# Two Techniques for Reconciling Algorithm Parallelism with Memory Constraints

U. Vishkin

The latency constraint:
number of clock cycles to reach main memory is increasing.

Suggestion [V94]: use parallel algorithms for prefetching.

The bandwidth constraint:
bandwidth to main memory is limited.

This paper:
some solutions.

## More motivation

Computer architecture trends [HP03]:
“Taking advantage of parallelism” only addition to the very short list of ”quantitative principle of computer design” since [HP90].
(The list:
- Make the common case fast/Amdahl’s Law.
- The CPU performance equation.
- Locality.)

Static and dynamic methods used to extract parallelism from code.
Problems:
1. Limited parallelism.
2. [BGK96]: Limited usefulness of prefetching, speculation, out-of-order execution, multithreading due to the bandwidth constraint.

What else can be done?

## Current paper revisits [V94]

Use parallel algorithms to enhance serial implementation

Very simple observation in [V94]:
Consider: serial emulation of a parallel algorithm. *Just before a round of the parallel algorithm, the whole list of memory addresses needed during this round is readily available; so start (pre)fetching all these addresses from slower memories at this time*.
Good news: great for the latency constraints.
Bad news: can be hurt by bandwidth constraints if between two visits to datum, the program visits more data than fits in the cache.

Current paper: provide optimizations to limit adverse effects of bandwidth constraints on [V94].

## The wavefront blocking technique

A parallel (PRAM) algorithm usually gives you in each step ALL the operations that can be done next in parallel.
The objective here is to do as much computing as possible on already cached data.

Guided by the size of the cache, the execution is designed to perform only a subset ("block") of those operations which can be done in parallel.
The execution would then proceed to "enabled" operations of subsequent steps of the parallel algorithm.
That is: at each phase of the execution a certain "parallel wave front" of operations is executed.
Then, the execution goes "back in time" in the parallel algorithm and proceeds to the next subset which fits in the cache, and so on.

The examples we have use "regular" data structures; e.g., two-dimensional arrays for matrix multiplication.

Example 1: Longest common subsequence (LCS)
Input: Two sequences $X[1..n]$ and $Y[1..m]$.
Dynamic programming: 2-D array $A[1..n; 1..m]$. $A(i, j)$ - length of the common subsequence of the sequences $X[1..i]$ and $Y[1..j]$.
A parallel algorithm: Advance the "wave front" of the computation from one ascending diagonal to the next. Ascending diagonal $k$ comprises all entries $(i, j)$, where $i + j = k$.
The value of $A(i, j)$ depends on 5 values:
$A(i-1, j)$ and $A(i, j-1)$ (both from diagonal $k-1$);
$A(i-1, j-1)$ (from diagonal $k-2$); $X(i)$ and $Y(j)$.
Case 1: the diagonal, $X$ and $Y$, all fit into cache. Easy.
Case 2: only a diagonal of size $d$ can be accommodated:
Run algorithm first for $X[1..d]$ and the original $Y[1..m]$.
Using prefetching refine $Y$, so that only $d$ of its values are needed at any given point in time.

Run time: dominated by time to access the cache for all data. Data (almost) always be ready in cache.

# Example 2: Matrix Multiplication

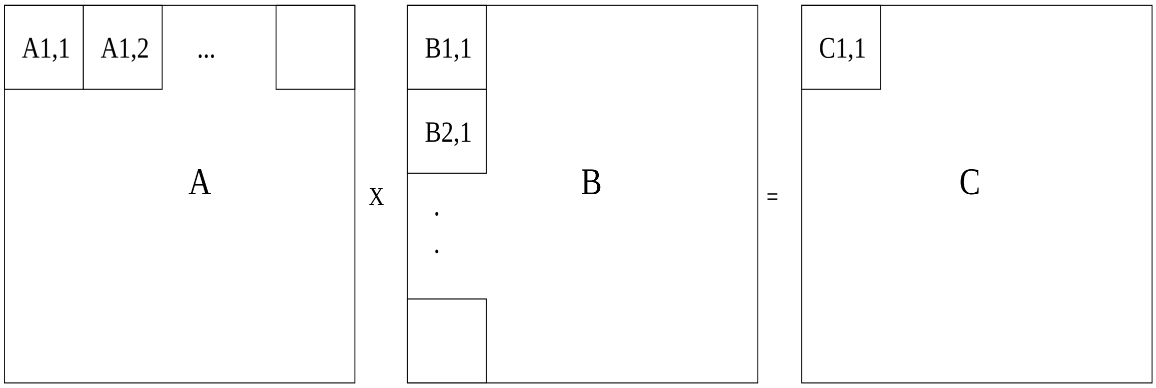


Figure 1: The submatrices

Let size of cache be $3t^2$; $n$ is divisible by $t$.
Partition each of the 3 matrices into $n^2/t^2$ square submatrices. Matrix multiplication is done in $n^2/t^2$ iterations. Each computes a different submatrix of $C$. Each iteration has $n/t$ subiterations: multiply submatrix of $A$ with a submatrix of $B$.

The data traffic of each iteration is at most $t^2(n/t) = nt$. With $n^2/t^2$ iterations, total traffic is $n^3/t$. Using current figures for bandwidth relative to processor speed: processing of $O(n^3)$ operation by the CPU will dominate the overall computation time.
Computational approach from [ACS90]. Different use.

## The not-to-cache label technique

Reasonable objective for cache use: store just used data and instructions till their next reuse.

Such an objective could explain policies, such as LRU for cache replacement. Namely, assuming locality, use an LRU priority for whom to replace.

HOWEVER:
- Suppose that some data is accessed only once.
- Suppose we can figure out which prior to the execution.

Suggestion: mark this data as "not to be cached". This frees the execution to use the cache where it can make a difference.

Example domain: graph algorithms.
The key property: each edge of the graph is accessed at most once.

Concrete examples:
- breadth-first search (BFS)
- Dijkstra's shortest-path algorithm, and

- a standard search procedure for directed acyclic graphs (DAGs).

In these examples, “irregular” data structures (e.g., lists and pointers) are often used.

## Note

- A compiler can do it. (Front end + back end.) Much easier from PRAM-on-chip type code, using the prefix-sum construct.

- Lots of literature on coping with memory constraints. Some reached homework assignments for undergraduate courses.

- This is a position paper. So, please go and find more techniques and examples and develop the compiler side.

## Original motivation for this work

General-purpose parallel computing is in a state of limbo.
- Parallel machines have not been cost-effective for programming. Is it fair to say that programmers try to avoid them?
- Easy-to-program programming and algorithms models (e.g., the PRAM) provided great virtual parallel scalability; unfortunately, it was not possible in the 1990s to build machines that efficiently support these models; the overheads for managing the parallelism provided by these models were too high.

## The PRAM-on-chip project at UMD.

PRAM (parallel random-access model) algorithmics: *knowledge-base 2nd only to serial algorithms*.

Under **Potential Breakthroughs**, in Culler and Singh, Parallel Computer Architecture, 1999: "... breakthrough may come from architecture ... that is, to truly design a machine that can look to the programmer like a PRAM".

**Unprecedented on-chip opportunities:**
- Amount of Hardware "Billion transistor chip era".
- Low latencies. Even more important,
- High bandwidth BUT: need to keep bandwidth use under control, the further from the processors (on- and off-chip) one gets.

This motivated the current work. However, it did not make sense to hide the relevance of this work to serial computing.

## A “Memory Wall” perspective

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.

PRAM-on-chip does better on both accounts:
- uses more the high bandwidth to (on-chip) 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.

Recall “independence of order semantics (IOS)”.
Example: breadth-first search (BFS).

The basic no-busy-wait finite-state-machine (NBW FSM) principle underlying PRAM-on-chip is like replacing a flat tire while continues to drive at high speed

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