Advanced OpenMP

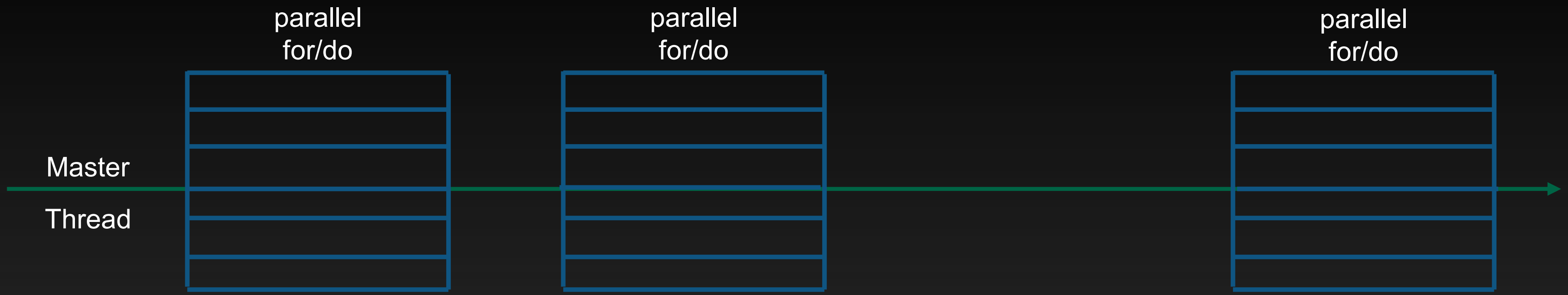
John Urbanic
Parallel Computing Scientist
Pittsburgh Supercomputing Center

Because Everything Isn't A Loop

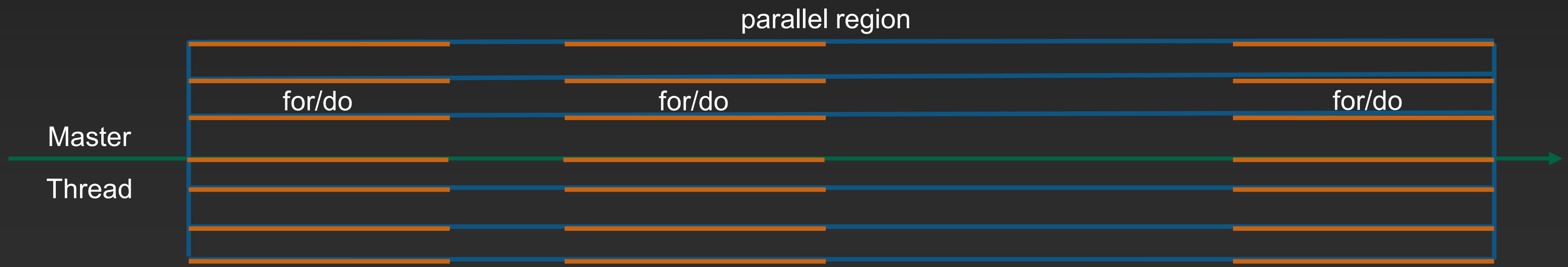
How do we deal with threads that do different things, or recursion, or anything that isn't a big loop?

We'll cover most of the remaining OpenMP capabilities, except the GPU stuff (we save that for a different talk).

Parallel Regions



What we have been doing



What we could do (*less overhead, no idle cores, finer control, more flexible algorithms*)

The parallel Construct

This sets the stage for most of the more advanced or flexible directives we are going to use. It tells the system to grab the specified number of threads and set them loose.

```
#pragma omp parallel [clause,  clause, ... ]
    structured-block
```

The clauses are

```
if([ parallel :] scalar-expression)
num_threads(integer-expression)
default(data-sharing-attribute)
private(list)
firstprivate(list)
shared(list)
copyin(list)
reduction([reduction-modifier ,] reduction-identifier : list)
proc_bind(affinity-policy) One of primary, close, spread
allocate([allocator :] list)
```

Multiple ways of specifying threads.

`In order of precedence:`

`if clause`	Logical value determines if this region is parallel or serial.
`num_threads clause`	Set this to specify how many threads in this region.
`omp_set_num_threads()`	A library API to set the threads.
`OMP_NUM_THREADS`	The environment variable we have been using.
`Default`	Often the number of cores on the node.

There is also, depending on the compute environment, the possibility of dynamic thread counts. There are a few library APIs to deal with that.

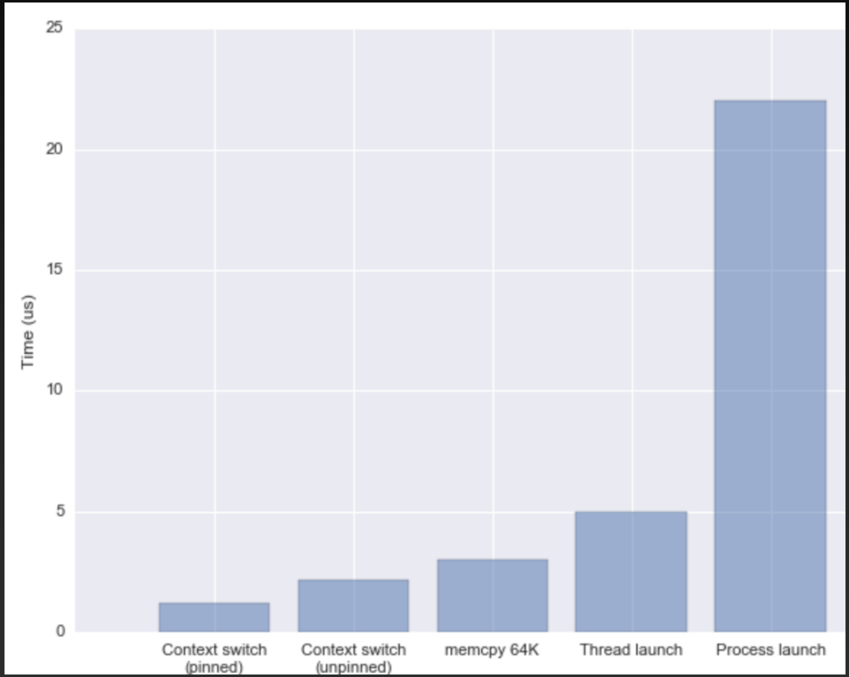
Threads Have Costs

Creating a thread takes thousands of cycles. The thread has to create its own stack, and do a context switch.

While this is much better than creating a process (which also has to create its own new virtual memory space), one must still be aware that this overhead should be justified by the amount of work done.

Switching threads also costs considerable cycles. This is why we see overhead with more threads than cores.

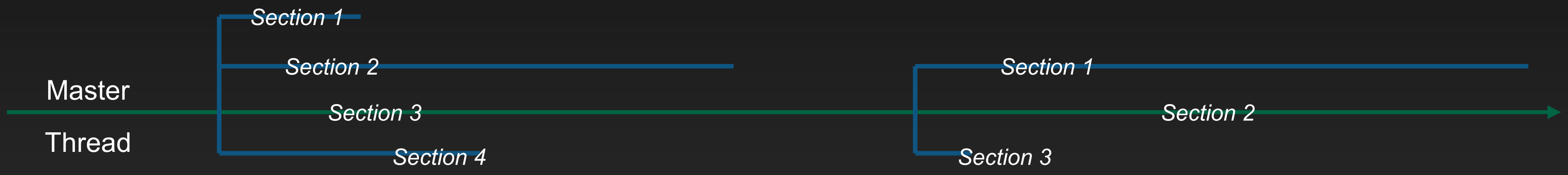
These numbers are very platform specific, and just suggestive of the typical order of magnitude.



From a very nice topical dive at
https://eli.thegreenplace.net/2018/measuring-context-switching-and-memory-overheads-for-linux-threads/

Just Do Different Things

Sections



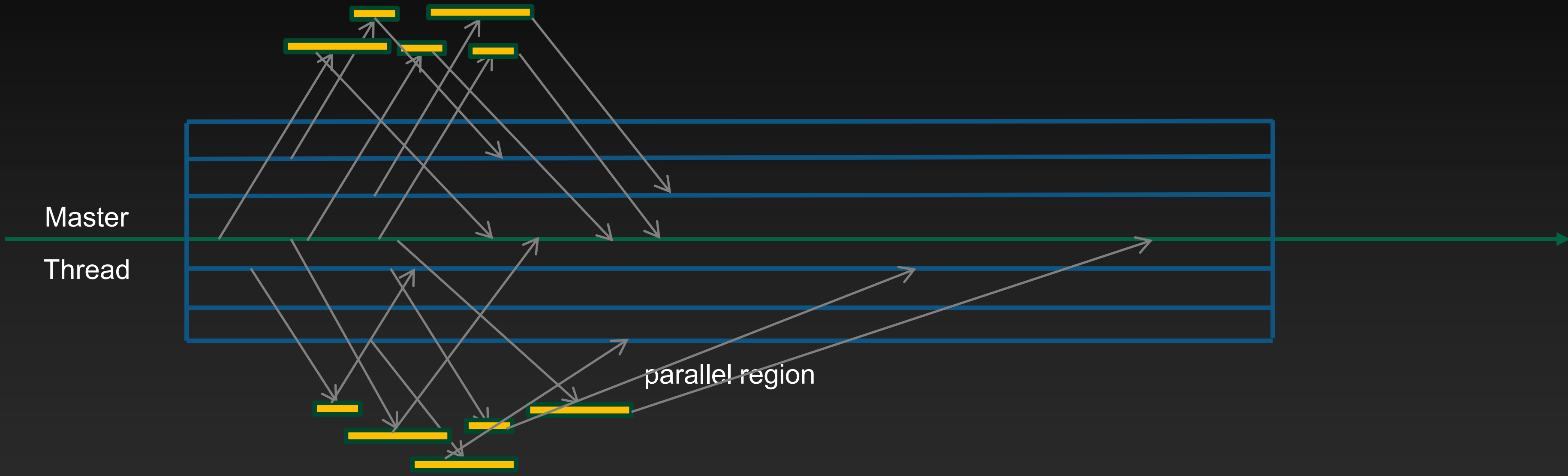
Each section will be processed by <u>one</u> thread. The number of sections can be greater or less than the number of threads available - in which case threads will do more than one section or skip, respectively.

Sections

```
.
.
.
#pragma omp parallel shared(a,b,x,y) private(index)
{

    #pragma omp sections
    {

           #pragma omp section
           for (index=0; index <n; index++)
                x[i] = a[i] + b[i];

           #pragma omp section
           if (option == 1)
                initialize();
           else
                step();

    }

}
.
.
```

```
.
.
!$OMP PARALLEL SHARED(A,B,X,Y), PRIVATE(INDEX)

!$OMP SECTIONS

      !$OMP SECTION
      DO INDEX = 1, N
         X(INDEX) = A(INDEX) + B(INDEX)
      ENDDO

      !$OMP SECTION
      IF (OPTION == 1) THEN
         CALL INITIALIZE()
      ELSE
         CALL STEP()
      END IF

!$OMP END SECTIONS

!$OMP END PARALLEL
.
.
```

For dynamic flexibility: Tasks



Actually, any thread can spin off tasks. And any thread can pick up a task. They will all wait for completion at the end of the region.

Summing An Array

Let's take the simple task of summing an array.

```
float array_sum(float *a, int length){

    float total=0;

    for (int i = 0; i < length; i++) {
        total += a[i];
    }

    return total;
}
```

Serial Code

```
float array_sum(float *a, int length){

    float total=0;

    #pragma omp parallel for reduction(+:total)
    for (int i = 0; i < length; i++) {
        total += a[i];
    }

    return total;
}
```

Easy OpenMP Version

Recursively Summing An Array

But maybe we are handed a recursive version of this same code. This represents a large class of algorithms.

```
float array_sum(float *a, int length){

    // terminal case
    if (length == 0) {
        return 0;
    }
    else if (length == 1) {
        return a[0];
    }

    // recursive case
    int half = length / 2;
    return array_sum(a, half) + array_sum(a + half, length - half);
}
```

Recursively Summing An Array With Tasks

OpenMP tasks allow us to naturally spin off threads of work.

```
float array_sum(float *a, int length){

    if (length == 0) {
        return 0;
    }
    else if (length == 1) {
        return a[0];
    }

    int half = length / 2;
    float x, y;

    #pragma omp parallel
    #pragma omp single nowait
    {
        #pragma omp task shared(x)
        x = array_sum(a, half);
        #pragma omp task shared(y)
        y = array_sum(a + half, length - half);
        #pragma omp taskwait
        x += y;
    }
    return x;
}
```

Optimized Recursively Summing An Array With Tasks

```
float array_sum(float *a, int length) {

float total;

    #pragma omp parallel
    #pragma omp single nowait
    total = parallel_sum(a, n);

    return total;
}


float serial_sum(float *a, int length)
{
    if (length == 0) {
        return 0;
    }
    else if (length == 1) {
        return a[0];
    }

    size_t half = n / 2;
    return serial_sum(a, half) +
           serial_sum(a + half, length - half);
}
```

```
float parallel_sum(float *a, int length){

     if (length <= CUTOFF) {
        return serial_sum(a, length);
    }

    int half = length / 2;
    float x, y;

    #pragma omp task shared(x)
    x = parallel_sum(a, half);
    #pragma omp task shared(y)
    y = parallel_sum(a + half, length - half);
    #pragma omp taskwait
    x += y;

    return x;
}
```

BTW, we have essentially reproduced the functionality here of the newish *taskloop* directive.

Fibonacci Tasks

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

int main()
{
  int n = 10;

  #pragma omp parallel shared(n)
  {
    #pragma omp single
    printf ("fib(%d) = %d\n", n, fib(n));
  }
}
```

```c
int fib(int n)
{
  int i, j;

  if (n<2)
    return n;

  else {

       #pragma omp task shared(i) firstprivate(n)
       i=fib(n-1);

       #pragma omp task shared(j) firstprivate(n)
       j=fib(n-2);

       #pragma omp taskwait
       return i+j;
  }
}
```

Here is one that is almost always presented as a recursive algorithm.

Task Capability

Tasks have additional directives and clauses. The most important are:

- taskwait (wait for completion of child tasks, should almost always use)
- taskgroup (can wait on child & descendants)
- taskyield (can suspend for another task, avoid deadlock)
- final (no more task creation after this level)
- untied (can change thread dynamically)
- mergable (can merge data with enclosing region)
- depend (list variable dependencies between tasks [in/out/inout]
  This provides a way to order workflow.)

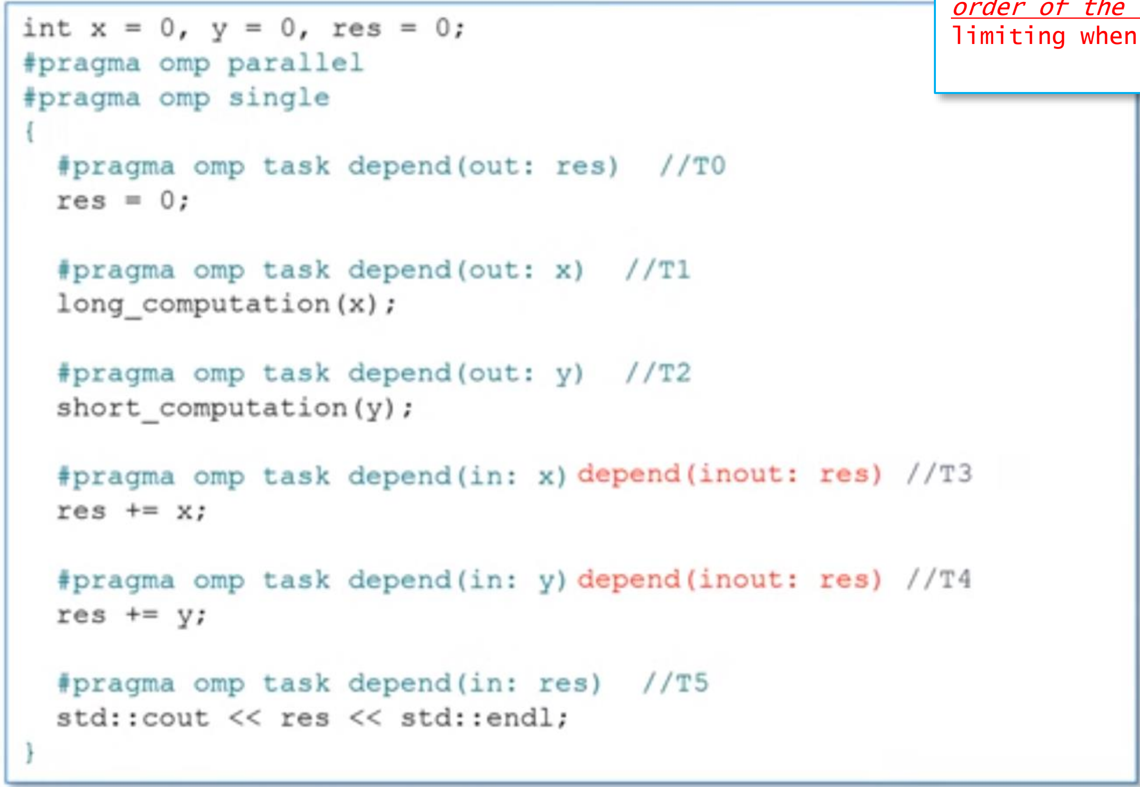
This last one gives us some very powerful capabilities to efficiently manage order dependencies, and has been an active area of OpenMP development in versions 3.0 through the latest 5.0.

From the very nice OpenMP.org video https://www.youtube.com/watch?v=YZCWPkKLVYM

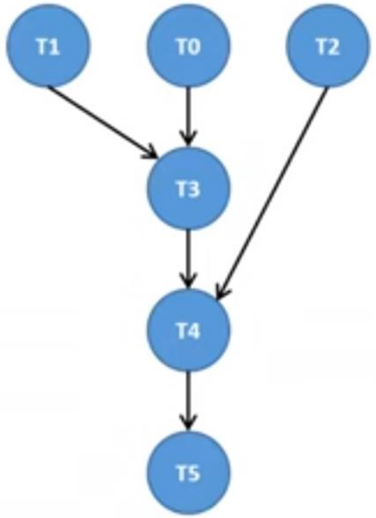
New Task Dependencies

Use the dependencies to describe what is happening to the data, not to force some execution order.

The execution order will depend up upon the *actual order of the source code*, with the dependencies limiting when tasks may be executed.

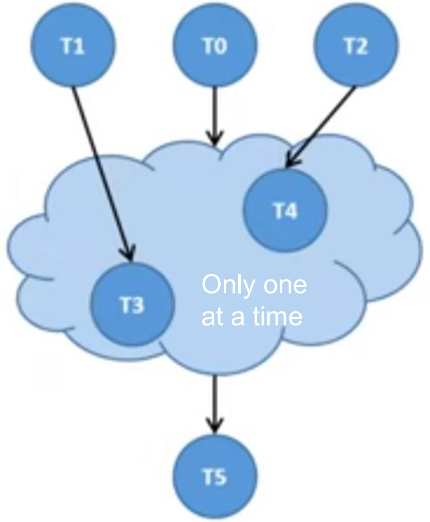


```
int x = 0, y = 0, res = 0;
#pragma omp parallel
#pragma omp single
{
  #pragma omp task depend(out: res)  //T0
  res = 0;

  #pragma omp task depend(out: x)  //T1
  long_computation(x);

  #pragma omp task depend(out: y)  //T2
  short_computation(y);

  #pragma omp task depend(in: x) depend(inout: res) //T3
  res += x;

  #pragma omp task depend(in: y) depend(inout: res) //T4
  res += y;

  #pragma omp task depend(in: res)  //T5
  std::cout << res << std::endl;
}
```



New Task Dependencies

```
int x = 0, y = 0, res = 0;
#pragma omp parallel
#pragma omp single
{
  #pragma omp task depend(out: res)  //T0
  res = 0;

  #pragma omp task depend(out: x)  //T1
  long_computation(x);

  #pragma omp task depend(out: y)  //T2
  short_computation(y);

  #pragma omp task depend(in: x) depend(mutexinoutset: res) //T3
  res += x;

  #pragma omp task depend(in: y) depend(mutexinoutset: res) //T4
  res += y;

  #pragma omp task depend(in: res)  //T5
  std::cout << res << std::endl;
}
```



Dynamic Dependencies

We can also now (as of OpenMP 5.0) deal with dynamically defined dependencies, so a list of items may include array sections.

```
#pragma omp parallel
#pragma omp single
{

     for (int i = 0; i < n; ++i)
          #pragma omp task depend(out: array[i])
          compute_element(array[i]);

     #pragma omp task depend(iterator(k=0:n),in: array[k])
     use_elements(array);


}
```

Here n is evaluated at runtime, and is the equivalent of creating n different in dependency clauses (*depend (in: array[0], array[1], array[2],…)* .

Tasks Are Very Powerful

If you really embrace this task paradigm, there is now even a *taskloop* directive that allows you to decompose for/do loops into tasks in a very controlled manner. We won't go into it here.

However before we leave these elegant heights and descend into some much grittier low-level detail, I want to emphasize that this task approach provides a powerful, and robust (as in, not error prone) framework that would have been a dream for any pthreads programmer of yesteryear. You are getting all the scheduling that they have to do at no cost.

Now, let's go back to our original parallel for/do loops and see what happens if we want to manage them at a low level ourselves...

For/Do loops in a Parallel Region

Since you already have an effective way to parallelize loops, you may wonder what the following capability is good for.

The usual answer is that you may want to avoid the overhead of starting and stopping threads, or the wasted cycles when cores go unused if they finish with one loop.

You may also have loops mixed in with a parallel region of code using one of the prior techniques.

But, as we are about to see, this approach requires more attention to detail, so you may wish to seriously consider if it is worth the optimization gains, if any.

Parallel Region Loops with C

```
#pragma omp parallel shared(t, t_old) private(i,j, iter) firstprivate(niter)
for(iter = 1; iter <= niter; iter++) {

   #pragma omp for
   for(i = 1; i <= NR; i++) {
      for(j = 1; j <= NC; j++) {
          t[i][j] = 0.25 * (t_old[i+1][j] + t_old[i-1][j] +
                            t_old[i][j+1] + t_old[i][j-1]);
      }
   }

   dt = 0.0;

   #pragma omp for reduction(max:dt)
   for(i = 1; i <= NR; i++){
      for(j = 1; j <= NC; j++){
        dt = fmax( fabs(t[i][j]-t_old[i][j]), dt);
        t_old[i][j] = t[i][j];
      }
   }
   if((iter % 100) == 0) {
       print_trace(iter);
   }
}
```

This is a simpler loop than our actual exercise's condition while loop.

Working example in slide notes below is not that complicated, but we will skip it for the nonce.

Parallel Region Loops with Fortran

```fortran
!$omp parallel shared(T, Told) private(i,j,iter) firstprivate(niter)
       do iter=1,niter
          !$omp do
          do j=1,NC
             do i=1,NR
                T(i,j) = 0.25 * ( Told(i+1,j)+Told(i-1,j)+
     $                           Told(i,j+1)+Told(i,j-1) )
             enddo
          enddo
          !$omp end do

          dt = 0

          !$omp do reduction(max:dt)
          do j=1,NC
             do i=1,NR
                dt = max( abs(t(i,j) - told(i,j)), dt )
                Told(i,j) = T(i,j)
             enddo
          enddo
          !$omp end do

         if( mod(iter,100).eq.0 ) then
            call print_trace(t, iter)
         endif
       enddo
!$omp end parallel
```

Thread control.

If we did this, we would get correct results, but we would also find that our output is a mess.

```
How many iterations [100-1000]? 1000
---------- Iteration number: 100 ------------
[995,995]: 63.33  [996,996]: 72.67  [997,997]: 81.40  [998,998]: 88.97  [999,999]: 94.86  [1000,1000]: 98.67  ---------- Iteration number:
100 ------------
[995,995]: 63.33  [996,996]: 72.67  [997,997]: 81.40  [998,998]: 88.97  ---------- Iteration number: 100 ------------
[995,995]: 63.33  [996,996]: 72.67  [997,997]: 81.40  [998,998]: 88.97  [999,999]: 94.86  [1000,1000]: 98.67
---------- Iteration number: 100 ------------
[995,995]: 63.33  [996,996]: 72.67
[999,999]: 94.86  [1000,1000]: 98.67
```

All of our threads are doing output. We only want the master thread to do this. This is where we find the rich set of thread control tools available to us in OpenMP.

Solution with Master

```
.
.
.
#pragma omp master
if((iter % 100) == 0) {
       print_trace(iter);
}
.
.
```

```
.
.
!$omp master
         if( mod(iter,100).eq.0 ) then
            call print_trace(t, iter)
         endif
!$omp end master
.
.
```

The Master directive will only allow the region to be executed by the master thread. Other threads skip. By skip we mean race ahead - to the next iteration.

We really must have an "omp barrier" after this or threads could already be altering *t* as we are writing it out. Life in parallel regions requires attention to detail!

Barrier

```c
.
.
.
#pragma omp master
if((iter % 100) == 0) {
       print_trace(iter);
}
#pragma omp barrier
.
.
```

```fortran
.
.
!$omp master
         if( mod(iter,100).eq.0 ) then
            call print_trace(t, iter)
         endif
!$omp end master

!$omp barrier
.
.
```

A barrier is executed by all threads only at:

- A **barrier** command
- Entry to and exit from a parallel region
- Exit only from a worksharing command (like do/for)
  - Except if we use the **nowait** clause

There are no barriers for any other constructs including **master** and **critical**!

Implicit Barriers

```
#pragma omp parallel shared(t, t_old) private(i,j, iter) firstprivate(niter)
for(iter = 1; iter <= niter; iter++) {

   #pragma omp for
   for(i = 1; i <= NR; i++) {
      for(j = 1; j <= NC; j++) {
          t[i][j] = 0.25 * (t_old[i+1][j] + t_old[i-1][j] +
                            t_old[i][j+1] + t_old[i][j-1]);
      }
   }

   dt = 0.0;

   #pragma omp for reduction(max:dt)
   for(i = 1; i <= NR; i++){
      for(j = 1; j <= NC; j++){
        dt = fmax( fabs(t[i][j]-t_old[i][j]), dt);
        t_old[i][j] = t[i][j];
      }
   }
   if((iter % 100) == 0) {
       print_trace(iter);
   }
}
```

The implicit barrier after the loop is saving us from another race condition.

Otherwise, some threads could finish early and go on to the second loop and modify values of t_old that are still being used in the first loop.

So, we aren't really getting much of a speedup using parallel regions here.

Other Synchronization Directives & Clauses

`single`	Like Master, but any thread will do. Has a `copyprivate` clause that can be used to copy its private values to all other threads.
`critical`	Only one thread at a time can go unnamed (only one thread in *all*
`atomic`	Eliminates data race on this one efficient than critical.
`ordered`	Forces serial order on loops.
`nowait`	This clause will eliminate implied barriers on certain directives.
`flush`	Even cache coherent architectures need this to eliminate possibility of register storage issues. Tricky, but important *iff* you get tricky. We will return to this.

Hints

These two directives now have *hint* clauses. We will wait discuss those with locks in a few slides. But they have great potential to allow your code to *automagically* avoid unnecessary waits to enter these regions.

Solution with thread IDs

```
.
.
.
tid = omp_get_thread_num();
if (tid == 0) {
   if((iter % 100) == 0) {
       print_trace(iter);
   }
}
.
.
```

```
.
.
    tid = OMP_GET_THREAD_NUM()
    if( tid .eq. 0 ) then
      if( mod(iter,100).eq.0 ) then
          call print_trace(t, iter)
      endif
    endif
.
.
```

Now we are using OpenMP runtime library routines, and not directives. We would have to use ifdef if we wanted to preserve the serial version. Also, we should include a barrier somewhere here as well.

Run-time Library Routines

OMP_SET_NUM_THREADS	Sets the number of threads that will be used in the next parallel region
OMP_GET_NUM_THREADS	Returns the number of threads that are currently in the team executing the parallel region from which it is called
OMP_GET_MAX_THREADS	Returns the maximum value that can be returned by a call to the OMP_GET_NUM_THREADS function
OMP_GET_THREAD_NUM	Returns the thread number of the thread, within the team, making this call.
OMP_GET_THREAD_LIMIT	Returns the maximum number of OpenMP threads available to a program
OMP_GET_NUM_PROCS	Returns the number of processors that are available to the program
OMP_IN_PARALLEL	Used to determine if the section of code which is executing is parallel or not
OMP_SET_DYNAMIC	Enables or disables dynami regions
OMP_GET_DYNAMIC	Used to determine if dynan
OMP_SET_NESTED	Used to enable or disable n
OMP_GET_NESTED	Used to determine if neste
OMP_SET_SCHEDULE	Sets the loop scheduling po
OMP_GET_SCHEDULE	Returns the loop schedulin ive
OMP_SET_MAX_ACTIVE_LEVELS	Sets the maximum number
OMP_GET_MAX_ACTIVE_LEVELS	Returns the maximum num
OMP_GET_LEVEL	Returns the current level o
OMP_GET_ANCESTOR_THREAD_NUM	Returns, for a given nested
OMP_GET_TEAM_SIZE	Returns, for a given nested
OMP_GET_ACTIVE_LEVEL	Returns the number of nest
OMP_IN_FINAL	Returns true if the routine
OMP_INIT_LOCK	Initializes a lock associated
OMP_DESTROY_LOCK	Disassociates the given lock variable from any locks
OMP_SET_LOCK	Acquires ownership of a lock
OMP_UNSET_LOCK	Releases a lock
OMP_TEST_LOCK	Attempts to set a lock, but does not block if the lock is unavailable
OMP_INIT_NEST_LOCK	Initializes a nested lock associated with the lock variable
OMP_DESTROY_NEST_LOCK	Disassociates the given nested lock variable from any locks
OMP_SET_NEST_LOCK	Acquires ownership of a nested lock
OMP_UNSET_NEST_LOCK	Releases a nested lock
OMP_TEST_NEST_LOCK	Attempts to set a nested lock, but does not block if the lock is unavailable
....	

Don't be intimidated.

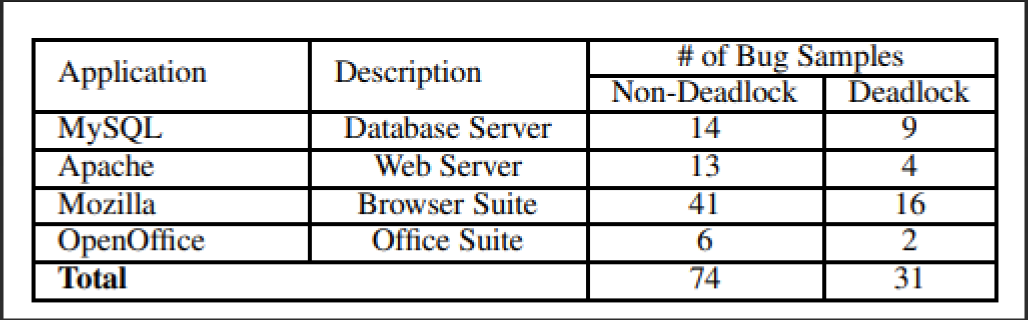
These are either the equivalent of directives, or complementary.

They can easily by mixed and matched with directives.

Concurrent Programming Is Scary!

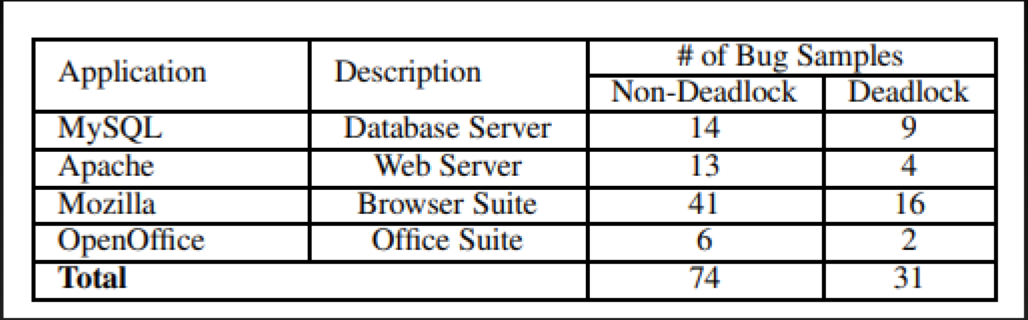
Concurrent programming, very much including threading, is considered quite intimidating and error prone by many. We have already seen, and surmounted, some of the commonly cited issues.

We are now in a position to discuss this in an informed way. A very nice study on the topic is *Learning from Mistakes — A Comprehensive Study on Real World Concurrency Bug Characteristics* by Lu, et. al. It is worth a read, and you should be able to understand it well, armed with what we have learned here.



Application	Description	# of Bug Samples	
		Non-Deadlock	Deadlock
MySQL	Database Server	14	9
Apache	Web Server	13	4
Mozilla	Browser Suite	41	16
OpenOffice	Office Suite	6	2
Total		74	31

Not Really That Many Ways To Go Wrong



Application	Description	# of Bug Samples	
		Non-Deadlock	Deadlock
MySQL	Database Server	14	9
Apache	Web Server	13	4
Mozilla	Browser Suite	41	16
OpenOffice	Office Suite	6	2
Total		74	31

The good news is that there are not really that many common ways to go wrong. The "non-deadlock" category was almost all either atomicity mistakes or race/ordering bugs like we have already discussed.

The second category is broken out for a reason. It is a common source of errors that is unique to parallel programming, and that new or inexperienced parallel programmers continually introduce to their serial code ports. And it is intermittent enough that that detecting or debugging it can be very frustrating.

Locks

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

omp_lock_t my_lock;

int main() {

  omp_init_lock(&my_lock);

  #pragma omp parallel
  {

    int tid = omp_get_thread_num( );
    int i;

    omp_set_lock(&my_lock);

    for (i = 0; i < 5; ++i) {
      printf("Thread %d - in locked region\n", tid);
    }

    printf("Thread %d - ending locked region\n", tid);

    omp_unset_lock(&my_lock);

  }

  omp_destroy_lock(&my_lock);
}
```

Output

```
Thread 2 - in locked region
Thread 2 - in locked region
Thread 2 - in locked region
Thread 2 - in locked region
Thread 2 - in locked region
Thread 2 - ending locked region
Thread 0 - in locked region
Thread 0 - in locked region
Thread 0 - in locked region
Thread 0 - in locked region
Thread 0 - in locked region
Thread 0 -
Thread 1 -
Thread 1 -
Thread 1 -
Thread 1 -
Thread 1 -
Thread 1 -
Thread 3 -
Thread 3 -
Thread 3 -
Thread 3 -
Thread 3 -
Thread 3 -
```

mutex and semaphore

These are two very similar and overlapping objects. Google them to see endless arguments over their precise definitions.

BTW, all of these are built on top of something like the CAS instruction.

This could have been done with just an omp critical!

Deadlocks

We now have the ability to create the dreaded deadlock situation.

```
Thread A                  Thread B

Lock(USB Drive)           Lock(File)
Lock(File)                Lock(USB Drive)
Copy(File)                Copy(File)
Unlock(File)              Unlock(USB Drive)
Unlock(USB Drive)         Unlock(File)
```

And many applications have multiple data structures that we want to protect with locks, so it isn't hard to introduce this bug.

Avoiding Deadlocks

This isn't an intractable problem. Your operating system has thousands of locks in use right now. An obvious solution is to have a strict ordering of all of your locks, and only acquire them in that order.

In a big code (like an OS) that may be impractical. Although one can always use the memory addresses of the locks as a free ordering scheme! Even then we do have to be aware of the order that multiple routines in the code path could be acquiring locks.

There is also a tradeoff in lock granularity. Do we have one lock to protect a whole database, or locks on each record? One lock is simpler and less error prone, but could leave a lot of threads needlessly waiting.

For the *Python* fans out there, perhaps now you understand why the Global Interpreter Lock (GIL) was introduced (especially for all the object reference counts), and why it is taking so long to eliminate. And, why it prevents true multi-threading in Python.

Lockless Data Structures

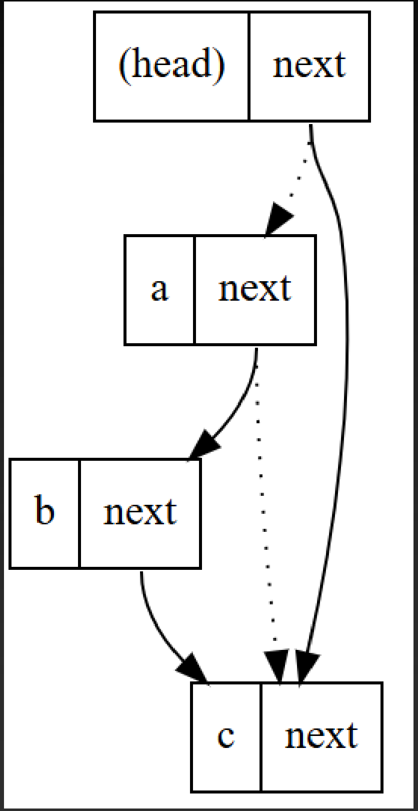
Wouldn't it be nice if we had data structures the didn't need locks? Indeed, these are possible, but require varying degrees of cleverness. A great example is the linked-list.

Inserting or deleting from a linked list has all sorts of obvious, and some subtle, ways to go wrong. We can just lock the whole list, but that is a bottleneck.

There are multiple solutions (all requiring that CAS primitive), but this remains an open area of research.

If you are interested, I recommend the Wikipedia page as a good starting point.

BTW, trying to implement a linked list in *Rust* is a very enlightening exercise! I think doubly-linked may be impossible.

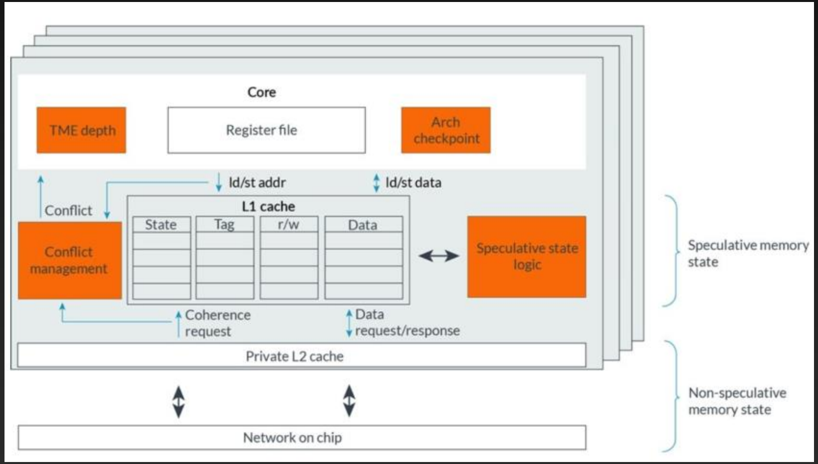


https://en.wikipedia.org/wiki/Non-blocking_linked_list

Transactional Memory

As multi-core threading became dominant, the hardware vendors saw the need to help increase the efficiency of access to contended data structures. The answer we find on modern processors is transactional memory.

Transactional memory is hardware support to capture the full state of the memory access code and data, such that it can be done speculatively and rolled back if there is a conflict. If contention is low, this allows the thread to behave as though it is lock-free.



Arm Transactional Memory Implementation
From their online guide.

This is tricky stuff. It is one of the things that bit Intel with security problems, and AMD and Arm took a long time to deploy it themselves.

Hints

OpenMP gives us an easy way to let our *atomic* or *critical* regions, and our *omp_init_lock_with_hint* and *omp_init_nest_lock_with_hint* use this underlaying hardware to our benefit. Just add one of the following hint clauses (or as a parameter to the lock).

- omp_sync_hint_uncontended: low contention is expected in this operation, that is, few threads are expected to perform the operation simultaneously in a manner that requires synchronization.
- omp_sync_hint_contended: high contention is expected in this operation, that is, many threads are expected to perform the operation simultaneously in a manner that requires synchronization.
- omp_sync_hint_speculative: the programmer suggests that the operation should be implemented using speculative techniques such as transactional memory.
- omp_sync_hint_nonspeculative: the programmer suggests that the operation should not be implemented using speculative techniques such as transactional memory.

** Nested locks are locks that can be set multiple times, and keep a count.*

flush - a step too far?

An example of the kind of low-level control you can achieve is the flush directive. An experienced concurrent programmer may want to do risky stuff like *reading and writing shared variables from different threads* (perhaps for rolling your own locks or mutexes). As shared memory machines have cache issues and compiler instruction reordering that can cause shared values to get out of sync, this is tricky business.

- implicit barriers (as mentioned previously)
- barrier (incurs synchronization penalty)
- flush (no sync)

If you think you are wandering into this territory, a good reference for examples and warnings is:

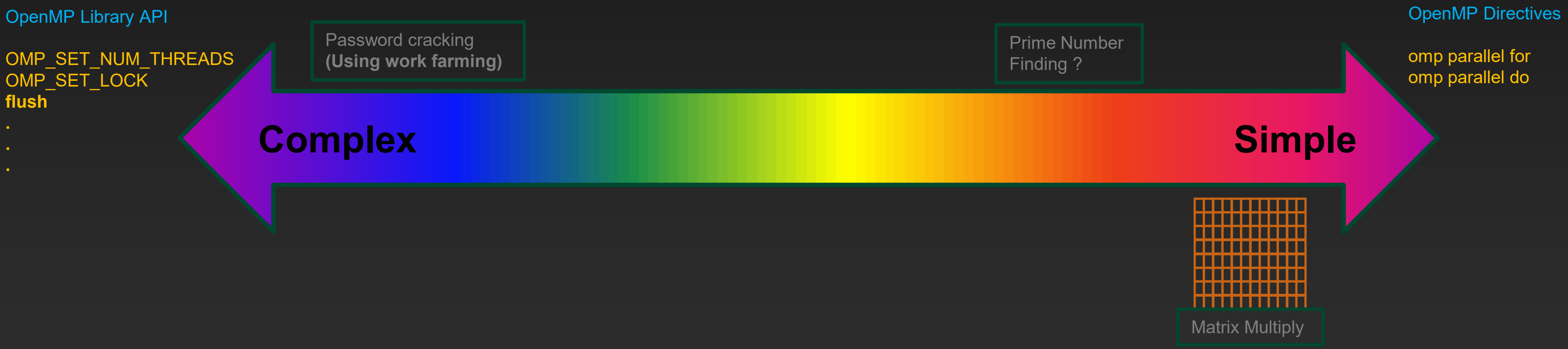
OpenMP Application Program Interface

http://openmp.org/mp-documents/OpenMP_Examples_4.0.1.pdf

Most likely none of you will find this level of control advantageous.

Complexity vs. Efficiency

How much you will gain in efficiency by using these more flexible (dangerous) routines depends upon your algorithm. How asynchronous can it be?



For HPC coders, a good general question is, how much time are threads spending at barriers? If you can’t tell, profiling will.

Opposite of tricky: Fortran 90

Fortran 90 has data parallel constructs that map very well to threads. You can declare a workshare region and OpenMP will do the right thing for:

- FORALL
- WHERE
- Array assignments

```
      PROGRAM WORKSHARE

      INTEGER N, I, J
      PARAMETER (N=100)
      REAL AA(N,N), BB(N,N), CC(N,N), DD(N,N)
      .
      .
      .
!$OMP PARALLEL SHARED(AA,BB,CC,DD,FIRST,LAST)

!$OMP WORKSHARE
      CC = AA * BB
      DD = AA + BB
      FIRST = CC(1,1) + DD(1,1)
      LAST = CC(N,N) + DD(N,N)
!$OMP END WORKSHARE

!$OMP END PARALLEL

      END
```

Scheduling

```
#pragma omp parallel for private (j) \
        reduction(+:not_primes)
for ( i = 2; i <= n; i++ ){
  for ( j = 2; j < i; j++ ){
    if ( i % j == 0 ){
      not_primes++;
      break;
    }
  }
}
```

C Version

```
!$omp parallel do reduction(+:not_primes)
      do i = 2,n
         do j = 2,i-1
            if (mod(i,j) == 0) then
               not_primes = not_primes + 1
               exit
            end if
         end do
      end do
!$omp end parallel do
```

Fortran Version

We do have a way of greatly affecting the thread scheduling while still using do/for loops. That is to use the schedule clause.

Let's think about what happens with our prime number program if the loop iterations are just evenly distributed across our processors. Some of our iterations/threads will finish much earlier than others.

Scheduling Options

`static, n`	Divides iterations evenly amongst threads. You can optionally specify the chunk size to use.
`dynamic, n`	As a thread finishes, it is assigned another. Default chunk size is 1.
`guided, n`	Block size will decrease with each new assignment to account for remaining iterations at that time. Chunk size specifies minimum (and defaults to 1).
`runtime`	Decided at runtime by OMP_SCHEDULE variable.
`auto`	Let the compiler/runtime decide.

OpenMP 5 has now added modifiers (`monotonic`, `nonmonotonic`, `simd`) for use with the above, but they seem not to be widely implemented yet.

Exercise 2: Improving Prime Number
(About 20 minutes)

Speed up the prime number count just using the scheduling options you have available.

1) Start with the prime_serial.c/f version in the OpenMP/Prime folder and then add the parallel directives as per the previous lecture slides. See how much it speeds up on various thread counts. Then…

2) Try various scheduling options to see if anything is effective at optimizing further. This "empirical" approach is a perfectly reasonable, and safe, way to find some low-hanging fruit.

One Scheduling Solution

```c
#pragma omp parallel for private (j) \
        reduction(+:not_primes) \
          schedule(dynamic)
for ( i = 2; i <= n; i++ ){
  for ( j = 2; j < i; j++ ){
    if ( i % j == 0 ){
      not_primes++;
      break;
    }
  }
}
```

C Version

```fortran
!$omp parallel do reduction(+:not_primes) schedule(dynamic)
      do i = 2,n
         do j = 2,i-1
            if (mod(i,j) == 0) then
               not_primes = not_primes + 1
               exit
            end if
         end do
      end do
!$omp end parallel do
```

Fortran Version

Dynamic scheduling with a default chunksize (of 1).

Results

We get a pretty big win for little work and even less danger. The Fortran and C times are almost exactly the same for this code.

Threads	Default (s)	Dynamic	Speedup
Serial	30.0		
2	22.3	15.2	1.5
4	13.0	8.1	1.6
8	7.6	4.2	1.8
16	4.2	2.2	1.9
28	2.4	1.2	2

25X Serial!

500,000 iterations.

OpenMP SIMD Extension

Much earlier I mentioned that vector instructions fall into the realm of "things you hope the compiler addresses". However as they have become so critical achieving available performance on newer devices, the OpenMP 4.0 standard has included a simd directive to help you help the compiler. There are two main calls for it.

1) Indicate a simple loop that should be vectorized. It may be an inner loop on a parallel for, or it could be standalone.

```
#pragma omp parallel
{
  #pragma omp for
  for (int i=0; i<N; i++) {
    #pragma omp simd safelen(18)
    for (int j=18; j<N−18; j++) {
      A[i][j] = A[i][j−18] + sinf(B[i][j]);
      B[i][j] = B[i][j+18] + cosf(A[i][j]);
    }
  }
}
```

There is dependency that prohibits vectorization. However, the code can be vectorized for any given vector length for array B and for vectors shorter than 18 elements for array A.

OpenMP SIMD Extension

2) Indicate that a function is vectorizable.

```
#pragma omp declare simd
float some_func(float x) {
       ...
       ...
}

#pragma omp declare simd
extern float some_func(float);

void other_func(float *restrict a, float *restrict x, int n) {
  for (int i=0; i<n; i++)  a[i] = some_func(x[i]);
}
```

There are a ton of clauses (private, reduction, linear, reduction, etc.) that help you to assure safe conditions for vectorization. They won't get our attention today.

We won't hype these any further. Suffice it to say that if the compiler report indicates that you are missing vectorization opportunities, this adds a portable tool.

Affinity

Memory affinity has been a non-portable pain for decades. It has steadily grown to be a very important performance consideration. Thanks to OpenMP, there is finally a portable way to deal with it.

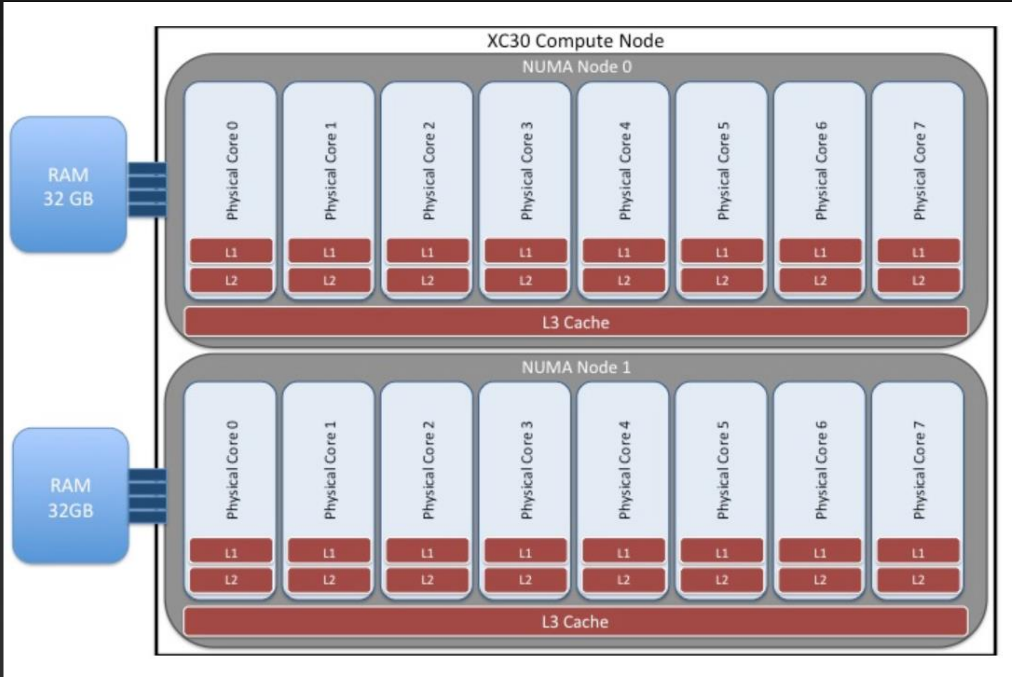
Just on a single node (our concern for OpenMP) we have:

- Registers (including vector registers)
- Caches (multiple levels)
- RAM (processor local or NUMA memory)
- HBM?
- Accelerators?
- NVM?

These are being accessed in various patterns by:

- Loops (hopefully vectorized)
- Threads
- Processes
- Cores
- Processors

ORNL Cray XC30 Node



Easy Data Affinity

Here is a good example of how easy it can be to request data/thread affinity for a couple of tasks that we know share data.

```
void related_tasks( float* A, int n ){

      float* B;

      #pragma omp task shared(B) depend(out:B) affinity(A[0:n])
      {
            B = compute_B(A,n);
      }
      #pragma omp task firstprivate(B) depend(in:B) affinity (A[0:n])
      {
            update_B(B);
      }
      #pragma omp taskwait
}
```

Thread Placement and Memory Allocation

We can also mange these issues with explicit control of our thread placement or closely controlled management of our memory allocation. These approaches have also lacked any standard methods. We only have time to present the basics here. The documentation is comprehensive:

Thread placement:

OMP_PLACES environment variable. It has lots of options and fine control mapping.
Clauses on *parallel* directive: *primary*, *close*, *spread*

Memory Allocation:

allocate clause on all data sharing directives
allocate directive
omp_alloc() and associated functions

The specifiers on these follow, and give you some idea of the kinds of hints/suggestions you can provide:

OpenMP 5.0 Memory Hierarchy Awareness

The specifiers in the new spec give you some idea of how many ways we can characterize this.

distance ≈ near, far   Specifies the relative physical distance of the memory space with respect to the task the request binds to.

bandwidth ≈ highest, lowest   Specifies the relative bandwidth of the memory space with respect to other memories in the system

latency ≈ highest, lowest   Specifies the relative latency of the memory space with respect to other memories in the system.

location = Specifies the physical location of the memory space.

optimized = bandwidth, latency, capacity, none   Specifies if the memory space underlying technology is optimized to maximize a certain characteristic. The exact mapping of these values to actual technologies is implementation defined.

pagesize = positive integer   Specifies the size of the pages used by the memory space.

permission = r, w, rw   Specifies if read operations (r), write operations (w) or both (rw) are supported by the memory space.

capacity ≥ positive integer   Specifies the physical capacity in bytes of the memory space. available ≥ positive integer Specifies the current available capacity for new allocations in the memory space.

OpenMP Environment

We've talked about a lot of tweakable configuration, and many of those parameters have multiple ways to set them (which is helpful). One convenient way I like to get a snapshot of the system is to use the OMP_DISPLAY_ENV variable to display most of the parameters. Just export OMP_DISPLAY_ENV=TRUE, or set it to VERBOSE for even more info.

```
OPENMP DISPLAY ENVIRONMENT BEGIN
   _OPENMP='201611'
  [host] OMP_CANCELLATION='FALSE'
  [host] OMP_DEFAULT_DEVICE='0'
  [host] OMP_DISPLAY_ENV='TRUE'
  [host] OMP_DYNAMIC='FALSE'
  [host] OMP_MAX_ACTIVE_LEVELS='2147483647'
  [host] OMP_MAX_TASK_PRIORITY='0'
  [host] OMP_NESTED='FALSE'
  [host] OMP_NUM_THREADS: value is not defined
  [host] OMP_PLACES: value is not defined
  [host] OMP_PROC_BIND='false'
  [host] OMP_SCHEDULE='static'
  [host] OMP_STACKSIZE='4M'
  [host] OMP_THREAD_LIMIT='2147483647'
  [host] OMP_WAIT_POLICY='PASSIVE'
OPENMP DISPLAY ENVIRONMENT END
```

C++

- private /shared, etc. work with objects
  - constructors/destructor are called for private
  - things can get complicated with firstprivate, threadprivate, etc.

- Probably biggest question is std:vector
  - Safe if no reallocation: No push_back(), pop_back(), insert()
  - Iterators are even allowed in for loop here

- Other containers less likely to just work
  - For example, std::list (a doubly linked list) updated by multiple threads would be a nightmare

- Note: MPI 3 and newer have dropped C++ specific API, so be aware if aiming for larger scalability

Information Overload?

We have now covered just about everything with the exception of the GPU oriented stuff. I hope you recall how much we accomplished with just a parallel for/do. Let's recap. In HPC the most common approach is to:

- Look at your large, time-consuming for/do loops first
  - Deal with dependencies and reductions
  - Using private and reductions
  - Consider scheduling

- If you find a lot of barrier time (via inspection or profiler) *then:*
  - Sections
  - Tasks
  - Run-time library
  - Locks
  - Barriers/nowaits

There will be projects, such as graph oriented algorithms, where it will be more natural to just start with tasks, or another paradigm.

Some Alternatives

- OpenCL (Khronos Group)
  - Everyone supports, but not as a primary focus
  - Intel - OpenMP
  - NVIDIA - CUDA, OpenACC
  - AMD - now HIP
  - Khronos has now brought out SYCL
- Fortran 2008+ threads (sophisticated but not consistently implemented)
- C++11 threads are basic (no loops) but better than POSIX
  - C++17 brings parallel STL
  - C++20 atomic smart pointers, futures, latches and barriers, coroutines, transactional memory, task blocks
- Python threads are fake (due to Global Interpreter Lock)
- DirectCompute (Microsoft) is not HPC oriented
- C++ AMP (MS/AMD)
- TBB (Intel C++ template library)
- Cilk (Intel, now in a gcc branch)
- Intel oneAPI (Includes DPC++ and extends SYCL)
- Kokkos and Raja: https://www.exascaleproject.org/research-project/kokkos-raja/