The University of Hertfordshire

The Challenges facing Libraries and Imperative Languages from Massively Parallel Architectures

Jason McGuiness
Computer Science
University of Hertfordshire
UK

Building Futures in Computer Science
empowering people through technology

Colin Egan
Computer Science
University of Hertfordshire
UK

Presentation Structure

- Parallel processing
  - Pipeline processors, MII architectures, Multiprocessors
- Processing In Memory (PIM)
  - Cellular Architectures: Cyclops/DIMES and picoChip
- Code-generation issues arising from massive parallelism
- Possible solutions to this issue:
  - Use the compiler or some libraries
- An example implementation of a library, and the issues
- Questions?
  - Ask as we go along, but we'll also leave time for questions at then end of this presentation

Parallel Processing

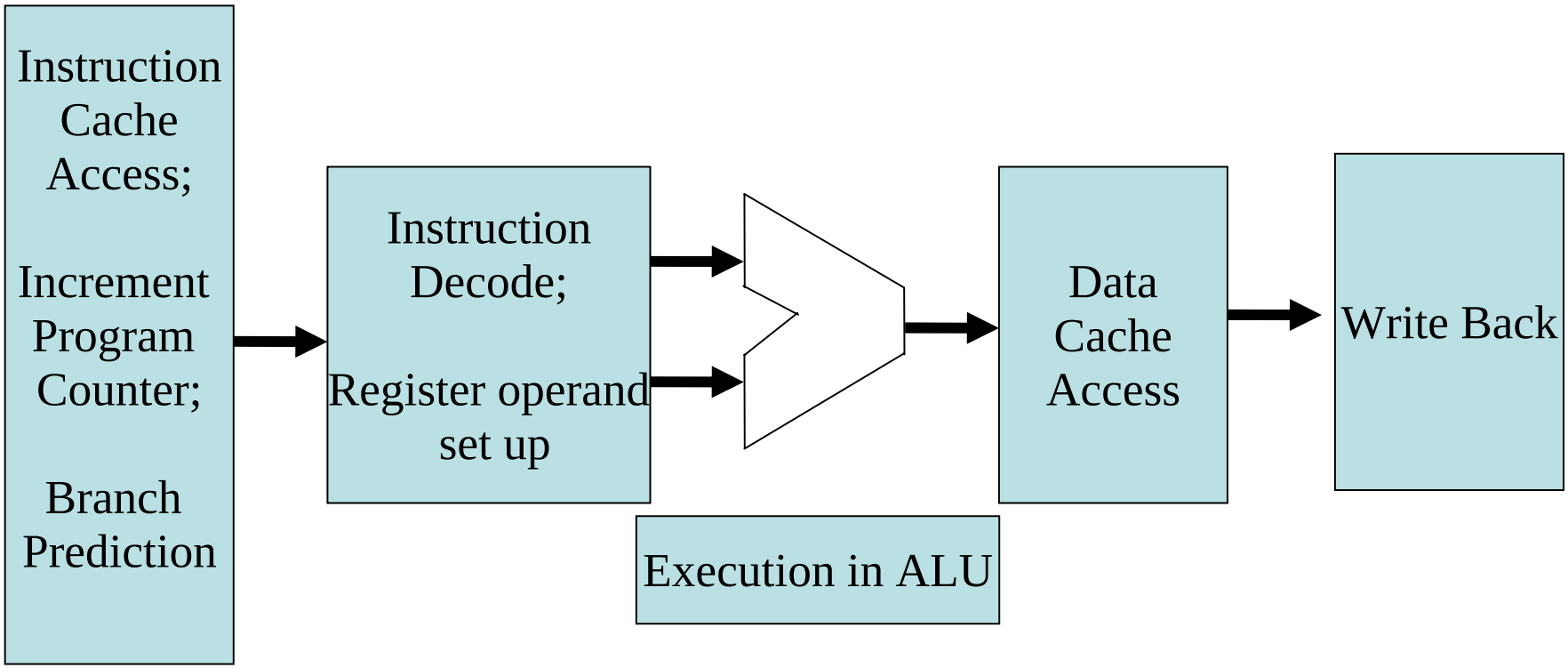
- How can parallel processing be achieved?
  - By exploiting:
    - Instruction Level Parallelism (ILP)
    - Thread Level Parallelism (TLP)
    - Multi-processing
    - Data Level Parallelism (DLP)
    - Simultaneous Multi-Processing (SMP)
    - Concurrent processing
    - etcetera

Pipelining

- Exploits ILP by overlapping instructions in different stages of execution:
  - ILP is the amount of operations in a computer program that can be performed on at the same time (simultaneously)
- Improves overall program execution time by increasing throughput:
  - It does not improve individual instruction execution time

Pipelining

- A simple 5-stage pipeline



Pipelining hazards

- Pipelining introduces hazards which can severely impact on processor performance:
  - Data (RAW, WAW and WAR)
  - Control (conditional branch instructions)
  - Structural (hardware contention)

- To overcome such hazards complex hardware (dynamic scheduling) or complex software (static scheduling) or a combination of both is required

Multiple Instruction Issue

- MII
  - A processor that is capable of fetching and issuing more than one instruction during each processor cycle
  - A program is executed in parallel, but the processor maintains the outward appearance of sequential execution
  - The program binary must therefore be regarded as a specification of what was done, not how it was done
  - Minimise program execution time by:
    - by reducing instruction latencies
    - by exploiting additional ILP

Multiple Instruction Issue

- Superscalar Processor:
  - An MII processor where the number of instructions issued in each clock cycle is determined dynamically by hardware at run-time
    - Instruction issue may be in-order or out-of-order (Tomasulo or equivalent).
- VLIW Processor (Very Long Instruction Word):
  - An MII processor where the number of operations (instructions) issued in each clock cycle is fixed and where the operations are selected statically by the compiler at compile time

Problems of MII

- To sustain multiple instruction fetch, MII architectures require a complex memory hierarchy:
  - Caches
    - l1, l2, stream buffers, non-blocking caches
  - Virtual Memory
    - TLB
  - Caches suffer from:
    - Compulsory misses
    - Capacity misses
    - Collisions

Memory Problems

- Compulsory misses:
  - The first time a processor address is requested it will not be in cache memory and must be fetched from a slower level of the memory hierarchy:
    - Hopefully main (physical) memory
    - If not from Virtual Memory
    - If not from secondary storage
  - This can result in long delay (latency) due to large access time(s)

Memory Problems

- Capacity misses:
  - There are more cache block requests than the size of the cache
- Collisions:
  - The processor makes a request to the same block but for different instructions/data
- For both:
  - Blocks therefore have to be replaced
  - But a block that has been replaced might be referenced again resulting in yet more replacements

Thread Level Parallelism

- A thread can be considered to be a '*light weight process*'
  - Where a thread consists of a short sequence of code, with its own:
    - registers, data, state and so on
    - but shares process space
- TLP is exploited by simultaneously executing different threads on different processors:
  - TLP is therefore exploited by multiprocessors

Multiprocessors

- Should be:
  - Easily scalable
  - Fault tolerant
  - Achieve higher performance than a uni-processor

- But …
  - How many processors can we connect?
  - How do parallel processors share data?
  - How are parallel processors co-ordinated?

Multiprocessors

- Shared Memory Processors (SMP)
  - All processors share a single global memory address space
  - Communication is through shared variables in memory
  - Synchronisation is via locks (hardware) or semaphores (software)

Multiprocessors

- Uniform Memory Access (UMA)
  - All memory accesses take the same time
  - Do not scale well
- Non-uniform Memory Access (NUMA)
  - Each processor has a private (local**)** memory
  - Global memory access time can vary from processor to processor
  - Present more programming challenges
  - Are easier to scale

Multiprocessors

- NUMA
  - Communication and synchronization are achieved through message passing:
    - Processors could then, for example, communicate over an interconnection network
    - Processors use *send* and *receive* primitives

Multiprocessors

- The difficulty is in writing effective parallel programs:
  - Parallel programs are inherently harder to develop
  - Programmers need to understand the underlying hardware
  - Programs tend not to be portable
  - Amdahl's law; a very small part of a program that is inherently sequential can severely limit the attainable speedup
- "It remains to be seen how many important applications will run on multiprocessors via parallel processing."
- "The difficulty has been that too few important application programs have been written to complete tasks sooner on multiprocessors."

Multiprocessors

- Multiprocessors suffer the same memory problems as uni-processors and in addition:
  - The problem of maintaining *memory coherence* between the processors

Cache Coherence

- A read operation must return the value of the latest write operation

- But in multiprocessors each processor will (probably) have its own private cache memory
  - There is no guarantee of data consistency between private (local) cache memory and shared (global) memory

Processing in memory

- The idea of PIM is to overcome the bottleneck between the processor and main memory by combining a processor and memory on a single chip
- The benefits of a PIM architecture are:
  - Reduced memory latency
  - Increases memory bandwidth
  - Simplifies the memory hierarchy
  - Provides multi-processor scaling capabilities:
    - Cellular architectures
  - Avoids the Von Neumann bottleneck

Processing in memory

- This means that:
  - Much of the expensive memory hierarchy can be dispensed with
  - CPU cores can be replaced with simpler designs
  - Less power is used by PIM
  - Less silicon space is used by PIM

Processing in memory

- But …
  - Processor speed is reduced
  - The amount of available memory is reduced
- However, PIM is easily scaled:
  - Multiple PIM chips connected together forming a network of PIM cells
  - Such scaled architectures are called Cellular architectures

Cellular architectures

- Cellular architectures consist of a high number of cells (PIM units):
  - With tens of thousands up to one million processors
  - Each cell (PIM) is small enough to achieve extremely large-scale parallel operations
  - To minimise communication time between cells, each cell is only connected to its neighbours

Cellular architectures

- Cellular architectures are fault tolerant:
  - With so many cells, it is inevitable that some processors will fail
  - Cellular architecture simply re-route instructions and data around failed cells

- Cellular architectures are ranked highly as today's Supercomputers:
  - IBM BlueGene takes the top slots in the Top 500 list

Cellular architectures

- Cellular architectures are threaded:
  - Each thread unit:
    - Is independent of all other thread units
    - Serves as a single in-order issue processor
    - Shares computationally expensive hardware such as floating-point units
  - There can be a large number of thread units:
    - 1,000s if not 100,000s of thousands
    - Therefore they are massively parallel architectures

Cellular architectures

- Cellular architectures are NUMA
  - Have irregular memory access:
    - Some memory is very close to the thread units and is extremely fast
    - Some is off-chip and slow
- Cellular architectures, therefore, use caches and have a memory hierarchy

Cellular architectures

- In Cellular architectures multiple thread units perform memory accesses independently
- This means that the memory subsystem of Cellular architectures do in fact require some form of memory access model that permits memory accesses to be effectively served

Cellular architectures

- Uses of Cellular architectures:
  - Games machines (simple Cellular architecture)
  - Bioinformatics (protein folding)
  - Imaging
    - Satellite
    - Medical
    - Etcetera
  - Research
  - Etcetera

Cellular architectures

- Examples:
  - BlueGene Project:
    - Cyclops (IBM) – next generation from BlueGene/P, called BlueGene/C
      - DIMES – a prototype implementation
  - Gilgamesh (NASA)
  - Shamrock (Notre Dame)
  - picoChip (Bath, UK)

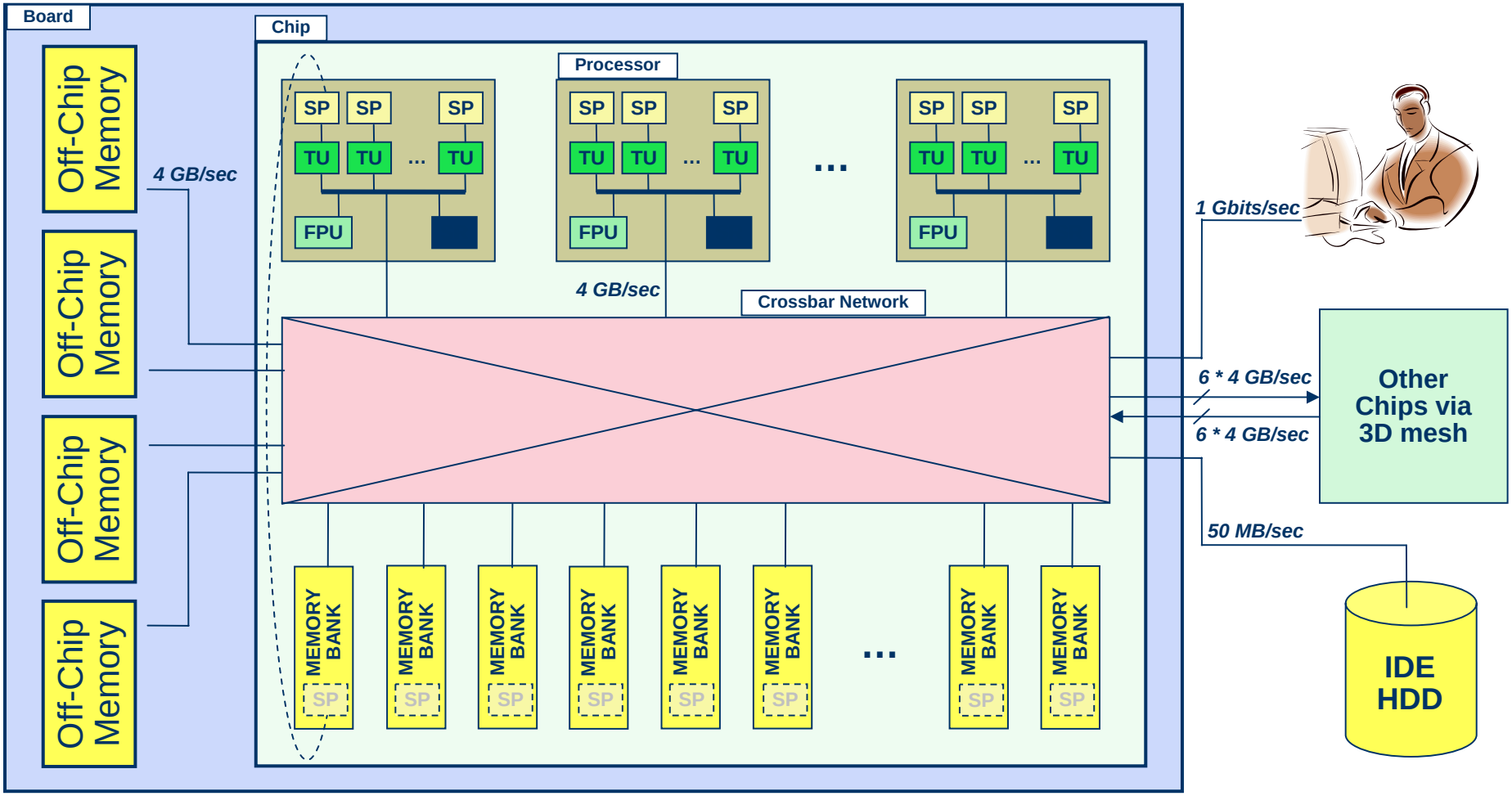
IBM Cyclops or BlueGene/C

- Developed by IBM at the Tom Watson Research Center
- Also called BlueGene/C in comparison with the earlier version of BlueGene/L and BlueGene/P

Cyclops

- The idea of Cyclops is to provide around one million processors:
  - Where each processor can perform a billion operations per second
  - Which means that Cyclops will be capable of one petaflop of computations per second (a thousand trillion calculations per second)

Cyclops



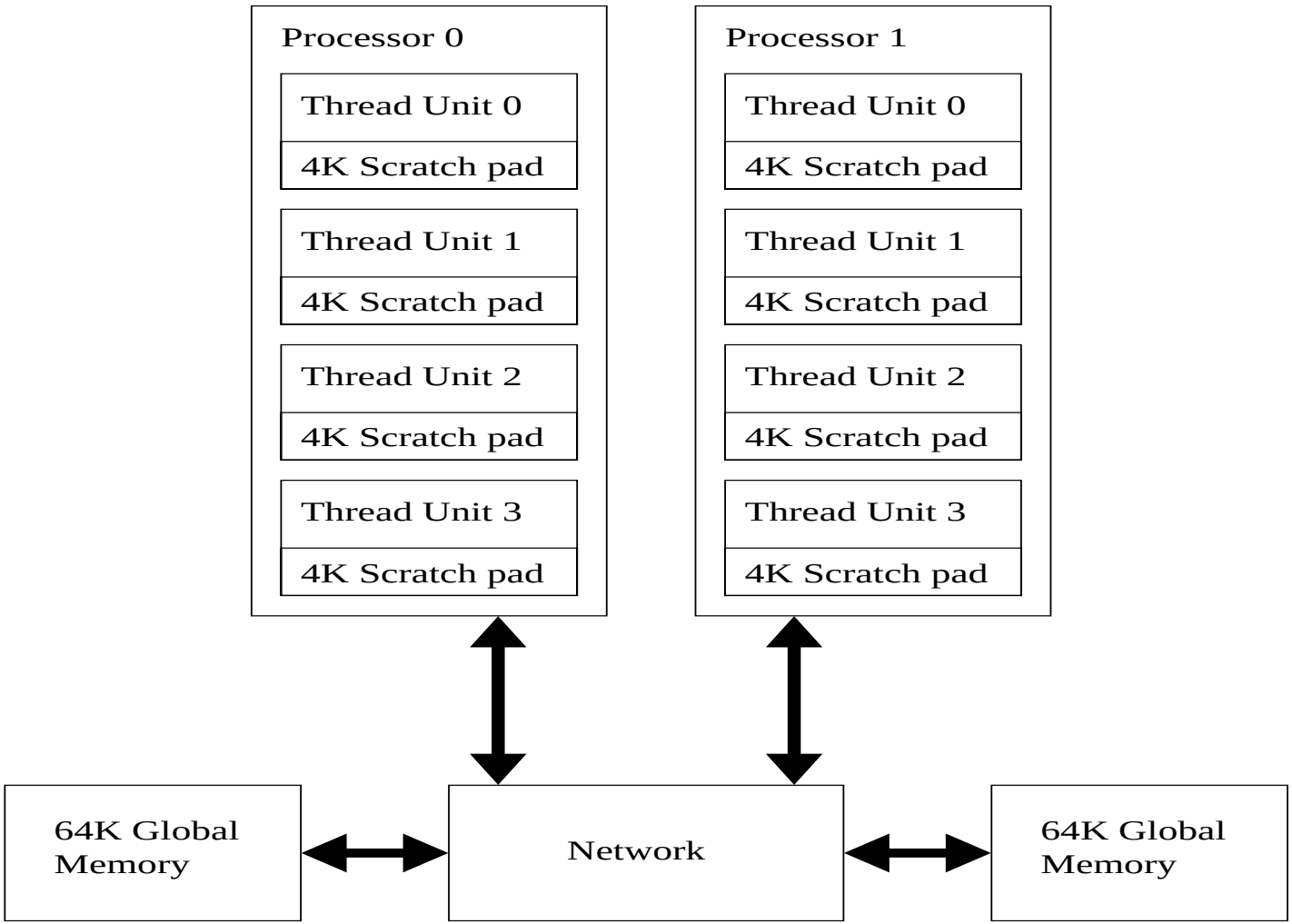
DIMES

- DIMES:
  - Is the first hardware implementation of a Cellular architecture
  - Is a simplified ‘cut-down’ version of Cyclops
  - Is hardware validation tool for Cellular architectures
  - Emulates Cellular architectures, in particular Cyclops, cycle-by-cycle
  - Is implemented on at least one FPGA
  - Has been evaluated by Jason

DIMES

- The DIMES implementation that Jason evaluated:
  - Supports a P-thread programming model
  - Is a dual processor where each processor has four thread units
  - Has 4K of scratch-pad (local) memory per thread unit
  - Has two banks of 64K global shared memory
  - Has different memory models:
    - Scratch pad memory obeys the program consistency model for all of the eight thread units
    - Global memory obeys the sequential consistency model for all of the eight thread units
  - Is called DIMES/P2

DIMES/P2



DIMES

- Jason's concerns were:
  - How to manage a potentially large number of threads
  - How to exploit parallelism from the input source code in these threads
  - How to manage memory consistency

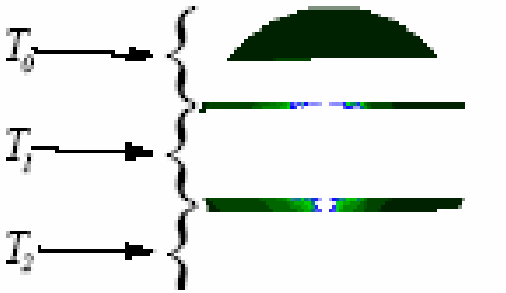
DIMES

- Jason tested his concerns by using an "*embarrassingly parallel program which generated Mandelbrot sets*"

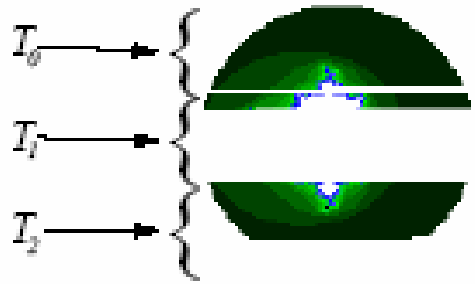
- Jason's approach was to distribute the work-load between threads and he also implemented a work-stealing algorithm to balance loads between threads:
  - When a thread completed its '*work-load'*, rather than remain idle that thread would '*steal-work*' from another '*busy*' thread
  - This meant that he maximised parallelism and improved thread performance and hence overall program execution time

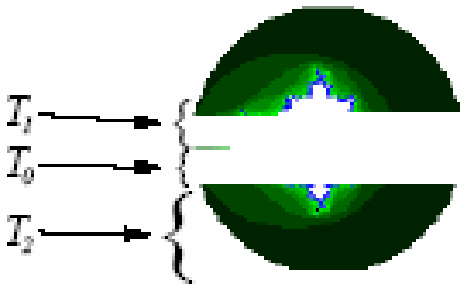
DIMES

Shortly after program start

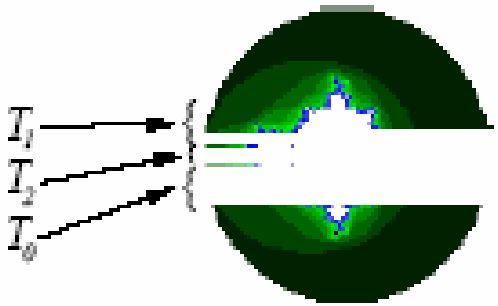


Shortly before work stealing





Just after work stealing



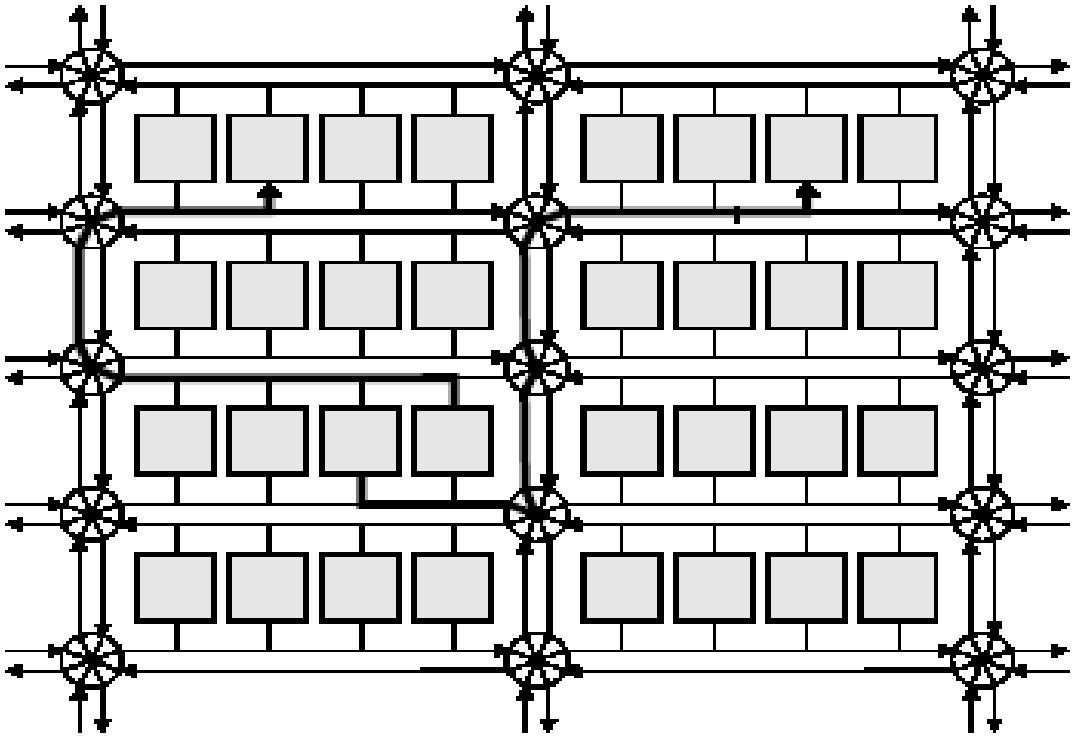
More work stealing

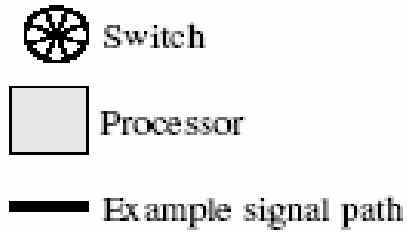
picoChip

- picoChip are based in Bath
  - *"… is dedicated to providing fast, flexible wireless solutions for next generation telecommunications systems."*
- picoArray™
  - Is a tiled architecture
  - 308 heterogeneous processors connected together,
  - The interconnects consist of bus switches joined by picoBus™
  - Each processor is connected to the picoBus™ above and below it

picoChip

- picoArray™





picoChip and Parallelism

- The parallelism provided by picoChip is synchronous
  - This avoids many of the issues raised by the other architectures that expose asynchronous parallelism
  - But it is at the cost of the flexibility that asynchronous parallelism provides

Abstraction of the Parallelism

- This may be done in various ways:
  - For example within the compiler:
    - Using trace scheduling, list-based scheduling or other data-flow based means amongst others
  - Using language features:
    - HPF, UPC or the additions to C++ in the IBM Visual Age compiler and Microsoft's additions
  - Most commonly using libraries:
    - For example: Posix threads, Win32 threads, OpenMP, boost.thread, home-grown wrappers

Parallelism using Libraries

- Using libraries has a major advantage over implementing parallelism within the language:
  - It does not require the design of a new language, nor learning a new language
  - Novel languages are traditionally seen as a burden and often hinder adoption of new systems due to the volume of existing source-code
  - But libraries are especially prone to mis-use and are traditionally hard to use

Issues of Libraries: part I

- The model exposed is very diverse:
  - Message passing, e.g. OpenMP
  - Exposes loop-level parallelism (e.g. “forall ...” constructs) exposes very limited parallelism in general-purpose code
  - Is very low-level, e.g. the primitives exposed in Posix Threads are extremely basic: mutexes, condition variables, and basic thread operations
- The libraries require experience to use well, or even correctly

Part II: Non-composability of atomic operations!

- The fact that atomic operations do not compose is a major concern when using libraries!

- The composition of thread-safe data structures does not guarantee thread-safety:
  - At each combination of the data structures, more locks are required to ensure correct operation!
  - This implies more and more layers of locks, that are slow...

Parallelism in the Compiler

- Given the concerns with regards to libraries, what about parallelising compilers?
- The fact is that auto parallelising compilers exist, e.g. list-based scheduling implemented in the EARTH-C (circa 1999) has been proven to be optimal for that architecture
- Data-flow compilers have existed for years
  - Why aren't they used?

Industrial Parallelising Compilers

- Microsoft is introducing OpenMP-based constructs into their compiler, e.g. “forall”
- IBM Visual Age has similar functionality
- Java has a thread library
- C++0x: much work has been done regarding standardisation with respect to multiple threads

C++ as an example

- The uses of C++ makes it an interesting target for parallelisation:
  - Although imperative, so arguably flawed for implementing parallelism
  - It has great market penetration, therefore there is much demand for using parallelism
  - Commonly used in high-performance, multi-threaded systems
  - General-purpose nature and quality libraries are increasing the appeal to super-computers

Parallelism support in C++

- Libraries exist beyond the usual C libraries:
  - boost.thread – exists now, requires standards-compliant compilers
  - C++0x: details of the threading support are becoming apparent that appear to include:
    - Atomic operations (memory consistency), exceptions, machine-model underpins approach
    - Threading models: thread-as-a-class, lock objects
    - Thread pools – coming later – probably
    - More details on the web, or at the ACCU - in flux

Experience using C++

- Recall DIMES:
  - Prototype of massively parallel hardware
  - Posix-threads style library implementing threads
  - C++ thread-as-a-class wrapper implemented

- Summary of experience:
  - Hardly object-orientated: no separation in design of the Mandlebrot application with the work-stealing algorithm and thread pool
  - The software insufficiently separated the details of the hardware features from the design

Further experiences using C++

- From this work and other experiences, I developed a more interesting thread library:
  - Traits abstract underlying OS thread API from wrapper library
  - Therefore has hardware abstractions too
  - Provision of higher-level threading models:
    - Primarily based on futures and thread pools
  - Use of thread pools and futures creates a singly rooted-tree of threads:
    - Trivially *deadlock free* – a holy grail!

C++0x threads now? *No!*

- Included in libjmmcg:
  - Relies upon non-standard behaviour and broken optimisers! For example:
    - Problems with global code-motion moving apparently const-objects past locks
    - Exception stack is unique to a thread, not global to program and currently unspecified
    - Implementation of std::list
  - DSEL has syntax limitations due to design of C++
  - Doesn't use boost ...
  - Has example code and test cases
  - Isn't complete! (e.g. Posix & sequential specialisations incomplete, some inefficiencies)
  - But get it from libjmmcg.sourceforge.net, under LGPL

Trivial example usage

```
struct res_t { int i; };
struct work_type {
    typedef res_t result_type;
    void process(result_type &) {}
};
pool_type pool(2);
async_work work(creator_t::it(work_type(1)));
execution_context context(pool<<joinable()<<time_critical()<<work);
pool.erase(context);
context->i;
```

- The devil is in the omitted details: the typedefs for:
  - `pool_type`, `async_work`, `execution_context`, `joinable`, `time_critical`
- The library requires that the work to be mutated has the items in *`italics`* defined

Explanation of the example

- The concept is:
  - that asynchronous work (`async_work`) that should be mutated (*`process`*) to the specified output type (*`result_type`*) is transferred into a thread pool (`pool_type`) of some kind
- This transfer (`<<`) may, optionally (`joinable`), return a future (`execution_context`)
  - Which can be used to communicate (`->`) the result of the mutation, executed at kernel priority (`time_critical`), back to the caller
- The future also allows exceptions to be propagated

More details regarding the example

- The thread pool (`pool_type`) has many traits:
  - Master-slave or work-stealing
  - The thread API (Win32, Posix or sequential)
  - The thread API costs in very rough terms
  - Implies a work schedule that is a fifo baker's ticket schedule, *implementation of GSS(k) is in progress*
- The library effectively implements a software simulation of data-flow
- Wrapping a function call, plus parameters, in a class converts Kevlin's threading model to this one

Time for controversy.... What faces programmers...

- Large-scale parallelism is here, now:
  - Blade frames at work:
    - 4 cores x 4 CPUs x 20 frames per rack = 320 thread units, in a NUMA architecture
- The hardware is expensive!
- But so is the software ...
  - It must be programmed, economically
  - The programs must be maintained ...
- Or it will be an expensive failure?

Talk summary

- In this talk we have looked at parallel processing and justified the reasons for Processing In Memory (PIM) and therefore cellular architectures
- We have briefly looked at two example architectures:
  - Cyclops and picoChip
- Jason has worked on DIMES, the first implementation of a (cut-down) version of a cellular architecture
- The issues of programming for these massively parallel architectures has been described
- We focussed on the future of threading in C++

The University of Hertfordshire

The Challenges facing Libraries and Imperative Languages from Massively Parallel Architectures

Questions?

Jason McGuiness
Computer Science
University of Hertfordshire
UK

Building Futures in Computer Science
empowering people through technology

Colin Egan
Computer Science
University of Hertfordshire
UK