PRAM on Chip: Advancing from Parallel Algorithms to a Parallel System

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

I believe that the approach described in the current talk leads to answers for most of the questions raised in the two talks by Intel and IBM people that we just heard. I realize that claiming to offer the solution sought by the previous two talks is a bit humbling. Still, this is exactly what I am doing.
Parallel Algorithms Based on Parallel Architectures, or Vice Versa

Focal point of my work since 1979: seek to improve general-purpose single task completion time by parallelism.

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 understood and developed 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
Current post-theory chapter: PRAM-On-Chip framework that extends from high level APIs to VLSI implementation.
Sanity Check

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...

Specs need to account for:
- the programming productivity problem: run-time and development time.
- applications.
The parallel programming problem

Currently, parallel programming is simply too difficult, which makes it unacceptable for many current objectives.

Tribal lore, parallel programming profs, DARPA HPCS Development Time study (2004-2008):
Parallel algorithms and programming for parallelism is easy. What is difficult is the programming/tuning for performance that comes after that.
In a nutshell, we try to avoid the latter stage.
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? Asked at Intel 4/05
Hopefully not:
- Justin Rattner, CTO, Intel, Electronic News, March 13, 2006: “It is better for Intel to get involved in this now so when we get to the point of having 10s and 100s of cores we will have the answers. There is a lot of architecture work to do to release the potential, and we will not bring these products to market until we have good solutions to the programming problem.”

Still missing by many: at the heart of every program there is .. an algorithm. Every computer science program recognizes that: see how programming is taught, courses on algorithms and data structures, etc. Still, many speak about the mechanics of MPI, OpenMP, or multithreading programming without giving enough attention to the underlying intellectual component. What added-value will PC/workstation vendors provide with
more than a few cores? Currently they don’t seem to have good ideas on how to harness multi-cores toward faster single completion time. Should they be losing sleep over that?!
**Things I Want to Say Today**

**Horizon** 1. An overall environment from high level languages to a complete system: *not fall behind the serial world on anything; significantly ahead on performance* 2. Prototype applications that benefit. 

**Status** PRAM-On-Chip is becoming ready for building, programming, teaching and developing applications:

**Building** PRAM-On-Chip is now only a matter of opportunity. Based on synthesizable Verilog, etc., we will know what to do.

**Programming** Developed methodology for advancing from PRAM algorithms to efficient programs and significant understanding for backwards compatibility with (standard) higher level programming interfaces.

Namely: Used to apologize since PRAM was too high-level (did not represent actual performance of what could be built). Becoming more appropriate to apologize for PRAM being too low-level and plan around it.

**Teaching** Developed “kit” for teaching the programming methodology including a tool chain, comprising compiler and cycle-accurate simulator.
**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:
  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

Observer:

(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!

DC&VC expect “solid and QUICK”; but: Calculus-like basics.
In reality

Only multi-chip multi-processors were available:
Take home: What’s buildable was not programmable and vice versa
Faster clock: 2x/18months; then stagnation since 2003

Finally, good news: # transistors/chip
1980: 29K. 2006: 1B. Factor: 30,000X!

My Take on the Programming Problem
Enough bandwidth: ideal (PRAM) programming model.
Limited bandwidth: programming difficulties.

PRAM-On-Chip

Newer insight: lower overheads are possible with on-chip multi-processors. PRAM On Chip relies on ability to get enough bandwidth and much better latencies on chip.
Corollary: Option 1 is alive again!
Roll-back clock to 1991-3: The conclusion is clear. Just teach the PRAM chapter.
2006: what has changed besides not having the chapter in the text?!
The case only became stronger:
• Migration path from serial. On-chip implementation allows down scaling: (i) competitive performance on serial code; and (ii) gains over serial even if small amount of parallelism are available.
• The productivity argument (next).
• Scaling to larger problem size. New (3/2006) technology ideas for significant improvement in off-chip bandwidth.
The productivity picture

- Performance programming:
  Hochstein-Basili for DARPA development-time study. Multiply sparse matrix by vector. 4th MPI programming assignment, Prof. Gilbert, UCSB. 2nd PRAM-On-Chip programming assignment, UV, UMD. On average: less than 50% development time.

- 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 - Olano and students, work in progress
  VHDL - Gu-Vishkin, JEC’06, 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 for compiler extraction of parallelism from serial code. Detours it.
PERFORMANCE PROGRAMMING & ITS PRODUCTIVITY

Basic Algorithm (sometimes informal)

Add data-structures (for serial algorithm)

Serial program (C)

Add parallel data-structures (for PRAM-like algorithm)

Parallel program (XMT-C)

Low overheads!

Standard Computer

Parallel Programming (Culler-Singh)

Decomposition

Assignment

Orchestration

Mapping

Parallel computer

XMT architecture (Simulator)

- 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

Application programmer’s interfaces (APIs) (OpenGL, VHDL/Verilog, Matlab)

Serial program (C)

Compiler

Parallel program (XMT-C)

XMT architecture (Simulator)

Standard Computer

Automatic? Yes

Parallel Programming (Culler-Singh)

Decomposition

Assignment

Orchestration

Mapping

Parallel computer

Automatic? Maybe
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”.
2006 - 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, then 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 $\rightarrow$ architecture $\rightarrow$ 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

```
Parallel states from a Spawn to a Join and serial states

```

```
“Standard” PRAM/Vector

```

```
“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).
Claim: Teaching Parallel Algorithms and Programming is on Par with Serial

Milestones in demonstrating claim:
1. Programming assignments in Parallel Algorithmics, UMD-Spring06: parallel MATVEC, general deterministic (Bitonic) sort, breadth first search on graphs, sample sort and parallel connectivity. Since 1st class on serial algorithms/programming typically does not require more demanding assignments, suggests: PRAM theory coupled with PRAM-On-Chip programming are on par with serial algorithms and programming.
2. Tutorial paper to augment a PRAM algorithms text with a proposed methodology for gradually evolving a PRAM algorithm into an effective PRAM-On-Chip program. Through proper modeling of stages in this process, abstract algorithmic thinking and reasoning gradually give in to growing level of detail. (Extends the [SV-82] “work-depth” methodology for description of parallel algorithms.) Coupled with a tool-chain: allows such teaching elsewhere.
Problem design

High-level Work-Depth Description
"How to think in parallel"
Sequence of steps
Each step: Set of concurrent operations
Informal Work/Depth complexity

Work-Depth Model
Sequence of steps
Each step: Sequence of concurrent operations
Work/Depth complexity

PRAM Model
Allocate/schedule processors
Each step: Sequence of p concurrent operations
No. of parallel steps

Legend:
→ original "parallel thinking to PRAM" methodology
→ proposed "parallel thinking to PRAM-on-chip program" methodology
→ proposed shortcut for PRAM-on-chip programmers
Issues: (1) Length of sequence of round trips to memory; (2) nesting of spawns, and (3) QRQW.

Some BFS Example conclusions: (1) Describe using simple nesting: for each vertex of a layer, for each of its edges...; (2) since only single-spawns can be nested (reason beyond current presentation), for some cases (generally smaller degrees) nesting single-spawns works best, while for others flatening works better; (3) let compiler derive best implementation.
**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 $C$.

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>D</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td></td>
</tr>
</tbody>
</table>

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)**

```c
int x = 0;
Spawn(0, n) /* Spawn n threads; $ ranges 0 \rightarrow 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**, \(PS \ R_i \ R_j\), has an inseparable ("atomic") outcome: (i) Store \(R_i + R_j\) in \(R_i\), and (ii) store original value of \(R_i\) in \(R_j\).

Several successive PS instructions define a **multiple-PS instruction**. E.g., the sequence of \(k\) instructions:

\[
PS \ R_1\ R_2; \ PS \ R_1\ R_3; \ldots; \ PS \ R_1\ R(k + 1)
\]

performs the prefix-sum of the **base** \(R_1\) and the **elements** \(R_2, R_3, \ldots, R(k + 1)\) to get:

\[
R_2 = R_1;
R_3 = R_1 + R_2; \ldots; \ R(k + 1) = R_1 + \ldots + R_k;
R_1 = R_1 + \ldots + R(k + 1).
\]

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.

When PC1 hits Spawn, a spawn unit broadcasts 1000000 and the code to PC1, PC 2, PC1000 on a designated bus
**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)**

- **Start**
  - $\$: := TCU-ID
  - Use PS to get new $\$
  - **Is $\$ > n$ ?**
    - Yes: **Done**
    - No: **Execute Thread $\$**

Next: use “Billion transistor” chip, for: memory, interconnect and low overhead management of parallelism.
Status and Plans

Hardware Thread: - 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: 6-24 clocks, depending on interconnection network and its clock
- good news: bandwidth indeed not a problem
- completed synthesizable Verilog
- completed cycle accurate simulator; under testing
- on track to FPGA prototyping by Fall’06

- toolchain (compiler+simulator) in use by parallel algorithms class
- major upgrade to a “professional” compiler; first draft 5/06
- together with extensive documentation (tutorial and manual for XMT-C, tutorial for advancing from PRAM algorithm to an XMT-C program) will allow portable kit for “mission partners”.
- class programming homework on par with serial algorithms class. Given, or will be given: multiplication of
sparse matrix by vector, general sorting, randomized sorting, integer sorting, selection, BFS, log-time connectivity. Consistent with claim that PRAM is an alternative to serial RAM. **Who else in parallel computing can say that?**

- Perspective on challenges: leveling-off algorithm maturity, including: SW prefetching & HW support; programming; compiler; power; debugging; applications;
Relative Maturity
- Prior to XMT  - Now  - Future work

Algorithms  Programming, System SW  Architecture feasibility
**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

SPEED-UP RELATIVE TO SERIAL COMPUTING

- Q_Sort-EMT-1K
- DAGs-EMT-g6
- DAGs-SpMT-g6
- List_R-SpMT-50K
- Int_Sort-SpMT-50K
- DAGs-EMT-g4
- List_R-SpMT-1K
- DAGs-SpMT-g4
- **Parallel computing versus XMT.** List ranking results for the Intel Paragon reported in Sibeyn-97 versus XMT results.

![Graph showing speed-up vs input size for parallel computing and XMT results.](image)

**Legend:**
- □ Means Multi-processing with 100 processors.
- ○ Means XMT with ILP of 200

Input size: powers of 10
• Relative to serial computing.

4 last transparencies
Conclusion

Objectives
- General-purpose computing
- Single task completion time

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

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.

**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/+  
Second generation: MTEAC’00 at MICRO (MTEAC Best Paper Award), HIPS’01, TOCS’03 (special journal issue of SPAA’01), ISCAS’04.

Backup slides:
3 options for growing to MPP

Single chip

Mainstream computing (serial)

Aspiration: easy fit for MPP, Minimize falling-of-a-cliff to multi-chip

Aspiration: Mainstream computing:
- Serial compatibility
- Parallel for speed

Massively parallel Processing (MPP)

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

<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>image processing</td>
<td>4. treeadd</td>
<td>Olden</td>
</tr>
<tr>
<td></td>
<td>5. perimeter</td>
<td>Olden</td>
</tr>
</tbody>
</table>

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

Speedups

<table>
<thead>
<tr>
<th>Speedups</th>
<th>1</th>
<th>4</th>
<th>16</th>
<th>64</th>
<th>256</th>
</tr>
</thead>
</table>

![Graph showing speedups for various programs and sizes](image)

- Programs: jacobi, tomcatv, mmult, dot, dbscan, dbtree, convolution, perimeter, quicksort, radix, treeadd, dag

- Comparison with best serial performance

- Speedup over best serial shown for different sizes (1, 4, 16, 64, 256)