2005 International Conference on Analysis of Algorithms DMTCS proc. **AD**, 2005, 49–58

Cache efficient simple dynamic programming

Cary Cherng[1†] and Richard E. Ladner[1‡]

[1] *Department of Computer Science and Engineering, Box 352350, University of Washington, Seattle, WA 98195.* (ladner,ccerng)@cs.washington.edu

---

New cache-oblivious and cache-aware algorithms for simple dynamic programming based on Valiant's context-free language recognition algorithm are designed, implemented, analyzed, and empirically evaluated with timing studies and cache simulations. The studies show that for large inputs the cache-oblivious and cache-aware dynamic programming algorithms are significantly faster than the standard dynamic programming algorithm.

Keywords: Dynamic Programming, Cache-Oblivious Algorithms, Cache-Aware Algorithms

---

1 Introduction

Simple dynamic programming is the solution to one special type of dynamic program with many applications such as context-free language recognition, matrix-chain multiplication, and optimal binary trees. The solutions are easy to implement and run in time $O(n^3)$. Unfortunately, these standard implementations exhibit very poor cache performance (measured in the number of cache misses) when the input data is large. The purpose of this paper is to design, implement, analyze, and empirically evaluate divide-and-conquer algorithms that exhibit good cache performance this problem. We consider *cache-oblivious* algorithms where the algorithms do not have any knowledge of the cache parameters (bytes per memory block and cache capacity), yet have good cache performance regardless of the cache parameters [3, 7]. We also consider *cache-aware* algorithms where the algorithms may have even better cache performance with knowledge of the cache parameters. Amazingly, a divide-and-conquer algorithm for simple dynamic programming has been in existence for around thirty years and can be adapted to have good cache performance in both the cache-oblivious and cache-aware sense.

In modern computer systems, the memory is divided into levels, low to high: registers, L1 cache, L2 cache, main memory, and external memory. Data can be accessed at a lower level much more quickly than at a higher level. Rather than dealing with multiple levels of memory, for the remainder of the paper we just consider two levels: cache and memory. Because it is so costly to access memory, it is divided into memory blocks which contain multiple bytes. When a byte is accessed, the entire memory block where it resides is brought into the cache with the hope that nearby blocks will be accessed in the near future. Algorithms with good *spacial locality* have this feature.

There are several features of an algorithm that may potentially lead to it having good cache performance. First, an algorithm that processes its data *in-place* often has better cache performance than one that uses extra memory in addition to the memory needed for the input and output. For example, the small memory footprint of in-place quicksort means that it can handle larger input sizes before the cache miss penalty is incurred. By contrast, mergesort is not in-place because it uses an auxiliary array as the target of merging so that its cache miss penalty occurs with smaller inputs. Second, an algorithm that processes its data in a *divide-and-conquer* fashion often has better cache performance than one that uses iteration. This is because, in the divide-and-conquer approach a large problem is divided into subproblems, solved recursively, then the subproblem solutions are combined together to solve the original problem. When the subproblems are small enough to fit in the cache, then the only cache misses that occur are the compulsory misses needed to load the subproblem for the first time. Subsequent accesses to data in the subproblem

[†] Research partially supported by NSF-REU grant CCR-0098012.
[‡] Research partially supported by NSF grant CCR-0098012.

are found in the cache, that is, the algorithm has good *temporal locality*. For example, a divide-and-conquer approach to matrix multiplication has much better cache performance than the standard matrix multiplication algorithm [3].

1.1 Contributions

In 1975, Valiant showed that context-free language recognition can be solved in less than cubic time [11]. He did this by showing that context-free language recognition is reducible to Boolean matrix multiplication in a time preserving way. That is, context-free language recognition has the same time complexity as Boolean matrix multiplication, which had been shown to be solvable in less than cubic time by 1975. The first step in Valiant's proof is essentially giving a divide-and-conquer algorithm for simple dynamic programming.

We show that Valiant's divide-and-conquer algorithm can be done in-place in a general setting of a nonassociative semi-ring. We bound its cache performance, giving the coefficient of the dominant term. The pure form of Valiant's algorithm is cache-oblivious. We present a blocked version of Valiant's algorithm that is cache-aware that performs well for all input sizes. We compare the standard simple dynamic programming algorithm with Valiant's algorithm and its blocked version using timing studies and cache simulations. As expected, Valiant's algorithm and its blocked version have superior cache performance on all data sets and run in less time than the standard simple dynamic programming solutions on large data sets.

Our contribution is to demonstrate that a 30 year old algorithm that was designed for a theoretical goal is actually practical and useful on today's computers with memory hierarchies.

1.2 Other Related Prior Work

Dynamic programming is a powerful algorithmic tool that was identified in the pioneering work of Bellman [2]. Because most dynamic programs tend to have relatively high complexity (n^2, n^3, and worse), much effort, with significant success, has been put into trying to reduce their complexity. Several techniques stand out: the monotonicity property [6, 12], the Monge property [1], and convexity/concavity [14]. For example, establishing the monotonicity property reduces the complexity of finding the optimal binary search tree from $O(n^3)$ to $O(n^2)$. These algorithmic approaches are not always successful, so improving the complexity may entail reducing just the constant factor, which is the goal of cache-oblivious and cache-aware algorithms.

Expert programmers have known for many years that reducing cache misses can significantly improve the running time of programs. Some techniques for improving performance are cache-aware in that the program has tuning parameters that depend on the cache parameters of the target computer. The cache-oblivious approach pioneered by Frigo *et al* [3] does not use tuning parameters but relies on the structure of the algorithm to achieve good cache performance. A survey of results on cache-oblivious algorithms can be found in [7].

2 Matrix Multiply and Accumulate

Although the focus of the paper is on simple dynamic programming we will use a form of matrix multiplication as a building block. In particular we will use the following version of matrix multiplication that we call the *matrix multiply and accumulate* operation which is the assignment

$$U := U + W \cdot Z,$$

where U is stored disjoint from W and Z. In our application, U, W, and Z are submatrices of some larger matrix. The well-known divide-and-conquer algorithm for matrix multiplication, which probably goes back to Strassen [10], is cache-oblivious [3]. It can be modified to an algorithm for multiply and accumulate that is also in-place.

Lemma 2.1 *Let U, W, and Z be Boolean matrices of size $n = 2^k$, and with U in disjoint memory from W and Z. The matrix multiply and accumulate operation $U := U + WZ$ can be done in-place using divide-and-conquer.*

Proof: The proof is by induction on k. If $k = 0$, then the operation on single elements can be done in-place. If $k > 0$, then partition U into four submatrices of size 2^{k-1}

$$U = \begin{bmatrix} U_{11} & U_{12} \\ U_{21} & U_{22} \end{bmatrix}, \qquad W = \begin{bmatrix} W_{11} & W_{12} \\ W_{21} & W_{22} \end{bmatrix}, \qquad Z = \begin{bmatrix} Z_{11} & Z_{12} \\ Z_{21} & Z_{22} \end{bmatrix}$$

The assignment $U := U + W \cdot Z$ is equivalent to the parallel assignment

$$\left[\begin{array}{cc} U_{11} & U_{12} \\ U_{21} & U_{22} \end{array} \right] := \left[\begin{array}{cc} U_{11} & U_{12} \\ U_{21} & U_{22} \end{array} \right] + \left[\begin{array}{cc} W_{11} & W_{12} \\ W_{21} & W_{22} \end{array} \right] \cdot \left[\begin{array}{cc} Z_{11} & Z_{12} \\ Z_{21} & Z_{22} \end{array} \right].$$

Compute U_{11} by the assignments

$$\begin{array}{rcl} U_{11} & := & U_{11} + W_{11} \cdot Z_{11} \\ U_{11} & := & U_{11} + W_{12} \cdot Z_{21} \end{array}$$

which by the induction hypotheses can be done in-place. All these other submatrices of U can be computed in-place similarly. □

The asymptotic cache performance analysis is similar to analysis in [3]. We assume a fully associative cache and an optimal replacement policy. In addition, we ignore the small number of cache misses when accessing the constant number of global variables and temporaries. We will use two parameters, C the cache size in matrix items and B the number of items of a matrix that can fit in one memory block. Note that B can be fractional if a matrix item requires more bytes than a memory block. We assume that B is very small compared to C (In [3] $C = \Omega(B^2)$). This allows us to ignore low order edge effects where some memory blocks are brought into the cache but only partially accessed. Assume we have $n \times n$ matrices where n is a power of 2. Define $Q_M(n)$ to be the number of cache misses incurred by the divide-and-conquer multiply and accumulate operation. We have the following inequalities:

$$\begin{array}{rcll} Q_M(n) & \leq & 3n^2/B & \text{if } 3n^2 \leq C \\ & \leq & 8Q_M(n/2) + O(1) & \text{otherwise.} \end{array}$$

The base case of the recurrence comes from the fact that if all three matrices fit in the cache then at most $3n^2/B$ cache misses are incurred in bringing them into the cache. The additive constant term in the recurrence is for the additional storage needed for the recursive calls. Expanding out the recursion gives

$$Q_M(n) \leq 8^k Q(n/2^k) + O(1) \sum_{i=0}^{k-1} 8^i.$$

The base case implies $Q_M(n/2^k) \leq C/B$, and

$$k = \left\lceil \log \left(\frac{3n^2}{C} \right)^{1/2} \right\rceil.$$

We can then derive the solution

$$Q_M(n) \leq 8 \cdot 3^{3/2} \frac{n^3}{BC^{1/2}} + O\left(\frac{n^3}{C^{3/2}} \right). \tag{1}$$

There are a number of ways to express the solution to the recurrence. We choose this representation because it gives the constant factor in front of the dominant term $n^3/(BC^{1/2})$. This will allow us to estimate the number of cache misses in an actual implementation. Since we are interested in the constant factor of the high order term, we do not use the coarse bound $C = \Omega(B^2)$ in solving the recurrence.

3 Simple Dynamic Programming

In simple dynamic programming a problem $x_1, \cdots, x_n$ of size n is solved by solving all contiguous subproblems of smaller size. Each subproblem can be indexed by a pair of integers (i, j) with $1 \leq i \leq j \leq n$ which represents the subproblem $x_i, \cdots, x_j$ of size $j - i + 1$. To be more specific, we assume the input elements come from a set A which is the domain of a *nonassociative semi-ring* $(A, +, \cdot, 0)$ where $+$

is an associative ($x+(y+z) = (x+y)+z$), commutative ($x+y = y+x$), idempotent ($x+x = x$), binary operator and $\cdot$ is a nonassociative, noncommutative, binary operator. The value 0 is the additive identity ($x+0 = x$) and multiplicative annihilator ($x \cdot 0 = 0 \cdot x = 0$). Finally, the operators satisfy the distributive laws (($x \cdot (y+z) = x \cdot y + x \cdot z$ and $(y+z) \cdot x = y \cdot x + z \cdot x$). The simple dynamic programming problem with input $x_1, \ldots, x_n$, has as its goal to compute the sum of products of $x_1, \ldots, x_n$ in this order, under all possible groupings for the products. For example, if $n = 3$, compute $x_1 \cdot (x_2 \cdot x_3) + (x_1 \cdot x_2) \cdot x_3$ and for $n = 4$, compute the five-term sum:

$$x_1 \cdot ((x_2 \cdot x_3) \cdot x_4) + x_1 \cdot (x_2 \cdot (x_3 \cdot x_4)) + (x_1 \cdot x_2) \cdot (x_3 \cdot x_4) + ((x_1 \cdot x_2) \cdot x_3) \cdot x_4 + (x_1 \cdot (x_2 \cdot x_3)) \cdot x_4.$$

More generally, for n inputs there will be $\frac{(2(n-1))!}{(n-1)!n!}$ (the $(n-1)$-st Catalan number) terms in the sum. There are many examples of simple dynamic programming problems. Two that come to mind are the *matrix-chain multiplication* problem and the *context-free language membership* problem.

The matrix-chain multiplication problem is: given n matrices $M_1, \ldots, M_n$, where matrix M_i has dimension $p_{i-1} \times p_i$, find the minimal number of scalar multiplications needed to compute the matrix product $M_1 \times M_2 \times \cdots \times M_n$. The universe A consists of ∞ (the additive zero) and the triples (a, b, c) where a and b are positive integers, and c is a nonnegative integer. The addition operator is defined by $(a, b, c) + (a, b, c') = (a, b, \min(c, c'))$, $(a, b, c) + \infty = \infty + (a, b, c) = (a, b, c)$, and $\infty + \infty = \infty$. The multiplication operator is defined by $(a, b, c) \cdot (b, d, c') = (a, d, c + c' + abd)$ and all other cases multiply to ∞. Given a sequence of matrices $M_1, \ldots, M_n$ where matrix M_i has dimension $p_{i-1} \times p_i$, define $x_1, x_2, \ldots, x_n$ to be the input to the simple dynamic program where $x_i = (p_{i-1}, p_i, 0)$. The third component of the solution to the simple dynamic program is the minimal number of scalar multiplications needed to compute the matrix product.

The setup for the context-free language membership problem is quite different. Let (V, Σ, R, S) be a fixed Chomsky normal form context-free grammar where V is the set of nonterminals, Σ is the set of terminals, R is the set of production rules, and S is the start symbol. The production rules must have one of the two forms $A \to BC$ or $A \to a$ ($A, B, C \in V$ and $a \in \Sigma$). The membership problem is: given a string $w = w_1 w_2 \cdots w_n$ where $w_i \in \Sigma$, determine if w can be generated by the grammar. In this case the universe A is the set of subsets of V, and ϕ (the empty set) is the additive zero. The addition operator is union. The multiplication operator is defined by

$$x \cdot y = \{A \in V : \text{ for some } B \in x \text{ and } C \in y,\ A \to BC \text{ is a production rule in } R\}.$$

Given the input $w = w_1 w_2 \cdots w_n$, define $x_1, x_2, \ldots, x_n$ to be the input to the simple dynamic program where $x_i = \{A \in V : A \to w_i\}$. The input w is generated by the grammar if and only if the solution to the dynamic program contains the start symbol S.

There is a very elegant solution to the simple dynamic programming problem that runs in time $O(n^3)$. Indeed, this solution corresponds exactly to known solutions for the matrix-chain multiplication problem [4, 8] and the context-free language membership problem, which is commonly known as the Cocke-Kasami-Younger algorithm [5, 13]. Let $x_1, x_2, \ldots, x_n$ be an input. This solution, that is learned by countless computer science undergraduates, computes the solutions subproblems $x_i, \ldots, x_j$ in the order of larger and larger size of the subproblem, $d = j - i + 1$. The solution employs the upper-right of a two-dimensional array $D[1..n, 1..n]$. Initially, we set $D[i, i] := x_i$ and $D[i, j] := 0$ for $1 \leq i < j \leq n$. The algorithm then proceeds to fill in the remainder of the upper-right of D one diagonal at a time. The final result is in $D[1, n]$.

Diagonal Algorithm for Simple Dynamic Programming

```
for d = 2 to n
    for i = 1 to n − d + 1
        j := d + i − 1
        for k = i to j − 1
            D[i, j] := D[i, j] + D[i, k] · D[k + 1, j]
```

Horizontal Algorithm for Simple Dynamic Programming

```
for i = n − 1 to 1
```

for $j = i + 1$ to n
  for $k = i$ to $j - 1$
    $D[i, j] := D[i, j] + D[i, k] \cdot D[k + 1, j]$

Vertical Algorithm for Simple Dynamic Programming

for $j = 2$ to n
  for $i = j - 1$ to 1
    for $k = i$ to $j - 1$
      $D[i, j] := D[i, j] + D[i, k] \cdot D[k + 1, j]$

There are several things to note about the Diagonal Algorithm. First, there is no need to allocate an entire square array for D because only the upper right is used. Second, the algorithm can be done in-place, where we define in-place for simple dynamic programming to be the storage for $D[i, j], 1 \leq i \leq j \leq n$. Third, the algorithm has poor cache performance. To see this consider the computation of the d-th diagonal. In computing the entries of the d-th diagonal all the entries of D to the left and below the d-th diagonal are accessed. Fourth, there are two alternative algorithms, the *Horizontal Algorithm* and *Vertical Algorithm* that solve the subproblems $x_i, \ldots, x_j$ in a different orders. In the Horizontal Algorithm all the subproblems of the form $x_i, \ldots, x_j$ are solved before the subproblems of the form $x_{i-1}, \ldots, x_j$ and in the Vertical Algorithm all the subproblems of the form $x_i, \ldots, x_j$ are solved before the subproblems of the form $x_i, \ldots, x_{j+1}$. Both these solutions have slightly better cache performance than the Diagonal Algorithm because they have more temporal locality (see Section 5). For example, in the Horizontal Algorithm, when a row of the result is computed, the items in the same row are accessed repeatedly.

4 Valiant's DP-Closure Algorithm

As suggested by Valiant [11] it is very helpful to represent the simple dynamic programming problem in a matrix form. Given an input $x_1, x_2, \cdots, x_{n-1}$ we construct an $n \times n$ matrix X as follows: $X_{i,i+1} = x_i$, for $1 \leq i < n$, and all other entries are 0. In matrix notation the dynamic programming problem can be expressed by the closure operation, the *DP-closure* of X, X^+, defined by:

$$\begin{aligned} X^+ &= X^{(1)} + X^{(2)} + \cdots \quad \text{where} \\ X^{(1)} &= X \\ X^{(i)} &= \sum_{j=1}^{i-1} X^{(j)} \cdot X^{(i-j)} \qquad \text{for } i > 1 \end{aligned}$$

The entry X^+_{1n} contains the solution to the dynamic program.

The cache-oblivious algorithm we present is based on Valiant's algorithm for general context-free recognition [11]. Because we want to consider matrices that have size a power of two, for the remainder of the paper we will assume that n is a power of two and that the input to the dynamic program is $x_1, \ldots, x_{n-1}$, that is we only consider inputs of size one less than a power of two. To handle inputs not of that size, the input can be padded with the zero of the nonassociative semi-ring. Valiant's algorithm has two recursive routines, the star ($\star$) and the plus ($+$), which we summarize in our matrix notation. Let Y be a square matrix of size $n = 2^k$ whose upper left and lower right hand quarters are already DP-closed. For this special case, we let $Y^\star$ denote the DP-closure of Y. If $n = 2$, then $Y = Y^\star$ and there is nothing to do. Otherwise, partition Y into sixteen submatrices of size 2^{k-1} (six of which are zero).

$$Y = \begin{bmatrix} Y_{11} & Y_{12} & Y_{13} & Y_{14} \\ & Y_{22} & Y_{23} & Y_{24} \\ & & Y_{33} & Y_{34} \\ & & & Y_{44} \end{bmatrix}$$

The following recursive algorithm computes its DP-closure $Y^\star$. To emphasize, the preconditions are that

$$\begin{bmatrix} Y_{11} & Y_{12} \\ & Y_{22} \end{bmatrix}, \begin{bmatrix} Y_{33} & Y_{34} \\ & Y_{44} \end{bmatrix}$$

are DP-closed.

Valiant's Star Algorithm ($n = 2^k > 2$)

$$
\begin{aligned}
\begin{bmatrix} Y_{22} & Y_{23} \\ & Y_{33} \end{bmatrix} &:= \begin{bmatrix} Y_{22} & Y_{23} \\ & Y_{33} \end{bmatrix}^\star \\
Y_{13} &:= Y_{13} + Y_{12} \cdot Y_{23} \\
\begin{bmatrix} Y_{11} & Y_{13} \\ & Y_{33} \end{bmatrix} &:= \begin{bmatrix} Y_{11} & Y_{13} \\ & Y_{33} \end{bmatrix}^\star \\
Y_{24} &:= Y_{24} + Y_{23} \cdot Y_{34} \\
\begin{bmatrix} Y_{22} & Y_{24} \\ & Y_{44} \end{bmatrix} &:= \begin{bmatrix} Y_{22} & Y_{24} \\ & Y_{44} \end{bmatrix}^\star \\
Y_{14} &:= Y_{14} + Y_{12} \cdot Y_{24} \\
Y_{14} &:= Y_{14} + Y_{13} \cdot Y_{34} \\
\begin{bmatrix} Y_{11} & Y_{14} \\ & Y_{44} \end{bmatrix} &:= \begin{bmatrix} Y_{11} & Y_{14} \\ & Y_{44} \end{bmatrix}^\star
\end{aligned}
$$

For the general problem, let X be a square matrix of size $n = 2^k$ with the input $x_1, \dots, x_{n-1}$ just above the diagonal and the rest of the matrix zero. If $n = 2$, then $X = X^+$ and there is nothing to do. Otherwise, partition X into sixteen matrices of size 2^{k-1} (nine of which are zero).

$$
X = \begin{bmatrix} X_{11} & X_{12} & & \\ & X_{22} & X_{23} & \\ & & X_{33} & X_{34} \\ & & & X_{44} \end{bmatrix}
$$

Valiant's DP-Closure Algorithm ($n = 2^k > 2$)

$$
\begin{aligned}
\begin{bmatrix} X_{11} & X_{12} \\ & X_{22} \end{bmatrix} &:= \begin{bmatrix} X_{11} & X_{12} \\ & X_{22} \end{bmatrix}^+ \\
\begin{bmatrix} X_{33} & X_{34} \\ & X_{44} \end{bmatrix} &:= \begin{bmatrix} X_{33} & X_{34} \\ & X_{44} \end{bmatrix}^+ \\
X &:= X^\star
\end{aligned}
$$

These descriptions of Valiant's Star and DP-closure Algorithms are different from what appeared in Valiant's paper [11]. For example, from Valiant's original description is was not clear that the algorithms could be done in-place. The algorithms can be proved to be in-place by induction on k. They are certainly in-place for $k = 1$. For $k > 1$, each of the lines of the algorithm is in-place either by the induction hypothesis or by lemma 2.1.

Now we shall prove that Valiant's DP-closure Algorithm and Valiant's Star Algorithm are correct. The proofs are by induction on the size 2^k of the matrix X.

For Valiant's DP-closure Algorithm, in the base case where $k = 1$ the matrix X has only one nonzero element x_{12}. In this case $X = X^+$ and so nothing is done by the algorithm. Now assume the DP-closure Algorithm is correct for matrices of size 2^{k-1} where $k > 1$. Then the first two steps correctly compute the DP-closure of X for the submatrices

$$
\begin{bmatrix} X_{11} & X_{12} \\ & X_{22} \end{bmatrix}, \begin{bmatrix} X_{33} & X_{34} \\ & X_{44} \end{bmatrix}
$$

Then X has the necessary form so that the last operation $X := X^\star$, invoking the Star Algorithm, will finish computing X^+.

As for the correctness of Valiant's Star Algorithm, in the base case where $k = 1$ the matrix Y has only one nonzero element y_{12}. In this case $Y^\star = Y$ and so nothing is done by the algorithm. Now assume the Star Algorithm is correct for matrices of size 2^{k-1} where $k > 1$. As defined in Star Algorithm if, Y is the input then $Y_{13}, Y_{23}, Y_{24}, Y_{14}$ are the only submatrices where the DP-closure of Y is not known. By the inductive hypothesis the first step gives the DP-closure of Y for submatrix Y_{23}. After the second and third steps, the DP-closure of Y for submatrix Y_{13} is known. To see this let us consider a fixed entry y_{ij} of Y_{13}.

The standard algorithm computes the sum

$$y_{ij} := \sum_{k=i+1}^{j-1} y_{ik} y_{kj} \tag{2}$$

where the y_{ik}'s and y_{kj}'s have already been DP-closed; the sum being taken in any order. We will show that the second and third steps compute the above sum. By examining the indices we see that y_{ik} is an element of Y_{12} if and only if y_{kj} is an element of Y_{13}. We rewrite (2) as

$$y_{ij} := \sum_{y_{ik} \in Y_{12}} y_{ik} y_{kj} + \sum_{y_{ik} \in Y_{11} \cup Y_{13}} y_{ik} y_{kj}. \tag{3}$$

The second step $Y_{13} := Y_{13} + Y_{12} \cdot Y_{23}$ accounts for terms in the first sum of (3). The third operation has the same effect as the standard algorithm applied to the submatrix

$$\begin{bmatrix} Y_{11} & Y_{13} \\ & Y_{33} \end{bmatrix},$$

but that accounts for the terms in the last sum of (3). Hence, the second and third steps compute the DP-closure of Y for submatrix Y_{13}. Likewise, the fourth and fifth steps compute the DP-closure of Y for submatrix Y_{24}, and the last three steps compute the DP-closure of Y for submatrix Y_{14}. We underscore the following observation by stating it as a theorem. Valiant's algorithm for simple dynamic programming can be implemented recursively and in-place.

4.1 Analysis of Valiant's DP-Closure Algorithm

An elementary analysis shows that the time complexity of Valiant's Star and DP-closure algorithms is $O(n^3)$. We will focus on the cache performance analysis. Define $Q_\star(n)$ and $Q_+(n)$ to be the number of cache misses incurred by the algorithm that computes $Y^\star$ and X^+, where Y and X are square matrices of size $n = 2^k$ and Y has its upper left and lower right quarters are already DP-closed. We have the following inequalities, where $Q_M(n)$ is the number of cache misses incurred by the divide-and-conquer multiply and accumulate operation (Section 2).

$$\begin{array}{rcll} Q_\star(n) & \leq & n(n-1)/(2B) & \text{if } n(n-1)/2 \leq C \\ & \leq & 4Q_\star(n/2) + 4Q_M(n/4) + O(1) & \text{otherwise,} \end{array}$$

$$\begin{array}{rcll} Q_+(n) & \leq & n(n-1)/(2B) & \text{if } n(n-1)/2 \leq C \\ & \leq & 2Q_+(n/2) + Q_\star(n) + O(1) & \text{otherwise.} \end{array}$$

Expanding out the recursion for $Q_\star(n)$ gives

$$Q_\star(n) \leq 4^k Q_\star(n/2^k) + \sum_{i=1}^{k} 4^i Q_M(n/(2^{i-1}4)) + O(1)\frac{4^k - 1}{4 - 1}$$

The base case implies $Q_\star(n/2^k) \leq C/B$, and

$$k = \left\lceil \log\left(\frac{n}{4C}(-1 + (1 + 8C)^{1/2})\right) \right\rceil. \tag{4}$$

Then using inequality (1) and $C = \Omega(B^2)$ we can show

$$Q_\star(n) \leq 3^{3/2} \frac{n^3}{BC^{1/2}} + O\left(\frac{n^3}{C^{3/2}} + \frac{n^2}{B}\right). \tag{5}$$

As for $Q_+(n)$ we begin by expanding out the recursion which gives

$$Q_+(n) \leq 2^k Q_+(n/2^k) + \sum_{i=0}^{k-1} 2^i Q_\star(n/2^i) + O(1) \sum_{i=0}^{k-1} 2^i.$$

The base case implies $Q_+(n/2^k) \leq C/B$, and the same value for k as in (4). Then using inequality (5) and $C = \Omega(B^2)$ we can show

$$Q_+(n) \leq 4 \cdot 3^{1/2} \frac{n^3}{BC^{1/2}} + O\left(\frac{n^3}{C^{3/2}} + \frac{n^2}{B} + \frac{n}{B/C^{1/2}}\right). \tag{6}$$

4.2 Blocked Valiant's DP-Closure Algorithm

The pure version of Valiant's DP-Closure Algorithm is cache-oblivious but has a lot of overhead because of the recursive calls and calls to matrix multiply and accumulate. An alternative is a cache-aware algorithm which chooses two parameters S and M. The parameter S is such that if $n \leq S$ then X^+ and $Y^\star$ are computed using the standard iterative dynamic program. In particular, computing $Y^\star$ requires only computing the upper right hand quarter of the DP-closure of Y. The parameter M is such that if $n \leq M$ then the matrix multiply and accumulate operations are done in the standard way, not with recursive divide-and-conquer. The good settings for the parameters S and M can be determined experimentally or can be estimated. For example, S can be chosen to be the largest number such that $S(S-1)/2 \leq C$ and M can be chosen to be the largest number such that $3M^2 \leq C$. If there is more than one cache, then there is no clear choice for S and M, so experiments can used to find the best parameters.

5 Experiments

We implemented, in C++, the Diagonal, Horizontal, and Vertical algorithms for simple dynamic programming. We also implemented Valiant's and Blocked Valiant's DP-Closure algorithms. In simple dynamic programming a problem of size $n-1$ corresponds to an $n \times n$ upper triangular matrix. The lower triangle of the matrix is unused. Hence, only the upper triangle of the matrix needs to be stored, and we chose to store it in row-major order. The experiments were run under Red Hat Linux 9 on a 1 GHz AMD Athlon with 64 KByte L1 data cache (2-way) and 256 KByte L2 data cache (16-way). The memory block size is 64 Bytes for both caches. The compiler used was g++ (GCC) 3.2.2 20030222 (Red Hat Linux 3.2.2-5, with optimization -O3). We also used the Cachegrind profiler as part of the Valgrind trace driven debugger and profiler [9]. The simulator used the same cache parameters as the AMD Athlon. The implementation was for the specific problem of matrix-chain multiplication which used 4 Byte integers chosen randomly. Thus, $B = 64/4 = 16$, and for L1, $C = 2^{16}/4 = 2^{14}$ and for L2, $C = 2^{18}/4 = 2^{16}$. In our studies where the input size varies we report normalized means which is the average of two medians of ten experiments divided by n^3.

In order to establish good block parameters S and M for the Blocked Valiant's DP-closure algorithm we experimented to find the best parameters for $n = 2,048$. Figure 1 gives a plot of the running time versus S and M. The best choices are $S = 256$ and $M = 64$ which we used for all the experiments.

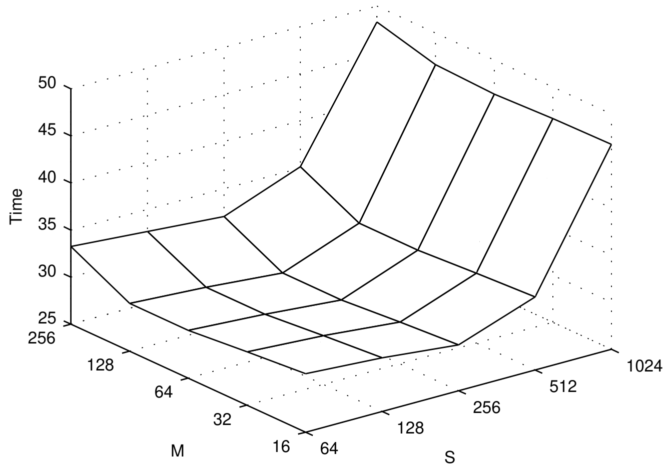

Fig. 1: Time in seconds are plotted as a function of the two block parameters S and M for inputs matrices of size 2,048.

Figure 2 shows instruction count (left) and running time (right) for Valiant's DP-closure algorithm (V), Blocked Valiant's DP-closure algorithm with $S = 256$ and $M = 64$ (V256-64), the Diagonal Algorithm (Diag), Horizontal Algorithm (Horz), and the Vertical Algorithm (Vert). It is clear that the instruction counts for the cache-oblivious V algorithm are much larger than all the other algorithms. All the standard dynamic programming algorithms suffer greatly in running time when the problem size is greater than 255. The cache-aware V256-64 algorithm out performs the cache-oblivious algorithm V in both instruction count and running time.

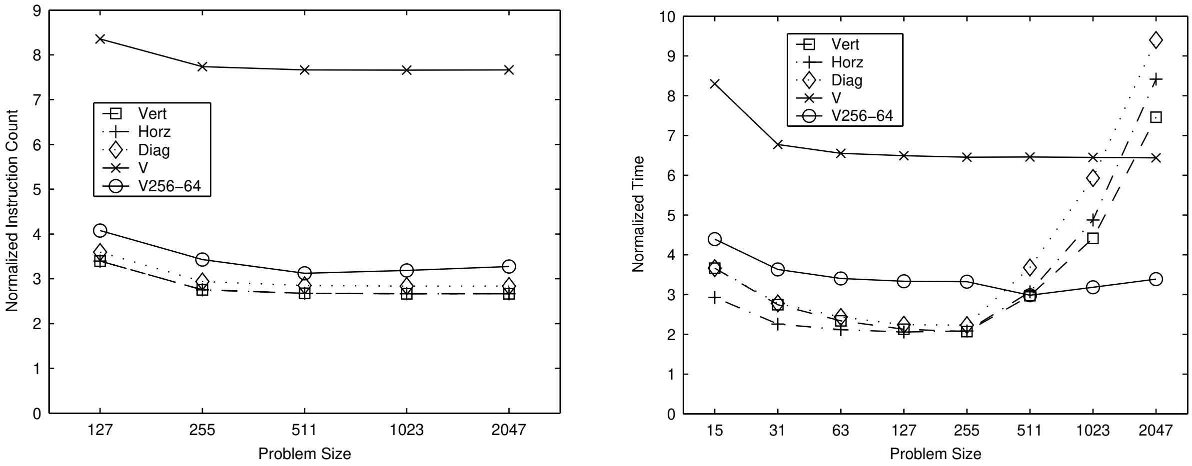

Fig. 2: Normalized instructions (left) and time in nanoseconds (right) are plotted for the five algorithms.

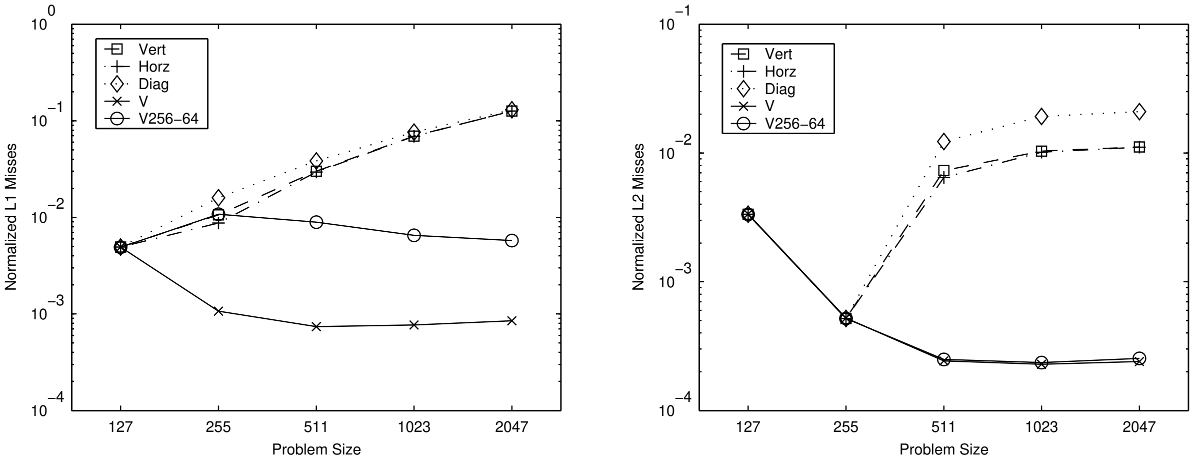

Fig. 3: Normalized cache misses for L1 (left) and L2 (right) are plotted for the five algorithms.

Figure 3 shows the cache performance of all the algorithms in the L1 and L2 caches, on the left and right, respectively. The normalized cache misses are shown on a log scale because of the huge difference between the three standard algorithms and the two versions of Valiant's algorithm. The standard algorithms have a factor of 100 more cache misses on the largest problem size. It is interesting to note that V and V256-64 have essentially the same number of cache misses in the L2 cache but V256-64 has more cache misses than V in the L1 cache.

It is interesting to calculate how well the theoretical upper bound constant $4 \cdot 3^{1/2} \approx 6.93$ of inequality (6) holds ups in the cache simulations. For a cache with B items per memory block and C items per cache we would expect this 6.93 to be a reasonable upper bound for the number of cache misses $\times BC^{1/2}/n^3$. For $n = 2047$, for the L1 cache we get 1.74 and for the L2 cache we get 0.98. It appears that our formula (6) has some predictive value.

6 Conclusions

We have demonstrated empirically that adaptations of Valiant's 1975 algorithm for solving the simple dynamic programming problem have good cache performance in both the cache-oblivious and cache-aware sense.

7 Acknowledgments

We would like to thank David Wise who read an early version of the paper and helped us understand some of the history of cache efficient matrix multiplication.

References

[1] A. Aggarwal, M. M. Klawe, S. Moran, P. Shor, and R. Wilber. Geometric applications of a matrix-searching algorithm. *Algorithmica*, 2(2):195–208, 1987.

[2] R. E. Bellman. *Dynamic Programming*. Princeton University Press, Princeton, New Jersey, 1957.

[3] M. Frigo, C. E. Leiserson, H. Prokop, and S. Ramachandran. Cache-oblivious algorithms. In *40th Annual Symposium on Foundations of Computer Science (FOCS '99)*, pages 285–298, 1999.

[4] S. S. Godbole. On efficient computation of matrix chain products. *IEEE Transactions on Computers*, C-22(9):864–866, 1973.

[5] T. Kasami. An efficient recognition and syntax algorithm for context-free languages. Scientific Report AFCRL-65-758, Air Force Cambridge Research Laboratory, Bedford, MA.

[6] D. E. Knuth. Optimum binary search trees. *Acta Informatica*, 1:14–25, 1971.

[7] P. Kumar. Cache oblivious algorithms. In U. Meyer, P. Sanders, and J. Sibeyn, editors, *Algorithms for Memory Hierarchies: Advanced Lectures*, pages 193–212. Springer-Verlag, 2003.

[8] Y. Muraoka and D. J. Kuck. On the time required for a sequence of matrix products. *Communications of the ACM*, 16(1):22–26, 1973.

[9] J. Seward and N. Nethercote. Valgrind (debugging and profiling tool for x86-linux programs). http://valgrind.kde.org/index.html.

[10] V. Strassen. Gaussian elimination is not optimal. *Numer. Math.*, 13:357–361, 1969.

[11] L. G. Valiant. General context-free recognition in less than cubic time. *Journal of Computer and Systems Sciences*, 10:308–315, 1975.

[12] F. F. Yao. Efficient dynamic programming using quadrangle inequalities. *Proceedings of the 12-th Annual ACM Symposium on Theory of Computing*, pages 429–435, 1982.

[13] D. H. Younger. Recognition of context-free languages in time n^3. *Information and Control*, 10(2):189–208, 1967.

[14] Z. Galil Z and K. Park. Dynamic programming with convexity, concavity and sparsity. *Theoretical Computer Science*, 92(1):49–76, 1992.