Why are Multicores a Challenge?

David May

Bristol University and XMOS

Multicore, Manycore, Millioncore

What are the challenges?

What are the new challenges?

We've been doing parallel computing for 50 years

... with a wide variety of architectures

Background

We can build chips with hundreds of processors

We can build computers with millions of processors

We can support parallel programming in hardware

We can build digital systems in software

But our thinking is still dominated by sequential ideas

Parallel Program Patterns

Parallel Random Access Machines (PRAMs/BSP)

Data Parallelism / Process Arrays

Directed Dataflow Graphs

Task Farms and Server Farms

Event handlers

Recursive use of any of the above

Multicores

Four things that multicores need to do

concurrency

communication

synchronisation

event handling

Multicores

Four things that multicore programmers need to understand

concurrency

communication

synchronisation

event handling

Multicores

Four things that (most) designers of multicores are trying to do

shared memory

shared memory

shared memory

shared memory

Why shared memory is difficult to use

Concurrent write-write, write-read or read-write $\Longrightarrow$ non-determinacy

Formal verification of shared memory programs is ‘challenging’

Empirical verification is impossible except for the simplest cases

Access collisions even for read-read $\Longrightarrow$ latency

Why shared memory is not enough

The *writer* can put the data where the *reader* can access it

But the *reader* doesn't know it's there

Either the *reader* has to look over and over again (polling) - or it has to be notified by an interrupt

Polling uses processor and memory cycles and wastes energy

Interrupts take a lot of cycles and introduce timing issues

And what about synchronisation?

Why timing matters

In parallel processors, there are many potential sources of timing jitter - in processors, caches, memory and interconnect

Have to manage latencies in communication
... can't manage latency if you don't know how big it is

Have to provide synchronisation (barriers) in control flow or data flow
... these are delayed by the *slowest* participant

May need to use buffers to maintain rates when processing streams
... can't determine buffer sizes if you don't know the variance in timing

Some Probabilities

Probability p of no cache miss when executing program P

Suppose we execute n copies of P in parallel, terminating in a barrier

Probability that the barrier will not be delayed by a cache miss = p^n

For n = 100 and p = 0.99, p^n = 0.37

For n = 1000 and p = 0.99, p^n = 0.00004

If there are caches or other sources of jitter the performance of sequential composition behaves as expected but the performance of parallel composition doesn't

Three programs (1)

```
{ int i;
  int n;
  int sum;
  sum = 0;
  n = 0;
  for (i=0; i<maxdata; i++)
  { sum = sum + a[n];
    n = n + 1;
    if (n > maxdata)
      n = n - maxdata;
  }
}
```

Three programs (2)

```
{ int i;
  int n;
  int sum;
  sum = 0;
  n = 0;
  for (i=0; i<maxdata; i++)
  { sum = sum + a[n];
    n = n + 1000;
    if (n > maxdata)
      n = n - maxdata;
  }
}
```

Three programs (3)

```
{ int i;
  int n;
  int sum;
  sum = 0;
  n = 0;
  for (i=0; i<maxdata; i++)
  { sum = sum + a[n];
    n = a[n] + 1000;
    if (n > maxdata)
      n = n - maxdata;
  }
}
```

Three programs - performance

In 1980, these three programs would have had almost the same performance

Today, the difference on a state-of-the-art computer is at least 50

How can efficient software be designed for unpredictable machines?

How much software is running 50 times slower than it should?

How many 50-CPU clusters have been bought where a single computer would do?

And on thousands of processors

... the opportunity for inefficiency is much greater

A recent post about Hadoop (map-reduce):

"... this technique is an astonishing factor of 1340 times less efficient than an alternative technique for processing sub-graph pattern matching queries ..."

... problems with misuse of abstraction: abstraction should be about *managing complexity*, not *hiding* things people need to know about

General Purpose Parallel Processing

Think in terms of *Processes and Communication Patterns*, not *Algorithms and Data Structures*

Scalable interconnect to support *any* communication patterns (eg Clos networks, not 2-D meshes)

Low latency communication enables high parallelism and rapid spread of computation

Bounded latency enables latency hiding by the processors

Collections of processors can be used to execute large sequential programs

Communication

... is not an “overhead”

Most programs spend a lot of instructions moving data around

Re-arrangement of data is often a key part of the algorithm

“The majority of actual instruction tables will consist almost entirely of the initiation of subsidiary operations and transfers of material”

Alan Turing, 1945

Design the interconnect first; then design the processors to keep it busy

Communication Patterns

Communication and data access patterns are often known, especially in embedded processing

They often involve a series of permutation routing operations between known endpoints

Compilers can allocate processors and network routes

Timing can be deterministic - potential of *real time* parallel processing

For unknown patterns, use randomisation

For many-to-one, use hashing and combining (or replication)

Composition

Communication patterns can be composed and embedded within each other

Sometimes the entire program evolution is visible to a compiler

Sometimes the evolution is data-sensitive

The issues in allocating processors and network routes mirror those of allocating memory in sequential processing

... global, stack, heap

Want to optimise cases which can be compiler analysed

Why we need parallel languages

We need to express processes and communication patterns easily and clearly

We want to be able to use different patterns in combination

We want to be able to optimise parallel programs

We need to program events and interactivity - there will be lots of it

And ideally, we'd like to unify hardware and software design

Big Data Structures

Implement big data structures as server farms

Use hashing to even distribution and access load

Provide concurrent access (with combining or replication if needed)

Implement application-specific caching (if needed)

This follows known techniques for implementing scalable PRAMs

Executing Sequential Algorithms

Implement global memory as a server farm

Implement groups of procedures, functions and objects as servers

Accesses to global data are less than 10% of instructions; calls are less than 5%

Network is under-loaded

Optimisations: concurrent accesses and concurrent calls

We can think of a universal computer as an infinite collection of small processors, instead of a small collection of infinite processors

Universality and Efficiency

Universal parallel machine = Universal processors + Universal interconnect

In latency tolerant patterns (such as streaming), optimal

In latency sensitive patterns, subject to $log(p)$ runtime overhead

Potential to replace runtime overhead with $log(p)$ *excess parallelism* hiding latency

In sequential patterns, $log(p)$ overhead only affects global accesses

... simulations suggest a factor of 2-3 overhead implementing 4GB memory using 64-4096 processors

Non-Blocking Networks

Clos networks implement *permutations* on their inputs

A *strict-sense* network can always allocate a new route

A *re-arrangeable* network needs fewer routers but may require re-arrangement of existing routes

Compiler can implement known permutation patterns on a re-arrangeable network

Multiple routes per processor can be implemented using time-division multiplexing

Redundant processors, routers and links can be provided

Processor Node Architecture

Source and sink messages concurrently as fast as the interconnect can handle them

Low latency on very short messages

Multi-threading for latency hiding

Time-deterministic execution

Simple architecture enabling on-the-fly compilation/optimisation is useful

... although in fact, almost any architecture can be used provided it doesn't introduce too much jitter

Time-deterministic execution

Use simple execution architecture

Eliminate caches - memory operations should take one CPU cycle

... or at least, reduce memory hierarchy using stacked DRAM

Use non-blocking networks

Use lots of processors - idle processors are good for responsiveness

Summary

Scalable, real-time, parallel architecture

Supports all patterns with low overhead

Software topology independent of physical topology

Homogeneous or Heterogeneous processing nodes

Energy efficient - idle processors can be switched off

Redundancy

Commodity parallel processing

Ideally, we want to build processing like memory

Choose an economical chip size ($70mm^2$ for DRAM), $100mm^2$ for logic

... this will hold hundreds of processors

Stack them up in 3D using through silicon vias.

Connect them using silicon photonics and wavelength division multiplexing

General purpose components with behaviour defined by software

Heterogeneous architectures

Really want heterogeneity in the *implementation*, not the *architecture*

... one programming model, several optimisations

Even then, unless you're optimising for a specific application you don't know what the ratio between the heterogeneous components should be

... potentially need a whole range of products with varying ratios

Attached accelerators have been coming and going for a long time!

Parallelising compilers

... for ‘legacy’ software have been promised for a long time!

It seems unlikely that programs expressed in terms of sequentially traversing large data structures can be automatically converted into programs expressed in terms of efficient communication patterns

... with or without cache coherent shared memory

And the legacy programs keep on growing and becoming more complex

... so the prospects for auto-parallelisation are probably not as good now as they were in the 1980s!

Verification

Eliminate race conditions: disjointness of variables for parallelism; synchronised communication for event-handling

Make a clear distinction between data-sensitive and data-insensitive communication patterns

Ideally, communication patterns and their evolution is data insensitive; optimisable and verifiable

If it's data sensitive, it should probably be expressed in ways that can be reasoned about (eg recursion)

And sometimes we need to verify timing; sometimes we need to verify external interactions

The education project

We have an opportunity to build systems of unprecedented capability: we need to educate a generation of *concurrent thinkers*

Processes and Communication Patterns, not *Algorithms and Data Structures*

They should learn about Sequence, Concurrency and Event-handling (interaction) *at the same time*

In schools, a good way to do this is through robotics: “I want my robot to dance and sing at the same time”

In Universities, we need to re-think the curriculum: today most systems involve concurrency but most courses don’t

The open computing project

How simple can we make a parallel processor?

Simple deterministic processing node

Non-blocking network

Concurrent programming language and operating system

Implementable in anything - including emerging technologies

Nothing hidden by abstraction layers

The embedded HPC project

There will be many new applications involving embedded intelligence and real-time HPC

interaction: sensors, actuators and haptics

vision, language

robotics

real-time emulation

high-performance control

...

The Multicore Opportunity

Commodity high performance processing

Scalability: processing, communication, event-handling

Rapid design of innovative consumer products

Embedded Intelligence and Real-time HPC

Systems that *learn* replacing systems that have to be *programmed*

Energy efficiency: the *dark silicon* opportunity