

## Slide 1

Hello. My name is Darryl Gove, I work in the compiler team on the analysis and optimisation of applications and benchmarks. In this session I'll be talking about utilising CMT systems. I hope that you find the presentation interesting, there will be opportunities at the end to ask questions, and I will be happy to take questions during the presentation – particularly if I need to further explain some of the content.

The final slide of the presentation contains both my e-mail address and the address of my blog. Slides and a transcript of this presentation will be available from my blog later today. If you have more questions after this presentation, do contact me for follow up discussions.

## Slide 2:

There are three main topics I want to talk about. First of all what is meant by CMT. Secondly I want to cover ways of using CMT systems to get the best throughput. Finally, I want to discuss how the developer can leverage CMT systems. In this section I will cover automatic parallelisation, and parallelisation through OpenMP. I will also cover reductions as well as data races.

I'm keeping the content of this talk at a relatively high level. If you want to explore the details, please either ask me at the end, or ask me off-line.

## Slide 3:

CMT stands for Chip Multi-Threading. The point being to focus on the fact that the chip is able to run multiple simultaneous threads, and to not worry about the implementation details of the chip.

The most well known examples of this are the UltraSPARC T1 and T2. The illustration on the page shows a stylised version of the UltraSPARC T2. The abstraction is that this is a single chip which contains 64 virtual processors. If you look under the wrapper of that, you'll find that the implementation is eight cores, and each of those cores is capable of running eight threads. Part of the point of the CMT moniker is that you should not need to be concerned about this level of detail; to first order this is a CPU that can handle 64 simultaneous threads.

It should immediately be apparent that the only way to get the best performance from a CMT system is to keep the virtual processors busy. A single thread will only use 1/64<sup>th</sup> of the chip.

## Slide 4:

There are a variety of ways that a CMT system can be utilised effectively. The easiest thing to do might be to run 64 processes, each process can occupy a single core, so the system is fully occupied. There is a concern with this approach. If the system becomes over subscribed, then critical processes could get starved of compute resources.

An improvement over this is to use processor sets. A processor set is a group of processors. Only tasks assigned to that set can execute on the processors in that set.

With both processes sharing an OS, and processor sets, all the processors share the

same view of the system. It is possible for one process to communicate with another processor, or perhaps even to overwrite data from another process.

Solaris Zones can be used to improve the security of the applications running on a system. Each Zone appears to the applications running in it to be the entire system – the applications within a Zone cannot “break out” and see the applications running on the system outside of that Zone. This gives much improved security. Actions taken by one application in a Zone cannot affect a different application in another Zone.

All the Zones share a single Solaris instance. So if an application were to somehow cause a problem with this single operating system, then it would have an impact across all Zones.

One further step that can be taken is to use logical domains to split the hardware into separate machines. Each machine can have its own operating system – they could be different versions of the same OS, or even different OSs.

All these approaches are basically about running multiple applications on a single system. In some situations the user might want to run a single application, and still fully utilise the system.

Slide 5:

There are a number of technologies that can be used for developing multi-threaded applications. The common options are POSIX threads, OpenMP, and automatic parallelisation.

POSIX threads present the developer with the most complete set of options for developing parallel code. The developer is responsible for controlling all aspects of the threading, which can make them hard to use, and can also make the process of writing parallel code error-prone.

OpenMP is a specification for compiler directives that tell the compiler how to parallelise the application. An example of this is a directive that tells the compiler that the next loop should be run in parallel over multiple threads. Since the parallelisation is directive-based, it is relatively simple for the user since they just use the directives to tell the compiler what to do. The compiler does the hard work of managing and assigning work to the threads.

The OpenMP specification was recently enhanced to reach 3.0. This new version enables the parallelisation of many different types of applications.

Automatic parallelisation is enabled through a couple of compiler flags. This makes it incredibly easy to use. Currently it is limited to loop-based parallelism, which does limit the situations where the compiler can identify code that can safely be run in parallel.

In this presentation I am going to focus on automatic parallelisation and OpenMP since these two approaches represent the easiest ways of developing parallel applications.

Slide 6:

The first example I'm going to show is this simple loop that clears an array by writing zeros to it. There are three compiler flags used. The flags `-xvpara` and `-xloopinfo` tell the compiler

to output verbose information about the parallelisation generated. These flags are useful for both automatic parallelisation and for OpenMP, and enable you to check whether the loops in your code are actually parallelised by the compiler.

The flag that enables automatic parallelisation is `-xautopar`. You can see from the output that a parallel version of the loop is generated, together with a serial version which is called when the trip count of the loop is too low to make using the parallel version profitable.

Slide 7:

It is very useful to understand the concept of “reductions”. These are situations where a range of data is reduced down to a single value.

In this example we are calculating the sum of the elements in an array.

Notice that when the code is compiled, the compiler does not parallelise that loop, reporting an 'unsafe dependence on the variable “ret”'. The problem is that performing the summation in parallel may cause a difference in the final value, any difference would typically be small, but the compiler has to make the safe choice.

The flag to tell the compiler that it is safe to generate reductions is `-xreduction`. In the presence of this flag the compiler does generate a parallel version of the loop.

Slide 8:

This is the same loop parallelised using OpenMP. The compiler flag to get enable the generation of parallel code using OpenMP directives is `-xopenmp`.

The directive is a single line in the source code which specifies that the next for loop should be made parallel. It is also necessary to specify how the variables within the parallel region are used. In this case, we specify that the variable “total” is a reduction.

Slide 9:

One of the most difficult aspects of adding the OpenMP directives is identifying the scoping of the variables used in the parallel region. There is a Sun-specific extension which tells the compiler to work out the appropriate scoping for the variables.

Slide 10:

The other parallelisation directive supported in the OpenMP 2.5 specification (which is the specification supported by Sun Studio 12) is parallel sections.

The developer can use parallel sections to identify regions of code that can be safely executed in parallel. In this example there are two blocks of code that can be executed in parallel.

Slide 11:

OpenMP 3.0, which is supported in the recent Sun Studio Express releases, introduces the idea of task based parallelism. This is a very flexible parallelisation mechanism. The

basic idea is that a thread can queue a task for later execution. The task will either be executed by the thread doing the queuing, or by some other available thread.

Slide 12:

There are a number of advantages to OpenMP.

The specification is directive based, so the modifications to the code are the addition of a few parallelisation directives. This typically makes it very easy to add to a serial code.

More importantly, the work of generating the parallel code is done by the compiler – reducing the chance of error. A good example of why this is helpful is that the user does not have to explicitly manage the creation or destruction of the worker threads.

Applications can be parallelised incrementally, so the developer can initially start with a single time consuming routine, rather than having to do major restructuring of the code.

Finally, the directives in the code are only recognised when the compiler flag is used. This means the same source base can be used to generate a serial and a parallel version of the application. These two versions can be compared to check whether any problems reported against the application are down to the parallelisation of the code, or are present in the serial version.

Slide 13:

There are some things to consider about writing parallel codes.

First of all the threads in an application can execute at different rates. It's not possible to assume that one thread will always finished before another thread. If synchronisation is necessary, then it has to be coded into the application.

The next thing to consider is that data is shared between threads through memory. A modified variable is not visible to another thread until after that value has been stored to memory and then reloaded by the second thread. This is usually ensured by annotating all shared variables as being volatile.

The final thing to realise is that you need to be careful to avoid one thread modifying a variable that another thread is using. This kind of error is called a data-race.

Slide 14:

This is an example of a data-race. Two threads are reading, modifying, and then writing the variable A. If the two threads ran at different times, then A would contain the value 7. However, both threads run at the same time, so the value left in the variable A depends on the result of the last thread to write its value back to memory – in this case the value 4 is left in the variable.

As with most bugs, data-races can be hard to detect because the effect of the error may not be visible until long after the error has occurred. Fortunately the Thread Analyzer in Sun Studio can be used to identify these kinds of errors.

Slide 15:

The steps to parallelising a code are relatively straightforward in abstract. Of course, following them on a real application can be tricky.

The first thing to do is to profile the application. You need to identify where the time is being spent. Sometimes the time is spent in a single loop, often it is going to be spent on a callstack.

The trick is to identify the largest possible region of code that can safely be executed in parallel. You need large regions of code to ensure that each thread performs a significant amount of work before having to synchronise with the other threads. The synchronisation costs are often what will limit the scalability of the application.

Once the code has been made parallel it is important to check for data races. Using `autopar` or `OpenMP` is a great way of reducing (or eliminating) the chance that the code will contain data-races. However, it is not always clear that parallelising an application in a certain way will introduce a data-race, so it is best to check.

The final thing to do is to profile the application and check that the changes have lead to performance gains.

The process is likely to be iterative. Once one region of code has been made parallel, there are going to be other regions which still consume significant runtime.

Slide 16:

So I hope that this talk has given you a good overview of the technologies that are available for fully utilising CMT systems.

In particular, I hope that you feel more comfortable with some of the technologies that are available for writing parallel applications. In particular I would be thrilled if you tried using `OpenMP` on some codes that are important to you.

Slide 17:

This final slide contains my e-mail address and a link to my blog. I'd like to open up the discussion for questions.