Two Techniques for Reconciling Algorithm Parallelism with Memory Constraints (Extended Abstract)

Uzi Vishkin*
University of Maryland
vishkin@umiacs.umd.edu

ABSTRACT

The utility of algorithm parallelism for coping with increased processor to memory latencies using "latency hiding" is part of the folklore of parallel computing. Latency hiding techniques increase the traffic to memory and therefore may "hit another wall": limited bandwidth to memory. The current paper attempts to stimulate research in the following general direction: show that algorithm parallelism need not conflict with limited bandwidth.

A general technique for using parallel algorithms to enhance serial implementation in the face of processor-memory latency problems is revisited. Two techniques for alleviating memory bandwidth constraints are presented. Both techniques can be incorporated in a compiler.

There is often considerable parallelism in many of the algorithms which are known as useful serial algorithms. Interestingly enough, all the examples provided for the use of the two techniques come from such serial algorithms.

Categories and Subject Descriptors

C.0 [**Computer Systems Organization**]: General

General Terms

Algorithms, Management

Keywords

Parallel algorithms, memory systems constraints, prefetching

1. INTRODUCTION

Performance growth for computer systems must take into account constraints on computer memory systems. Unfortunately, computer memory systems are becoming increasingly constrained. The two main concerns (see [WM95] and [S96]) have been: (i) the increasing (counting processor clock cycles) latency to main memory, (ii) the limited bandwidth to main memory. Known optimizations to bridging the processor-memory latency gap include prefetching, speculation, out-of-order execution, and multithreading. However, some of these optimizations turned out not to be very effective due to processor-memory bandwidth constraints [BGK96].

The current paper *revisits a rather general technique* proposed in [V94] for using parallel algorithms to enhance implementation on serial machines. The main concrete observation of that paper is very simple. Consider the serial emulation of a parallel algorithm. *Just before the serial emulation of a round of the parallel algorithm begins, the whole list of memory addresses needed during this round is readily available; and, we can start fetching all these addresses from slower memories at this time,* so that these data will be ready at the fast memories when requested by the CPU. The effectiveness of this technique for the purpose of alleviating the latency problem is hurt by bandwidth constraints, similar to the known optimizations mentioned above. The key purpose of the current paper is to provide optimizations to significantly limit the adverse effects of *bandwidth constraints* on the technique of [V94].

In this paper, we discuss two techniques. Both techniques cope with a situation where between two visits of the same datum, a program "recycles" though more data than can fit into the cache. Sometimes this happens due to a high rate of parallelism in the underlying algorithm. Such massive hauling of data in and out of the cache could make the overall performance vulnerable to the limited bandwidth to memory. A preview of the two techniques follows.

1.0.0.1 The wavefront blocking technique.

The technique is based on using a degree of freedom provided by the parallelism of the algorithm. Guided by the size of the cache, the execution is designed to perform only a subset ("block") of the operations that the algorithm can do in parallel ("the parallel wave front") at a time, before proceeding to the next subset. The objective is to do as much computing as possible on already cached data. The examples we have for the wavefront blocking technique use "regular" data structures, such as two-dimensional arrays (e.g., for matrix multiplication).

1.0.0.2 The not-to-cache label technique.

*Supported by NSF grant 9820955

A typical use of cache memories has been for storing just used data and instructions till their next reuse. Suppose that some of the data is accessed only once and this can be figured out prior to the execution. The technique simply marks this data as "not to be cached". This frees the execution to use the cache where it can make a difference. Examples that we have for the not-to-cache label technique include a variety of graph algorithms, where each edge of the graph is accessed at most once. This includes breadth-first search (BFS), Dijkstra's shortest-path algorithm, and a standard search procedure for directed acyclic graphs (DAGs), where "irregular" data structures (e.g., lists and pointers) are often used.

The presentation illustrates how each of the techniques addresses latency and bandwidth concerns.

Consider the following idealized memory-centric *computing model* as a framework for empirical and analytical work. There are two levels of memory: cache and main memory. The cache has limited size (64MB is one forward looking possibility). There is unlimited bandwidth to the cache. Access time to the cache is 1 time unit. The main memory has unlimited size. There is limited bandwidth to the main memory. Access time to the main memory is high (100 time units is one current possibility). Other issues are suppressed in this model.

Ideally, the optimizations described in our examples should be done by the computer system itself, whether using static or run-time methods. The purpose of this paper is to present optimizations which are feasible. The author's hope is that our results will motivate incorporating some new optimizations in future systems; this should mostly, though not exclusively, be a compiler effort.

It is important to understand that the bandwidth concerns which motivate the current paper are different, though not completely unrelated, than space concerns which motivated papers such as [BGMN] and [BL93].

2. WAVEFRONT BLOCKING

Consider the dynamic programming algorithm for the longest common subsequence (LCS) problem (see, e.g., page 314 of [CLR90]), among two sequences $X[1..n]$ and $Y[1..m]$. This algorithm computes values into a two dimensional array $A[1..n;1..m]$, where $A(i,j)$ is the length of the common subsequence of the sequences $X[1..i]$ and $Y[1..j]$. The algorithm can be described as a parallel algorithm. Each step of the parallel algorithm consists of advancing the "wave front" of the computation from one ascending diagonal to the next. Formally, given a positive integer k, *ascending diagonal* k consists of all entries (i,j) in A for which $i + j = k$. The value of $A(i,j)$ is a simple function of 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 $Y(j)$. Concretely, $A(i,j) := max(A(i-1,j), A(i,j-1), if\ X(i) = Y(j)\ then\ A(i-1,j-1) + 1\ else\ A(i-1,j-1))$

If the diagonal, as well as X and Y, fit into the cache, then the wavefront blocking technique does not do anything. However, if the diagonal is too big for this, the progress of the computation is updated to avoid the need to repeatedly flush the cache. In other words, the principled question addressed by the wavefront blocking technique is: given the size of the cache, "massage" the wave front of the underlying parallel algorithm to get the best performance in the above model.

Assumptions made: (i) The size of the cache could become known to the compiler. (ii) Prefetching is used. Overall, Assumption (ii) tends to strengthen the alternative, where the wavefront blocking technique is not used; while prefetching alleviates the latency to main memory, it is still constrained by bandwidth limitations.

Next we provide some run time estimates. They are all done using pencil and paper. Suppose that all memory references require data from main memory (since the requested data is not in the cache). Since the number of operations relative to each diagonal element is small, the performance of the algorithm will be dominated by the bandwidth to memory.

Observe that, in case the diagonal is too long for fitting into the cache, the run time of the algorithm in our model will be dominated by the rate at which data can be transmitted from memory. This is since each cycling through the diagonal will happen after the previously stored data has been replaced (assuming that LRU is used for data replacement). In the computer architecture literature, this case is referred to as cache thrashing.

With the wavefront blocking technique, we serialize the algorithm into several rounds, where in each round a fraction (of the diagonal) that fits into the cache is used. That is, if the size of the cache can accommodate a diagonal of lengh d, then we will first run the LCS algorithm for the first d characters of X, $X[1..d]$, and the original $Y[1..m]$. (Using prefetching, it is easy to refine the computation to have only the d values from $Y[1..m]$, which are needed for computation of the "current" diagonal, if m is too large.) Second, for $X[d+1..2d]$, and $Y[1..m]$. Third, for $X[2d+1..3d]$, and $Y[1..m]$; and so on. The running time will be dominated by the time to access the cache for all data, since the data will (almost) always be ready in the cache.

2.0.0.3 *Second example: matrix multiplication*

Let A and B be two nXn matrices. We want to compute their product into matrix C. Assume that the cache is too small to include the matrices. If we use a standard matrix multiplication algorithm, the total traffic of data that needs to be brought to the processor can reach n^3, and this will dominate the overall execution time.

Suppose that the size of the cache is $3t^2$ for some integer t, and that n is divisible by t. Figure 1 illustrates how to partition each of the 3 matrices into n^2/t^2 square submatrices. The matrix multiplication will proceed in n^2/t^2 iterations, each will be devoted to compute a different submatrix of C. Each of these iterations works in n/t subiterations, where in a subiteration the product of a submatrix of A with a submatrix of B is computed. The data traffic of each iteration is at most $t^2(n/t) = nt$. Since there are n^2/t^2 iterations, the total traffic does not exceed n^3/t. Using current figures for bandwidth relative to processor speed imply that the processing of $O(n^3)$ operation by the CPU will dominate the overall computation time.

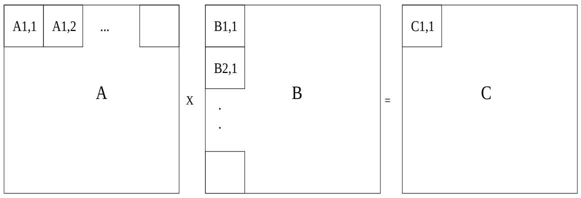

Figure 1: The submatrices

Any reader familiar with current practice will have no problem noticing several issues that have been suppressed. The way in which a matrix is stored in the memory should make a difference. That is, should it be stored column-wise, row-wise, or diagonal-wise? Each of these options will affect differently the content of cache lines.

The matrix multiplication approach shown here is similar to [ACS90] where it was used for studying communication complexity of PRAMs.

3. NOT-TO-CACHE LABEL

Our first example in the current section is breadth-first search (BFS). We first recall that the list of edges of a certain vertex is visited *once* only (from that vertex). The *key observation* here is as follows: *If we know that each edge is visited only once (from a vertex), we can actually avoid ever putting it in the cache, since it will never be reaccessed.*

Consider now two data structures that the BFS code uses. One which stores the "unvisited"/"visited" label for each vertex, and another which provides the (input) edges of the graph, which can be in the form of adjacency lists, or incidence matrix. Suppose, for example that the size of the cache suffices to hold the first data structure, but not the second. The above key observation guides the system to store the first data structure, and not the second. This would provide better performance than storing the just-used entries of the second data structure, since entries of the second data structure are never reused.

For simplicity, suppose that the memory operations of the BFS algorithm consist of iterations; in each iteration, another edge (u, v) of vertex u is visited and then the label of vertex v is looked up. In the case where the size of the cache suffices to hold the first data structure, but not the second, 50% of the maximum possible traffic to memory will be saved. If traffic to memory is the bottleneck then the saving in overall performance can be significant. The exact number would depend on the structure of the graph. Simulations will reveal the answer to the graphs checked, but the value of simulations will be limited since the issue of what represents a typical family of graphs has been remained largely unresolved in the area of experimental algorithms.

To put the not-to-cache technique in context we need to discuss two issues: (i) For the key observation to be useful, we must have a mechanism for marking datum as: "not to be cached". (ii) How could one figure out from the code that the edges of each vertex are visited only once? In BFS, a certain label of each vertex changes at most once from "unvisited" to "visited"; the label never changes after that; this one time change occurs prior to the only piece of code which visits its outgoing edges. In other words, it should be feasible for a compiler to conclude from the code that the edges of a certain vertex are visited only once. We refer the reader to Section 4.1 for a discussion of the two techniques in the context of the explicit-multi threading (XMT) PRAM-On-Chip project. Deriving the conclusion that the edges of each vertex are visited only once from parallel XMT code will be even easier, since it can be done by analyzing the use of Prefix Sum (PS) in the code.

3.0.0.4 More examples

Dijstra's shortest-path algorithm and a standard search procedure for directed acyclic graphs (DAGs).

Some work on cache bypass has been done in [RD96] and [TFMP95]. Run-time statistics identifies data which is likely not to be used more than once. This data is put in a buffer rather than in the cache.

4. DISCUSSION

The area of extracting parallelism from code by compilers has received considerable attention. Our work is orthogonal to this kind of efforts, although it should be recognized that significant compiler work (and sometimes a bit more) is needed in order to upgrade a serial code of any of the algorithms mentioned above into a form where our optimizations have their full impact on performance.

For this reason, our analysis in the current paper is not in the strict quantitative mode, where hard numbers are provided. We defer this kind of simulation-based work which assumes a particular cache replacement policy (such as LRU - least recently used), or a given environment, where the only update needed is the one proposed in the paper, to later work; it was more important at the current stage to bring out the main ideas. While a quantitative analysis would have been possible for XMT (see the following subsection), a focus on that would have diluted the generality of the approach and its potential relevance to serial computing platforms.

Finally, we would like to reiterate that, perhaps the most interesting comment in the current paper, is that both techniques can be incorporated in a compiler.

4.1 The XMT Context

Performance gaps (or mismatches) between the multiprocessors of the 1990s and the theory of parallel algorithms were aggravated by difficulties in reconciling the simple memory model of parallel algorithms with memory systems of those multiprocessors. This created an unhealthy situation of machines with a rather thin algorithmic theory, on one hand, and an algorithmic theory with no machines, on the other hand. We believe that ideas similar to the ones presented here could lead to streamlining parallel algorithms and parallel machines, using the large on-chip caches which are becoming available. This concern is being addressed by our PRAM-On-Chip project [VDBN98] and [NNTV01].

For concreteness, we note one significant difference between a serial computing implementation of BFS and an XMT implementation. Using its IOS (independence of order semantics), XMT does not enforce a serial order on the accesses to "visited"/"unvisited" labels of the vertices. This means the following. In serial computing, if there is delay in the arrival of some datum from the memory, processing at the processing unit must halt. But, in XMT the processing can proceed with whatever data which has arrived.

Next are a few general words of comparison with serial computing. Recall that: (i) Larger on chip caches are possible, but for serial computing, the return on using them is diminishing. (ii) When very few cache misses can overlap (in time), as in serial computing, even the (limited) bandwidth to memory is underused.
Our intuition is that XMT does better than serial computing on both accounts: (a) latency, by overlapping cache misses, and (b) bandwidth, by generating concurrently memory requests, while the use of the cache alleviates a possible penalty from overuse. Coupled with IOS, XMT reduces the effect of cache stalls.

5. CONCLUSIONS

It has been previously observed that algorithm parallelism allows to use prefetching to "hide" the increasing processor to memory latencies. However, a straightforward application of prefetching could aggravate performance degradation due to another memory constraint: limited bandwidth to main memory. The current paper demonstrates that there are ways to reconcile algorithm parallelism with the limited bandwidth constraint, as well. The current paper actually becomes a position paper in its call for applications and extensions of this point.

We finally note that the literature reports a considerable amount of related work on coping with similar memory constraints, although most of this work has not been in the context of algorithm parallelism. We are even aware of examples where some of this work has reached homework assignments given to undergraduate computer science students. A small sample of recent work includes [BAYT01, DK00, DK01] and some of the references in these papers.

6. ACKNOWLEDGMENTS

Helpful comments by R. Barua and D. Yeung are gratefully acknowledged.

7. REFERENCES

A. Aggarwal, A.K. Chandra and M. Snir. Communication complexity of PRAMs. In *Theoretical Computer Science* 71,1, 3–28, 1990.

A-H.A. Badaway, A. Aggarwal, D. Yeung and C-W. Tseng. Evaluating the impact of memory system performance on software prefetching and locality optimizations. In *15th Int. Conf. on Supercomputing*, Sorrento, Italy, June 2001.

G.E. Blelloch, P.B. Gibbons, Y. Matias and G.J. Narlikar. Space-efficient scheduling of parallelism with synchronization variables. In Proc. 9th ACM-SPAA, 1997, 12–22.

R.D. Blumofe and C.E. Leiserson. Space-efficient scheduling of multi-threaded computations. In *Proc. 25th ACM-STOC*, 362–371, 1993.

D. Burger, J. Goodman, and A. Kagi. Quantifying memory bandwidth limitations in future microprocessors. In the *International Symposium on Computer Architecture*, Philadelphia, Pennsylvania, May 1996. ACM.

T. Cormen, C.E. Leiserson, and R. Rivest. Introduction to Algorithms. MIT Press, 1st edition, 1990.

C. Ding and K. Kennedy. Memory bandwidth bottleneck and its amelioration by a compiler. In *Proc. Int. Parallel and Distributed Processing Symp.*, Cancun, Mexico. May 2000.

C. Ding and K. Kennedy. Improving effective bandwidth through compiler enhancement of global cache reuse. In *Proc. Int. Parallel and Distributed Processing Symp.*, San Francisco, California, April 2001.

D. Naishlos, J. Nuzman, C-W. Tseng and U. Vishkin. Towards a First Vertical Prototyping of an Extremely Fine-Grained Parallel Programming Approach. In *Proc. 13th ACM Symposium on Parallel Algorithms and Architectures (SPAA)*, 2001.

J.A. Rivers and E.S. Davidson. Reducing conflicts in direct-mapped caches with a temporality-based design. In *Proc. 1996 Int. Conf. on Parallel Processing*, 154–163, 1996.

R. Sites. Its the memory, stupid!. *Microprocessor Report*, Vol. 10,10, August 5, 1996, 2-3.

A. Saulsbury, F. Pong, and A. Nowatzyk. Missing the Memory Wall: The Case for Processor/Memory Integration. In *Proceedings of the 23rd Annual International Symposium on Computer Architecture*, 90-101, May 1996.
`http://www.ee.ualberta.ca/~elliott/ee602/seminars/` 2000w/ISCA96.Saulsbury.pdf

G. Tyson, M. Farrens, J. Matthews and A. R. Pleszkun. A modified approach to data cache management. In *Proc. 28th ACM/IEEE Int. Symp. on Microarchitecture*, 1995.

U. Vishkin, S. Dascal, E. Berkovich and J. Nuzman. Explicit Multi-Threading (XMT) bridging models for instruction parallelism (extended abstract). In *Proc. 10th ACM Symposium on Parallel Algorithms and Architectures (SPAA)*, 1998. For greater detail see the extended summary under
`http://www.umiacs.umd.edu/~vishkin/XMT/` .

U. Vishkin. Can parallel algorithms enhance serial implementation? In *Proc. 8th IEEE International Parallel Processing Symposium (IPPS)*, 1994, 376–385. Also, in *Communications of the ACM*, 39,9 (1996), 88-91.

M. Wilkes. The memory gap. Keynote address at the *Workshop on Solving the Memory Wall Problem*, held with ISCA 2000.
http://www.ece.neu.edu/conf/wall2k/wilkes1.pdf,

W. Wulf and S. McKee. Hitting the memory wall: Implications of the obvious. Computer Architecture News, 23(1), March 1995.