NTNU – Trondheim
Norwegian University of
Science and Technology

OpenMP: Tasks

Jan.Christian.Meyer@ntnu.no

Worksharing-style OpenMP

- The way we have been partitioning computational workloads so far depends strongly on the layout
  - We expect to have some large, multidimensional arrays
  - Iterations can proceed along each dimension in turn
  - The size of the workload is fixed and known when the loops begin
- This is pretty close to optimal for parallel for-loops
  - Great for linear algebra
  - Great for Fourier transforms

  ...but not all programs simulate physical phenomena

Issues with the worksharing view

- Suppose we start a bunch of threads and inside the parallel region, we have a loop that calls a function with an array of points:

```
#pragma omp parallel for
for ( int i=0; i<N; i+=BLOCK )
    do_something ( &(data_points[i]), BLOCK );
```

- How do we deal with parallelism at the receiving end?

```
void do_something ( mytype_t *data_points, int count )
{
    #pragma omp for
    for ( int x=0; x<count; x++ )
        data_points[x] = sqrt(x);          // ...or whatever
}
```

Here we have assumed that the threads are already online

Issues with the worksharing view

- Suppose we **don't** assume that the threads have already gone live

```
void do_something ( mytype_t *data_points, int count )
{
    #pragma omp parallel for
    for ( int x=0; x<count; x++ )
    data_points[x] = sqrt(x);       // ...or whatever
}
```

- Now we can only call the function from outside of parallel regions

  ...and get the stop/start-effect of spawning and joining threads for every block we transfer

Nested parallelism

- Nested parallelism is when one work-package spawns more work-packages that should be distributed among threads
- The worksharing directives aren't very good at this kind of thing
  - They practically assume that all threads will participate in one big loop-schedule that is ready when the loop begins
  - As we know, this is not **100%** necessary

    (we can have multiple loops with no-wait clauses in a parallel region)

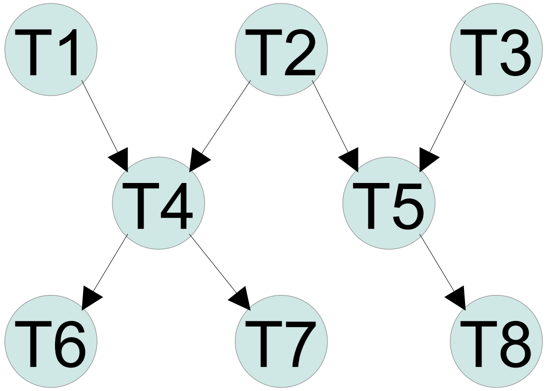
    but it's definitely the default assumption
  - Demonstrably clumsy when we want to separate the list of things to do from the team of threads that do them

Task-based programming

- This is an alternative view of how to express parallelism
- The idea is similar to the original thought behind pthreads:
  - Take a block of work and dispatch it for background execution
  - Record which blocks of work depend on the other ones
  - Assign them to the team of threads in an order that matches their dependencies

From a bird's eye view

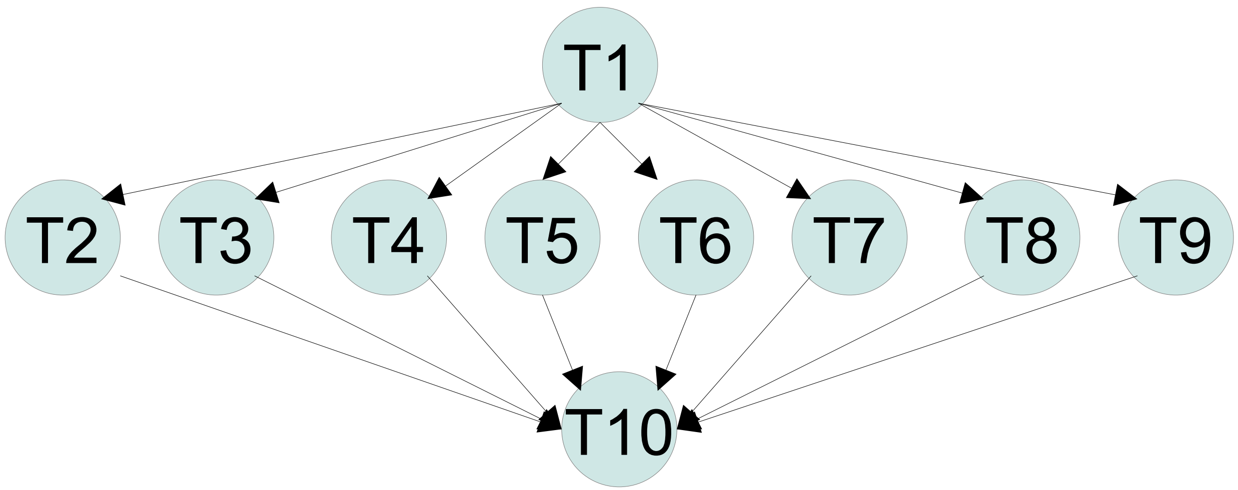
- Task-based programs generate dependency graphs behind the scenes:



- This arbitrary example-graph would suggest that
  - Tasks T1, T2, T3 can run in parallel
  - Outputs from T1,T2 are needed for T4, and outputs from T2,T3 are needed for T5
  - T4, T5 can run in parallel
  - *Etc. etc.*

Workmaking directives can be task graphs too

- Their shape is just very trivial and uninteresting:



i.e. loop iterations are independent, and synchronize when they are finished

OpenMP tasks

- OpenMP admits the creation of arbitrary dependency graphs through the *task* directive
- We can write

```
#pragma omp task
{
    do_some_stuff();
}
```

  and the block's context will be whisked away into an internal queue somewhere, to be executed at the first opportunity
- If we want to wait for all spawned tasks to be finished, there is

```
#pragma omp taskwait
```

NTNU – Trondheim
Norwegian University of
Science and Technology

Function calls can be tasks

- This maneuver

```
#pragma omp task
some_useful_function ( arg1, arg2, arg3 );
```

  will take the whole function call to ‘some_useful_function’ and make a background task out of it
- We can also declare functions to be tasks by definition

```
#pragma omp task
void some_useful_function ( int arg1, int arg2, int arg3) { … }
```

  which will task-ify every call we make to it

NTNU – Trondheim
Norwegian University of
Science and Technology

Tasks with and without threads

- You can make tasks out of things without having started a parallel region
  - They'll just be added to a list and run in sequence
- When there *is* a live team of threads active, they'll pick up tasks from the task graph in parallel

The wonderful part of this

- The body of a task is at liberty to create more tasks
  - Their dependencies can be inferred from their arguments and uses of their return values
  - Alternatively, *taskwait* directives, if you want to be explicit about it
- It's not necessary to assume any particular relationship between the thread count and the number of tasks
  - Tasks-spawning-tasks-spawning-tasks can nest as deeply as you like, they will all be run in due time

Impractical application

- Making tasks out of loop iterations serves little purpose
  - As we have demonstrated, you can do it just fine
  - There's even a directive **#pragma omp taskloop** that automates making a task out of each iteration in a loop
- It doesn't work very well with the loops we've been using worksharing directives on:
  - This only exposes the same amount of parallelism as we did before
  - It comes with the additional overhead of constructing the trivial taskgraph internally

Practical application

- Tasks come into their own when you're solving problems that are impractical to express as loops
- Divide-and-conquer algorithms are a splendid example
  - i.e. problems where the parallelizable work comes out of each nesting level in a recursive function call:

    Make a task out of the first call...

    Spawn a couple of smaller tasks at the 2nd nesting level...

    Make even smaller tasks at the 3rd level...

    Split them up into more tasks at the 4th level...

    ...you get the picture...

Starting the chain reaction

- With a recursive divide-and-conquer problem, say

```
void here_we_go() {
    #pragma omp task
    do_the_first_half();
    #pragma omp task
    do_the second_half();
}
```

it’s natural to try and write

```
#pragma omp parallel
{
    #pragma omp task
    here_we_go();
}
```

- This is a mistake
  - N threads will start N individual task-trees that all do the same thing

The common pattern

- If you have a recursive tree of function calls that spawn tasks, the top level tends to look like this

```
#pragma omp parallel
{
    #pragma omp single
    start_the_circus();
}
```

Prepare some threads to pick up the tasks

One thread starts the recursion, it will soon create enough parallel work for everyone

NTNU – Trondheim
Norwegian University of Science and Technology

Example time

- We need a divide/conquer type of algorithm
  - This is different from our classical HPC number-crunching applications
- I've gone with *quicksort*
  - You have (supposedly) already encountered this method in Algorithms & Data Structures, but we can repeat it briefly

A quick review of quicksort

- Pick a range in an unsorted array of numbers

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

(This one is only very mildly out of order, in the interest of brevity)

- Choose a *pivot* number
  - Say, 8
- Search from the low end until you find a number >pivot

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

A-ha!

- Search from the high end until you find a number <pivot

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

A-ha!

A quick review of quicksort

- When you have found two suitable numbers, swap them:

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

- When your search-pointers pass each other, the array is a little bit more sorted than it was:

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

  – One part is an unordered list of <pivot numbers
  – The other is an unordered list of >pivot numbers

Divide and conquer

- These two parts can now be passed along for further quick-sorting:

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

- Each part will have its own beginning and end
  - ...and we'll pick a new pivot in the range of each of them
- When this process has reached a list length of 1, a single number is sorted by default

NTNU – Trondheim
Norwegian University of Science and Technology

Choosing pivots

- The algorithm behaves a little bit differently depending on how you choose the pivot
  - I've gone with the MOT (median-of-three) approach:

    Compare the first, last, and middle elements, and use the median
- There are a few different ways to manipulate memory as well
  - The example implementation sorts in-place, i.e. it overwrites the un-sorted array with its sorted equivalent

Choosing programming languages

- Today’s example is written in C++
  - Sorry about that
  - The reason is that I wrote it in order to compare OpenMP with another task-friendly programming model that only exists for C++
- Hopefully, you can read it anyway
  - It’s not doing anything super object-oriented, functional, or any meta-programming, so it’s pretty C-like after all
  - If you feel it is unfair of me to swap languages mid-semester, just tell me, and I will happily translate it into plain C
  - It’s not a lot of work, I’m just not eager to rewrite things unless I know they will be useful to someone

A small disclaimer

- OpenMP implementations differ from compiler to compiler
  - I get speedup out of this implementation with GCC/g++
  - I **don't** get speedup out of it with LLVM/clang++

  - This isn't necessarily universally true, I'm just mentioning it
  - If you're on MacOS, the 'gcc' package may have installed a version of clang masquerading under the name gcc
  - Call it with '--version' if you're uncertain, and see what it answers
  - Don't ask me why the responsible package-manager people have chosen to do this, because I don't understand it

(If you understand it, I'd love to hear what the reason is)