Tutorial:
Parallel Computing in HEP

Ben Wynne

02/06/14

Tutorial: Parallel Computing with Algorithms that are Pretty Inconvenient

Ben Wynne

02/06/14

Introduction

You can find a lot of tutorials that will teach you how to parallelise some example algorithm

Typically the example will be something nicely parallelisable, like matrix multiplication

In HEP, we spend most of the CPU time in algorithms that are not so neat

This tutorial will give you an example algorithm more like those in HEP, and some tips on how to go about parallelising it

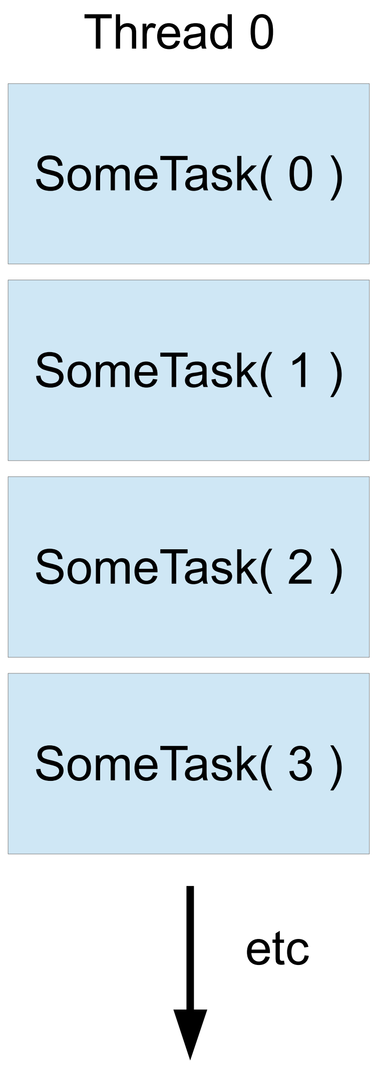
Then it's hands-on time!
How fast can you make it run?

Multithreading

To multithread a piece of code, find independent tasks that are executed in series, and execute them in parallel instead

Mostly this means looking at for loops:

```
for ( int i = 0; i < iMax; i++ )
{
        SomeTask( i );
}
```

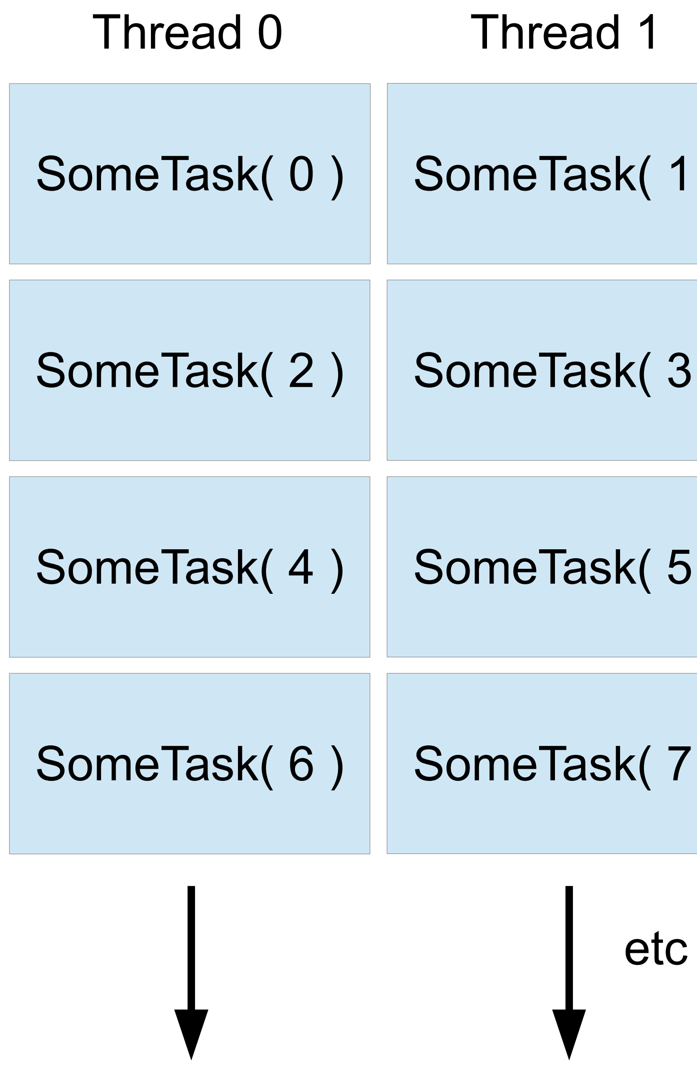


Multithreading

OpenMP provides a neat way of parallelising loops like this:

```
omp_set_num_threads(2);

#pragma omp parallel for
for ( int i = 0; i < iMax; i++ )
{
	SomeTask( i );
}
```

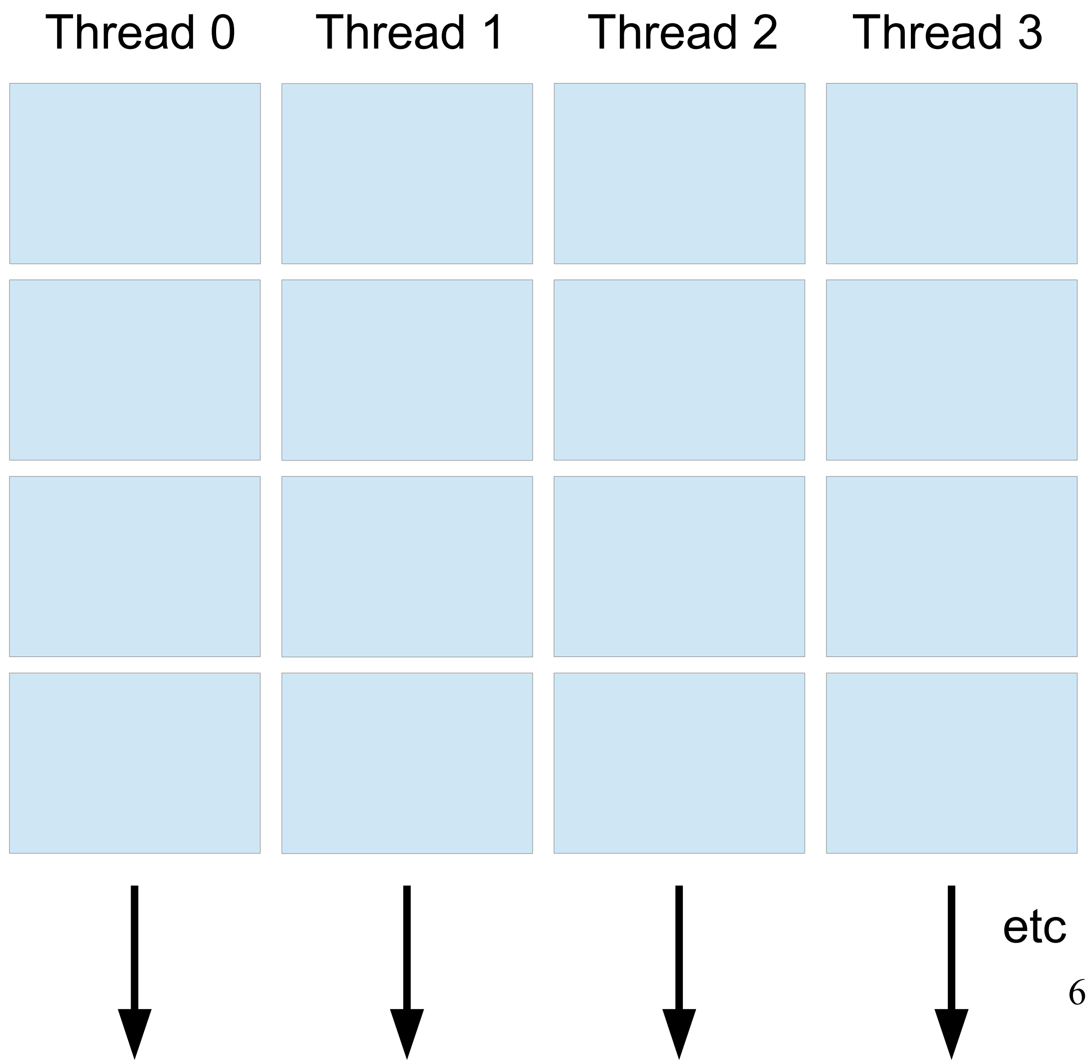


Multithreading

You can set an arbitrary number of threads, but it's usually best to limit it to the number of CPU cores available:

```
omp_set_num_threads(4);

#pragma omp parallel for
for ( int i = 0; i < iMax; i++ )
{
        SomeTask( i );
}
```



Number of CPU cores you have:

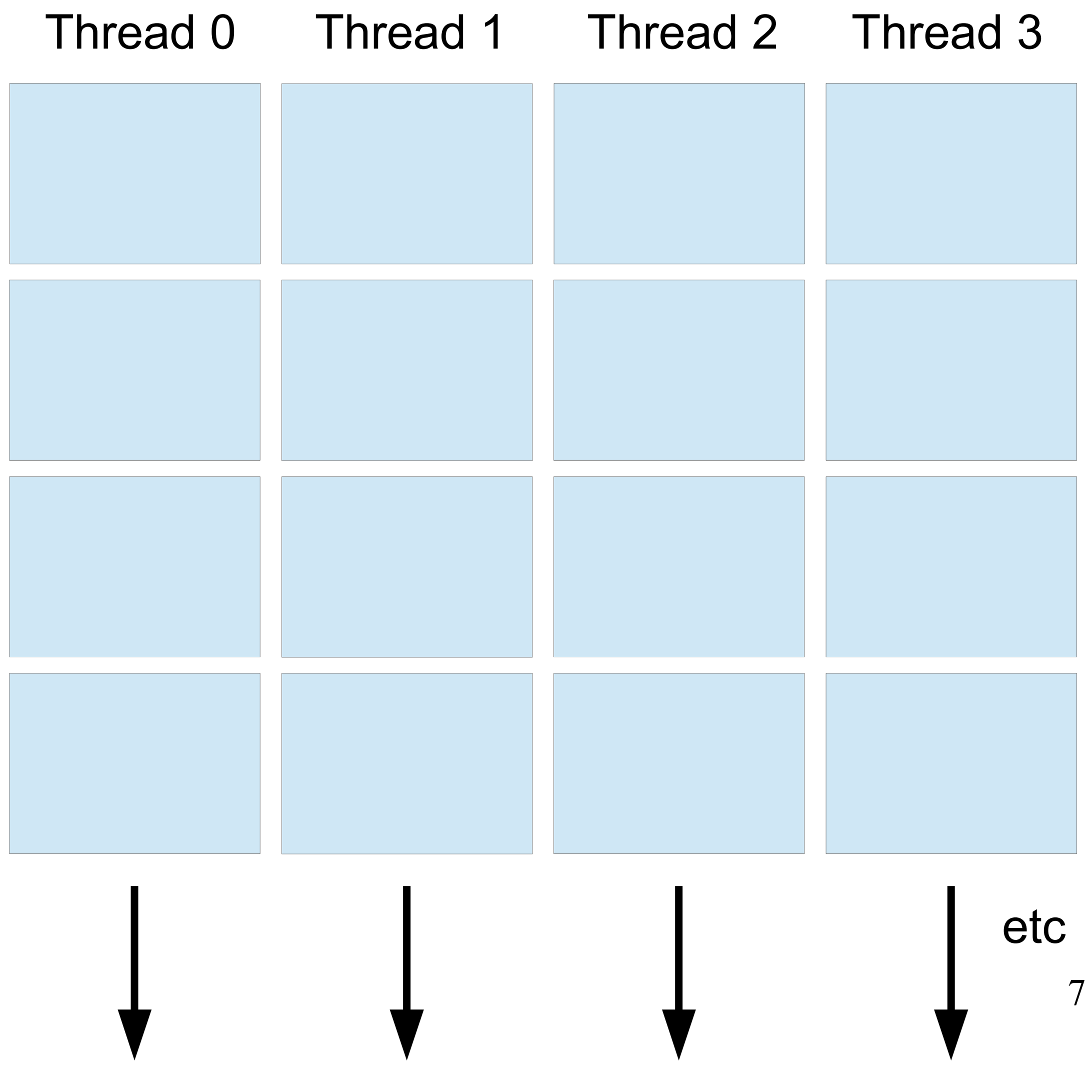
```
cat /proc/cpuinfo | grep processor | wc -l
```

Multithreading

You can set an arbitrary number of threads, but it's usually best to limit it to the number of CPU cores available:

```
omp_set_num_threads(4);

#pragma omp parallel for
for ( int i = 0; i < iMax; i++ )
{
	SomeTask( i );
}
```



Number of CPU cores you have:

```
cat /proc/cpuinfo | grep processor | wc -l
```

Thread safety

If one thread is modifying a location in memory (i.e. a variable) when another thread could be trying to read it or to modify it as well, the code is "thread unsafe."

The result is unpredictable: it will vary depending on which thread gets there first

```
omp_set_num_threads(4);

#pragma omp parallel for
for ( int i = 0; i < iMax; i++ )
{
        double result = SomeTask( i );

        total += result;
}
```

This is not thread safe

Thread safety

The best approach to a situation like this is to try and redesign the algorithm to make it thread safe

However, if you really can't get around it then OpenMP provides a way to indicate that part of the code should only be executed by one thread at a time:

```
omp_set_num_threads(4);

#pragma omp parallel for
for ( int i = 0; i < iMax; i++ )
{
	double result = SomeTask( i );

	#pragma omp critical
	total += result;
}
```

This also will be useful later when considering outputting results to disk...

NB: The sum of results from different threads is called "reduction." It is a common problem, and there are actually much better ways to solve it than this. Google if interested...

Thread safety

Since the critical pragma forces serial execution of the code, there is a corresponding performance penalty: threads have to wait for each other to get past this point

For example, this code is almost the same as on the previous page, but now the multithreading does nothing at all

```
omp_set_num_threads(4);

#pragma omp parallel for
for ( int i = 0; i < iMax; i++ )
{
        #pragma omp critical
        total += SomeTask( i );
}
```

So, think of critical as a workaround, rather than a good solution. Always try to minimise the amount of work done in critical sections

Tutorial example

The example code for this tutorial is attached to the Indico page:
https://indico.cern.ch/event/321764/

Please download and extract HandsOnCode.tgz

There is a README file inside the tarball with instructions on how to run the code. While this example is not multithreaded yet (that's for you to do!) it does have all the appropriate OpenMP linking set up, and will set the number of threads for you based on a command line argument

Once you've got the code working, take a look at the part within the comments saying "your code here." See what you make of it!

Hands-on time!
Get the code to compile and run

Tip 1: understand the algorithm

If you are presented with a piece of code to optimise, without any context, then you are unlikely to be able to do much

You need to understand
- What result the code is trying to produce
- How the algorithm works

A lot of code in HEP is there to deal with unusual cases, so it may be hard to determine what the core of the algorithm is

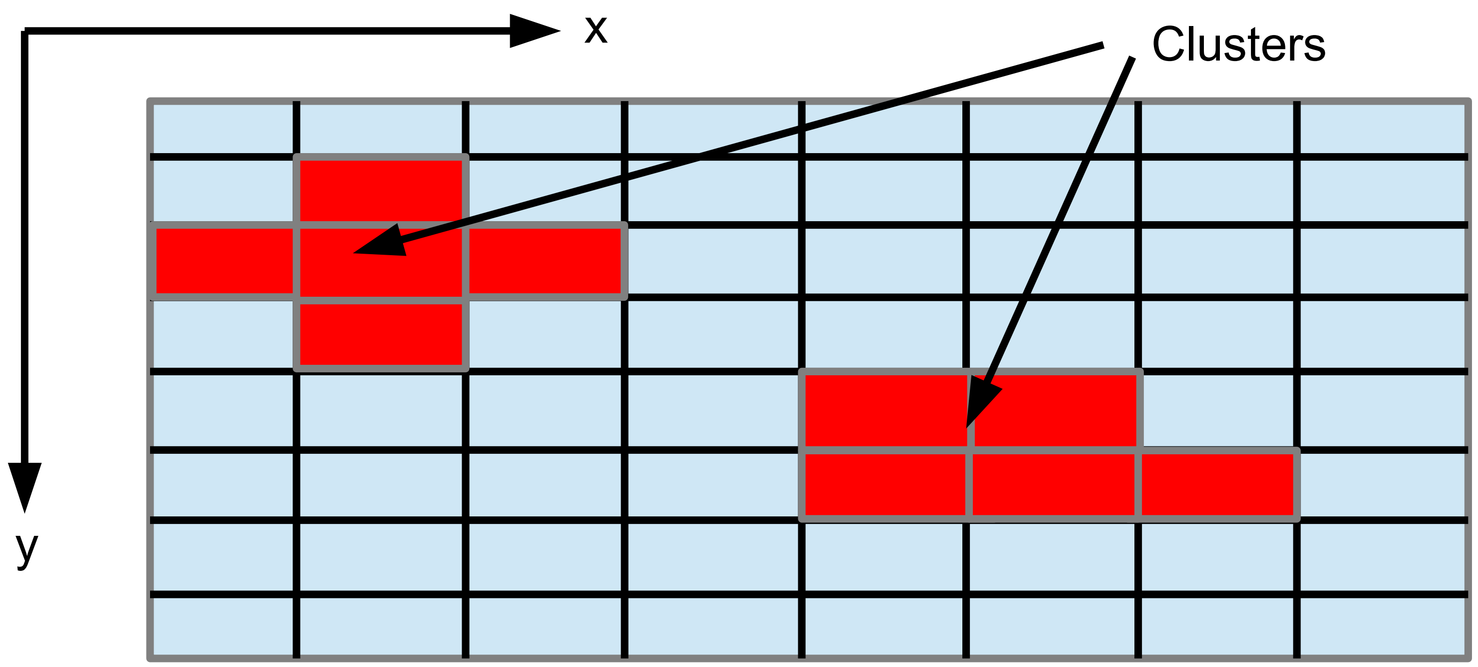
Also, any CPU-intensive code may well have had single-threaded optimisations already performed. This will tend to make it harder to understand, and much harder to parallelise

Your first step should always be to try and understand the purpose of the code you are looking at, and how it achieves that purpose. You need to go and find the documentation

Since documentation is often light on the details, or missing, you might well need to track down the original developers of the code. Check the SVN history

Example algorithm

The algorithm we're looking at is a (very simple) example of “clusterisation,” i.e. assembling adjacent deposits of charge in the detector into space-points where a charged particle is likely to have passed



The input is a list of the pixels that have had charge deposited: their x and y positions, and how much charge was deposited

We make clusters by grouping any of these input pixels that are nearest neighbours, horizontally or diagonally

Example algorithm

The first version of the algorithm has a very simple approach to making clusters

1) Create a cluster containing just the first input pixel

2) Compare each input pixel to every pixel in every existing cluster

3) If the input pixel is adjacent to any of the pixels in the cluster, add the input to the cluster

4) If the input pixel has not been added to any existing cluster, use it to start a new cluster

Once all clusters are made, find the x and y position of each one

As it stands, the code is very difficult to parallelise...

Hands-on time!
Read the code and think how to improve it

Tip 2: split up the input

Can you run your algorithm on part of the input, and get the corresponding part of the output?

If yes, then you can run the algorithm in parallel on each separate part of the input

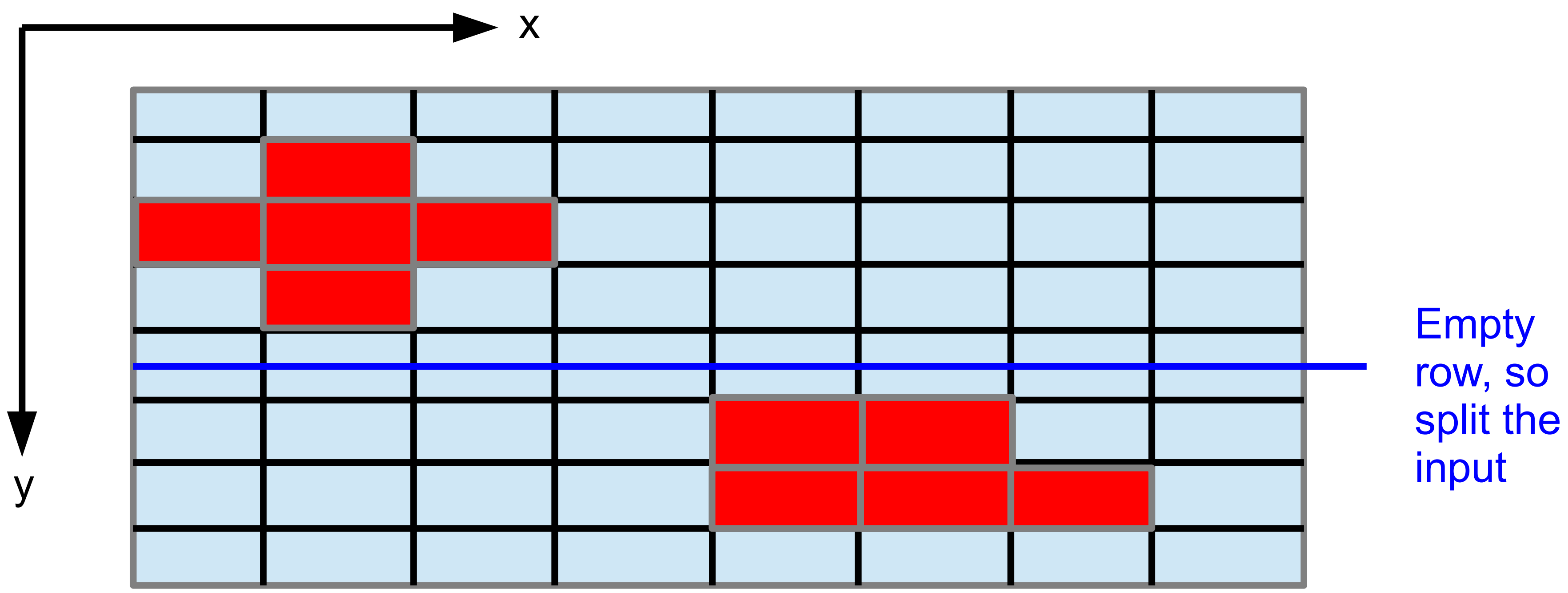
This question might not have an obvious answer – consider our example code. We have to compare each pixel with each possible cluster, right?

Wrong!

Tip 3: make the most of what you know

If the input has some structure or ordering, you can use this when you split it

Since we are only considering nearest neighbours for clustering, we can split our input every time there is an empty row or column



Since our input pixels are ordered by their y-position, we know there is an empty row whenever an input pixel is not adjacent in y to the previous input

Empty columns are hard to find though, so I haven't tried that. Any good ideas?

Tutorial example 2

Add a step to your code that splits the input whenever there is an empty row

Alternatively download my version from the agenda: HandsOnCode2.tgz

Try using OpenMP to run the main algorithm in parallel on the split parts of the input – remember to set the number of threads with the command line argument!

What are you going to do about the output?

Hands-on time!
Parallelise using the split input

Tip 4: test, test, test

The performance of a multithreaded piece of code is extremely hard to predict

- 2x the threads does not guarantee 2x the speed, that's just the ideal case
- If your code is not thread safe, the numerical results may have changed

Run your code before and after any modification, and take a look at the timing and the output

The easier it is to test your code, and the more information you have, the easier it will be to multithread it successfully

- It is quite possible to use extra CPU without speeding up the code!

It may seem like a hassle to set up a nice standalone testing environment, but you'll benefit from it in the long run

If you did the input splitting right (or used my version) then you should see that the code runs >100x faster than it did before

Wow! Multithreading is great, right?

Tip 5: better algorithms always win

The speed improvement you saw has nothing to do with multithreading

Our clustering algorithm compares each input pixel to each existing cluster. This scales badly as the size of the input increases, i.e. doubling the number of input pixels will more than double the time taken

By splitting up the input we get a huge reduction in the time taken for the clustering, even before we try to multithread

It is also what allows us to multithread the algorithm, which might give us another 2x or 4x improvement

Conversely, sometimes multithreading requires that you remove some existing optimisations that are not thread-safe

Tip 6: never stop testing

Always compare the performance of your multithreaded code with different numbers of CPU threads including just using one

To paraphrase a lot of people, “multithreading forces you to write better code”

You should always try to be clear how much of the performance gain you see is from the code improvement, rather than from multithreading that code

Tutorial example 3

There are some other example input datasets to take a look at

Try running the code on bigInput.txt and bigInput.2.txt (attached to the agenda)

How do they perform?

Hands-on time!
Take a look at the performance with the other inputs

Tip 7: data affects performance

You should have noticed that bigInput.2.txt takes far more time to process than bigInput.txt, despite the fact that it is 10x smaller

What is going on?

Remembering how important the splitting of the input was, add a debug output showing how many different parts the input is split into:

```
Using 1 threads

Finished reading 1792180 pixels from file bigInput.txt

Protoclusters by y-axis splitting: 10735

Time elapsed: 3822.98 milliseconds
```

```
Using 1 threads

Finished reading 179269 pixels from file bigInput.2.txt

Protoclusters by y-axis splitting: 1

Time elapsed: 142582 milliseconds
```

In this example, bigInput.2.txt has no empty rows, and so the splitting of the dataset doesn't work

The properties of the data affect how we can use multithreading!

Tip 8: parallel doesn't have to be clever

Now that our splitting of the input data is ineffective, we have to find another way to parallelise. Take a look at the central loops of the clustering algorithm:

```
//Loop over existing clusters
for ( unsigned int testClusterIndex = 0; etc... )
{
        //Loop over pixels in the existing clusters
        for ( unsigned int testPixelIndex = 0; etc... )
        {
                //Look for adjacency
                 if ( adjacent... )
                {
                        //Store the match
                        //Break out of the loop – job done!
                }
         }
}
```

It would make sense to search the existing clusters in parallel, as the number of clusters should rapidly exceed the number of CPU cores (the number of pixels within a single cluster is unlikely to ever grow very large)

Unfortunately, the command to break out of the loop will prevent us from parallelising

Tip 8: parallel doesn't have to be clever

So, why break out of the loop? If we skip this part then we can parallelise:

```
//Loop over existing clusters
#pragma omp parallel for
for ( unsigned int testClusterIndex = 0; etc... )
{
        //Loop over pixels in the existing clusters
        for ( unsigned int testPixelIndex = 0; etc... )
        {
                //Look for adjacency
                 if ( adjacent... )
                {
                        //Store the match
                        //Don't stop searching!
                }
         }
}
```

It seems stupid to keep looking for something once you've already found it, but the results may surprise you

In my tests, not breaking out of the loop has little effect on single-threaded performance. Parallelism then makes the algorithm up to 3x faster!

Tutorial example 4

Try adding this alternative parallel approach to your code, or you can download my version from the agenda page: HandsOnCode3.tgz

Now test out the performance again with bigInput.2.txt

Have the numerical results changed (check output.txt)? If so, why?

Hands-on time!
Try out the new parallel approach

Tip 9: parallel doesn't have to be perfect

Our new parallel method can change the results of the code

If an input pixel is adjacent to more than one existing cluster, the behaviour is undefined – it will be added to whichever cluster the threads reach last

However, our current algorithm had no strategy for dealing with this situation anyway!

While the result would have been predictable (the first matching cluster would always have been chosen), there was no physics motivation to do it that way

Now our result can be different from this, but physically it is no less valid

In real code, we probably would have a second pass through the results to deal with overlapping clusters...

Tip 10: better algorithms ALWAYS win

Doesn't it seem strange that we didn't see much performance change when we changed the algorithm to keep testing clusters even after an adjacent cluster was found?

Maybe we should think about that a bit more...

Tip 3 says “make the most of what you know” - remember that our input pixels are sorted by y, and then by x

Any clusters adjacent to our input pixel are always going to be near the end of the list of existing clusters!

Because we were searching forwards through the list, we were always going to have to wait until nearly the end of the search to find an adjacent cluster

Let's try searching backwards through the list, with the “stop if match found” command reinstated

In my tests, a reverse search is 5x faster when you exit it as soon as a match is found

Tutorial example 5

Try implementing the reverse search in your code, or download my version: HandsOnCode4.tgz

If the improved algorithm beats the multithreaded approach, what do we do now?

Does anyone have another way to parallelise the code?

Give it a try!

Hands-on time!
Experiment with new parallelism ideas

Tip 11: cheer up!

I didn't come up with any better way of parallelising the code when the y-axis splitting doesn't work

But I made bigInput.2.txt to be deliberately awkward

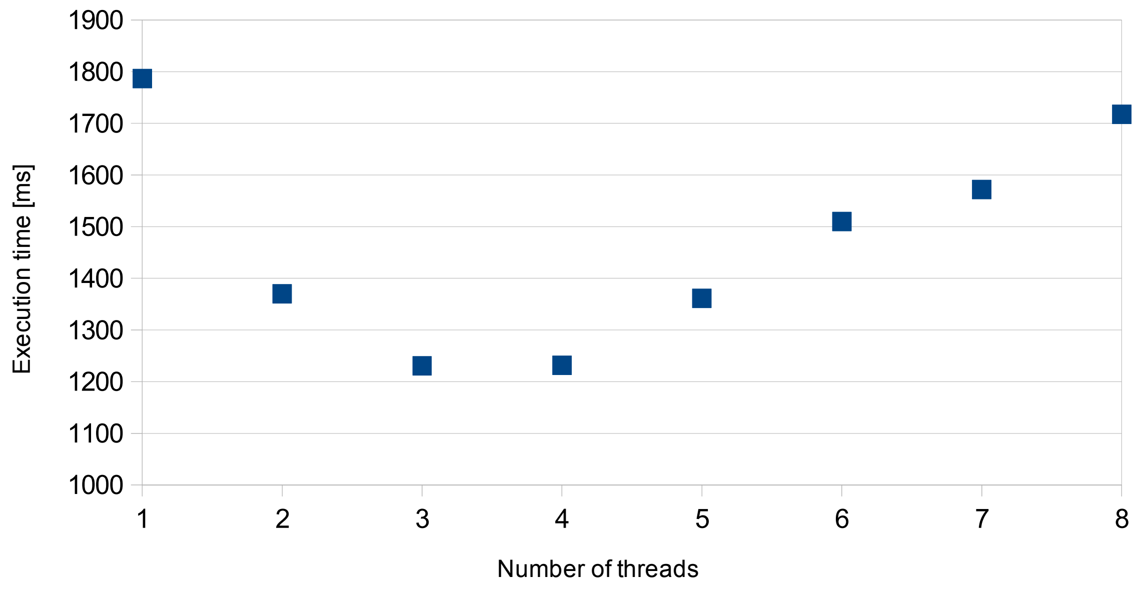
Our original method of running in parallel with the split data input is still effective when there are empty rows in the input data

Now, our algorithm improvements have made it even faster!

Tip 12: seriously, never stop testing

I found that sometimes running more threads could make performance worse

Take a look at these results, using the improved algorithm and bigInput.txt (i.e. data splitting is effective):



Why is this? Perhaps only the 4 real cores of my hyperthreaded CPU are effective? Perhaps the serial I/O step causes more congestion with more threads?

Probably a combination of factors!

Late night addendum: flatter is better

I came up with a better way to do the search

Reduce loop nesting wherever possible. Rather than searching the pixels within the existing clusters, you can search back through all pixels already examined, then look up which cluster they are in

This requires only one for loop, rather than the two nested loops

It seems to be about 10x faster this way for bigInput.2.txt

We can then parallelise the single for loop that does this search, with a critical section for storing the results. Depending on the number of threads, this is 2 or 3x faster still

Try it yourself or download HandsOnCode5.tgz

Hands-on time!
Hopefully I'm not crazy. Can you get this to work?

Summary

Hopefully this has given you a decent taste of parallelising “real” HEP code, rather than an idealised example

Parallel programming is hard! Getting it to work can be difficult, and the results may not be as you expect

The most important things to remember are:

1) Set up a good test environment, and TEST EVERYTHING you do

2) Improving the algorithm will always be the most effective approach
    - understanding the algorithm is crucial!

3) Data parallelism (i.e. splitting the input) is the easiest to use
    - figure out how to split it up
    - different inputs may give different performance

Blatant plug

We'd like to run a 2(ish) day tutorial at Edinburgh

This will cover the topics already mentioned, and will also extend into GPU and MIC devices (we have some to play with)

Probably in September/October

Please email me if interested!
bwynne@cern.ch