



---

## **Using DDT to debug applications on the IBM Cell BE processor**

August 2007

---

## Using DDT to debug applications on the IBM Cell BE processor

The Cell Architecture is the result of collaboration between IBM, Sony and Toshiba to design a high-performance and power-efficient processor that could drive applications in the fields as diverse as gaming, HDTV and supercomputing.<sup>1</sup>

The Cell is an innovative heterogeneous multiprocessor consisting of:

- The PPE – The Power Processing Element containing an IBM 64-bit Power Architecture™ core, the so-called Power Processing Unit (PPU).
- Eight SPEs – Specialized co-processor units each containing a Synergistic Processing Unit (SPU).

with a coherent on-chip bus for communication between the elements.

Whilst the PPE has a familiar processor, with ordinarily at least 256Mb RAM (global memory) available, each SPE has a large register set and a local store of 256Kb. Access to the global memory from the SPEs is performed using DMA via the bus, or by exchanging messages with the PPE through a “mailbox” mechanism.

### *Programming the Cell*

Standard PowerPC programs will run unmodified on a Cell system such as the IBM QS20 BladeCenter or the Sony PS3™ which can both run the Fedora™ or Yellow Dog™ distributions of the Linux operating system – using just the PPE. However high-performance computing users of the Cell will be keen to exploit the capability of the SPUs fully. The choices for this fit broadly into three models:

- Transparently – using libraries that have been optimized for the Cell architecture. Many HPC codes use proprietary or open-source libraries for the computational kernel of the application, and these may have been ported to the Cell. Simple re-linking will enable the application to use the new

libraries.

- Intermediately – using advanced languages or compiler directives. A number of third parties provide compilers and/or libraries that can optimize the transfer of data and the computation between the SPUs and the PPU. This can involve rewriting the computational kernel of an application, or just adding compiler directives rather similar to parallelizing using OpenMP.<sup>2</sup>
- Directly – the developer writes applications for both the SPUs and the PPU and has responsibility for data transfer and synchronization. This is typically used when wanting to directly control the behaviour of the Cell, or to hand-optimize the performance of the computational kernel, or for code that does not fit well with the patterns supported by the advanced languages.

### *The Need to Debug*

Inevitably, where there is programming, there is debugging. This white paper introduces Allinea's DDT, a debugger for multi-threaded and parallel applications which has enhanced support for the Cell.

Like programming the Cell, the process of debugging can also differ from a user's previous experience. Historically trying to find bugs by placing “print” statements in code has been popular, but this has always led to a repeated cycle of modify-compile-run that can be rather slow to achieve results.

This method does not work well in multi-threaded environments, and this is particularly true with the Cell where a print issued by the SPU will lead to communication between the PPU and SPU to produce the output which can also change the behaviour of the bug.

Graphical debugging tools, like Allinea's DDT, represent a leap forward beyond the print solution. At runtime the user can control progress of a program, see all the values of its variables, its

memory and the current execution stack – providing far more information, more flexibly, than print statements can. This makes bug fixing a quicker and less frustrating task.

## Debugging Cell Applications

With applications written in the “direct” model for Cell, the debugging need is greatest. At this level extensions to enable SPU debugging and PPU debugging concurrently are essential. Developers must be able to see variables and memory across every part of the processor.

For the “intermediate” model of programming – using the advanced compiler tools and languages for Cell – a full Cell debugger can give a better view of program state than a standard debugger – for example by allowing users to see the active SPE threads and check their progress at the same time as monitoring the PPU.

DDT is a powerful cross-platform graphical debugger, used widely in industry, government and academia for high-performance computing applications – for programs that typically can have high degrees of parallelism, sometimes thousands of processors simultaneously running the same application, but often on smaller clusters of Linux machines. It is DDT's capability to debug these kinds of application that make DDT a great choice for debugging Cell – with its inherently parallel model of execution. In addition to support for parallelism through intuitive process controls, DDT also has features not found in many debuggers – for example sophisticated memory debugging that can find common errors with heap memory or detect illegal reads.

## How to Debug with DDT

DDT is a source level debugger – it allows the user to see source files and examine the detail of various threads as they progress through execution.

Cell applications consist of two components: the code for the PPU, and the code for the SPUs. The SPU code is usually embedded in the PPU binary at

link time using special tools in the IBM SDK for Cell programming.<sup>3</sup>



Source File Showing Thread Position

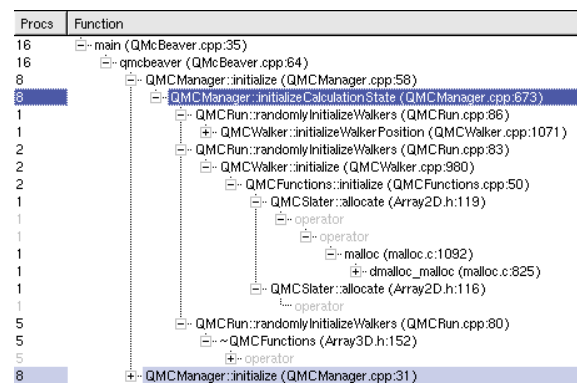
When DDT loads an application for debugging, it will automatically detect the presence of both PPU and SPU code, and find the source files for both sections.

Initially the program will stop at main(), the entry point to the code. Then, as each SPU thread is started, DDT can stop the process, so that the user can see how the thread was created, and, at termination of an SPU thread, DDT can stop allowing the reason for termination to be easily determined. Tracking thread termination is an important task – many errors inside SPU threads can cause immediate, and often difficult to understand, thread termination. Examples include exhausting the stack or the available space in the local store.

## At a Glance Program State

When a program is first paused inside DDT, there are many parts of the DDT interface that can help the developer get an understanding of what is going on with the code.

The source code is highlighted showing which lines have threads on them, and by simply hovering the mouse, the actual threads present are identified.



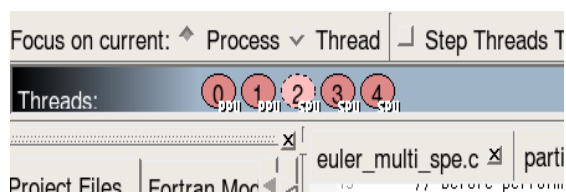
Parallel Stack View Showing All Threads

The Parallel Stack View is another very popular feature for showing divergent behaviour across processors and threads. This window displays the call stacks of every thread – optionally including every thread of every process for a program with multiple Cells – in a single tree view.

Where threads share a common stack – this meaning they are in the same part of a program – then they will be on the same branch of the tree. The number of threads at each part of the tree is shown, which means it is simple to find the threads that are behaving differently from the norm just by looking at those branches containing only one thread.

## Controlling the PPU and SPUs

Across the top of DDT's main window, the SPU and PPU threads are shown – they are identified by the “PPU” and “SPU” suffixes underneath the thread number. By clicking on a thread to select it, this thread becomes current – which means that the variables being shown are those present on that particular thread.



DDT Showing SPU and PPU Threads

Breakpoints can be added to program to stop PPU and SPU threads when a particular point in the program is reached. DDT also allows breakpoints to be set that apply to all, or just one of the threads.

DDT can control all the threads, or just one at a time, this allows the user to focus on an individual PPU or SPU thread – stepping one line of code at a time whilst the other threads are paused. Following individual threads through a computation is one of the most useful capabilities of a debugger.

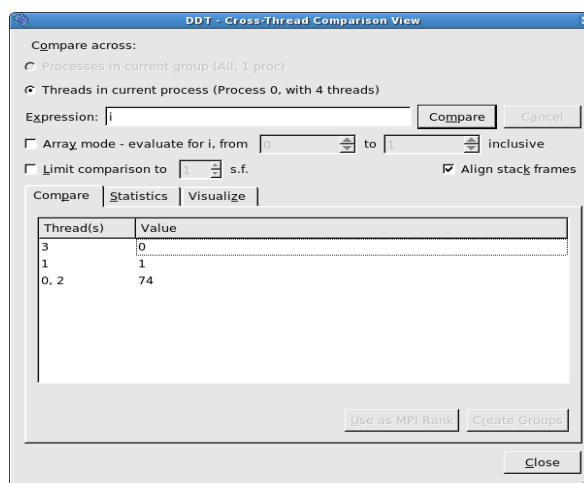
Often a Cell code will exhibit SIMD (Single Instruction Multiple Data) parallelism – each SPU thread executing the same part of a code at the same time. In these situations it can be instructive to step each thread ahead a line at a time – DDT has a special mode for this, selected by toggling the “step

threads together” button. When threads diverge, it can be the source of a problem.

## Examining Variables

Each SPE and PPE has its own addressable memory region. For the SPE, the local store holds variables – such as automatic stack variables, or global or static variables – and also local heap memory allocated using the malloc() system call within the SPE.

There are some traps to SPU programming – a global variable is only “global” on the same SPE – each SPE and PPE will have a different variable, unlike conventional multi-threaded programming where processes share the same memory.



Comparing Variables to Detect Bugs

DDT can help spot problems like this – it has tools such as the Cross Thread Comparison window which will examine the variables with the same identifier across each SPU and PPU thread. This is ideal for spotting rogue values, the common values are also grouped together so that the developer can really focus on the differences.

If an individual SPU or PPU thread is selected, the values of its variables are shown in the Local Variables tab. Dragging the current line markers in the source code will select the variables shown on each selected line, and evaluate these too in the Current Line tab. For user defined data types – structures, classes and derived types – the contents can be opened up by just clicking on the variables.

Locals		Current Line(s)	Stack
Variable Name	Value		
argc	1		
argv	0xbfa92ac4		
beingWatched	7177107		
bigArray			
dest	2		
dynamicArray	0xb7e86008		
environ	0xbfa92c00		

*Examining Local Variables with DDT*

Variables can also be dropped into the Evaluations box for more detailed probing. Pointers can be followed from the evaluation window, by right clicking and selecting “dereference”.

## Low-Level Cell Status

There are additional tabs in DDT that show the current status of DMA requests and mailbox events – this information is directly provided by the kernel and can show important data such as the current contents of the mailboxes that are used to communicate between the PPU and SPUs. SPUs and PPUs have, in total, three mailboxes – two for SPU

Locals		Current Line(s)	Stack	Event	Mailbox	Proxy-DMA
Variable Name	Value					
j	509			<b>SPU Outbound Mailbox</b>		
left	976			0xc0000000		
parm	4268421168			<b>SPU Outbound Interrupt Mailbox</b>		
spu_id	28901968			0xc0000000		
tag_id	0					
time	7					

*Directly Displaying Mailbox Status*

to PPU communication (one interrupting the PPU, the other must be polled) and one for PPU to SPU communication (polled).

## Summary

The IBM Cell brings extensive benefits to demanding users in terms of performance, but to exploit these fully requires a new way of programming. This paper has shown how DDT can be used to simplify the transition to Cell programming by providing an easy to use and highly capable debugger which is able to debug complete Cell applications. Not only can DDT debug multiple

threads within a single Cell, DDT can also be used to debug clusters of Cells.

DDT for Cell is available from <http://www.allinea.com>. Supported platforms are the IBM QS20 Blade Center and Sony PS3™ running Fedora Core 6 and IBM Cell SDK 2.1.

## About Allinea Software

Allinea Software is a leading supplier of development tools for parallel and high performance computing. Allinea was founded by computer scientists from Warwick and Oxford Universities, giving the company unrivalled expertise in scientific and parallel computing and an insight into the challenges of exploiting parallelism as it enters the mainstream.

## References

1. The Cell Project at IBM Research (<http://www.research.ibm.com/cell>)
2. RapidMind (<http://www.rapidmind.net>) and Cell Superscalar (<http://www.bsc.es/cellsuperscalar>)
3. The IBM Cell Software Development Kit (<http://www.ibm.com/developerworks/power/cell/>)