PRAM on Chip
What to do with all this hardware? Could the PRAM-On-Chip architecture lead to upgrading the Wintel performance-to-productivity platform?

U. Vishkin, vishkin@umiacs.umd.edu.
Worked with 4 students, since 1997:

My strategic research objective, since 1979:
Seek to improve general-purpose single task completion time by parallelism.

Note 4: Reaxiomatization for better performance.

- Speed-up: X50-100 over serial for general-purpose applications. For: single task completion time.
Note 5: performance orders of magnitude better.

- Applications:
  - Simpler APIs for games+graphics yet good performance.
  - Molecular simulation, e.g., drug design. Aspire: Nanosecond per step; feasible to simulate many more steps.
  - General purpose: “iceberg effect”.
Note 6: Market/science/defense apps.

- Conclusion: elementary science; innovative timely technology; huge apps.
Note 7: Fascinating science. Competitive technology. Big money. Big Splash...

Note 8: much higher level of abstraction in XMT. Easier to program. Argue: still good performance.

Highlights

- Technological opportunity - bandwidth and latencies orders of magnitude better on chip.
  Note 1: Unprecedented bandwidth; better latencies.

- The PRAM (parallel random-access model) – the “ultimate” for scalable parallel thinking/programmability. BUT, was impractical for the 1990s multi-chips: overheads for managing parallelism too high.
  Note 2: Could not be done before.

- PRAM-on-Chip – a concrete virtual-PRAM design...
  NDA with Major Elec Des company led to prelim cost est.
  Note 3: Practical! Building is in D stage!

- Wait a minute: but how do you manage parallelism?
  Through reaxiomatization of von-Neumann’s (“mathematical machines”) 1946 design:
  - Control (program counter + stored program) of computer systems same as in the 1940s. New: upgraded.
  - “Billion transistor” chip, for: memory, interconnect and low overhead management of parallelism.

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

Premise: algorithmist states (or, does not hide..) what can be done concurrently.
Algorithmic knowledge-base 2nd only to serial algorithms. Simplest parallel model.

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) 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 par
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-Thread (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

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: \( e_0 \), \( e_1 \) and \( e_6 \) 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 multipopem and instructions.

Algorithm level (high-level program)

\[
\begin{align*}
\text{int } x &= 0; \\
\text{Spawn}(0, n) &\quad /\!\!\!/ \text{ Spawn } n \text{ threads; } S \text{ ranges } 0 \rightarrow n - 1 \star / \\
\{ \text{int } e &= 1; \\
\text{if } (B[S] == 1) &\quad \{ \text{PS}(x, e); \\
\text{D}[e] = A[S] \} \} \\
\text{if } (B[S] == 0) &\quad \{ \text{W}(x, e\text{-id}); \} \\
\text{D}[e] = -1 \} \\
\text{end} \\
\text{end}
\end{align*}
\]

Notes: (i) PS is defined next (think F&A). See results for \( e_0, e_2, e_6 \) and \( x \). (ii) Using C-style scoping, Join instructions are implicit.

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).
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.

Machine extensions to the von Neumann model

Program counter -> stored program:
– a remarkable Darwinistic success story.
Pragmatic: Extend rather than head-on competition.

If successful, Encyclopedia Britannica will need to update its Computer entry...

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

Snippet of a machine execution model

Given: a Spawn of \( \pi \) threads. The hardware determines which virtual thread to execute next in a distributed fashion.

The program of a thread control unit (TCU)
The Memory Wall

Concerns: 1) latency to main memory, 2) bandwidth to main memory.
Position papers: “the memory wall” (Wulf), “it’s 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 current practice.
- High bandwidth on-chip interconnects
- Allow infrequent global synchronization (with IOS). Attractive: lower energy.

Empirical results

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

<table>
<thead>
<tr>
<th>Input size: powers of 10</th>
<th>Speed-up</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>60</td>
</tr>
<tr>
<td>3</td>
<td>100</td>
</tr>
<tr>
<td>4</td>
<td>100</td>
</tr>
<tr>
<td>5</td>
<td>100</td>
</tr>
<tr>
<td>6</td>
<td>100</td>
</tr>
<tr>
<td>7</td>
<td>100</td>
</tr>
<tr>
<td>8</td>
<td>100</td>
</tr>
</tbody>
</table>

LEGEND: 100 = Means Multi-processing with 100 processors. 200 = Means XMT with ILP of 200

Relative to serial computing.

<table>
<thead>
<tr>
<th>SPEED-UP RELATIVE TO SERIAL COMPUTING</th>
</tr>
</thead>
<tbody>
<tr>
<td>Q_Sort-EMT-1K</td>
</tr>
<tr>
<td>DAGs-EMT-g6</td>
</tr>
<tr>
<td>DAGs-SpMT-g6</td>
</tr>
<tr>
<td>DAGs-SpMT-g4</td>
</tr>
<tr>
<td>DAGs-EMT-g4</td>
</tr>
<tr>
<td>List_R-SpMT-50K</td>
</tr>
<tr>
<td>Int_Sort-SpMT-50K</td>
</tr>
<tr>
<td>DAGs-EMT-g4</td>
</tr>
<tr>
<td>List_R-SpMT-1K</td>
</tr>
<tr>
<td>DAGs-SpMT-g4</td>
</tr>
</tbody>
</table>

4 last transparencies: much more data
Conclusion

Objectives
- General-purpose computing
- Single task completion time

Predicaments in the past
- Good returns on using on-chip growth for caches. Now diminishing...
- Architecture techniques. Mined out..

Outcomes in 2-5 years
(2-3 years) - Speed-up: > 30.
(3-5 years) - Speed-up: > 100.
Some applications (productivity):
- Molecular simulations (e.g., for drug design, 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; note: Intel’s Vtune, Sony PS2, etc., appear to lead in the opposite direction.
- 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.

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).
Needed prototype study: Applications, programmer's productivity, architecture. Successful? involve the other.

Documentation
See http://www.umiacs.umd.edu/~vishkin/XMT/
First generation: SPAA’98, WAE’98 (special issue).
Second generation: MTEAC’00 at MICRO (MTEAC Best Paper Award), HIPS’01, SPAA’01 (special issue, invited)

Backup slides:
Legend of the Lazy Performance Programmer

Current practice (some repetition):
- SP2 non trivial programming
- Intel’s Vtune
- Write your own memory manager, Game Developer, 2/2002
- Current parallel programming: decomposition-first. Programmability problems

The ideal: non-"programming" APIs. Many new products for games/graphics. Performance needed.

Backup slides:

How come that WE need somebody like you?
Even the world best race car builder needs to rely on ships to transport his/her cars. The “ship” here takes you from the 1946 strategic design point to the new one (buildability of a first design), and the explicit parallel programmability (theory and results). The “race car” are all the elegant static and dynamic optimizations, and the productivity enhancing SDKs/APIs. You need a ship (out-of-the-box mean of transportation) to cross water (the paradigm upgrade) to put your race car to use

An EDA company wants to make some money; what’s the big deal?

[CS-99] Parallel Computer Architecture asserts:
- the PRAM algorithmic model is DESIRABLE, but NOT feasible (for the 1990s type multi-chip MPPs)
- a virtual-PRAM machine (that can look to the programmer like the PRAM) becoming FEASIBLE will be a BREAKTHROUGH for general-purpose computing
### First Application Set

<table>
<thead>
<tr>
<th>Domain</th>
<th>Program</th>
<th>source</th>
</tr>
</thead>
<tbody>
<tr>
<td>Scientific Computation</td>
<td>1.jacobi</td>
<td></td>
</tr>
<tr>
<td></td>
<td>2.tomcatv</td>
<td>SPEC95</td>
</tr>
<tr>
<td>Linear Algebra</td>
<td>3.mmult</td>
<td>Livermore Loops</td>
</tr>
<tr>
<td></td>
<td>4.dot</td>
<td>Livermore Loop</td>
</tr>
<tr>
<td>Database</td>
<td>5.dbscan</td>
<td></td>
</tr>
<tr>
<td></td>
<td>6.dbtree</td>
<td>MySQL</td>
</tr>
<tr>
<td>Image processing</td>
<td>7.convolution</td>
<td></td>
</tr>
</tbody>
</table>

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

### Second Application Set

<table>
<thead>
<tr>
<th>Domain</th>
<th>Program</th>
<th>source</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sorting Algorithms</td>
<td>1.quicksort</td>
<td></td>
</tr>
<tr>
<td></td>
<td>2.radixsort</td>
<td>(SPLASH)</td>
</tr>
<tr>
<td>graph traversal</td>
<td>3.dag</td>
<td></td>
</tr>
<tr>
<td></td>
<td>4.treeadd</td>
<td>Olden</td>
</tr>
<tr>
<td>image processing</td>
<td>5.perimeter</td>
<td>Olden</td>
</tr>
</tbody>
</table>

- **Computation:**
  - irregular,
  - unpredictable
  - synchronization needed

### Summary

#### Speedups

- **jacobi**
- **tomcatv**
- **mmult**
- **dot**
- **dbscan**
- **dbtree**
- **convolution**

- **perimeter**
- **quicksort**
- **radix**
- **treeadd**
- **dag**

### 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