From Algorithm Parallelism to Instruction-Level Parallelism: An Encode-Decode Chain Using Prefix-Sum

Uzi Vishkin

2nd title:

Instruction-Level Parallel Processing: Why Inertia and Progress Need Not Be at Odds

Reality: Underused technologies

- "Parallel software crisis" -
too few parallel machines actually used for faster single-task computation:
hardly any software for the non-specialist.

- Huge parallel algorithmic knowledge-base that waits to be implemented.

Possible Explanation: Fundamental Gap

- Programming models - extensive "fine-grained" use of shared variables

- Machines - cellular-MIMD ("coarse-grained")

Proposed strategy

Stay on the fine-grained side.

New architecture: Recruit uniprocessor instruction-level parallelism (ILP).

ILP: main stream technology (AMD,DEC,HP, Intel,MIPS,PowerPC,SUN, who not?!).
[HP]: could reach hundreds by 2000; however, in standard code extractable parallelism is limited.

Contributions

(Theoretical) vision for new architecture
Add one unit-time instruction to standard instruction code: **multiple prefix-sums**

Insight There exists an integrative design where:
algorithm parallelism can fully translate into ILP.

New hardware tool Prefix-sum functional unit for degenerate multiple-PS.
"Within-budget" of k-fold ILP systems for cost and time - "relative complexity".
Numerous known uses. (Fetch-and-Add NYU-Ultra, V-83, Scan - Blelloch).
New application: **register file design**.

Individual prefix-sum $PS(A,B)$. Atomic:
(i) $A := A+B$, and
(ii) $B :=$ "the original value of A".

multiple-PS instruction. The sequence:
$PS(A,B_1)$
$PS(A,B_2)$
...
$PS(A,B_k)$
computes the prefix-sum of the base A and $B_1,B_2,...,B_k$ into $B_1,B_2,...,B_k$ and A. Namely:
$B_1 = A$
$B_2 = A + B_1$
$B_3 = A + B_1 + B_2$
...
$B_k = A + B_1 + B_2 + ... + B_{k-1}$
$A = A + B_1 + B_2 + ... + B_{k-1} + B_k$
Target assumption: unit-time for some k.

New insight: provable completeness for parallelism

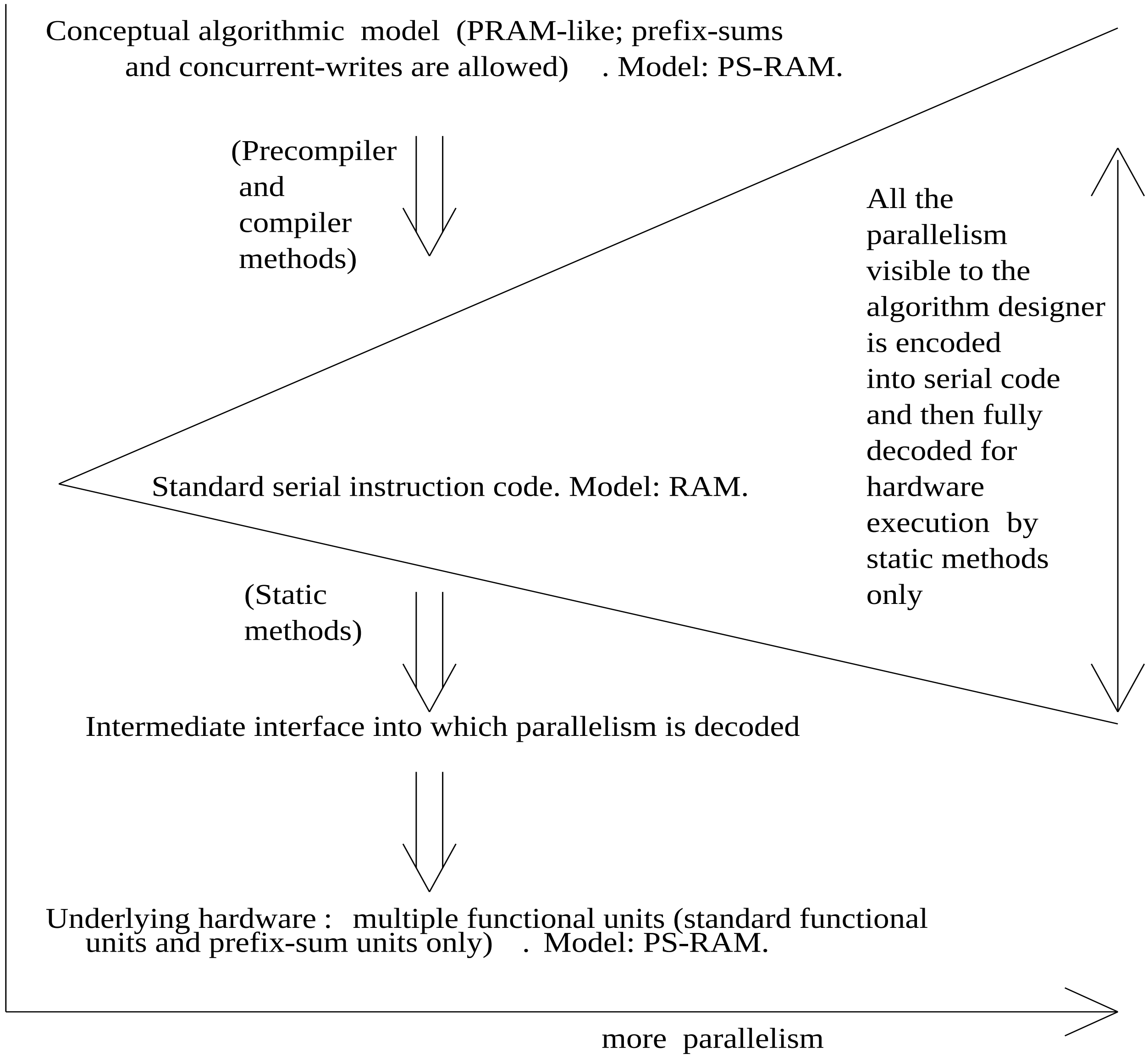

Figure 1: Encode-Decode of PS-RAM parallelism

Notice

- For modeling **parallel** algorithms and underlying hardware:
sequence of (sequences of) concurrent instructions.
Example (work-time):
$I_{1,1}, I_{1,2}, .., I_{1,k_1}$
$I_{2,1}, I_{2,2}, .., I_{2,k_2}$
...
$I_{t,1}, I_{t,2}, .., I_{t,k_t}$

- For standard serial-code: replace $PS(A, B)$ by $A = A + B$; $B = A - B$.

- Compiler resolution of memory disambiguation ("aliasing").

Key insight in the proof

Prefix-sum gatekeeper if concurrent-writes happen. Possible aborts.
Prefix-sum gatekeeper if concurrent-writes do not happen (!) but cannot be ruled out at compile time. If at run-time individual-PS only: simpler run-time implementation.

Example
For $i := 1$ to n pardo $\{A[P[i]] := B[i]\}$
Problem: how to get parallelism if **do** replaces **pardo**?

Proof idea
Initialize: For $i := 1$ to n $\{G[i] := 0; H[i] := 1\}$
For $i := 1$ to n $\{PS(G[P[i]], H[i])\}$
For $i := 1$ to n {if $H[i] = 0$ then $A[P[i]] := B[i]\}$

Note: The serial loops are independent (and, therefore, parallel!).
Simplifications Often gatekeepers are not

needed in software; and when in software often not needed in hardware.

Empirical result for test example

1st code: a standard RAM code for BFS.
2nd code: RAM code implementing a parallel version of BFS. **Note** same total number of operations.

Generating inputs: Knuth's GraphBase.

Lam-Wilson abstract ILP-machine models.
(i) CD-MF. Assumes perfect memory disambiguation, perfect register renaming, full control dependence analysis, ability to follow unlimited multiple flows of control.
(ii) SP-CD-MF. Also, ability to predict branches and speculatively execute instructions accordingly.

Measured Amount of parallelism: (total number of operations)/(parallel time)

Machine model	Nodes & Edges	First code	Second code	Factor Improved
(i)	100 & 750	26.59	216.74	8.15
	500 & 2500	20.77	248.24	11.95
(ii)	100 & 750	41.43	351.81	8.49
	500 & 2500	31.65	443.29	14.01

For 1st code: Lam-Wilson's simulator.
For 2nd code: enhancement to execute multiple-PS in one round.

Hardware implementation of small-integer multiple-PS

$PS\ R_0\ R_1,\ PS\ R_0\ R_2,\ ..\ ,\ PS\ R_0\ R_k$
is **small-integer** if R_1 through R_k have only small integers; e.g., between 0 and 3. No limits for R_0. If hardware can handle $h \le k$ PS, then do h at a time.

1. Compute "base-zero" prefix-sum. Compute suffix-sum.
2. Find the largest output element
$z_h = R_0 + (R_1 + R_2 + .. + R_h)$.

Goal Compute every output PS, by **few-bit** arithmetic. Let i be the most significant bit in which R_0 and z_h differ (in binary). Let w be $R_{0,31}, ..., R_{0,w+1}, 1, 0, 0, .., 0$.

Observation If output PS is $< w$, it is a few-bit add to R_0. If output PS is $\geq w$, it is a few-bit subtract from z_h.

3. Add each base-zero PS to R_0.
4. Subtract each suffix-sum from z_h.

Relative complexity: Relatively little, beyond adds and small-integer base-zero PS.

Two ideas for high-bandwidth fast memory

Register file organizations

- Off-line solution, based on edge-coloring of bipartite graphs. IPPS'97.

- Dynamic solution. Based on PS. SPAA'97.

Both improve on the classic context-matrix Cydra-5 solution.

Conflict-free rule: Values written or read simultaneously must reside in distinct modules.

Cycle #	1	2	3	4	5
	p		p	s	t
	q		q		u
	r		r		
	s				
	t				
	u				

Figure 2: Input to dynamic solution: instructions (values) allocated to cycle 1 together with their next use.

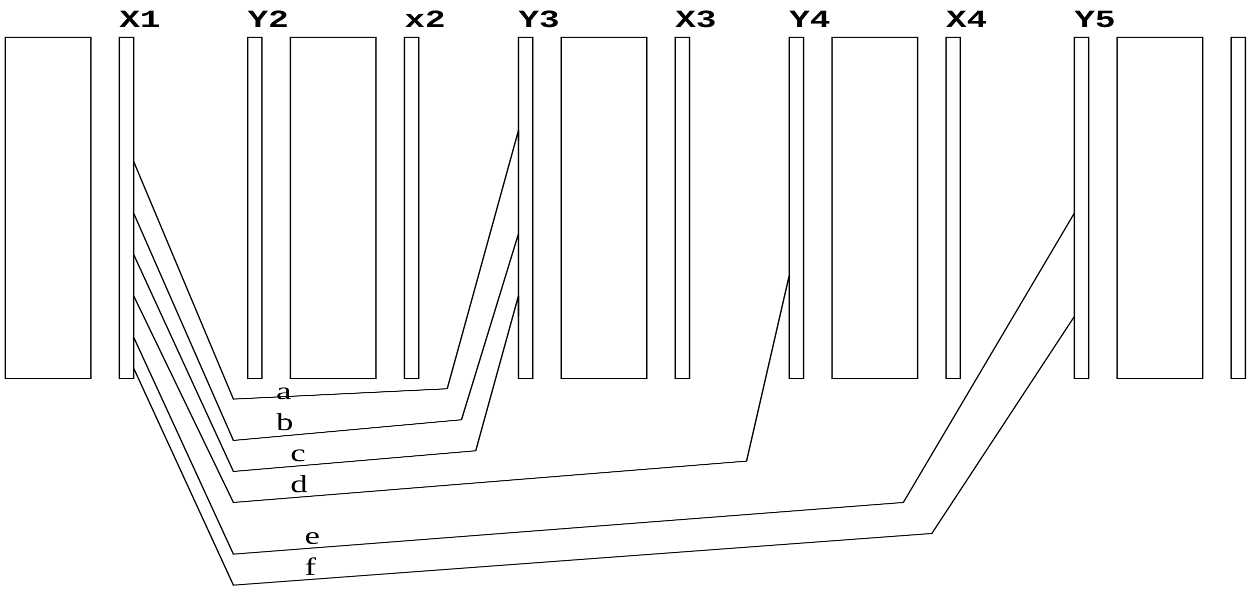

Figure 3: Illustration of input

Cycle #	1	2	3	4	5
	store p in R6		Fetch p from R6	Fetch s from R7	Fetch t from R1
	store q in R7		Fetch q from R7		Fetch u from R2
	store r in R8		Fetch r from R8		
	store s in R7				
	store t in R1				
	store u in R2				

Figure 4: Output of dynamic solution

Discussion

- **Compatible** with existing (serial) code.

- **Circumvents gap:** new architecture based on utilizing ILP.

- Takes advantage of the **ever increasing number of transistors** on a chip (10,000 over 20 years), **alleviating dead-ends** reached by current approaches to ILP.

- Critical long-term technology impact: once parallelism utilization on a chip is achieved, it is here to **stay as a competitive edge** (as technology keeps improving other aspects).

- **Paradigm shift** into "parallel computing on a chip".

- **Programmability:** builds on the theory of parallel algorithms, whose richness has no serious competition.

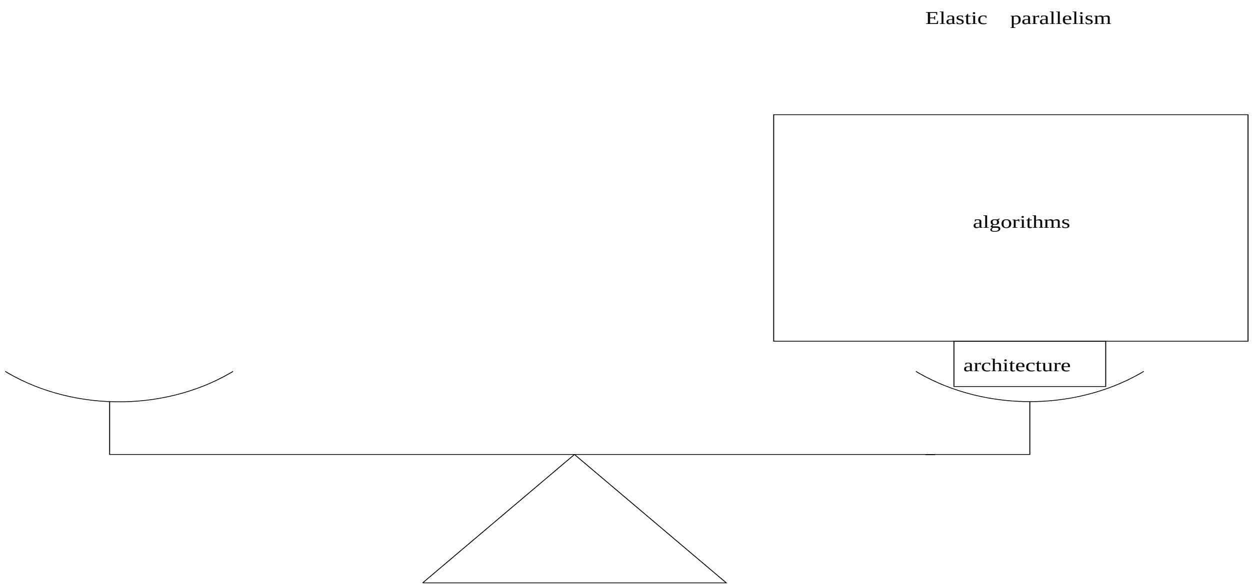

Figure 5: Advantage of new approach

- Recognizes uniprocessors (or unichips), as **pivot for the future of multi-processors**.

- Break aways from the **HPC inertia**, offering an evolutionary agenda and **desktop relevance**.

- **Algorithm-centric** approach. Programmers' time: the critical resource. One machine - many programmers; why do hardwarecentric approaches predominate?!

- **Myth** ILP is limiting. However:
Prediction 1B transistors on a chip within a decade (decent CPU's with 30K transistors).

Slogan-level

The "von-Neumann Bottleneck". Backus (1977 Turing Award Lecture).

• "I propose to call this tube ("the connecting tube that transmits a single word between the CPU and the store", UV) the von Neumann bottleneck".

• "The assignment statement is the von Neumann bottleneck of programming languages and keeps us thinking in word-at-a-time terms in much the same way the computer's bottleneck does".

Does the approach annul "von-Neumann bottleneck" expectation? Using integrative redefinition of the von-Neumann architecture, **algorithm parallelism fully translates** into instruction-level parallelism.