

Implementing Applications on a Cellular Architecture - the Mandelbrot-set.

Jason McGuiness^{1,2}, Colin Egan², Guang Gao¹.

20th June 2003

¹University of Delaware, Newark, DE.

²University of Hertfordshire, Hatfield, Hertfordshire, U.K. AL10 9AB.

mcguines@capsl.udel.edu

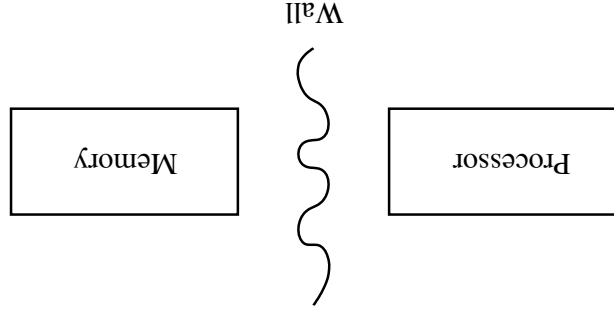
c.egan@herts.ac.uk

ggaoo@ee.udel.edu

Overview:

- The memory wall and cellular architectures: a solution?
- Programming models on Cellular Architectures.
- A Brief Overview of Cyclops and DIMES/P.
- An introduction to the Mandelbrot set.
- Threading and Work-Stealing applied to the Mandelbrot set.
- The programming implementation with regard to DIMES/P.
- Execution Details of the Mandelbrot-set application.
- Conclusions & Future Work.

A recap on the memory wall. Part I: The processor viewpoint.

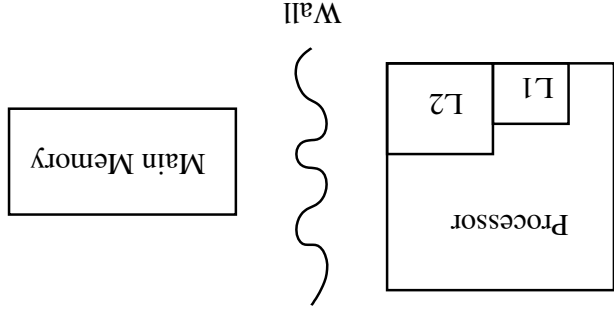


- Higher performance may be achieved through ILP by MII and/or pipelining. Various techniques are used to implement these goals, e.g. Register Renaming, Out-of-order instruction issue/execution, Branch Prediction, dynamic instruction scheduling, Value Prediction, Instruction Reuse, etc.

- But this causes a bottle-neck - upon a miss the recovery cost becomes increasingly high, because the memory cannot keep up with the required fetch rate.

- This leads to attempts to improve the performance of the memory.

A recap on the memory wall. Part II: The memory viewpoint.



- Increasing the levels of memory in the hierarchy, by placing levels of caches between the main memory and the CPU (or on the CPU).
- This reduces the memory wall, but on a cache miss the penalty is more severe. (Also this does not reduce the memory sub-system latency for an initial access, only upon subsequent access.)
- In both cases:

- The hardware complexity and cost is increased.
- The rewards obtained are balanced against known disadvantages.

The memory wall and cellular architectures: a solution?

- Why not place the processor in the memory, e.g. PIM architectures? Does this *remove* the memory wall?

- In principle due to the proximity of the execution units to the memory cells, the latency and bandwidth should be reduced.

- But due to the mixture of logic units on the silicon die, the gate density is reduced.

- To maintain gate density, more simple execution cores are used, such as RISC pipelines which may also omit branch prediction, for example.

- Thus the memory density and execution unit throughput are reduced. How may this be countered?

- With the addition of a network interface to interconnect between the PIM chips. Thus each chip becomes a *cell*.

- Thus reduced individual performance may be countered by interconnecting many of these cells together to build up a *cellular architecture*, e.g. Cyclops developed at IBM, Gilgamesh at NASA and Shamrock at Notre Dam.

Programming models on Cellular Architectures.

Cellular architectures have particular features that mean that their programming model is different to superscalar processors:

- They have large (100s) of execution (or in cellular architectures *thread units*) which are simple.
- Memory access is irregular: Some memory is very close, thus fast, the rest is off-chip, so much slower.

Research into appropriate programming models is on-going, but includes suggestions such as:

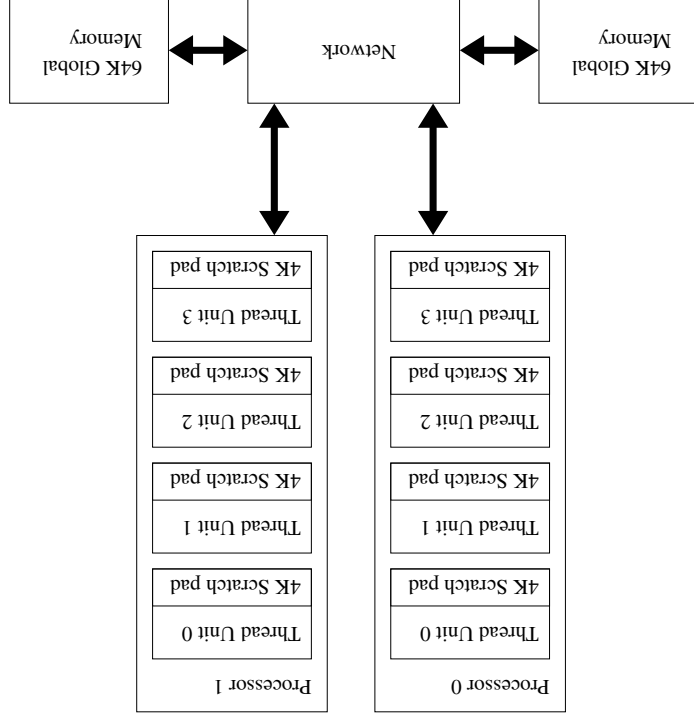
- Evaluation of multiple threads in each cell due to the large number of execution units. For example, one programming model uses thread percolation as a technique to perform dynamic load-balancing.

- Additionally, in cellular architectures, multiple threads perform memory accesses independently. As a result of this, the memory subsystem requires some form of access model that allows these memory references to be effectively served. For example, the use of the location-consistency model has been suggested as a memory access model.

A brief overview of Cyclops and DIMES/P.

At the University of Delaware the first hardware simulation of a cellular architecture has been built under Hiro Sakane's group:

- This is called DIMES/P.
- It is a simplified implementation of the CyclopsE design, one of the family of Cyclops architectures developed at the IBM T.J. Watson Research Center.



An introduction to the Mandelbrot set.

The Mandelbrot set is a *fractal* named after Professor B.B. Mandelbrot, who discovered the set in the 1960s. It is intimately related to the Julia set, also a *fractal*, discovered in the 1910s.

Both the Mandelbrot and Julia sets may be created by iteration of a very simple equation:

$$z_{n+1} = z_n^2 + c \quad (1)$$

In this equation, z_n is a complex number, where $z_0 = 0$. c is also a complex number, which is initialised to a value constant throughout the iterations. The iteration of equation terminates when:

1. Either n reaches the so-called "maximum iteration" value, m , a fixed constant, greater than zero.

2. Or $|z_n|$ exceeds the so-called "bailout" value, a fixed constant, usually set to the real value 4, for efficiency reasons.

The classic algorithm used to generate the Mandelbrot set:

1. Set the value of m , the maximum iterations, greater than zero.
2. Select a point from the complex plane, and set c to that value.
3. Initialise $n = 0$, $z_0 = 0$.
4. Execute equation 1.
5. Increment n .
6. If $|z^n| \geq 2$ then that c is not in the set of points which comprise the Mandelbrot set. Go to 2.
7. If $n > m$ then that c is in the Mandelbrot set, i.e. $c \in M$. Go to 2.
8. Go to 4.

Images of the Mandelbrot Set.

Figure 1:
The classic Mandelbrot set image generated by
"Fractint". Points coloured black are in M .

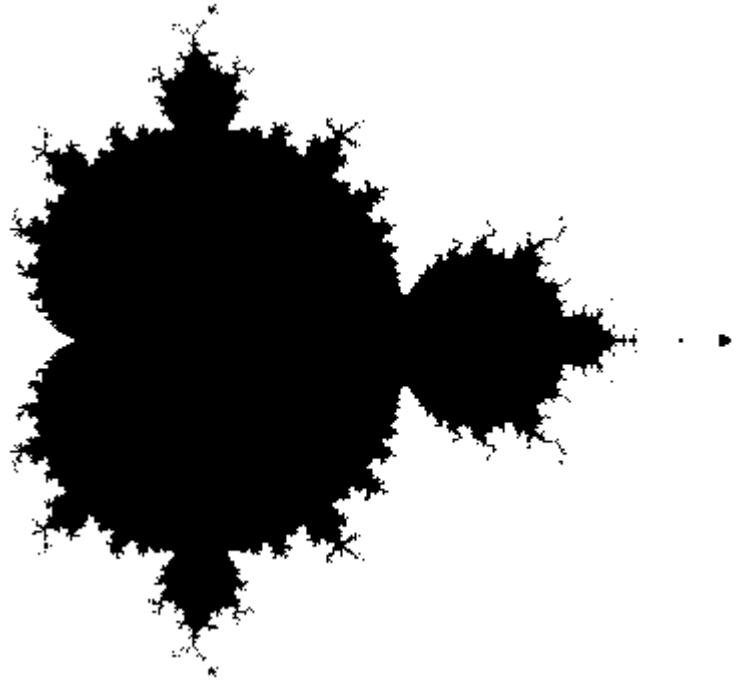
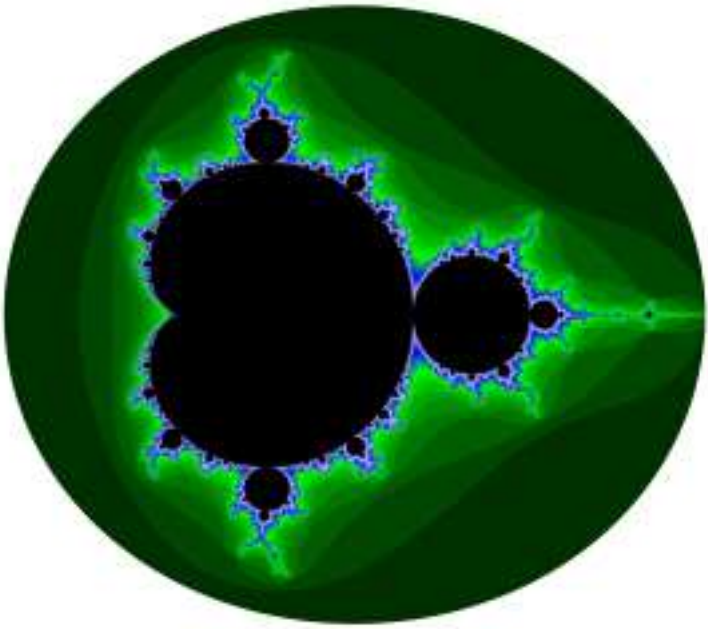


Figure 2:
A false-colour image of the Mandelbrot set gen-
erated by "Alpha One".



Threading applied to the Mandelbrot set.

An overview of threading the Mandelbrot-set generation algorithm:

- An important property of algorithm to generate the Mandelbrot set is that the classification of each c in the complex plane is independent of the classification of any other c . Thus the Mandelbrot set may be implemented as a massively parallel application, thus potentially suited to cellular architectures. Indeed the Mandelbrot has been used in as a benchmark for different architectures, such as fine-grain threaded-architectures and NUMA architectures.

The complex plane is divided into a series of horizontal strips. These strips may be calculated or rendered independently of each other, using separate render threads, as the classification of the points c within each strip is independent of such classification on other render threads. Therefore each render thread implements a slightly modified version of the classic algorithm, which is given in the threaded algorithm, given next.

The Render-Thread Algorithm.

1. The algorithm:

- (a) Set the value of m , the maximum iterations, greater than zero. Set the estimated completion-time, t , to ∞ .
- (b) Set $c = x$, where x is the top-left of the strip to be rendered.
- (c) Initialise $n = 0$, $z_0 = 0$.
 - i. Execute equation 1.
 - ii. Increment n .
 - iii. If $|z_n| \geq 2$ then that c is not in the set of points which comprise the Mandelbrot set. Go to 1d.
 - iv. If $n > m$ then that c is in the Mandelbrot set, i.e. $c \in M$. Go to 1d.
 - v. Go to 1(c)!
- (d) Increment the real part of c . If the real part of c is less than the width of the strip to be rendered, go to 1c.
- (e) Calculate the average of t and the time it took to render that line.
- (f) Set the real part of c to the left-hand of the strip. Increment the complex part of c . If the complex part of c is less than the height of the strip, go to 1c.
- (g) Signal work completed, set $t = 0$ (thus this thread is guaranteed not to be selected by the work-stealing algorithm).
- (h) Suspend.

A load-balancing algorithm was added to move uncompleted work to threads that have completed their assigned work. This is because each strip will take a different amount of time to render.

The Work-Stealing Algorithm.

1. Monitor render threads for a work-completed signal. That thread that completes we shall denote as T_c .

2. Find that render thread with the longest estimated completion-time, t , note that each render thread updates this time upon completion of a line. Call this thread T_l .

3. Stop T_l when it completes the current line it is rendering.

4. Split the remaining work to be done in the strip equally between the two render threads T_c and T_l .

5. Restart the render threads T_c and T_l .

6. Go to 1.

This is a dynamic-programming solution to the load-balancing problem of work distribution between the render threads. Due to the selection of the slowest render thread, this algorithm may be seen to be optimal. The author believes that this is an original application of work-stealing to Mandelbrot-set generation.

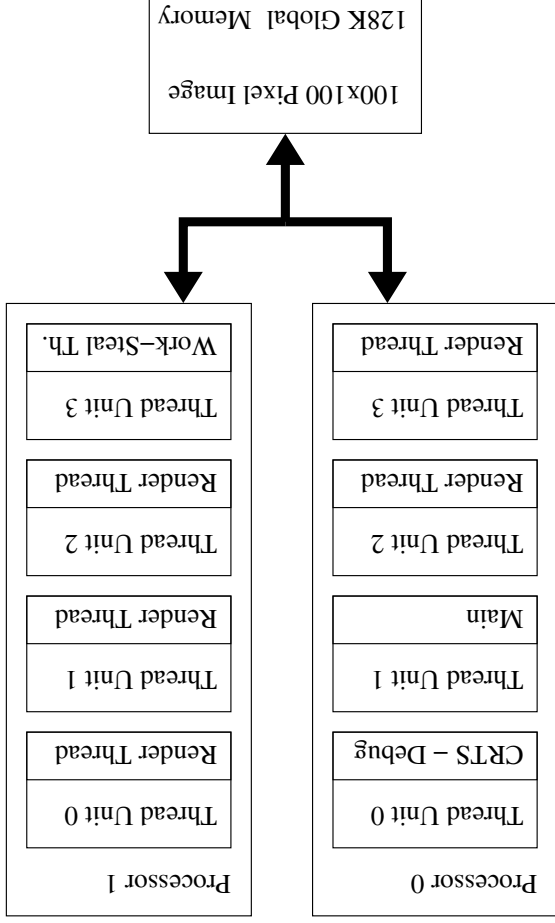
A Discussion of the Work-Stealing Algorithm.

- The bandwidth of the single thread that implements that algorithm is the limiting factor in its ability to scale.
- It is possible to scale this work-stealing algorithm, if one observes that the work-stealing algorithm operates upon a slice of the complex plane. This clue demonstrates that the work-stealing algorithm is recursive.

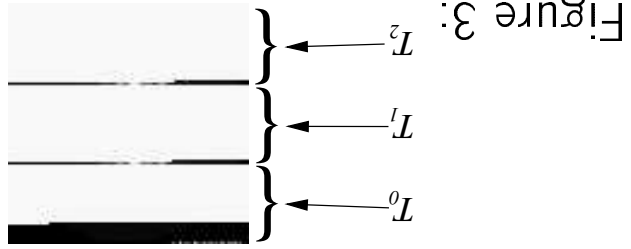
- Conversely this algorithm is able to tolerate failures in render threads. If a render thread stops responding, eventually it will be the slowest, unfinished render thread, and its work will be stolen. If robustness is not required, then the image generated may be viewed as an array values. Each of these values is the classification of c . Thus if one has $d_{0 \dots q}$ threads, each d_n thread initially classifies a point in the array offset by n , and once completed, moves along the array using a stride of q .

- This allows the use of a number of threads that is bounded by the number of points within the image. As this may be for an image of resolution 100×100 , thus 10,000 points, this maps well on to cellular architectures.

The static layout of the render and work-stealing threads within the DIMES/P system is shown below:



Execution Details of the Mandelbrot-set application.



The image generated shortly after program start-up.

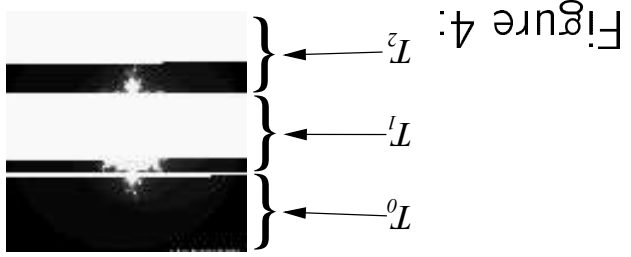
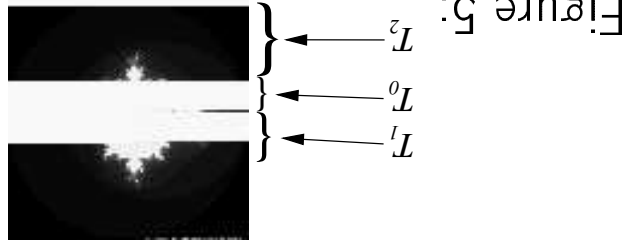
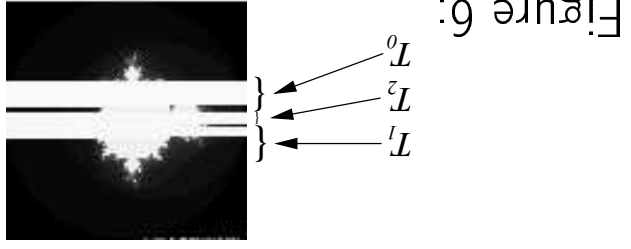


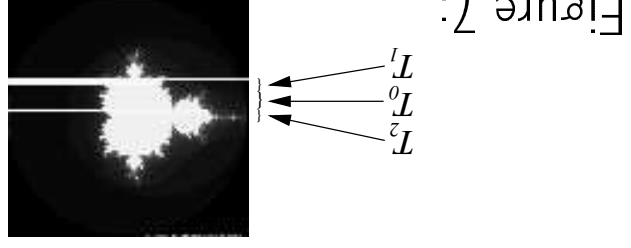
Image generation has progressed, shortly before a work-stealing event.



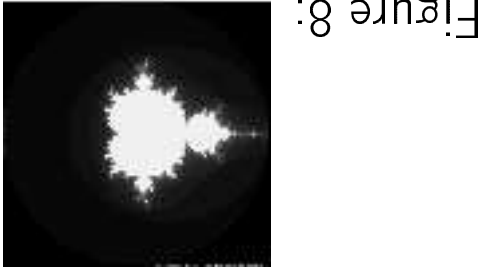
Just after the first work-stealing operation.



The second work-stealing operation.



The third work-stealing operation.



The completed Mandelbrot set.

Conclusion & Future Work.

- The Mandelbrot set is an ideal program to demonstrate and test massively parallel architectures, such as cellular architectures.
- The current run-time system, although simple, is sufficiently powerful for sophisticated applications.
- Future development of the Cyclops architecture towards Cyclops-64, with the development of hardware simulators with more hardware resources will allow the development and testing of more sophisticated programs that are based on more commonly known benchmarks. This will allow the testing of more the sophisticated programming models that have been suggested, such as thread percolation.