads in the team
- `firstprivate`
  - Combines the behavior of the **`private`** clause with automatic initialization of the variables in its list
- lastprivate
  - Combines the behavior of the **`private`** clause with a copy from the last loop iteration or section to the original variable object
- Reduction
- Other clauses include **`copyin`** and **`copyprivate`**

`firstprivate` Example

- Variables initialized from shared variable

```
incr = 0;
#pragma omp parallel for firstprivate(incr)

for (i=0;i <= Max; i++) {
   if ((i%2)==0) incr++;
   A(i)= incr;
}
```

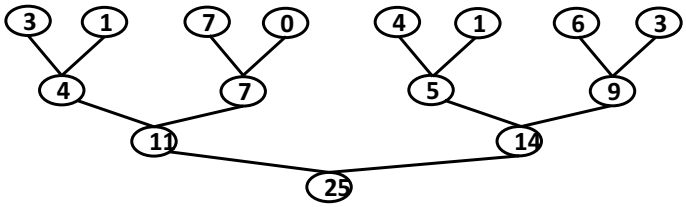
`lastprivate` Example

- Variables update shared variable using value from last iteration
- C++ objects are updated as if by assignment

```
void sq2(int n, double *lastterm)
{
  double x; int i;
  #pragma omp parallel
  #pragma omp for lastprivate(x)
  for (i = 0; i < n; i++){
     x = a[i]*a[i] + b[i]*b[i];
     b[i] = sqrt(x);
  }
  lastterm = x;
}
```


Reduction

- Perform a reduction of the data before transferring to the CPU
- **Tree based reduction approach** used within each thread block



Example of tree based SUM

- Reduction decomposed into multiple kernels to reduce number of threads issued in the later stages of tree based reduction

Reduction

- OpenMP reduction clause:
  - reduction (op : list)
- Inside a parallel or a work-sharing construct:
  - A local copy of each list variable is made and initialized depending on the "op" (e.g. 0 for "+").
  - Updates occur on the local copy.
  - Local copies are reduced into a single value and combined with the original global value.
- The variables in "list" must be shared in the enclosing parallel region.

```
    double  ave=0.0, A[MAX];
    int i;

     #pragma omp parallel for reduction (+:ave)
     for (i=0;i< MAX; i++) {
         ave + = A[i];
    }
   ave = ave/MAX;
```


Conditional Serial Execution of Parallel Regions

- At times it maybe useful to identify conditions when a parallel region should be executed by a single thread or using parallel threads

```
    double  ave=0.0, A[MAX];
    int i;

     #pragma omp parallel for reduction (+:ave) if (MAX > 10000)
     for (i=0;i< MAX; i++) {
         ave + = A[i];
    }
   ave = ave/MAX;
```

C/C++ Reduction Operations

- A range of associative operands can be used with reduction
- Initial values are the ones that make sense

Operand	Initial Value
+	0
*	1
-	0
^	0

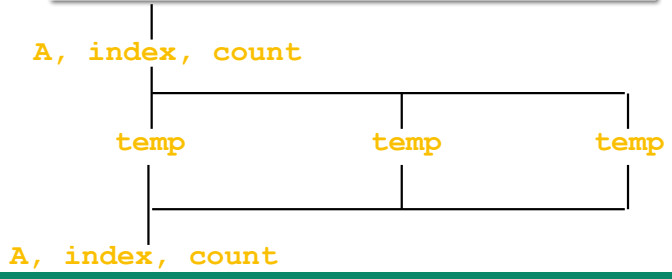
Operand	Initial Value
&	~0
|	0
&&	1
||	0

A Data Environment Example

```
float A[10];
main ()
{
  integer index[10];
  #pragma omp parallel
  {
      Work (index);
  }
    printf ("%d\n", index[1]);
}
```

***A*, *index*, and *count* are shared by all threads, but *temp* is local to each thread**

```
extern float A[10];
void Work (int *index)
{
  float temp[10];
  static integer count;
  <...>
}
```



Dot Product

```
float dot_prod(float* a, float* b, int N)
{
  float sum = 0.0;
  #pragma omp parallel for shared(sum)
   for(int i=0; i<N; i++) {
     sum += a[i] * b[i];
   }
  return sum;
}
```

What is Wrong?

Multiple threads modifying sum with no protection – this is a race condition!

LAU
الجامعة اللبنانية الأميركية
Lebanese American University

Synchronization

Programming Model: Synchronization

- OpenMP Synchronization
  - OpenMP Critical Sections
    - Defines a critical region on a structured code block
    - Named or unnamed
    - No explicit locks
  - Barrier directives
  - Explicit Lock functions
    - When all else fails – may require flush directive
  - Single-thread regions within parallel regions
    - master, single directives

```
#pragma omp critical [(lock_name)]
{
  /* Critical code here */
}
```

```
#pragma omp barrier
```

```
omp_set_lock( lock_l );
/* Code goes here */
omp_unset_lock( lock_l );
```

```
#pragma omp single
{
  /* Only executed once */
}
```


Barrier Construct

- Explicit barrier synchronization
  - Each thread waits until all threads arrive
  - We will talk about the shared construct later

```
#pragma omp parallel shared (A, B, C)
{
    DoSomeWork(A,B); // Processed A into B
    #pragma omp barrier

     DoSomeWork(B,C); // Processed B into C
}
```

Explicit `Barrier`

- Several OpenMP constructs have implicit barriers
  - Parallel – necessary barrier – cannot be removed
  - for
  - single
- Unnecessary barriers hurt performance and can be removed with the `nowait` clause

Avoiding Overhead: `nowait` Clause

- Use when threads unnecessarily *wait* between independent computations

```
#pragma omp for nowait
   for(...)
     {...};
```

```
#pragma single nowait
{ [...] }
```

```
#pragma omp for schedule(dynamic,1) nowait
 for(int i=0; i<n; i++)
   a[i] = bigFunc1(i);

#pragma omp for schedule(dynamic,1)
 for(int j=0; j<m; j++)
   b[j] = bigFunc2(j);
```

Explicit Barrier: Example

```
#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_calc4(id);
}
```

implicit barrier at the end of a for work sharing construct

no implicit barrier due to nowait

implicit barrier at the end of a parallel region

Avoiding Overhead: `if` clause

- The if clause is an integral expression that, if evaluates to true (nonzero), causes the code in the parallel region to execute in parallel
  - Used for optimization, e.g. avoid going parallel

```
#pragma omp parallel if(expr)
```

Avoiding Overhead: `if` clause

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

void test(int val)
{
    #pragma omp parallel if (val)
    if (omp_in_parallel())
    {
        #pragma omp single
        printf_s("val = %d, parallelized with %d threads\n",
                 val, omp_get_num_threads());
    }
    else
        printf_s("val = %d, serialized\n", val);
    }
```

```
int main( )
{
    omp_set_num_threads(2);
    test(0);
    test(2);
}
```


Synchronization

- Synchronization is used to impose order constraints and to protect access to shared data
- High level synchronization:
  - critical
  - atomic
  - barrier
  - ordered
- Low level synchronization
  - flush
  - locks (both simple and nested)

OpenMP `critical`: Example

- Threads wait their turn – only one at a time calls `consume()` thereby protecting `RES` from race conditions

- Naming the critical construct RES_lock is optional
- Good Practice – Name all critical sections

```
float RES;
#pragma omp parallel
{
float B;
#pragma omp for
  for(int i=0; i < niters; i++)
  {
    B = big_job(i);
    #pragma omp critical (RES_lock)
       consume (B, RES);
  }
}
```


Synchronization: `atomic`

- Very similar to the `critical` directive
  - Difference is that `atomic` is only used for the update of a memory location
  - `atomic` is referred to as a mini critical section with a block of one statement

```
#pragma omp parallel
{
  double tmp, B;
  B =  DoIt();
  tmp = big_ugly(B);
  #pragma omp atomic
    x +=  tmp;
}
```

Atomic only protects the read/update of X

Synchronization: atomic Example

```
#pragma omp parallel for shared(x, y, index, n)
   for (i = 0; i < n; i++) {
      #pragma omp atomic
        x[index[i]] += work1(i);
      y[i] += work2(i);
   }
```


Synchronization: Lock Functions

- 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()`

A lock implies a memory fence of all thread visible variables

Synchronization: Lock Functions

- 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.

`single` Construct

- The single construct denotes a block of code that is executed by only one thread (not necessarily the master thread).
  - First thread to arrive is chosen
- A barrier is implied at the end of the single block (can remove the barrier with a `nowait` clause).

```
#pragma omp parallel
{
   DoManyThings();
   #pragma omp single
   {
     exchange_boundaries();
   }  // threads wait here for single
   do_many_more_things();
}
```

`master` Construct

- A `master` construct denotes block of code to be executed only by the master thread
  - The other threads just skip it (no synchronization is implied).
  - Identical to the `omp` single, except that the master thread is the thread chosen to do the work

```
#pragma omp parallel
{
   DoManyThings();
   #pragma omp master // if not master skip to next stmt
   {
     ExchangeBoundaries();
   }
   DoManyMoreThings();
}
```


Synchronization: `ordered`

- Specifies that code under a parallelized for loop should be executed like a sequential loop.

```
#pragma omp parallel private (tmp)
#pragma omp for ordered reduction(+:res)

   for (i=0;i < n;i++){
      tmp = Neat_Stuff(i);
     #pragma ordered
      res += consum(tmp);
   }
```

Avoiding Overhead: collapse Clause

- Fuse or collapse perfectly nested loops to exploit a larger iteration space for the parallelization
  - Increase the total number of iterations that will be partitioned across the available number of OMP threads by reducing the granularity of work to be done by each thread
  - If the amount of work to be done by each thread is non-trivial (after collapsing is applied), this may improve the parallel scalability of the OMP application

Avoiding Overhead: `collapse` Clause

```
#pragma omp for collapse(2)
for(i = 1; i < N; i++)
   for(j = 1; j < M; j++)
      for(k = 1; k < K; k++)
         foo(i, j, k);
```

Iteration space from i-loop and j-loop is collapsed into a single one, if loops are perfectly nested and form a rectangular iteration space.

SPMD vs. Worksharing

- A parallel construct by itself creates an SPMD or "Single Program Multiple Data" program … i.e., each thread redundantly executes the same code.
- How do you split up pathways through the code between threads within a team?
  - This is called worksharing
    - Loop construct
    - Task construct
    - Sections/section constructs
    - Single construct

Worksharing

- Worksharing is the general term used in OpenMP to describe distribution of work across threads.
- Three examples of worksharing in OpenMP are:
  - `omp for` construct
  - `omp sections` construct
  - `omp task` construct

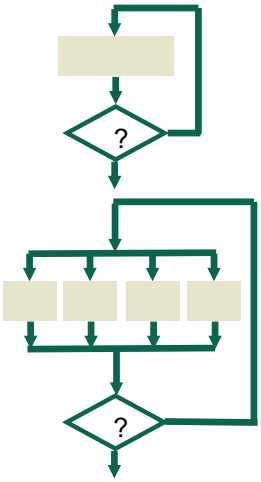
Automatically divides work among threads

OpenMP: Concurrent Loops

- OpenMP easily parallelizes loops
  - No data dependencies between iterations!

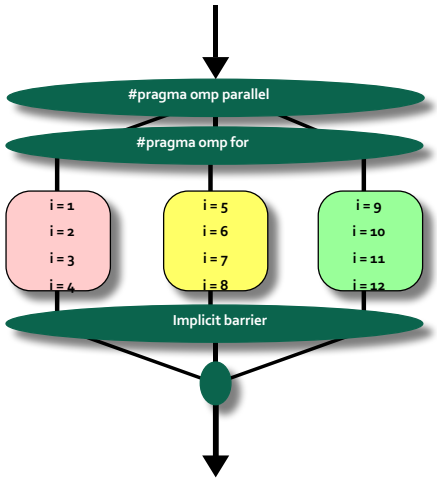
```
#pragma omp parallel for
for( i=0; i < 25; i++ ) {
  printf("Foo");
}
```

- Preprocessor calculates loop bounds for each thread directly from serial source



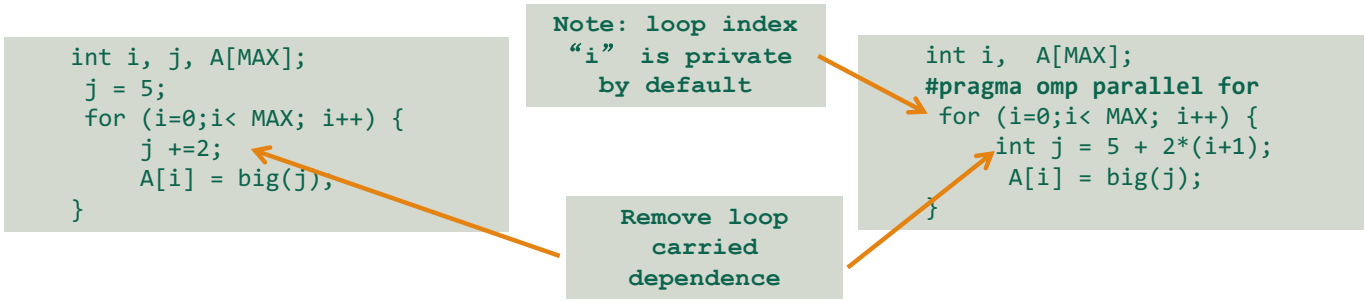
OpenMP: Concurrent Loops

```
// assume N=12
#pragma omp parallel for
for(i = 1; i < N+1; i++)
      c[i] = a[i] + b[i];
```



Working with loops

- Basic approach
  - Find compute intensive loops
  - Make the loop iterations independent so they can safely execute in any order without loop-carried dependencies
  - Place the appropriate OpenMP directive and test



Working with Loops: Subtle Details

- Dynamic mode (the default mode)
  - The number of threads used in a parallel region can vary from one parallel region to another.
  - Setting the number of threads only sets the maximum number of threads - you could get less.
- Static mode
  - The number of threads is fixed and controlled by the programmer.
- Although you can nest parallel loops in OpenMP, the compiler can choose to serialize the nested parallel region

OpenMP `schedule` Clause

- Determine how loop iterations are divided among the thread team
  - `static([chunk])` divides iterations statically between threads
    - Each thread receives `[chunk]` iterations, rounding as necessary to account for all iterations
    - Default `[chunk]` is `ceil( # iterations / # threads )`
  - `dynamic([chunk])` allocates `[chunk]` iterations per thread, allocating an additional `[chunk]` iterations when a thread finishes
    - Forms a logical work queue, consisting of all loop iterations
    - Default `[chunk]` is 1
  - `guided([chunk])` allocates dynamically, but `[chunk]` is exponentially reduced with each allocation

Loop Work-Sharing: The 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: Loop Scheduling

```
#pragma omp parallel for schedule(static)
for( i=0; i<16; i++ )
{
  doIteration(i);
}
```

```
// static scheduling

int chunk = 16/T;
int base = tid * chunk;
int bound = (tid+1)*chunk;

for( i=base; i<bound; i++ )
{
  doIteration(i);
}

Barrier();
```


OpenMP: Loop Scheduling

```
#pragma omp parallel for \
 schedule(dynamic)
for( i=0; i<16; i++ )
{
  doIteration(i);
}
```

```
// Dynamic Scheduling

int current_i;

while( workLeftToDo() )
{
  current_i = getNextIter();
  doIteration(i);
}

Barrier();
```

Schedule Clause Example

- Iterations are divided into chunks of 8
  - If start = 3, then first chunk is
  - i={3,5,7,9,11,13,15,17}

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


Sections worksharing Construct

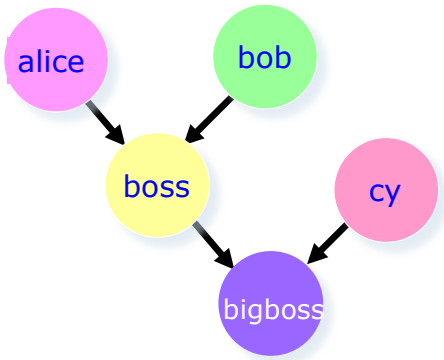
- OpenMP supports non-iterative parallel task assignment using the sections directive.
  - `#pragma omp sections`
    - Must be inside a parallel region
    - Precedes a code block containing of N blocks of code that may be executed concurrently by N threads
    - Encompasses each omp section
  - `#pragma omp section`
    - Precedes each block of code within the encompassing block described above
    - May be omitted for first parallel section after the parallel sections pragma
    - Enclosed program segments are distributed for parallel execution among available threads

Sections Worksharing Construct

- The `omp sections` directive supports the following OpenMP clauses:
  - shared(list)
  - private(list) firstprivate(list) lastprivate(list)
  - default(shared | none)
  - nowait
  - reduction

Decomposition

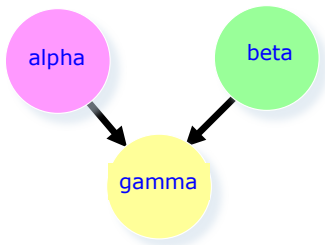
```
a = alice();
b = bob();
s = boss(a, b);
c = cy();
printf ("%6.2f\n", bigboss(s,c));
```



Alice ,bob, and cy can be computed in parallel

Sections work sharing Construct

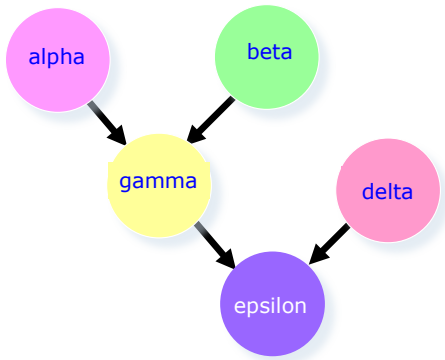
```
#pragma omp parallel sections
{
#pragma omp section  /* Optional */
   a = alpha();
#pragma omp section
   b = beta();
}

printf ("%6.2f\n”, gamma(a, b) );
```



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

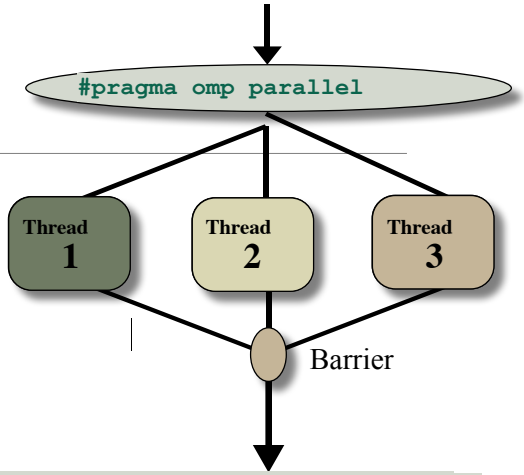
Sections work sharing Construct

```
#pragma omp parallel sections
{
#pragma omp section  /* Optional */
   a = alpha();
#pragma omp section
   b = beta();
}
#pragma omp parallel sections
{
#pragma omp section  /* Optional */
   c = delta();
#pragma omp section
   s = gamma(a, b);
}
printf ("%6.2f\n”, epsilon(s,c));
```



Parallel Construct Implicit Task View

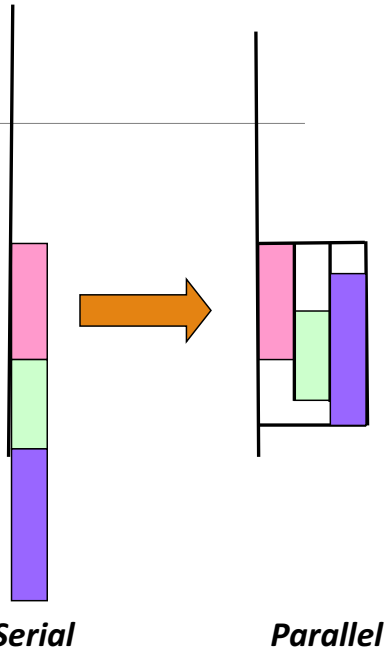
- Tasks are created in OpenMP even without an explicit `task` directive.
- Let's look at how tasks are created implicitly for the code snippet below
  - Thread encountering parallel construct packages up a set of implicit tasks
  - Team of threads is created.
  - Each thread in team is assigned to one of the tasks (and tied to it).
  - Barrier holds original master thread until all implicit tasks are finished.




```
#pragma omp parallel
    {
        int mydata;
        code…
    }
```

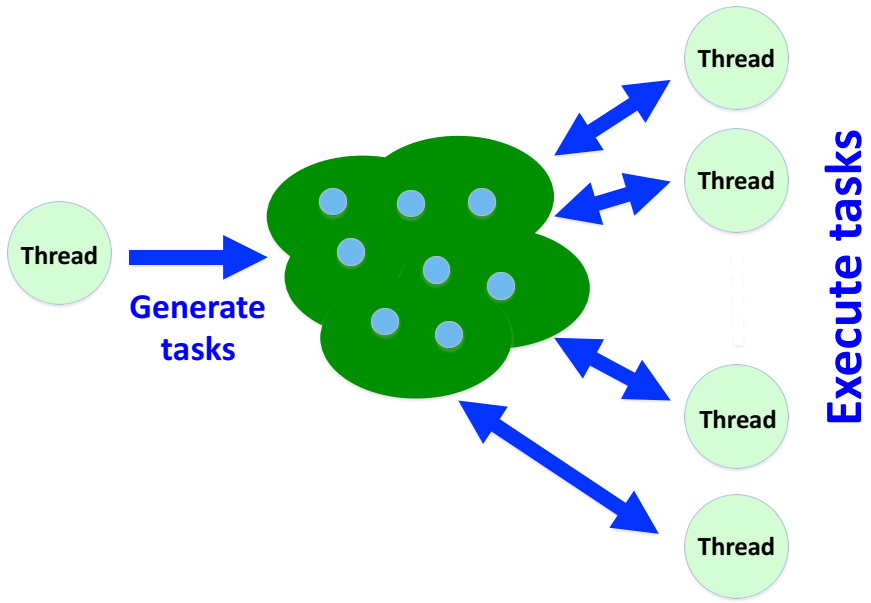

Tasks

- Tasks are independent units of work
- Threads are assigned to perform the work of each task
  - Tasks may be deferred or executed immediately
  - The system determines at runtime which case of the above
- Tasks are composed of:
  - code to execute
  - data environment
  - internal control variables (ICV)

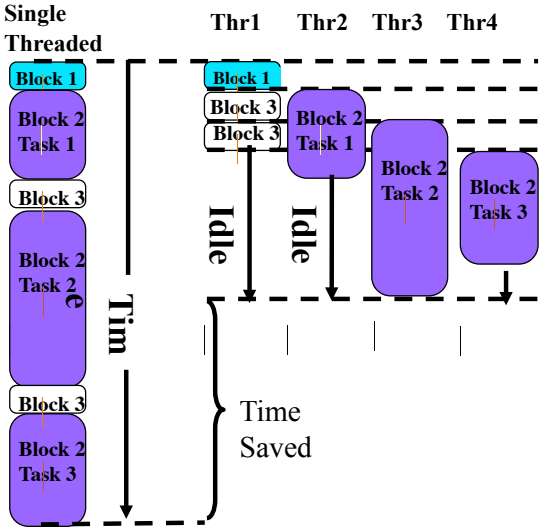


OpenMP Tasks

- Developers use a pragma to specify where the tasks are using the assumption that all tasks can be executed independently
- OpenMP Run Time System
  - When a thread encounters a task construct, a new task is generated
  - The moment of execution of the task is up to the runtime system
  - Execution can either be immediate or delayed
  - Completion of a task can be enforced through task synchronization



Why are tasks useful?



OpenMP task clause

#pragma omp task

- The **task** pragma can be used to explicitly define a task.
  - Used to identify a block of code to be executed in parallel with the code outside the task region
  - The **task** pragma can be useful for parallelizing irregular algorithms such as pointer chasing or recursive algorithms.

Simple Example 1

Write a program that prints either "A race car" or "A car race" and maximize the parallelism

Simple Example 1

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

int main(int argc, char *argv[]) {

          printf("A ");
          printf("race ");
          printf("car ");

   printf("\n");
   return(0);
}
```

Output?

Parallel Simple Example 1

```
#include <stdlib.h>
#include <stdio.h>
#include “omp.h”

int main(int argc, char *argv[]) {
#pragma omp parallel
{
      printf("A ");
      printf("race ");
      printf("car ");
}
     printf("\n");  return(0);
 }
```

```
A A A A A A A A A race A A A A A A A A race race
race race race race race race race car race race race
race race race race car car car car car car car
car car car car car car car car car
```


Parallel Simple Example 1 Using `Single`

```
#include <stdlib.h>
#include <stdio.h>
#include “omp.h”

int main(int argc, char *argv[]) {
#pragma omp parallel
{
    #pragma omp single
    {
       printf("A ");
       printf("race ");
       printf("car ");
     }
    }
    printf("\n");  return(0);
}
```

Output?

Parallel Simple Example 1 Using Tasks

```
#include <stdlib.h>
#include <stdio.h>
#include “omp.h”

int main(int argc, char *argv[]) {
#pragma omp parallel
{
     #pragma omp single
     {
         printf("A ");
         #pragma omp task
         printf("race ");
         #pragma omp task
         printf("car ");
      }
     }
     printf("\n");  return(0);
}
```

Output?

A car race

A race car

Parallel Simple Example 1 Using Tasks

```
#include <stdlib.h>
#include <stdio.h>
#include “omp.h”

int main(int argc, char *argv[]) {
#pragma omp parallel
{
     #pragma omp single
     {
         printf("A ");
         #pragma omp task
         {
         printf("race "); printf("race ");
         printf("race "); printf("race ");
         }
         #pragma omp task
         printf("car ");
      }
     }
     printf("\n");  return(0);
}
```

And Now?

A car race race race race

A race car race race race

Simple Example 2: || Linked List

```
#pragma omp parallel
// assume 8 threads
{
  #pragma omp single private(p)
  {
    …
    while (p) {
    #pragma omp task
      {
        processwork(p);
      }
      p = p->next;
    }
  }
}
```

A pool of 8 threads is created here

One thread gets to execute the while loop

The single "while loop" thread creates a task for each instance of processwork()

Task synchronization

```
#pragma omp parallel num_threads(np)
{
#pragma omp task
   function_A();
#pragma omp barrier
#pragma omp single
   {
#pragma omp task
      function_B();
   }
}
```

np Tasks created here, one for each thread

All Tasks guaranteed to be completed here

1 Task created here

B-Task guaranteed to be completed here

Avoiding Overhead: `final` Clause

```
#pragma omp task final(expr)
```

- `final` clause is useful for recursive problems that perform task decomposition
- Stop task creation at a certain depth in order to expose enough parallelism and reduces the overhead.
- The generated task will be a final one if the `expr` evaluates to nonzero value
- All task constructs encountered inside a final task create final and included tasks

Avoiding Overhead: `final` Clause

```
void foo(bool arg)
{
  int i = 3;
  #pragma omp task final(arg) firstprivate(i)
     i++;
  printf("%d\n", i); // will print 3 or 4 depending
on expr
}
```

Avoiding Overhead: `mergeable` Clause

- The mergeable clause allows the implementation to merge the task's data environment with the enclosing region
  - if the generated task is *undeferred* or *included*
    - undeferred: if clause present and evaluates to false
    - included: final clause present and evaluates to true

```
#pragma omp task mergeable
```


Avoiding Overhead: `taskyield` Clause

- The **`taskyield`** directive specifies that the current task can be suspended in favor of execution of a different task.
  - Hint to the runtime for optimization and/or deadlock prevention

```
#pragma omp taskyield
```

Avoiding Overhead: `taskyield` Clause

```
#include <omp.h>
void something_useful();
void something_critical();
void foo(omp_lock_t * lock, int n)
{
 for(int i = 0; i < n; i++)
 #pragma omp task
 {
    something_useful();
    while( !omp_test_lock(lock) ) {
    #pragma omp taskyield
 }
 something_critical();
 omp_unset_lock(lock);
 }
}
```

The waiting task may be suspended here and allow the executing thread to perform other work. This may also avoid deadlock situations

OpenMP: Recap

- OpenMP is a parallel programming model for Shared-Memory machines
  - All threads have access to a shared main memory
  - Each thread may have private data.
- Parallelism has to be expressed explicitly by the programmer.
  - Base construct is a Parallel Region which is a team of threads that is provided by the runtime system.
- Using the Worksharing constructs, the work can be distributed among the threads of a team.
  - The Task construct defines an explicit task along with it's data environment. Execution may be deferred.
- To control the parallelization, mutual exclusion as well as thread and task synchronization constructs are available.

Task Construct: Linked List Revisited

- A team of threads is created at the omp parallel construct
- A single thread is chosen to execute the while loop
  - Lets call this thread "L"
- Thread L operates 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 runs in its own thread
- All tasks complete at the barrier at the end of the parallel region's single construct

```
#pragma omp parallel
{
  #pragma omp single
  { // block 1
    node * p = head;
    while (p) {
    //block 2
    #pragma omp task private(p)
      process(p);
    p = p->next;  //block 3
    }
  }
}
```


When are tasks guaranteed to be complete?

- Tasks are guaranteed to be complete at thread or task barriers
  - At the directive: #pragma omp barrier
  - At the directive: #pragma omp taskwait
- Task barrier: `taskwait`
  - Encountering task is suspended until children tasks are complete
  - Applies only to direct children, not descendants!

Task Completion Example

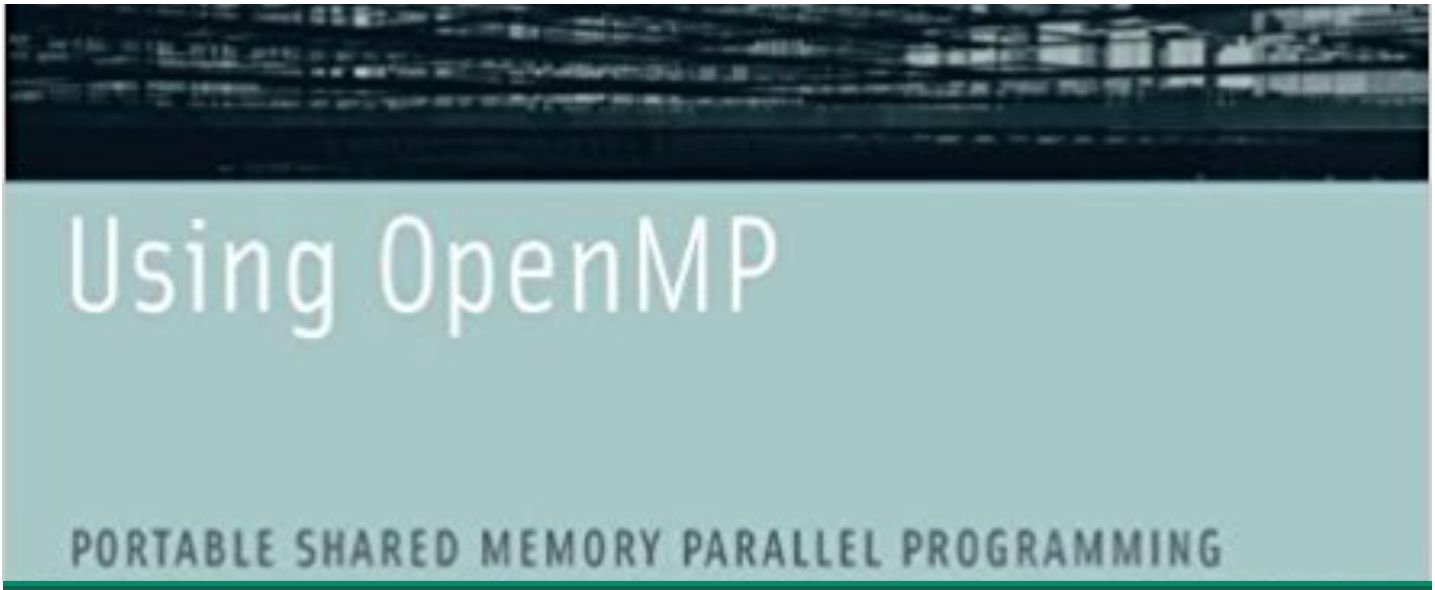
```
#pragma omp parallel
{
   #pragma omp task
   foo();
   #pragma omp barrier
   #pragma omp single
   {
      #pragma omp task
      bar();
   }
}
```

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



Environment Variables

Environment Variables

Name	Possible Values	Most Common Default
OMP_NUM_THREADS	Non-negative Integer	1 or #cores
OMP_SCHEDULE	„schedule [, chunk]“	„static, (N/P)“
OMP_DYNAMIC	{TRUE | FALSE}	TRUE
OMP_NESTED	{TRUE | FALSE}	FALSE
OMP_STACKSIZE	„size [B | K | M | G]“	-
OMP_WAIT_POLICY	{ACTIVE | PASSIVE}	PASSIVE
OMP_MAX_ACTIVE_LEVELS	Non-negative Integer	-
OMP_THREAD_LIMIT	Non-negative Integer	1024
OMP_PROC_BIND	{TRUE | FALSE}	FALSE
OMP_PLACES	Place List	-
OMP_CANCELLATION	{TRUE | FALSE}	FALSE
OMP_DISPLAY_ENV	{TRUE | FALSE}	FALSE
OMP_DEFAULT_DEVICE	Non-negative Integer	-

Nesting `parallel` Directives

- Nested parallelism can be enabled using the `OMP_NESTED` environment variable.
- If the `OMP_NESTED` environment variable is set to `TRUE`, nested parallelism is enabled.
- In this case, each parallel directive creates a new team of threads.

OpenMP Library Functions

- In addition to directives, OpenMP also supports a number of functions that allow a programmer to control the execution of threaded programs.

```
/* thread and processor count */
void omp_set_num_threads (int num_threads);
int omp_get_num_threads ();
int omp_get_max_threads ();
int omp_get_thread_num ();
int omp_get_num_procs ();
int omp_in_parallel();
```


OpenMP Library Functions

```
/* controlling and monitoring thread creation */
void omp_set_dynamic (int dynamic_threads);
int omp_get_dynamic ();
void omp_set_nested (int nested);
int omp_get_nested ();
/* mutual exclusion */
void omp_init_lock (omp_lock_t *lock);
void omp_destroy_lock (omp_lock_t *lock);
void omp_set_lock (omp_lock_t *lock);
void omp_unset_lock (omp_lock_t *lock);
int omp_test_lock (omp_lock_t *lock);
```

- In addition, all lock routines also have a nested lock counterpart for recursive mutexes.

Environment Variables in OpenMP

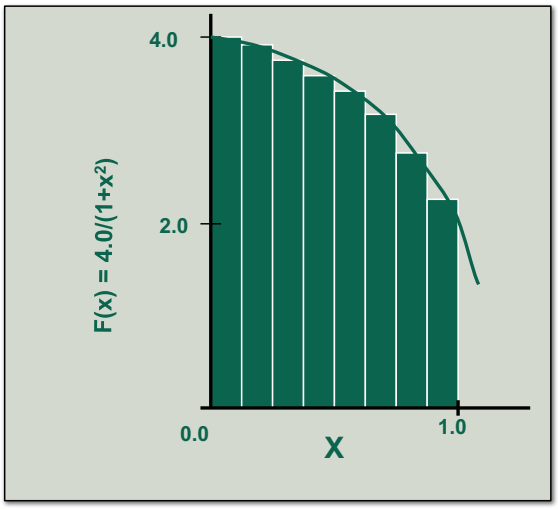
- OMP_NUM_THREADS: This environment variable specifies the default number of threads created upon entering a parallel region.
- OMP_SET_DYNAMIC: Determines if the number of threads can be dynamically changed.
- OMP_NESTED: Turns on nested parallelism.
- OMP_SCHEDULE: Scheduling of for-loops if the clause specifies runtime

Closing Comments: Explicit Threads Versus Directive Based Programming

- Directives layered on top of threads facilitate a variety of thread-related tasks.
- A programmer is rid of the tasks of initializing attributes objects, setting up arguments to threads, partitioning iteration spaces, etc.
- There are some drawbacks to using directives as well.
- An artifact of explicit threading is that data exchange is more apparent. This helps in alleviating some of the overheads from data movement, false sharing, and contention.
- Explicit threading also provides a richer API in the form of condition waits, locks of different types, and increased flexibility for building composite synchronization operations.
- Finally, since explicit threading is used more widely than OpenMP, tools and support for Pthreads programs are easier to find.

Examples

Back to the PI Problem …



Mathematically, we know that:

$$\int_0^1 \frac{4.0}{(1+x^2)} dx = \pi$$

We can approximate the integral as a sum of rectangles:

$$\sum_{i=0}^{N} F(x_i)\Delta x \approx \pi$$

Where each rectangle has width Δx and height $F(x_i)$ at the middle of interval i.

Back to the PI Problem: Serial Code

```
static long num_steps = 100000;
double step;
void main ()
{
   int i;
   double x, pi, sum = 0.0;

   step = 1.0/(double) num_steps;

   for (i=0;i< num_steps; i++){
        x = (i+0.5)*step;
        sum = sum + 4.0/(1.0+x*x);
    }
    pi = step * sum;
}
```


Calculate Pi by integration

$$\Pi = \int_0^1 \frac{4}{(1+x^2)} dx$$

```
double f(double x) {
return (double)4.0 / ((double)1.0 + (x*x));
}
void computePi() {
double h = (double)1.0 / (double)iNumIntervals;
double sum = 0, x;

...

for (int i = 1; i <= iNumIntervals; i++) {
    x = h * ((double)i - (double)0.5);
    sum += f(x);
}

myPi = h * sum;
}
```

Calculate Pi by integration

$$\Pi = \int_0^1 \frac{4}{(1+x^2)} dx$$

```
double f(double x) {
return (double)4.0 / ((double)1.0 + (x*x));
}
void computePi() {
double h = (double)1.0 / (double)iNumIntervals;
double sum = 0, x;

#pragma omp parallel for private(x) reduction(+:sum)

for (int i = 1; i <= iNumIntervals; i++) {
    x = h * ((double)i - (double)0.5);
    sum += f(x);
}

myPi = h * sum;
}
```


Reduction Example: Computing Pi

```
long num_steps=100000; double step;

void main()
{  int i;
   double x, sum = 0.0, pi;

    step = 1./(double)num_steps;
    start = clock();
    #pragma omp parallel for private(x) reduction (+:sum)
    for (i=0; i<num_steps; i++)
    {
      x = (i + .5)*step;
      sum = sum + 4.0/(1.+ x*x);
    }

    pi = sum*step;
    stop = clock();

    printf("The value of PI is %15.12f\n",pi);
    printf("Time to calculate PI was %f seconds\n",((double)(stop - start)/1000.0));
    return 0;
}
```

Calculate Pi by integration

$$\Pi = \int_0^1 \frac{4}{(1+x^2)} dx$$

# Threads	Runtime [sec.]	Speedup
1	0.002877	1.00
2	0.001777	1.62
4	0.002988	0.96
8	0.002050	1.40
16	0.105787	1.26

Number of Iterations: 1000,000
PI value: 3.141593

Architecture: Intel i7, 3 GHz, running Mac OS 10.11.3 with 16GB RAM

Useful MacOS/Linux Commands

To compile
gcc –Wall –fopenmp –o pi pi.c

To set the number of threads to 4 using OMP_NUM_THREADS:
In the bash shell, type: **export OMP_NUM_THREADS=4**
in the c shell, type: **setenv OMP_NUM_THREADS 4**

You can set the number of threads to different values (2, 8, etc) using the same command
To run the OpenMP example code, simply type **./pi**
You can use the time command to evaluate the program's runtime:
Not very accurate but will do!

```
voyager-2:~ haidar$ export OMP_NUM_THREADS=4
voyager-2:~ haidar$ /usr/bin/time -p ./pi
 The value of PI is  3.141592653590
 The time to calculate PI was 18037.175000 seconds
 real 6.52
 user 17.97
 sys 0.06
```

You can compare the running time of the OpenMP version with the serial version by compiling the serial version and repeating the above analysis

Parallel Tree Traversal

```
void traverse (Tree *tree)
{
   #pragma omp task
   if(tree->left)
     traverse(tree->left);

   #pragma omp task
   if(tree->right)
      traverse(tree->right);
   process(tree);
}
```


Useful MacOS/Linux Commands

- From within a shell, global adjustment of the number of threads:
  - export OMP_NUM_THREADS=4
  - ./pi
- From within a shell, one-time adjustment of the number of threads:
  - OMP_NUM_THREADS=4 ./pi
- Intel Compiler on Linux: asking for more information:
  - export KMP_AFFINITY=verbose
  - export OMP_NUM_THREADS=4
  - ./pi

Recursive Fibonacci

```
int main(int argc, char* argv[])

{
   [...]
   fib(input);
   [...]
```

```
int fib(int n)   {
   if (n < 2) return n;
   int x = fib(n - 1);
   int y = fib(n - 2);
   return
}  x+y;
```


Recursive Fibonacci: Discussion

```
int main(int argc, char* argv[])

{
   [...]
   fib(input);
   [...]
```

```
int fib(int n)   {
   if (n < 2) return n;
   int x = fib(n - 1);
   int y = fib(n - 2);
   return
}  x+y;
```

- Only one Task / Thread should enter `fib()` from `main()`, it is responsible for creating the two initial work tasks
- `taskwait` is required, as otherwise `x` and `y` would be lost

Recursive Fibonacci: Attempt 0

```
int main(int argc, char* argv[])

{
   [...]
#pragma omp parallel
{
#pragma omp single
{
   fib(input);
}
}
   [...]
}
```

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

What's wrong here?

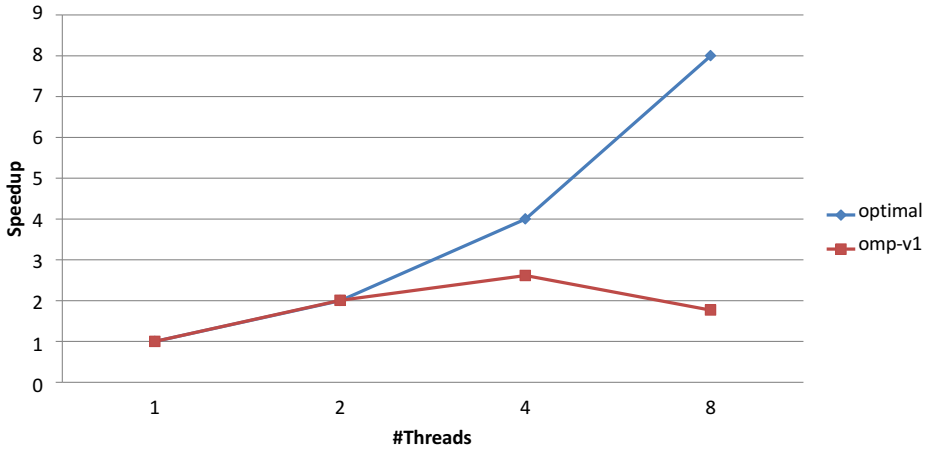
x and y are private. Can't use values of private variables outside of tasks

Recursive Fibonacci: Attempt 1

```
int main(int argc, char* argv[])

{
   [...]
#pragma omp parallel
{
#pragma omp single
{
   fib(input);
}
}
   [...]
}
```

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

Recursive Fibonacci: Attempt 1



Task creation overhead prevents better scalability

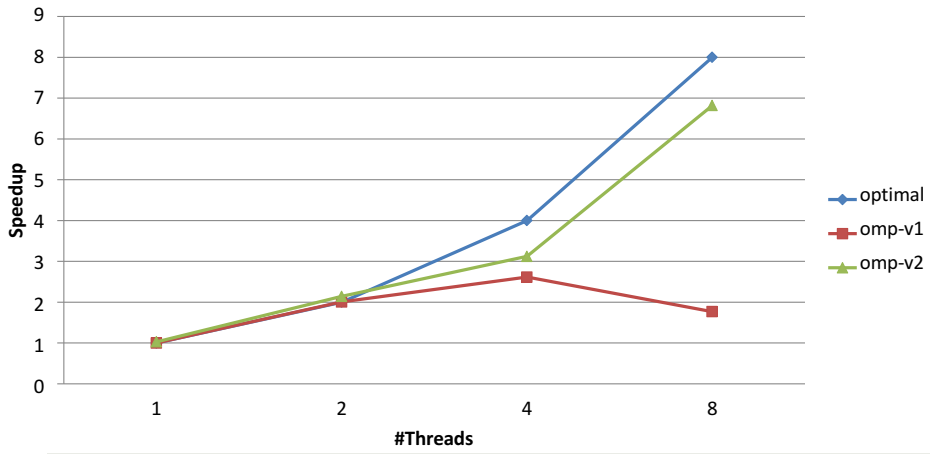
Recursive Fibonacci: Attempt 2

```
int main(int argc, char* argv[])

{
   [...]
#pragma omp parallel
{
#pragma omp single
{
   fib(input);
}
}
   [...]
}
```

```
int fib(int n)   {
 if (n < 2) return n;
 int x, y;
 #pragma omp task shared(x) if(n > 30)
 {
    x = fib(n - 1);
 }
 #pragma omp task shared(y) if(n > 30)
 {
    y = fib(n - 2);
 }
 #pragma omp taskwait
    return x+y;
}
```

Don't create yet another task once a certain (small enough) n is reached

Recursive Fibonacci: Attempt 2



Overhead persists when running with 4 or 8 threads

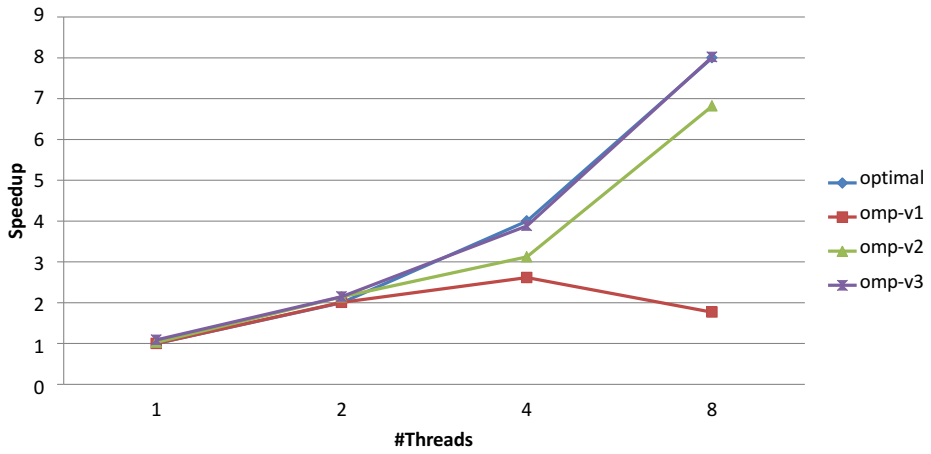
Recursive Fibonacci: Attempt 3

```
int main(int argc,   char* argv[])

{
   [...]
#pragma  omp parallel
{
#pragma  omp single
{
   fib(input);
}
}
   [...]
}
```

```
int fib(int n)  {
   if (n < 2) return n;
   if (n <= 30)
      return serfib(n);
   int x, y;
   #pragma omp task shared(x)
   {
      x = fib(n - 1);
   }
   #pragma omp task shared(y)
   {
      y = fib(n - 2);
   }
   #pragma omp taskwait
      return x+y;
```

Skip the OpenMP overhead once a certain n is reached (no issue w/ production compilers)

Recursive Fibonacci: Attempt 3



Example: Linked List using Tasks

- A typical C code that implements a linked list pointer chasing code is:

```
while(p != NULL){
  do_work(p->data);
  p = p->next;
}
```

- Can we implement the above code using tasks in order to parallelize the application?

Example: Linked List using Tasks

```
#pragma omp parallel
{
 #pragma omp single nowait
   {
     while (p != NULL) {
       #pragma omp task firstprivate(p)
       {
         processwork(p);
       }
       p = p->next;
     }
   }
 }
```


List Traversal

```
List ml; //my_list
Element *e;
#pragma omp parallel
#pragma omp single
{
   for(e=ml->first;e;e=e->next)
   #pragma omp task
      process(e);
}
```

What's wrong here?

Possible data race !
Shared variable e
updated by multiple tasks

List Traversal

```
List ml; //my_list
Element *e;
#pragma omp parallel
#pragma omp single
{
   for(e=ml->first;e;e=e->next)
#pragma omp task firstprivate(e)
      process(e);
}
```

Good solution – e is firstprivate

List Traversal

```
List ml; //my_list
Element *e;
#pragma omp parallel
#pragma omp single private(e)
{
   for(e=ml->first;e;e=e->next)
#pragma omp task
      process(e);
}
```

Good solution – e is private

List Traversal

```
List ml; //my_list
#pragma omp parallel
{
   Element *e;
   for(e=ml->first;e;e=e->next)
#pragma omp task
      process(e);
}
```

Good solution – e is private

Sudoko for Lazy Computer Scientists

- Find an empty field
- Insert a number
- Check Sudoku
  - (4 a) If invalid:
    - Delete number,
    - Insert next number
  - (4 b) If valid:
    - Go to next field

	6						8	11			15	14			16
15	11				16	14				12			6		
13		9	12					3	16	14		15	11	10	
2		16		11		15	10	1							
	15	11	10			16	2	13	8	9	12				
12	13			4	1	5	6	2	3					11	10
5		6	1	12		9		15	11	10	7	16			3
	2				10		11	6		5			13		9
10	7	15	11	16				12	13						6
9						1			2		16	10			11
1		4	6	9	13			7		11		3	16		
16	14			7		10	15	4	6	1				13	8
11	10		15				16	9	12	13			1	5	4
		12		1	4	6		16				11	10		
		5		8	12	13		10			11	2			14
3	16			10			7			6				12	

Sudoko for Lazy Computer Scientists

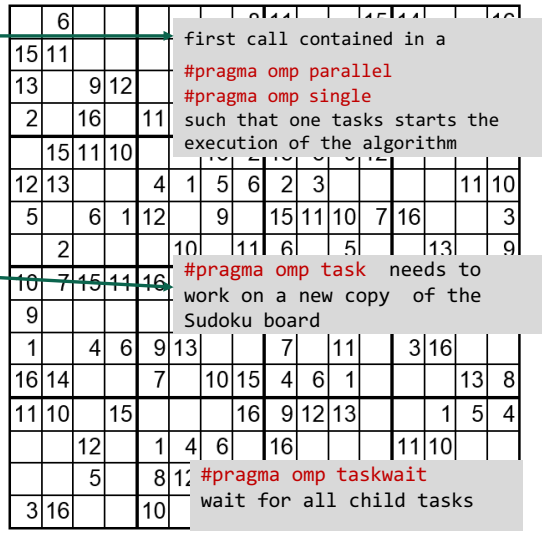
(1) Search an empty field

(2) Insert a number

(3) Check Sudoku

(4 a) If invalid:
Delete number,
Insert next number

(4 b) If valid:
Go to next field



```
first call contained in a
#pragma omp parallel
#pragma omp single
such that one tasks starts the
execution of the algorithm
```

```
#pragma omp task  needs to
work on a new copy  of the
Sudoku board
```

```
#pragma omp taskwait
wait for all child tasks
```


Brute-force Sudoku: Pseudocode

- OpenMP parallel region creates a team of threads

```
#pragma omp parallel
{
#pragma omp single
    solve_parallel(0, 0, sudoku2,false);
} // end omp parallel
```

  - Single construct: One thread enters the execution of `solve_parallel`
  - the other threads wait at the end of the single …
    - … and are ready to pick up tasks „from the work queue"

- Syntactic sugar (either you like it or you donot)

```
#pragma omp parallel sections
{
    solve_parallel(0, 0, sudoku2,false);
} // end omp parallel
```

Brute-force Sudoku: Implementation

```
for (int i = 1; i <= sudoku->getFieldSize(); i++) {
   if (!sudoku->check(x, y, i)) {
      #pragma omp task firstprivate(i,x,y,sudoku)
      {
             // create from copy constructor
             CSudokuBoard new_sudoku(*sudoku);
             new_sudoku.set(y, x, i);
             if (solve_parallel(x+1, y, &new_sudoku)) {
                 new_sudoku.printBoard();
             }
      } // end omp task
    }
  }
  #pragma omp taskwait
```

#pragma omp task needs to work on a new copy of the Sudoku board

#pragma omp taskwait
wait for all child tasks