PRAM on Chip

Can Parallel Computing Finally Impact Mainstream Computing?

U. Vishkin, vishkin@umiacs.umd.edu.
With graduate students (Balkan, Berkovich, Caragea, Dascal, Gu, Keceli, Naishlos, Nuzman, Wen), post-doc (Kupershtok), undegrads (Ba, Beytin, Mazzucco) micro-architecture (Franklin, Jacob), compiler (Barua, Tseng), VLSI/power (Qu), and graphics (Olano; student: Wang) experts.

A strategic question and what it implied

Essence of my work since 1979: seek to improve general-purpose single task completion time by parallelism.
Underlying assumption: some day parallel computing will affect mainstream computing.
Strategic question: what to do and in what order?

70s&80s: Much interest in parallel computing
Most pragmatists: build a good parallel architecture. Once built, figure out how to best program (or compile for) it. "Algorithms are worthwhile only if they fit real machines".
Minority opinion: parallelism must come from programmer; must be as simple as possible; thinking algorithmically in parallel is an "alien culture"; must be developed/understood first. Once developed: derive architecture specs; build architecture. Counter-intuitive to many: "parallel algorithms (theory...) should be prescriptive rather than descriptive".

A mostly-theory community developed the "PRAM theory". Many just enjoyed the theory

PRAM-On-Chip at UMD: current post-theory chapter

Goal Of This Talk

Describe the PRAM-On-Chip framework that extends from high level APIs to VLSI implementation.

Emphasis Of Presentation

More What and Why than How.

• What should the computer system be engineered for? in what ways/languages will the computer system be programmed? "specs".
Engineering from 30,000 feet: (i) specs (ii) design.
Past: Not much thought in mainstream computing. Specs for Pentium 4: same as for Pentium 3...
• Why: applications, run time, development time.
• How will the computer system be built? Time permitting.

Background in a nutshell

Tribal lore, parallel programming profs, DARPA HPCS Development Time study (2004-2008):
Parallel algorithms and programming for parallelism is easy. (Non-trivial: need to learn much to understand how easy they are... oxymoron?! how easy can be non-trivial?.)
However, "coarse-graining" this parallelism to optimize performance on multi-chip multi-processing (with their high coordination overhead) is hard.

Other side of the same coin: "decomposition-first" parallel programming. Known to be hard. Recall the 1990s "parallel software crisis": still valid.

Press Release, Intel, March 8, 2005

The game is over for software that is written only for a single processor... Intel is providing the platform and tools to help out the game developers as they start the transition to a multi-core and multi-thread environment.

Is it Deja vu all over again?

The PRAM: parallel random-access (virtual machine) model

• Ideal PRAM: latency for arbitrary number of memory accesses, same as for one access.
• Algorithmicist states (or, does not hide..) what can be done concurrently.
• Algorithmic knowledge-base *2nd only to serial algorithms*.
Simplest parallel model. Quite a few: surveys, whole books.
• CS archaeology:
1988-90: main algorithms textbooks; Baase 88, Manber-89, CLR-90, included PRAM algorithms chapters. Model of choice for parallel algorithms in all major algorithms/theory communities. Was taught everywhere.
1970s-1990s: Fierce "battle of ideas" by much CS talent: Seek the "ultimate" parallel programming model: (i) easy expression of parallel algorithms and their programs in the model, and (ii) validation of the model by algorithmic paradigms and solutions for as many problems as possible.
PRAM approach: a clear winner. Natural selection!
Take home: "Darwin has already spoken".

• Hypothetical: suppose PRAM-like programming ran well on a 1990 computer. Premise: PRAM algorithms would

be studied by every CS student.

Example

Given: (i) All commercial airports in the world. (ii) For each, all airports to which there is a non-stop flight

Task: Find the smallest number of flights from DCA to every other airport

Principle of PRAM algorithm

Parallel Step i: Given all the airports which require $i-1$ flights, find (concurrently!) all those that require i flights

Observe:

(i) "Concurrently": only change to serial algorithm

(ii) No "decomposition"

(iii) Inherent serialization: S - number of parallel steps

Total number of operations: T - O(number of flights)

Orders of magnitude gain relative to "serial":

$O(T/S)$ decisive also relative to coarse-grained parallelism.

(iv) Takes the better part of a semester to teach!

Expect: "solid and QUICK"; issue: Calculus-like basics.

A. Einstein: "Make everything as simple as possible, but not simpler".

In reality

Only multi-chip multi-processors were available:
Option 1. Run as is: trouble with performance. Overheads too high. 1993 LOGP paper: "PRAM unrealistic"; "should not be taught". Outcome:
PRAM non-complexity community dispersed. Funding dried up. CLRS-01: parallel algorithms chapter out. Fact of natural selection did not register in the collective memory of CS.
Option 2. Tune for machine. Dictates difficult "decomposition-first" parallel programming. High development time costs.
How successful?! Googled: Spring 2005 parallel programming classes. Surprisingly few. What's buildable was not programmable and vice versa.
New DARPA term: productivity.

Mainstream computing. Rely on serial code. Max issue/clock for successful commercial processor on singe task
1991-4: 2-6. 2005: 3-4. Stagnation!
Faster clock: not what used to be.

Quantitative HW enabler # transistors/chip
1980: 29K. 2005: 1B. Factor: 30,000X!

PRAM-On-Chip

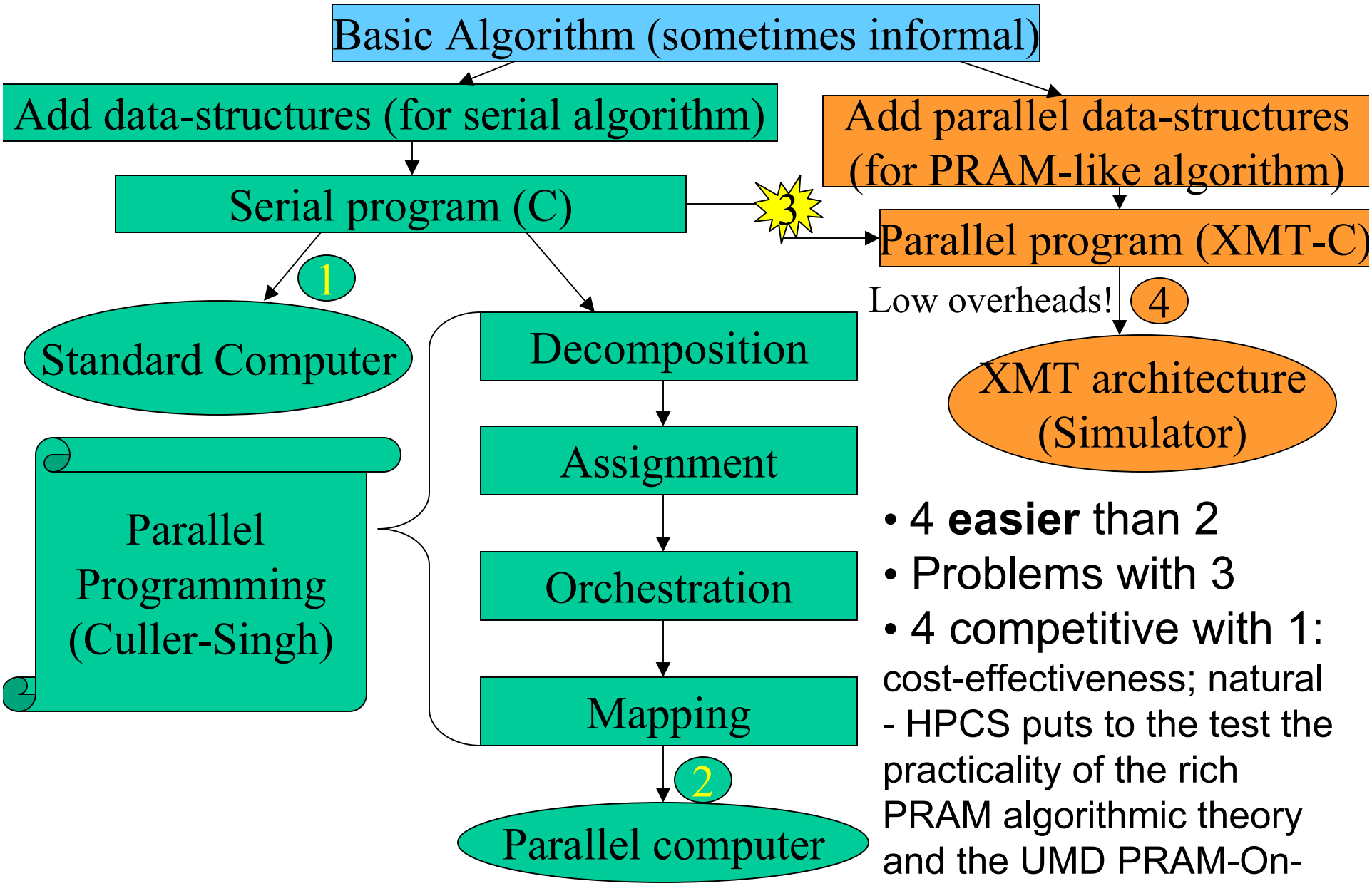
Put quantity advantage to good use Newer insight: lower overheads are possible with on-chip multi-processors. PRAM On Chip relies on ability to get much better bandwidth and latencies on chip.
Corollary: Option 1 is alive again!
Roll-back clock to 1991-3: The conclusion is clear. Just teach the PRAM chapter.
2005: what has changed besides not having the chapter in the text?!
Claim: the case only became stronger.

The productivity picture

- Performance programming, and
- The high-level API enabler:

Can you come up with the following win-win proposition?
An API for your favorite application whose:
(i) programming is so much easier than C programming that it could be done effectively by a non-computer-scientist application expert. May be even an existing (!) API:
OpenGL - Dr. Olano
VHDL - current MS student
and
(ii) PRAM-On-Chip performance is better than the best serial implementation of the C code.
The win-win proposition: better performance AND easier (cheaper!) to develop
Almost too good to be true: major impact across much of IT.
Note: Limited success: relying on compilers to extract parallelism from serial code. Detours it.

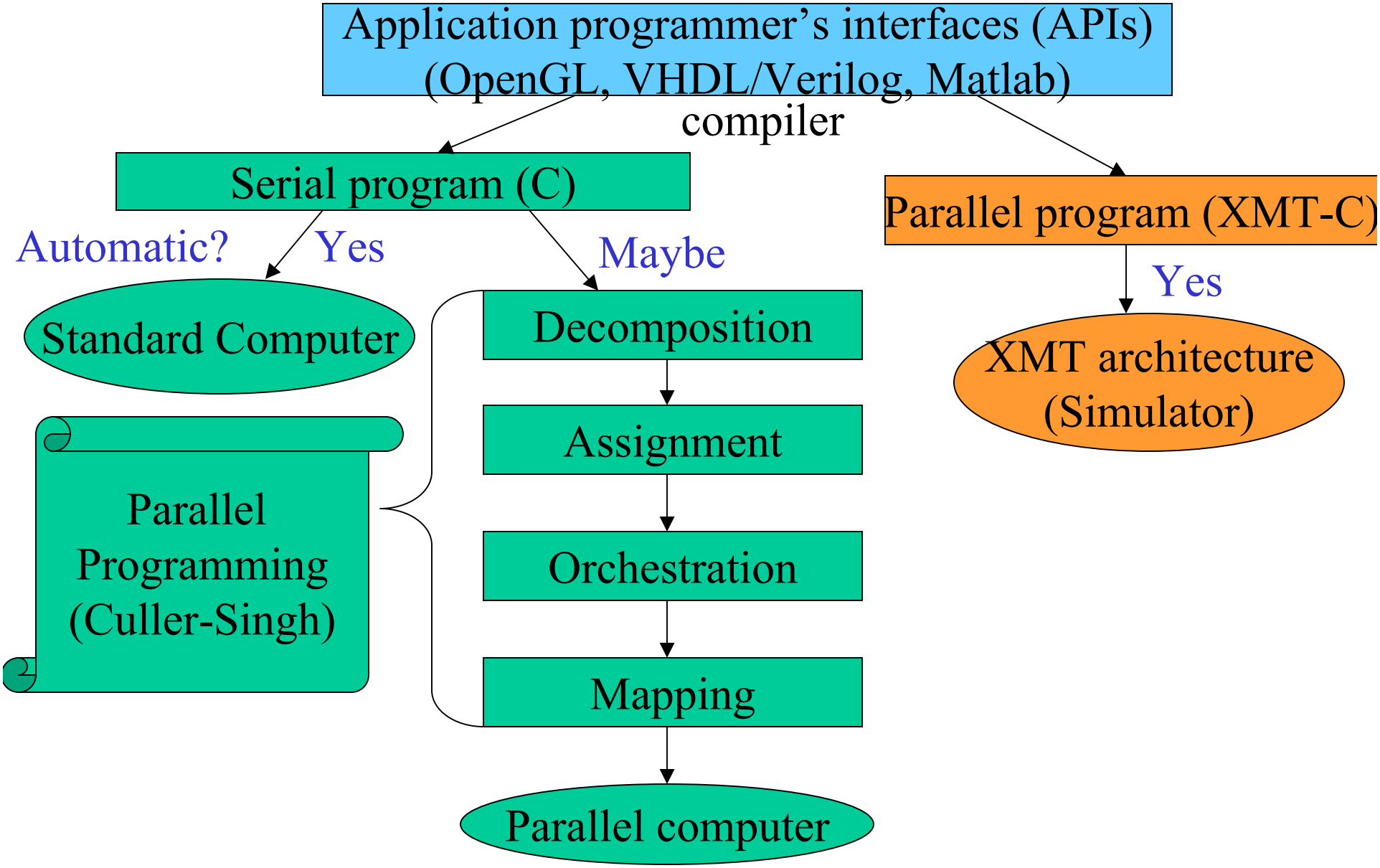
PERFORMANCE PROGRAMMING & ITS PRODUCTIVITY



- 4 **easier** than 2
- Problems with 3
- 4 competitive with 1:

cost-effectiveness; natural - HPCS puts to the test the practicality of the rich PRAM algorithmic theory and the UMD PRAM-On-Chip project

APPLICATION PROGRAMMING & ITS PRODUCTIVITY



The world is getting closer

Mark Twain once said: "The reports of my death are premature".
Some (early 1970s): "fast serial computing progress coming to an end".
2005 - Some: "this was wrong". Others: "premature"

The XMT vision was introduced in 1997. The world is getting closer:
(i) SUN: new Niagara architecture with 32 CPUs on chip.
(ii) Intel: all future processor chips will have multiple CPUs on the chip.
(iii) Herb Sutter, Microsoft 2005: "The Free Lunch Is Over: A Fundamental Turn Toward Concurrency in Software". States: hardware vendors have run out of room with most of their traditional approaches to boosting CPU performance (driving clock speeds and straight-line instruction throughput). Suggests that this puts us at a fundamental turning point in software development, where increased reliance on concurrency is expected. Not less significant than incorporation of OO Programming– his territory at MS.
(iv) Nov 2004 interview with Burton Smith, Cray's Chief Scientist (in HPCWIRE): "...the uniprocessor has pretty

well run out of steam. Parallelism to date has been a nice strategy for HPC users and an afterthought for microprocessor vendors. Now, it is becoming a matter of business survival for all processor vendors. Parallelism is going to be taken more seriously, starting with the idea of exploiting multi-threading and multiple cores on a single problem. This is a major change. Imagine if Microsoft wanted to write Office in a parallel language. What would that language be, and what would be the architecture to support it? We don't have good answers to these questions yet." XMT provides answers to these questions about language and architecture.

But, will languages prescribe architectures? time will tell; some recent reports suggest: OSs, applications are tailored to fit multi-core multi-threaded architectures. If true, will it work? or will they (again) find out the hard way?

Consortium of 5 companies and 10 (some top) universities. Seeks EU funding for realizing the PRAM-On-Chip vision (they said it): chip-Supercomputing using PRAM-like programming.

Application domains

Interactive visualization and virtual reality, CAD, control of switch fabric, ballistic missile defense, weather forecasting, radar processing, weapons design.
Special attention: high-end simulations. Where?
Molecular simulations (e.g., for drug discovery, protein folding). Suppose 10^{14} steps need to be simulated. A rate of step per nanosecond could be possible. Exciting: the whole simulation takes a day instead of 1000 days with multi-chip multiprocessing.
Namely, computationally demanding applications where the total number of rounds that needs to be simulated in a given time exceeds current parallel platforms.
Take note: (i) Pharma market larger than IT! (ii) Perhaps no better way to help mankind. (iii) System biology (e.g., SBML). (iv) Expediting drug approval (by the FDA).
"Best kept secret": almost nobody is working on this (1st: longest sequence; 2nd: as much parallelism).
Supercomputing companies focus on computing power.

An Overall Design Challenge

- Showed algorithm scalability.
- Hardware scalability: put more of the same
- ... but, how to manage parallelism coming from a programmable API?

Spectrum of Explicit Multi-Threading (XMT) Framework

Algorithms $--> $ architecture $--> $ implementation.

- XMT: strategic design point for fine-grained parallelism
- New elements are added only where needed

Attributes

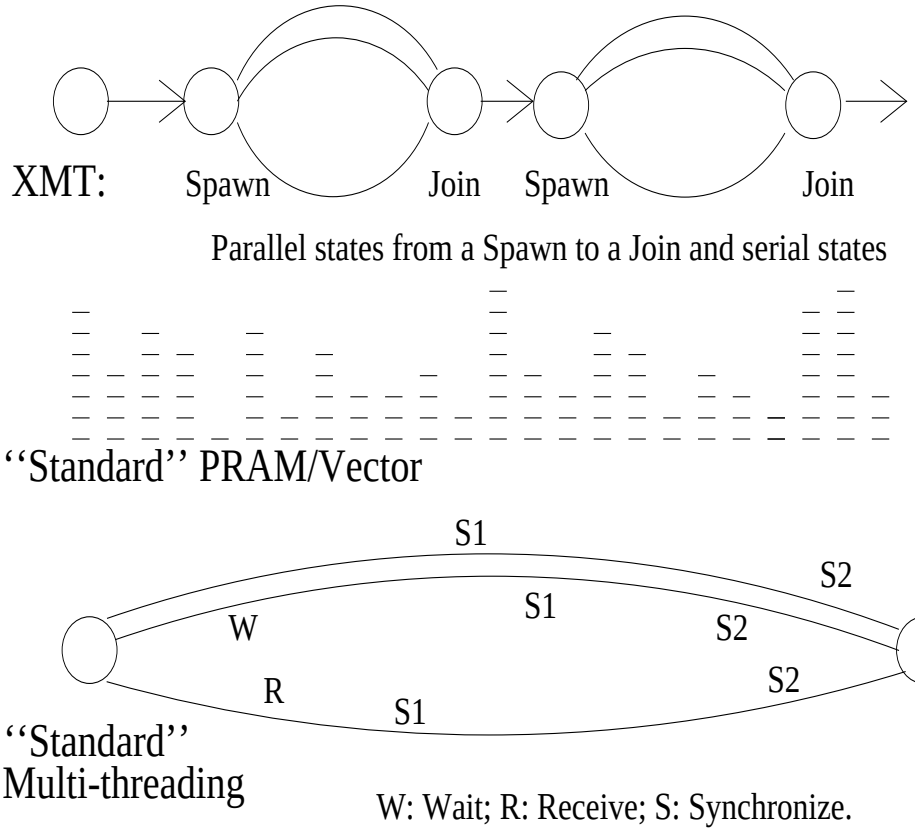
- Holistic

A variety of subtle problems across different domains must be addressed:

- Understand and address each at its correct level of abstraction

Snapshot: XMT High-level and assembly languages

For contrast: vector and standard multi-threading

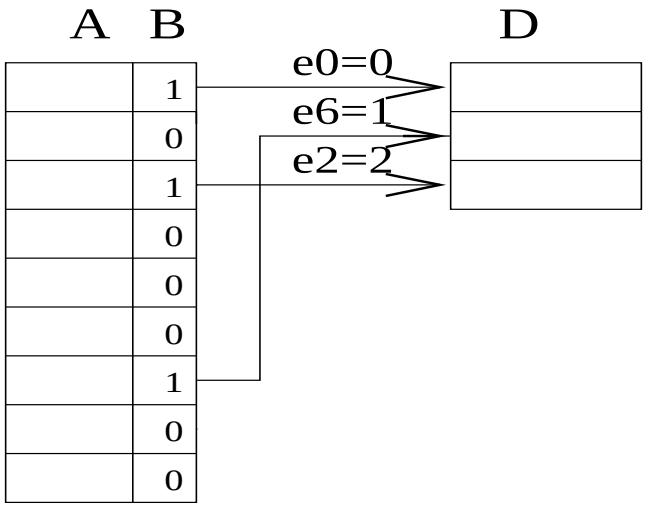


Cartoon Spawn creates threads; a thread progresses at its own speed and expires at its Join. Synchronization: only at the Joins. So, virtual threads avoid busy-waits by expiring. New: Independence of order semantics (IOS).

Programming interface: XMT-C and Assembly

The array compaction (artificial) problem

Input: Array $A[1..n]$ of elements; binary array $B[1..n]$.
Map in some order all $A(i)$, where $B(i) = 1$, to array C.



The array compaction problem.
Array B: bit values for the compaction.
For the program below: e0, e1 and e6
are the e values for threads 0, 2 and 6; x is 3.

XMT-C: High-level language

Single-program multiple-data (SPMD) extension of standard C. Includes Spawn and PS - a multioperand instructions.

Algorithm level (high-level program)

```
int x = 0;
Spawn(0,n)   /* Spawn n threads; $ ranges 0 → n − 1 */
{ int e = 1;
  if (B[$] == 1)
     { PS(x,e);
       D[e] = A[$] }
  }
n = x;
```

Notes: (i) PS is defined next (think F&A). See results for e0,e2, e6 and x. (ii) Using C-style scoping, Join instructions are implicit.

XMT Assembly Language

Standard assembly language, plus 3 new instructions: Spawn, Join, and PS.

The PS multi-operand instruction

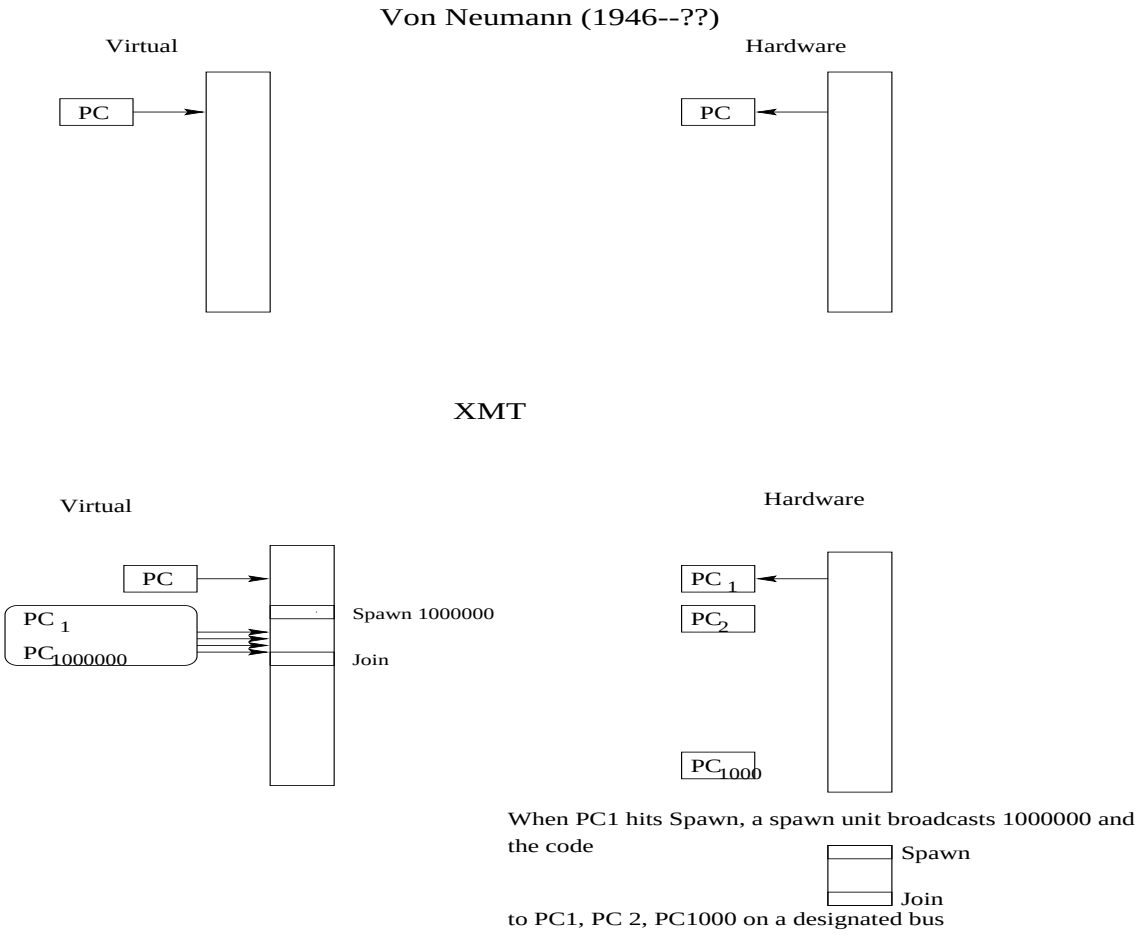
New kind of instruction: *Prefix-sum* (PS).
Individual PS, $PS\ Ri\ Rj$, has an inseparable ("atomic") outcome: (i) Store $Ri\ +\ Rj$ in Ri, and (ii) store original value of Ri in Rj.
Several successive PS instructions define a multiple-PS instruction. E.g., the sequence of k instructions:
$PS\ R1\ R2$; $PS\ R1\ R3$; ...; $PS\ R1\ R(k+1)$
performs the prefix-sum of the *base* $R1$ and the *elements* $R2, R3, ..., R(k+1)$ to get: $R2\ =\ R1$;
$R3\ =\ R1\ +\ R2$; ...; $R(k+1)\ =\ R1 + ... + Rk$;
$R1\ =\ R1 + ... + R(k+1)$.
Idea: (i) Several ind. PS's can be combined into one multi-operand instruction. (ii) Executed by a **new multi-operand PS functional unit**.

Mapping PRAM Algorithms onto XMT

(1) PRAM parallelism maps into a thread structure
(2) Assembly language threads are not-too-short (to increase locality of reference)
(3) the threads satisfy IOS

Machine extensions to the von Neumann model

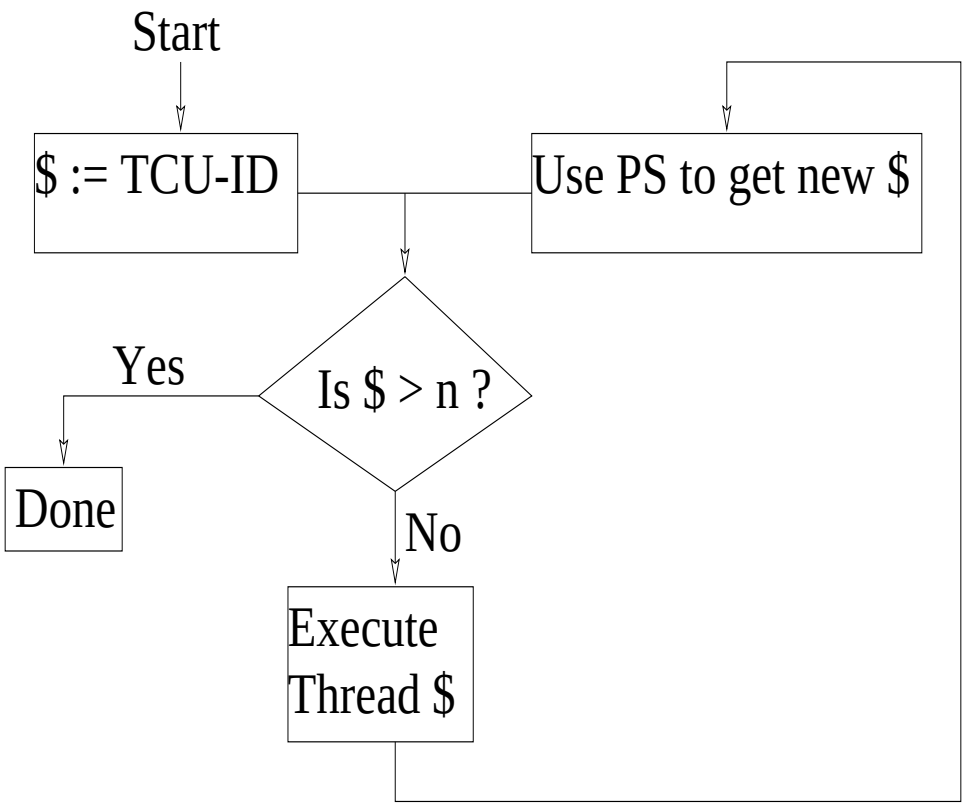
Program counter + stored program:
– a remarkable Darwinistic success story.
Pragmatic: Upgrade rather than replace.
Reaxiomatize 1946 von-Neumann's "mathematical machines";
yet, backwards software compatibility.



Snippet of a machine execution model

Given: a Spawn of n threads. The hardware determines which virtual thread to execute next in a distributed fashion

The program of a thread control unit (TCU)

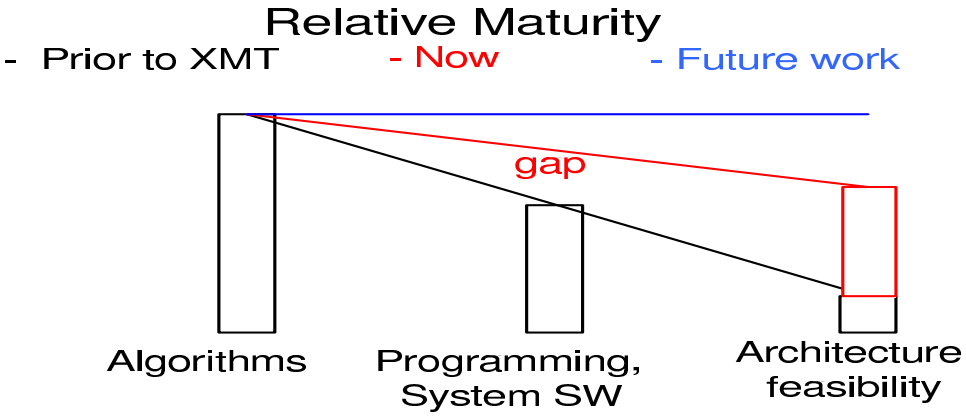


Next: use "Billion transistor" chip, for: **memory, interconnect and low overhead management of parallelism**.

Hardware Design Plans

- strong MTCU
- 1024 TCUs in 64 clusters
- 64 memory banks on chip
- first level of cache already shared
- cache-coherence defined away
- read latency 1st level cache: 20+ clocks!
- good news: bandwidth indeed not a problem

More on status later. Here in a nutshell:
- completing synthesizable Verilog
- leveling-off algorithm maturity: SW prefetching & HW support; programming; compiler; debugging; applications; team-up..



The Memory Wall

Concerns: 1) latency to main memory, 2) bandwidth to main memory.
Position papers: "the memory wall" (Wulf), "its the memory, stupid!" (Sites)

Note: (i) Larger on chip caches are possible; for serial computing, return on using them: diminishing. (ii) Few cache misses can overlap (in time) in serial computing; so: even the limited bandwidth to memory is underused.

XMT does better on both accounts:
- uses more the high bandwidth to cache.
- hides latency, by overlapping cache misses; uses more bandwidth to main memory, by generating concurrent memory requests; however, use of the cache alleviates penalty from overuse.

Conclusion: using PRAM parallelism coupled with IOS, XMT reduces the effect of cache stalls.

Memory architecture, interconnects

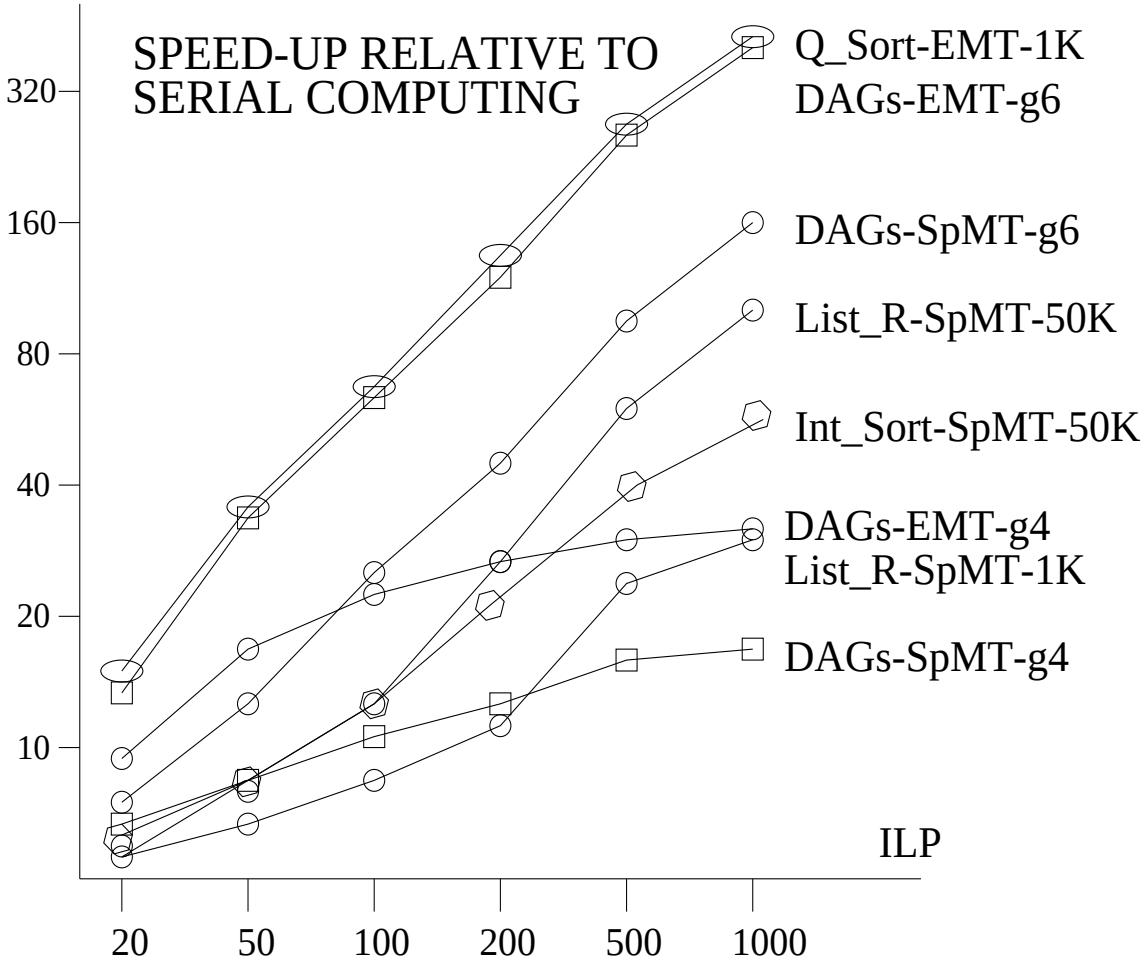
• High bandwidth memory architecture.
- Use hashing to partition the memory and avoid hot spots.
- Understood, BUT (needed) departure from mainstream practice.

• High bandwidth on-chip interconnects

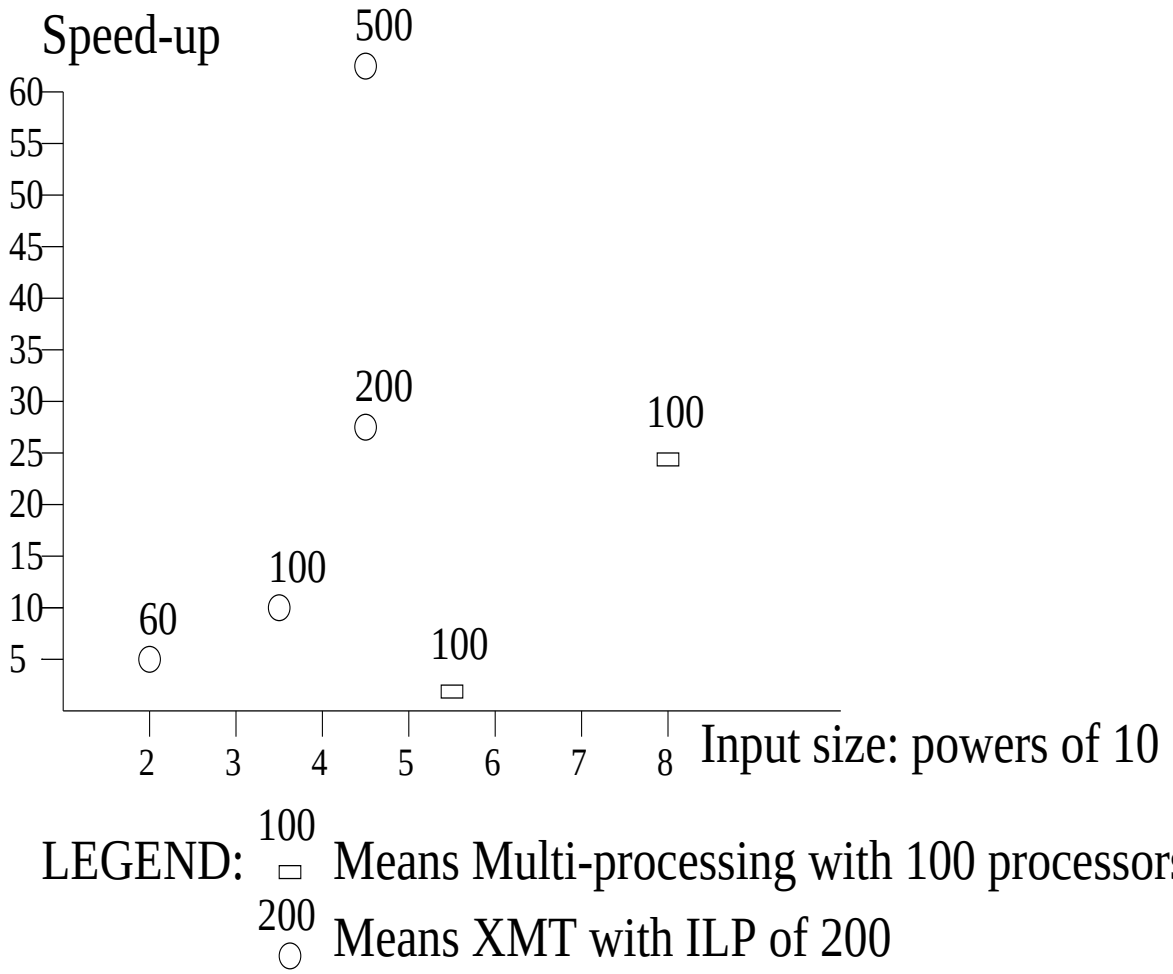
• Allow infrequent global synchronization (with IOS).
Attractive: lower energy.

• Couple with strong MTCU for serial code.

Empirical results



• **Parallel computing versus XMT**. List ranking results for the Intel Paragon reported in Sibeyn-97 versus XMT results.



- **Relative to serial computing**.

4 last transparencies

Conclusion

Objectives
- General-purpose computing
- Single task completion time

Predicaments in the past
- High overheads for managing parallelism in multi-chip multiprocessing. Dictated difficult "decomposition-first" parallel programming.
- Good returns on using on-chip growth for caches.

Now diminishing...
- Architecture techniques? Mined out. Popular micro-procs: max #instructions issued per clock–flat. Decade-long stagnation!

New, yet conservative approach

Architecture designed directly for the amounts of hardware that can fit on a single chip this decade. No scaling down; not incrementally upgrading decades-old architectures. Still backwards compatibility on software.

Outcomes speed-ups: 50-100.

Some applications (productivity):
- Molecular simulations (e.g., for drug discovery, protein

folding). Suppose 10^{14} steps need to be simulated. A rate of step per nanosecond could be possible. Exciting: the whole simulation takes a day instead of 1000 days with multi-chip multiprocessing.
- Hardware-enhanced software development kits (SDKs) for some application domains (e.g., graphics, desk-top data bases). Expert programmers will implement easy to use APIs. Such APIs alleviate barriers to entry to creative content producers who will generate greater demand.
- Applications are generally unlimited!
An "Iceberg Effect" in high-performance general-purpose computing systems: only a small fraction of the actual applications are visible at build-time.

Status

HW thread - completing synthesizable Verilog. On-chip interconnection network. Next: FPGA, cycle accurate simulator.
Architecture - much more to do. Nail down memory for MTCU.
Unending: 2005 papers on von-Neumann'46.
Overall power study, power options (short and long term) and implications - still ahead.
SW prefetching HW support

Compiler (+simulator) - used by class and applications. Some NESL type challenges.
Debugging - Only trivial serialization.
Programming - develop know-how and street smart. Example: how exactly to incorporate QRQW? Much to do.
Applications - OpenGL, VHDL. More

The WINTEL paradigm upgrade catch-22

1. Intel: but who will develop OS and SDKs/APIs?
2. Microsoft: who will build?
3. No action? performance improvement from VLSI only!

hurting both (and IT as a whole).
Prototype study needed: Applications, programmer's productivity, architecture. Successful? involve the other.

Documentation www.umiacs.umd.edu/ vishkin/XMT/+
First generation: SPAA'98, WAE'99 (special issue).
Second generation: MTEAC'00 at MICRO (MTEAC Best Paper Award), HIPS'01, TOCS'03 (special journal issue of SPAA'01), ISCAS'04.
End of semester. Material developed for parallel programming in Parallel Algorithmics, Spring 2005.
Optical interconnection idea: NSC'04, SPIE'04.

"Sermon" part of MFCS'04 talk: a proposed characteristic of "contribution"

You are told about a new technical idea and react: "but this is so obvious, I wonder how I missed it".
Problem: since hindsight is 20/20, how to reason about non-obviousness when it is not-so-obvious?

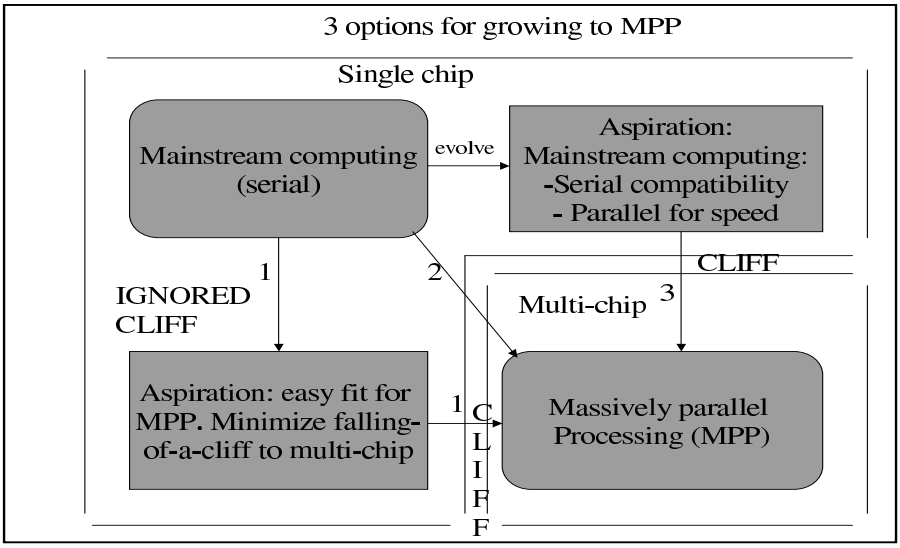
Found out recently: the US patent law practice provides an insight. That law protects inventions that meet three requirements: non-obviousness (most challenging to establish), utility, and novelty.
Legal definition of (non-)obviousness: A patent may not be obtained though the invention is not identically disclosed, if the differences relative to prior art are such that the subject matter as a whole would have been obvious at the time the invention was made to a person having ordinary skill in the art. But how is non-obvious demonstrated?
Using circumstantial evidence:
- experts stated disbelief in what the invention allows.
- unaddressed need.

"PRAM is unrealistic" statements showed to patent examiner that PRAM-On-Chip was non-obvious.

virtues of not-so-obvious non-obviousness

You can get a patent for it.
Simple. Many will be able to understand it.
Likely impact: may find its way to a textbook/encyclopedia, people may actually end up using it.
That is, more people may actually be interested in understanding it and... they will actually succeed.

Backup slides:



Compare Options 1-3

- MPP: Government big player
- Build on larger non-MPP market: 2 and 3. Not 1.
- Past record: Economics favored 2. Won (IBM, etc) even if not always best performance.
- Future: 1 versus 3. Issues:
- Which results in better final MPP?
- Is 1 economically viable?
- MPP funding fad: focus on 2nd falling –of-a-cliff in 1, ignoring the 1st

Experimental Methodology

- Simulator
  - SimpleScalar parameters for instruction latencies
  - 1, 4, 16, 64, 256 TCUs
- Configuration:
  - 8 TCUs per cluster
  - 8K L1 cache
  - banked shared L2 cache 1MB
- Programs rewritten in XMT
  - Speedups of parallel XMT program compared to best serial program
    - parallel applications: scalability to high levels
    - speedups for less parallel, irregular applications

First Application Set

Domain	Program	source
Scientific Computation	1.jacobi	
	2.tomcatv	SPEC95
Linear Algebra	3.mmult	Livermore Loops
	4.dot	Livermore Loop
Database	5.dbscan	[]
	6.dbtree	MySQL
Image processing	7.convolution	[]

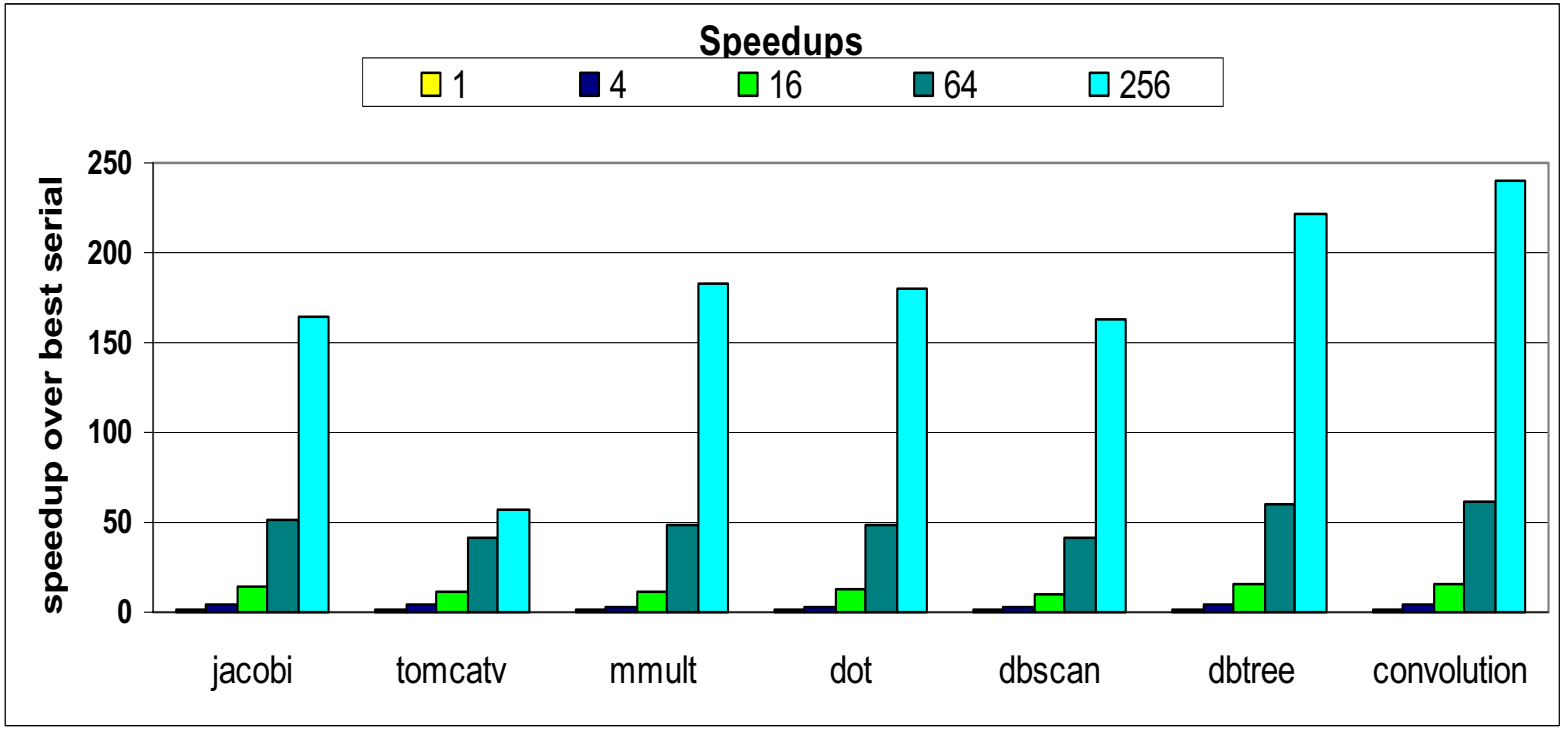
- Computation:
  - regular,
  - mostly array based,
  - limited synchronization needed

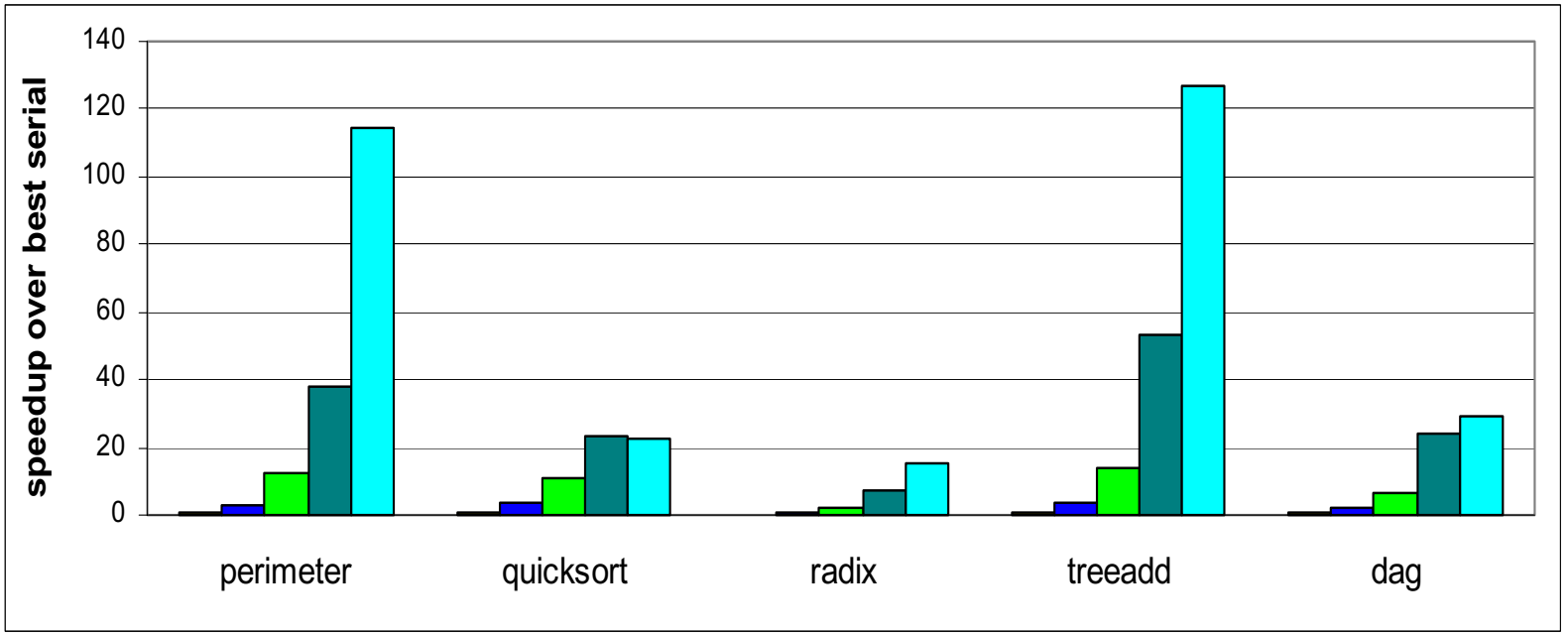
Second Application Set

Domain	Program	source
Sorting Algorithms	1.quicksort	
	2.radixsort	(SPLASH)
graph traversal	3.dag	
	4.treeadd	Olden
image processing	5.perimeter	Olden

- Computation:
  - irregular,
  - unpredictable
  - synchronization needed

Summary


Speedups
1
4
16
64
256
speedup over best serial
250
200
150
100
50
0
jacobi
tomcatv
mmult
dot
dbscan
dbtree
convolution


speedup over best serial
140
120
100
80
60
40
20
0
perimeter
quicksort
radix
treeadd
dag