The Challenges of Efficient Code-Generation for Massively Parallel Architectures.

Jason M M^{C}Guiness[1], Colin Egan[1], Bruce Christianson[1] and Guang Gao[2].

[1] Department of Compiler Technology and Computer Architecture, University of Hertfordshire, Hatfield, Hertfordshire, U.K. AL10 9AB.
`c.egan@herts.ac.uk`
[2] CAPSL, University of Delaware, Delaware, U.S.A.
`g.gao@capsl.udel.edu`

Abstract. The ever-widening gap between processor and memory speeds, has resulted in the *memory wall*[24]. Overcoming this memory wall may be achieved by increasing the bandwidth and reducing the latency of the processor to memory connection, for example by implementing processors-in-memory (PIM), or Cellular architectures, such as the IBM Cyclops. Such massively parallel architectures have sophisticated memory models. The authors contend that there is an open question regarding the potential, ideal approach to parallelism from the programmer's perspective. For example, expressed at language-level such as UPC or HPF, or using trace-scheduling within the compiler, or at a library-level, for example OpenMP or POSIX-threads, or within the architecture, such as data-flow. In this paper we used DIMES (the Delaware Iterative Multiprocessor Emulation System), developed by CAPSL at the University of Delaware, as a hardware evaluation tool for cellular architectures. As the programing example, we have chosen to use a threaded Mandelbrot-set generator with a work-stealing algorithm to evaluate the DIMES *cthread* programming model.

1 Introduction.

The memory-wall [24] is a limiting factor in CPU performance, which may be countered by introducing extra levels in the memory hierarchy . However, these extra levels increase the penalty associated with a miss in the memory-subsystem and increase in design complexity and power consumption of the overall system. Integrating the processing logic and memory [14,3,22], termed PIM, is an approach to overcome these issues. PIM architectures may improve both data-processing and data-access times, but the combined processor speed and the amount of memory may be reduced [3]. This may be overcome by connecting multiple, independent PIM cells, giving a *cellular architecture*. In this organisation, every thread unit is an independent single-issue, in-order processor, thus able to potentially access memory independently. This gives rise to a number of code-generation problems, centred around the fact that to provide computational power, these systems are massively parallel. Writing correct and efficient multi-threaded programs is known to be hard.

Thus, research also proceeded towards thread-generating compilers, for example, HPF and UPC [12], IBM XL Fortran and Visual Age C/C++, largely based upon OpenMP, all of which have their compromises. Higher-level approaches, such as the *EARTH compiler* [23], able to automatically create threads using the split-phase constraint have also been studied. The different memory hierarchies of cellular architectures, where memory-banks may have different access timings and consistency models, add to the multi-threaded code-generation problem. One possible solution, known as *location consistency* [9], may be to map the memory to different address ranges in the memory-map of the virtual machine. The EARTH compiler and UPC both provide language, hence compiler support, for such features using the split-phase constraint, or the use of the strict and relaxed keywords, respectively.

Because general-purpose languages have been slow to adopt a sophisticated abstraction of the machine model, library-based approaches have developed, for example, the various implementations of OpenMP. But, the authors contend that library-based solutions to threading are too dependant upon the programmer to use effectively. For example, the explicit use of locks in programs is prone to error, with deadlocks and race-conditions that are hard to track down easily, introduced, even on systems with only a few processors. The development of suitable tools to debug multi-threaded applications has also been slow. Debuggers are in development, for example for Cyclops [11], but there have been too few, with limited functionality.

As identifying parallelism both correctly and efficiently is very hard for the programmer to do, the authors contend that they should not do it. When such massively-parallel architectures are developed, this process should include time to develop libraries that plug into the target compilers to allow them to generate efficient code for that architecture. The compiler, equipped via these libraries with a detailed machine-model, could be able to use the programmer-identified parallelize-able variables and functions, to generate more efficient code. The authors identified little work investigating the software aspect of the code-generation problem for massively-parallel architectures. Unfortunately, if this case would continue, this shortcoming could adversely affect the popularity of such systems and maintain the perception that massively parallel architectures are too specialised and thus too expensive to be of more general use. Given the popularity of introducing multi-core processors, this position is set to become even more untenable.

2 Previous Work.

2.1 The VLIW Origins.

The VLIW architectures may have heavily influenced this field: a VLIW execution unit could be considered to have a limited number of "live" threads. The number of which has been limited by the ability of the compiler to identify available ILP (Instruction-Level Parallelism). Identifying ILP was vital, and research work [18] indicated that, in the SPEC95 benchmark suite, there was a potential for a large amount of ILP. But this was for machine models with an infinite number of available registers, or bypass buses, or an oracle branch predictor. The reality of the lack of an effective compile-time branch predictor or limited resources, contributed to a decrease in the instruction density in the VLIW code. Thus the effectiveness of the VLIW architecture as a technique to extract ILP and thus performance from re-compilation of source code was constrained.

2.2 Beyond VLIW: Super-scalar Architectures.

The development of dynamic branch predictors meant that speculative execution of code was much less likely to be wasted work. But now performance was hindered by slow memory speeds. Based upon evidence of data and control locality [17] caches were developed. An instruction cache could pre-fetch instructions to minimise pipeline stalls before the IF stage, using the pre-computed, predicted, branch-target address. Also register writes were directed to a data cache. When speculation was combined with branch prediction and out-of-order execution, even more ILP could be extracted. Thus the retirement of instructions was linked to an architectural state that would have to be rolled back, if a mis-prediction occurs and the instruction fetch re-started from the alternative branch.

This combination of data and instruction caches effectively decouples the processor from the speed of the main-memory. The complexity of the processor state is strongly related to the pipeline depth. If a mis-prediction occurs and the state is rolled back, the instruction fetch and pipeline are restarted, it takes increasingly long in a deeper pipelines to begin retiring instructions again. The accuracy of the branch predictor has become paramount, to avoid expensive restarts. But if the processor speed is to increase, then more stages are usually needed, increasing the cost of branch mis-predictions, giving rise to the *memory wall* [24].

2.3 Massively Parallel Architectures.

The problem of the memory wall may be viewed as an effect of the relative performance difference of main memory to processor speed. The instruction caches fetch from a relatively few memory banks, increasing the instruction retirement rate by increasing the number if IF stages, thus memory banks, indicated an architectural-level form of threading. Alternatively, multi-core processors take a different approach: if the OS supports pre-emptive multi-tasking, then large, OS-level threads which should have the inter-thread, data-dependencies explicitly specified using kernel-level (thus architectural) synchronisation primitives.

To counter the memory latencies inherent in the super-scalar designs, each instruction fetch stage and the data bus of a pipeline would be fed directly from an independent bank of memory on the same chip die. This integration had the advantage that the latency of memory accesses have been dramatically reduced. But to allow such integration, the pipelines are usually much more simple than a super-scalar pipeline, they may have no branch prediction, so no speculation, thus allowing more memory and more execution units per die. For example, in the picoChip [7] design uses a mixture of VLIW and DSP cores having direct, 1-clock cycle access to approximately 64K of RAM per core, the IBM BlueGene/C design [1], uses 64-bit cores with approximately 64K of software-controlled data cache, and another 4Gb of RAM on chip. To further increase the bandwidth, the picoChip has 4 ports implemented on it for accessing other picoChips in a grid arrangement, and a memory port for accessing off-chip memory. The IBM BlueGene/C design has 6 inter-chip connection ports, allowing a cubic array of chips. These aggregations are termed as a cellular architecture

2.4 The Programming Models: from Compiler to Libraries.

With such compute bandwidth, and parallelism, a number of problems for the programmer have been raised, primarily these are focused on the problems of memory reads and writes. Super-scalar chips have had mechanisms to hide these problems from the programmer, but the cellular chips such as picoChip and IBM BlueGene/C do not. Thus the programmer needs to know how memory reads and writes interact with:

- the software-controlled data-cache attached to that pipeline,
- the software-controlled data-cache of other on-chip pipelines,
- any global on-chip memory,
- the software controlled data-caches of other off-chip pipelines,
- the global on-chip memory that is on any other chips,
- any global memory that is not on any chip
- and finally, given the massive parallelism available, how to make efficient use of it.

These issues give rise to various programming models, but initially the last point will be discussed. Given the evidence of ILP studies, the efficient use of the massive parallelism for general purpose programs such as SPEC2000 is highly unlikely to be able to be parallelized to the extent of using a fraction of the resources of the IBM BlueGene design, similarly with the smaller picoChip. The answer would be that these architectures eschew the aspiration of being practical for general-purpose use. Instead they target specific, embarrassingly-parallel problem domains.

For a programmer, the memory access models are important to understand, or to have a library or compiler that hides the details from the applications programmer. In the remainder of the paper the authors will focus on the IBM BlueGene/C architecture, and a prototype implementation of it called Cyclops, that was implemented at CAPSL at the University of Delaware in collaboration with the University of Hertfordshire. In the following sections the memory access models will be discussed, leading on to a presentation of the authors' experience in developing a program for such an architecture. The experience gained from this will allow the authors to discuss the major problems that were faced, how, if at all, they were overcome, and the outstanding problem domains that, in the authors' experience, would hinder the acceptance of multi-core chips and, moreover such massively parallel designs as IBM BlueGene/C.

2.5 IBM BlueGene/C, Cyclops and DIMES/P.

The IBM BlueGene/C architecture is described in detail in [1]. Briefly, this architecture consists of a large number of thread units, an equal number of memory banks and a large crossbar on one silicone die. The execution thread-units are linked to each other and the memory banks via the crossbar, which also has at least 8 off-chip interconnects. These interconnects may be used to connect more of these chips together in a large 3-d mesh. Of the order of 160 thread units are on a single die, with the order of 2-4 Gbytes of DRAM, on-chip. This means that per chip there is a large amount of available parallelism, and considering that the 3-d mesh may contain of the order of 100,000 of such chips. A further factor in this design is that there is no data cache: instead there is a specialised portion of each DRAM bank that is directly accessible via a related thread unit. Such a portion of the DRAM is termed the *scratch-pad memory*, and is effectively a software controlled data cache. This scratch-pad memory is accessible from that related thread unit without having to access the crossbar. The other memory, not associated with any particular thread-unit is termed as *on-chip memory*. This gives rise to different memory access models, as already hinted previously. These memory access models are related to the work on location-consistency, described in [9,25]. In brief, this is the concept that if a set of memory locations are accessed from two different thread units, the thread units will experience different memory access models of those memory locations, upon simultaneous access. For example: simultaneous accesses, by different thread units, to location 1 might provide program consistency as the memory access model, whereas for location 2, with simultaneous accesses, by different thread units, this might provide sequential consistency. With regards to IBM BlueGene/C the scratch pad memory only guarantees program consistency with regards to memory accesses. But for any memory accessed via the cross-bar, it guarantees sequential consistency.

At CAPSL, in Delaware, much work was done in collaboration with IBM with regards to an implementation of the BlueGene/C architecture called Cyclops. This was initially with regards to CyclopsE [3] then progressed to Cyclops64, [5]. The CyclopsE architecture was prototyped in hardware, called DIMES/P, [21,20]. DIMES/P was used as the platform for executing the programming example, described later in this paper.

With regards to any later discussions, it is very important to remember that each of these architectures, IBM BlueGene/C, Cyclops64, CyclopsE and DIMES/P display the same features: multiple thread units and multiple memory consistency models. This is simply because they are all implementations of these same underlying concepts.

2.6 Programming Models on Cellular Architectures.

The hardware differences between cellular and super-scalar architectures indicate that different programming models, to those used for super-scalar architectures, are required to make effective use of the cellular architectures [9,10,11]. In the first two of those three papers, their authors propose the use of a combination of execution models and memory models, as already noted in this paper.

The primary concerns when programming DIMES/P, and thus any Cyclops-based architecture, were:

- How to manage the potentially large numbers of threads.
- How to easily express any parallelism within the input source-code.
- How to make correct, and most effective use, of the memory consistency models.

Some research has already been done regarding programming models for the threading, such as using thread percolation as a technique to perform dynamic load-balancing [2,13]. Another piece of research [4] investigated using multi-level scheduling-schemes: a work-stealing algorithm at the higher-level and a multi-threading technique at the lower-level to hide communication latencies. A further piece of research [8] investigated the use of filaments as lightweight threads to efficiently implement thread control.

3 Programming for Cyclops.

For Cyclops, a reasonable technique for implementing memory consistency models, thread management, and finally making use of any parallelism was investigated.

This started with investigating how to easily implement the memory consistency models. This was relatively simple: earlier, unpublished, work on the GCC-based compiler had implemented a simple algorithm: all static variables were stored in on-chip memory, and the function call stack, including all automatic variables was placed in the scratch-pad memory.

As there was no language-level support for thread management, a library had to be implemented to support the thread management instructions in the Cyclops ISA. An early version of TNT [6,11], called *cthreads* was used as the basis for creating a higher-level C++ abstraction. This was because the cthread implementation, that closely followed a POSIX-Threads API, was considered far too primitive by the authors to be effectively used for programming Cyclops. This C++ API also included critical-section, mutex and event objects to allow for easier management of the lower-level objects.

An abstraction of the extraction of parallelism from the range of possible example programs was not implemented for this paper, as this was considered to be potentially too closely coupled to the actual program in question. Ultimately this decision, in the authors' opinion, was flawed, and is where the crucial abstractions take place that allow a programmer to implement an algorithm with far less regard for the underlying architectural features. Thus the programmer would obtain much greater benefits from this more powerful abstraction.

To test these ideas, and the Cyclops architecture, a simple program was chosen. It had the properties that it was a small problem and embarrassingly parallel, ideally suited to CyclopsE. Thus an implementation of a program to generate Mandelbrot sets was created. In the following sections, a brief introduction to Mandelbrot sets will be given. Then how this program may be multi-threaded will be presented, with particular attention to the implementation used for DIMES/P. This will be followed by a description of how that threaded algorithm was implemented on the DIMES/P platform, including a brief recap of the DIMES/P architecture. Finally, the reader will be stepped through an example of the running application and the operation of the work-stealing algorithm, which was chosen to extract parallelism from the Mandelbrot-set generation algorithm.

3.1 A Brief Introduction to the Mandelbrot Set.

The Mandelbrot set [16] is so named after Professor B.B. Mandelbrot, who discovered the set in the 1960s. The Mandelbrot set may be created by iteration of a very simple equation:

$$z_{n+1} = z_n^2 + c \tag{1}$$

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, held constant throughout the iterations. The iteration of equation 1 terminates when:

1. Either n reaches the so-called "maximum iteration" value, m, a fixed constant, greater than zero, thus the point c identified as a member of M, the set of points that comprise the Mandelbrot set. (Thus $\mid z_n \mid < \infty$.)
2. Or $\mid z_n \mid \rightarrow \infty$.

3.2 Threading and the Mandelbrot Set.

For the Mandelbrot Set, an important property is that the classification of each c in the complex plane is independent of the classification of any other c. Thus generating the Mandelbrot Set is embarrassingly parallel, i.e. highly suited to cellular architectures. Alternative implementations for different architectures, such as fine-grain threaded-architectures [8] and NUMA architectures [4] also exist. For CyclopsE and DIMES/P in particular, which lack floating-point support, another important feature of this algorithm is that it does not require floating point support, it may be implemented in fixed-point arithmetic using up to 32 bits for the digits.

One might choose to implement the algorithm per thread unit within the cellular-architecture machine. This approach would work well for massive clusters of cellular computing nodes. (Recall that for an image of 100×100 points, c, 10,000 thread units would be required with this technique.) Moreover, the classification of any randomly selected c may take between 1 and m iterations of the algorithm. Also, in general, it is not possible to know in advance how long such a c will take to classify. Therefore the computation time would take approximately m times the time per iteration loop.

Due to the properties of DIMES/P, this technique was not possible, as there are only 8 thread units between two processors. Therefore a different approach was taken. In this implementation, the complex plane was divided into a series of horizontal strips. Those strips may be calculated independently of each other, using separate threads, as the classification of the points c within each strip is independent of classification within other threads, implemented as algorithm 1. Only the coordinates for the bounding rectangle

Algorithm 1 The render-thread algorithm.

```
1. Set the value of $m$, the maximum iterations, greater than zero. Set the estimated
   completion-time, $t$, to $\infty$.
2. Set $c = x$, where $x$ is the top-left of the strip to be rendered.
3. Initialise $n = 0$, $z_0 = 0$.
   (a) Execute equation 1.
   (b) Increment $n$.
   (c) If $| z_n | \geq 2$ then that $c$ is not in the set of points which comprise the Mandelbrot
       set. Go to 4.
   (d) If $n > m$ then that $c$ is in the Mandelbrot set, i.e. $c \subset M$. Go to 4.
   (e) Go to 3a.
4. 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 3.
5. Calculate the average of $t$ and the time it took to render that line.
6. 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 3.
7. Signal work completed, set $t = 0$ (thus this thread is guaranteed not to be selected
   by the work-stealing algorithm 2).
8. Suspend.
```

were inter-related between the threads. However, each strip will, in general, take a different amount of time to complete, thus the threads would have completed their assigned portion of work at different times. This lead to the addition of a load-balancing algorithm to move uncompleted work to threads that have completed their assigned work. Thus a work-stealing algorithm 2 was added to perform the load-balancing between the threads.

The updates to the start, x, and finish points of the strips, for the threads T_c and T_l were performed atomically - the threads were suspended whilst these updates were done, either because T_c was already stopped or because T_l was stopped by using a mutex. (A mutex was required as the data to be updated was a two complex numbers: x and the finish point, these should be updated as a pair, atomically. In this implementation a complex number consisted of two words - one for the real part, one for the imaginary part.)

Algorithm 2 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.
```

3.3 A Discussion of the Work-Stealing Algorithm 2.

This section gives a short discussion of some features of algorithm 2 that affect the ultimate performance of the program, especially for such architectures as Cyclops:

- The bandwidth of the single thread that implemented algorithm 2 was the limiting factor regarding scaling to more worker threads. But this algorithm was able to tolerate failures in worker threads. If a worker thread stopped responding, eventually it would have become the slowest, unfinished worker thread, and any work would have been continually stolen.
- If robustness is not required, then the image generated may be viewed as an array values. Each of these values would be the classification of c. Thus if one has $p_{0 \ldots q}$ threads, each p_n thread initially classifies a point in the array offset by n, and once completed, would move along the array using a stride of q. This would allow 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, which also maps well on to the cellular architectures described in [3]. For more thread units, the image resolution would need to be increased. Unfortunately this algorithm would not have a natural ability to tolerate failures in thread units, unlike the work-stealing algorithm that was used.

3.4 DIMES/P Implementation of the Mandelbrot-set application.

The pertinent features of this architecture, from [21,20], are that the memory model for the two types of memory, the scratch pad and the global memories are different:

- The global memory obeys the sequential consistency model for all thread units.
- The scratch pad memory obeys the program consistency model for all thread units, apart from the thread unit to which it is attached.

These different consistency models affected the way that the data for the Mandelbrot-set application was arranged in memory, but this will be discussed in more detail later.

The static layout of the render and work-stealing threads within the DIMES/P system is shown in figure 1.

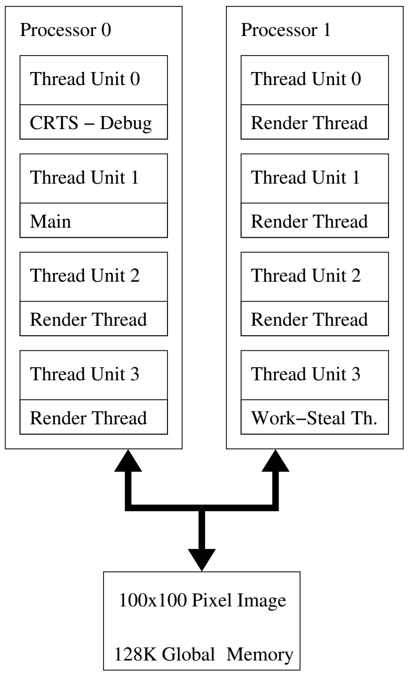

Figure 1. Layout of the render and work-stealing threads within the DIMES/P system.

The software threads that occupy the thread units are:

– The *CRTS - Debug* thread was required for the debugger [11], if it were to be run. As threads were statically allocated at program start-up, this had remain free for the debugger and Cellular Run-Time System (CRTS) support.
– *Main* is the main loop of the Mandelbrot-set application.
– The *Render Threads* are the worker threads that executed algorithm 1.
– *Work-Steal Th* was the thread that executed algorithm 2. Only one work-stealing thread was implemented, due to the limited number of thread units per processor. In principle, a worker thread could also run on this thread unit, but the CRTS did not support virtual threads, moreover the work-stealing thread actually had to spin in a busy wait monitoring for completion of a worker thread.

Further details regarding the implementation may be found in [15].

3.5 The Memory Layout.

For the programmer, the two 64k global, or on-chip, memory units comprised a single, contiguous 128k block of on-chip memory. This memory was used for code and any global data. The programmer had no way to differentiate between the two blocks. Moreover, the CyclopsE design was such that access times to them were the same, no matter which thread unit from which processor accessed the two blocks. But the programmer could ensure that data would be placed in global memory by the compiler by ensuring that it was static. The stack frame of the currently executing function was placed into the scratch-pad memory by the compiler. So function call depth was limited, as there was only 4K stack space, per thread. The Mandelbrot-set algorithm described did not need this much space for each thread unit, thus all thread local-data was placed into the corresponding scratch-pad memory for performance.

The 40,000 bytes of image data (100\times 100 words, 1 word = 4 bytes) was placed in global memory for implementation reasons. DIMES/P had no console, thus the only way that communication with DIMES/P could occur was via the global memory, from a specially written program that ran on the host computer.

4 Discussion.

The limitations of DIMES/P prevented further study of the properties of this program: scalability and timings were not done because of the limited number of thread units (8) and memory capacity. Despite this, the development of the program was instructive: an initial contention of this paper was that the memory models and massive parallelism inherent in cellular architectures would make programming for them hard. This was experienced to different measures, as will be discussed: initially the discussion will focus on the memory model support, then the thread library and parallelism support.

The memory model support, using the C/C++ keyword *static* by the compiler, made natural use of language-level syntax to map data into scratch-pad and on-chip memory made using these different memory models. Cyclops only has word-sized, atomic memory-operations, which were not used for this problem, because of the multiple, read-modify-write operations that had to be maintained as an atomic unit. This hampered the performance of the program, because barriers such as mutexes and critical sections had to be used instead. If the manual locking that had to be applied had been implemented within the support provided by the compiler, then it may have been possible for the compiler to perform optimization on the locking of access to the data, with apparently no impact upon the developer. It is the authors' contention that there should be support for such locking if programs more sophisticated than the one described in this paper are to be successfully written for these architectures.

With regards to the thread library: in the opinion of the author's, the complexity of POSIX-Threads has been a hindrance to successful multi-thread program creation. The creation of a C++ wrapper to hide thread creation and destruction, and combine with that thread, any local storage in an efficient manner, was only partially successful: The concept that a thread is an object has not been not universally accepted, because this means that the data to be manipulated becomes intimately intermingled with the thread-management code. Even for this simple example program, this was evident in the work-stealing algorithm, and the way it interacted with the start and end-points of the worker threads. For more complex, larger programs, such complexity would be likely to make writing them correctly, and modifying them later, very hard. Subsequent updates to the cthread model, which became TNT, described in [6], are still largely POSIX-Thread based, and which is a low-level API. The concept that data and execution should be kept separate is commonly and naturally embodied in programming via the syntax of "main". It has been contended that this pattern should be duplicated for thread libraries: that there should exist a pool of threads, to which work is passed. This work would be asynchronously executed, on a thread within the pool. With the results returned from that pool via a wait-able object. This concept is similar to the data-flow designs that preceded VLIW, indeed it has been argued that this concept is a software emulation of data-flow.

When considering the harder problem of creating an effective parallel algorithm and representing that in code, the example program described above was limited in its achievements: the work-stealing algorithm was intimately related to the program design, but this was related to the design of the thread-library, insufficient abstraction was available to the programmer. Alternative approaches have been examined, such as in [19], using OpenMP, which still has to be used as a library. Alternatively, if one has a thread pool into which work is submitted, then the details of how the pool works become separated from the work itself. The fact that the pool balances work between threads using a master-slave, or work-stealing algorithm is independent of the work: a natural division of concepts. Then the programmer would be free to add work to the pool as desired. The parallelism of the algorithm would be more naturally expressed in terms of operations on data. If one is to consider this further: the actual executable code (in terms of the function pointer) and the data are passed to the pool together. It could be then possible for the pool to be designed to make use of data locality and code locality: Did a previous thread execute that code before? If so, prefer to run that work on that thread. If there is resource asymmetry, then one might create a pool to represent the particular feature of that resource. For example a Cyclops chip might be represented as a thread pool, contained within a greater thread pool that represents the machine, due to the fact that off-chip memory access makes use of a message-passing protocol, rather than the cross-bar network embedded within the chip, that allows much more rapid memory access.

It is still an open question regarding what may be the ideal approach to parallelism: language-level support such as UPC, HPF or other language extensions, or within the compiler using trace-scheduling, or should it be at a library-level using, for example OpenMP or POSIX-Threads, or should it be within the architecture, such as the data-flow design.

The research presented in this paper is supported by the Engineering and Physical Research Council (EPSRC) grant number: GR/S58492/01.

References

1. Almásil, G., Cascaval, C., Castaños, J.G., Denneau, M., Lieber, D., Moreira, J.E. and Warren, H.S., "Dissecting Cyclops: Detailed Analysis of a Multithreaded Architecture.", ACM SIGARCH Computer Architecture News, Vol. 31, March 2003.
2. Cai, H., "Dynamic Load-Balancing on the EARTH-SP System.", Master's Thesis, McGill University, Montréal, May 1997.
3. Cascaval, C., Castaños, J.G., Ceze, L., Denneau, M., Gupta, M., Lieber, D., Moreira, J.E., Strauss, K. and Warren, H.S., "Evaluation of a Multithreaded Architecture for Cellular Computing.", 8th International Symposium on High-Performance Computer Architecture (HPCA), February 2002.
4. Cavalherio, G.G.H., Doreille, M., Galilée, F., Gautier, T., Roch, J-L., "Scheduling Parallel Programs on Non-Uniform Memory Architectures.", HPCA Conference – Workshop on Parallel Computing for Irregular Applications WPCIA1, Orlando, USA, January 1999.
5. del Cuvillo, J.B., Zhu, W., Hu, Z. and Gao, G.R., "FAST: A Functionally Accurate Simulation Toolset for the Cyclops-64 Cellular Architecture.", Workshop on Modeling, Benchmarking and Simulation (MoBS), held in conjunction with the 32nd Annual International Symposium on Computer Architecture (ISCA'05), Madison, Wisconsin, June 4, 2005.
6. del Cuvillo, J.B., Zhu, W., Hu, Z. and Gao, G.R., "TiNy Threads: a Thread Virtual Machine for the Cyclops64 Cellular Architecture.", Fifth Workshop on Massively Parallel Processing (WMPP), held in conjunction with the 19th International Parallel and Distributed Processing System, Denver, Colorado, April 3 - 8, 2005.
7. Duller, A., Towner, D., Panesar, G., Gray, A. and Robbins, W., "picoArray technology: the tool's story.", Proceedings of the Design, Automation and Test in Europe Conference and Exhibition, IEEE, 2005.
8. Engler, D.R., Andrews, G.R. and Lowenthal, D.K., "Filaments: Efficient Support for Fine-Grain Parallelism.",TR 93-13a, Dept. of Computer Science, University of Arizona, Tucson, 1993.
9. Gao, G.R. and Sarkar, V., "Location Consistency - a New Memory Model and Cache Consistency Protocol.", IEEE Transactions on Computers, Vol. 49, No. 8, August 2000.
10. Gao, G.R., Theobald, K.B., Hu, Z., Wu, H, Lu, J., Sterling, T.L., Pingali, K., Stodghill, P., Stevens, R. and Hereld, M., "Next Generation System Software for Future High-End Computing Systems.", International Parallel and Distributed Processing Symposium: IPDPS 2002 Workshops April 15 - 19, 2002 Fort Lauderdale, Florida.
11. Gao, G.R., Theobald, K.B., Govindarajan, R., Leung, C., Hu, Z., Wu, H., Lu, J., del Cuvillo, J., Jacquet, A., Janot, V. and Sterling, T.L., "Programming Models and System Software for Future High-End Computing Systems: Work-in-Progress.", International Parallel and Distributed Processing Symposium (IPDPS'03) April 22 - 26, 2003 Nice, France.

12. El-Ghazawi, T.A., Carlson, W.W., Draper, J.M., "UPC Language Specifications V1.1.1", October 2003.
13. Kakulavarapu, P., Morrone, C.J., Theobald, K., Amaral J.N. and Gao, G.R., "A Comparative Performance Study of Fine-Grain Multi-threading on Distributed Memory Machines.", 19th IEEE International Performance, Computing and Communication Conference-IPCCC2000, Phoenix, Arizona, USA, Feb. 20-22, 2000.
14. Kogge, P.M., Sterling, T.L. and Gao, G., "Processing in memory: Chips to petaflops.", In Workshop on Mixing Logic and DRAM: Chips that Compute and Remember at ISCA '97. `http://iram.cs.berkeley.edu/isca97-workshop/`, Denver, CO, June 1997.
15. M^{C}Guiness, J.M., "A DIMES Demonstration Application: Mandelbrot-Set Generation Using a Work-Stealing Algorithm.", CAPSL Technical Note 11, Department of Electrical and Computer Engineering, University of Delaware, Newark, Delaware, June 2003, `ftp://ftp.capsl.udel.edu/pub/doc/notes/`.
16. Mandelbrot, B.B., "The Fractal Geometry of Nature.", W.H.Freeman & Co., Sept., 1982.
17. Patterson, D.A. and Hennessy, L.J., "Computer Architecture: A Quantitative Approach.", 2$^{\text{nd}}$ Edition, Morgan Kaufmann Inc., San Francisco, pp. 374, 1996.
18. Postiff, M.A., Greene, D.A., Tyson, G.S. and Mudge, T.N., "The limits of instruction level parallelism in SPEC95 applications.", The Third Workshop on the Interaction between Compilers and Computer Architectures (INTERACT), in conjunction with the Eighth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-VIII), October 1998.
19. Rodenas, D., Martorell, X., Ayguade, E., Labarta, J., Almasi, G., Cascaval, C., Castanos, J. and Moreira, J., "Optimizing NANOS OpenMP for the IBM Cyclops Multithreaded Architecture.", 19th IEEE International Parallel and Distributed Processing Symposium, Vol. 1, pp. 110, 2005.
20. Sakane, H., Yakay, L., Karna, V., "DIMES/P Hardware Technical Manual.", CAPSL Technical Note 12, Department of Electrical and Computer Engineering, University of Delaware, Newark, Delaware, June 2003, `ftp://ftp.capsl.udel.edu/pub/doc/notes/`.
21. Sakane, H., Yakay, L., Karna, V., Leung, C. and Gao, G.R., "DIMES: An Iterative Emulation Platform for Multiprocessor-System-on-Chip Designs.", IEEE International Conference on Field-Programmable Technology, December 15-17, 2003, Tokyo, Japan.
22. Sterling, T.L., Zima, H.P., "Gilgamesh: A Multithreaded Processor-In-Memory Architecture for Petaflops Computing.", Proc.SC2002, Baltimore, November 2002.
23. Tang, X., "Compiling for Multithreaded *(sic)* Architectures," PhD dissertation, University of Delaware, Autumn 1999.
24. Wulf, W. and McKee, S., "Hitting the memory wall: Implications of the obvious.", Computer Architecture News, 23(1), pp. 20-24, 1995.
25. Zhang, Y., Zhu. W., Chen, F., Hu, Z. and Gao, G.R, "Sequential Consistency Revisited: The Sufficient Conditions and Method to Reason Consistency Model of a Multiprocessor-on-a chip Architecture.", The IASTED International Conference on Parallel and Distributed Computing and Networks (PDCN2005), February 15 - 17, 2005, Innsbruck, Austria.