Cache-efficient Dynamic Programming Algorithms for Multicores [*]

Rezaul Alam Chowdhury
The University of Texas at Austin
Department of Computer Sciences
Austin, TX 78712
shaikat@cs.utexas.edu

Vijaya Ramachandran
The University of Texas at Austin
Department of Computer Sciences
Austin, TX 78712
vlr@cs.utexas.edu

ABSTRACT

We present cache-efficient chip multiprocessor (CMP) algorithms with good speed-up for some widely used dynamic programming algorithms. We consider three types of caching systems for CMPs: *D-CMP* with a private cache for each core, *S-CMP* with a single cache shared by all cores, and *Multicore*, which has private L_1 caches and a shared L_2 cache. We derive results for three classes of problems: local dependency dynamic programming (LDDP), Gaussian Elimination Paradigm (GEP), and parenthesis problem.

For each class of problems, we develop a generic CMP algorithm with an associated tiling sequence. We then tailor this tiling sequence to each caching model and provide a parallel schedule that results in a cache-efficient parallel execution up to the critical path length of the underlying dynamic programming algorithm.

We present experimental results on an 8-core Opteron for two sequence alignment problems that are important examples of LDDP. Our experimental results show good speed-ups for simple versions of our algorithms.

Categories and Subject Descriptors

B.2.1 [**Arithmetic and Logic Structures**]: Design Styles—*Parallel*; B.3.2 [**Memory Structures**]: Design Styles—*Cache memories*; F.3.3 [**Studies of Program Constructs**]: Program and recursion schemes

General Terms

Algorithms, Theory, Performance, Experimentation

Keywords

cache-efficiency, shared cache, distributed cache, multicore, parallelism

1. INTRODUCTION

Chip multiprocessors (CMP) are rapidly becoming the dominant parallel computing platform. A key feature that sets CMPs apart from earlier parallel machines is the organization and cost measure for memory accesses. In a CMP, memory is organized in a hierarchy of caches, and two new features come with this:

- The cost of a memory access is determined by whether the accessed data item is a cache hit or a cache miss. This is in contrast to using the routing cost, or latency and gap parameters, to measure communication cost as in, e.g., [28, 16, 11].
- A cache has a small finite size, hence a data item brought into cache and needed at a later time may not reside there at that later time if it has been evicted to make space for other data items. This is a feature that has not been captured in any of the traditional parallel models and algorithms.

In this paper we consider some important dynamic programming (DP) applications, and we develop good CMP implementations under three types of caching models for CMP (private, shared, and multicore). We elaborate on these three caching models in Section 1.1. For all three CMP models, we develop parallelizations of dynamic programming algorithms that match the best sequential caching complexity while achieving the maximum speed-up available in the algorithm. We obtain results for the following classes of problems:

1. *LDDP (Local Dependency DP) problems,* where each update to an entry in the DP table is determined by contents of neighboring cells. The simplest example of this class is the well-known *LCS* (longest common subsequence) problem. Some widely-used string problems in bioinformatics including *PA* (pairwise sequence alignment with affine gap cost) and *Median* (3-way sequence alignment with affine gap) are examples of LDDP. We present experimental results for PA and Median that show simplified versions of our algorithms to run several times faster than currently used software for these problems even when using the default Linux scheduler for our parallel code.
2. *GEP (Gaussian Elimination Paradigm)*, which solves another wide class of problems including all-pairs shortest paths in weighted graphs (Floyd-Warshall DP), LU decomposition and Gaussian elimination without pivoting, and matrix multiplication. Here we improve the p-processor cache-efficient parallelism from $\Theta(\frac{n^3}{p} + n\log^2 n)$ in the earlier CMP algorithms in [8, 1] to

[*]This work was supported in part by NSF Grant CCF-0514876 and NSF CISE Research Infrastructure Grant EIA-0303609.

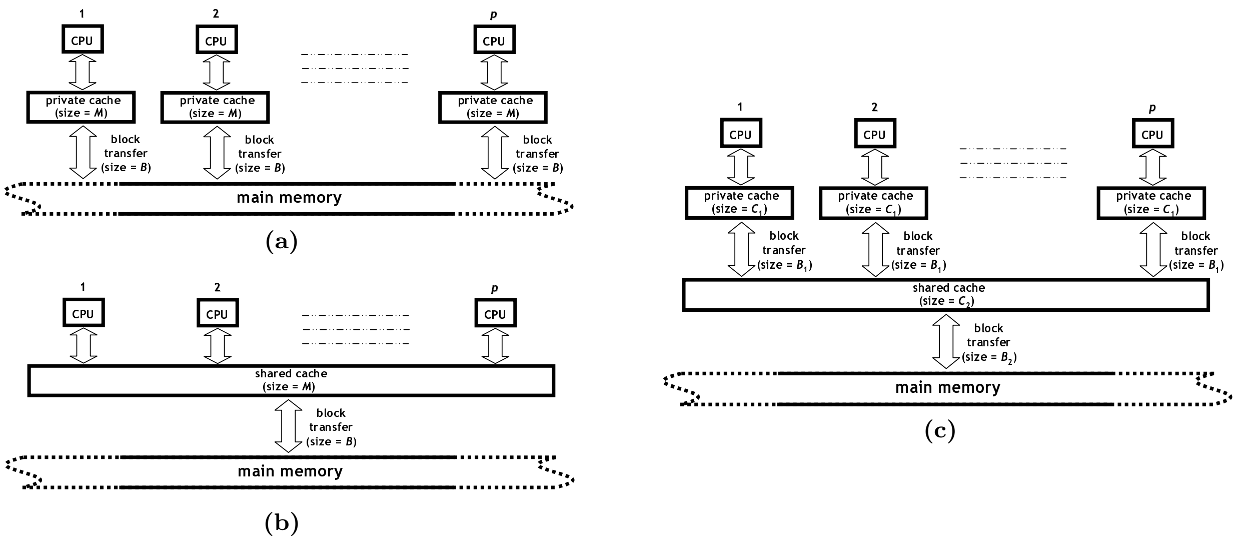

1
2
p
CPU
private cache (size = M)
block transfer (size = B)
main memory
(a)
shared cache (size = M)
(b)
private cache (size = C_1)
block transfer (size = B_1)
shared cache (size = C_2)
block transfer (size = B_2)
(c)

Figure 1: Caching models for CMP: (a) D-CMP, (b) S-CMP, and (c) Multicore.

$\Theta\left(\frac{n^3}{p}+n\right)$[1] on an $n \times n$ input, which is the best possible when staying within the GEP framework.

3. *Parenthesis problem*, which includes RNA secondary structure prediction, optimal matrix chain multiplication, construction of optimal binary search trees, and optimal polynomial triangulation.
4. *RNA-SP (RNA secondary structure prediction with simple pseudo-knots).* Combining our results for 3 dimensional LDDP and GEP it is straightforward to obtain results for RNA secondary structure prediction with simple pseudo-knots, using the sequential RNA-SP algorithm in [6]. We do not elaborate on this further.

Given a parallel algorithm, let $T_\infty(n)$ denote the number of parallel steps in the algorithm as a function of the input size n. This is normally referred to as the critical path length of the computation, and represents its intrinsic parallelism. We use the term *cache-efficient parallelism* or *cache-efficient critical path length* $I_\infty(n)$ to denote the number of parallel steps in a parallel algorithm that also matches the sequential work and I/O bound.

For all of the problems mentioned above we present CMP algorithms with $I_\infty(n) = \Theta(n)$ on all three models described in Section 1.1. Since the sequential running time is $\Theta(n^2)$ for LCS and PA, is $\Theta(n^3)$ for Median, GEP and Parenthesis, and $\Theta(n^4)$ for RNA-SP, we obtain a good amount of parallelism with $I_\infty(n) = \Theta(n)$. In fact, the defining DP algorithm for each of the problems, which is the one we use to derive our cache-efficient implementation, has critical path length $\Theta(n)$, hence we achieve the maximum parallelism possible while remaining as cache-efficient as the sequential case.

If we look purely for parallelism, there are NC algorithms known for all of these problems, and there are also work-optimal parallel algorithms that beat our parallel bound I_∞ (see, e.g., [15] and references therein for LDDP and Parenthesis problem). However, none of these results incorporate cache-efficiency in the parallel context, which is crucial for CMPs.

[1]In other words, we improve the critical pathlength of the computation from $\Theta\left(n \log^2 n\right)$ to $\Theta(n)$.

1.1 Caching models for CMP

We consider the following three models for CMPs. In all three models, data is transferred from one memory/caching level to another in a block of a given size.

1. **D-CMP**. Here the CMP is viewed as a p-processor machine, where each processor has its own private cache of size M [14, 8] (i.e., distributed caches). There is also a global shared memory that is arbitrarily large. The performance of a parallel algorithm is measured in terms of the number of parallel steps executed and the total number of block transfers of size B across all caches, assuming the caches are 'ideal' [13].

2. **S-CMP**. This is similar to D-CMP, except that there is a single cache of size $M \geq p \cdot B$ shared by all p processors [2, 8], where B is the block size. There continues to be a global shared-memory that is arbitrarily large, and performance is measured as in D-CMP.

3. **Multicore**. This models the trend in current CMPs: distributed L_1 caches (i.e., private to each core), each of size C_1, and a single shared L_2 cache of size $C_2 \geq p \cdot C_1$ [1]. There continues to be an arbitrarily large global shared memory. The cache complexity is specified by two parameters: the number of block transfers, each of size B_2, into L_2, and the total number of block transfers, each of size B_1, across all L_1 caches. One can consider a more general hierarchy of caches that are successively shared by larger groups of processors, and our results generalize to this model as well. We do not elaborate further on this.

We do not consider cache coherence protocols [17] since the updates that are performed in parallel in the algorithms we present are always on disjoint sets of data, and hence coherence is never invoked. We specify parallelism by parallel ***for*** loops and forking and joining through recursive calls. Our results continue to hold if we replace the ideal cache assumption by LRU.

Problem	I/O	Previous I_∞	Our I_∞	Tiling Parameters of Our Algorithms			
				Sequential	D-CMP	S-CMP	Multicore
LCS & PA	$\mathcal{O}\left(\frac{n^2}{MB}\right)$	n^2 (seq.)	$\mathcal{O}(n)$	$t[d]=2$, $\forall d$	$t[0]=p$, $t[d]=2,\ d>0$	$t[d]=2,\ d<r$, $r=\log(n/p)$, $t[r]=p$	$t[r]=p$, $r=\log(n/C_2)$, $t[d]=2,\ d\neq r$
Median	$\mathcal{O}\left(\frac{n^3}{B\sqrt{M}}\right)$	n^3 (seq.)	$\mathcal{O}(n)$	$t[d]=2$, $\forall d$	$t[0]=\sqrt{p}$, $t[d]=2,\ d>0$	$t[d]=2,\ d<r$, $r=\log(n/\sqrt{p})$, $t[r]=\sqrt{p}$	$t[r]=\sqrt{p}$, $r=\log(n/\sqrt{C_2})$, $t[d]=2,\ d\neq r$
Parenthesis		*					
GEP		$\mathcal{O}\left(n\log^2 n\right)$ [8, 1]					
RNA-SP	$\mathcal{O}\left(\frac{n^4}{B\sqrt{M}}\right)$	n^4 (seq.)	$\mathcal{O}(n)$	–	–	–	–

Table 1: Table of our results. Here the 'I/O' column lists the sequential cache-oblivious bound, and the I_∞ columns list the the number of parallel steps in a work-optimal parallel algorithm whose cache complexity matches the sequential bound. The tiling parameters are explained in the section for each problem. RNA-SP uses 3D-LDDP (similar to Median) and GEP as subroutines, and derives its tiling parameters from both. All results assume that the input is too large to fit into the available cache space, and some of them (i.e., Median, Parenthesis, GEP and RNA-SP) also assume that the cache is *tall* (i.e., $M=\Omega\left(B^2\right)$). * For the Parenthesis problem a cache-efficient parallel algorithm for the IBM Cyclops64 processor is given in [25].

1.2 On-line Schedules

We view the computation as a dynamic DAG that unfolds as the computation proceeds. Each node in the DAG represents a sequence of instructions being executed, and an edge from node u to node v indicates that computation at v uses a value computed at u. At any point in time, a node can be scheduled on a processor provided its predecessors in the DAG have been evaluated.

A sequential computation will compute in program order, which automatically satisfies the DAG constraints, and this is a depth-first topological sort of the DAG, called a *1DF schedule*. In the parallel context, we need a method to decide how the computation is distributed among the available processors as it unfolds. In *work-stealing* [4], the unfolding DAG is distributed across the processors, and an idle processor 'steals' some work from a random neighbor [4, 14]. Work-stealing has good performance for certain classes of algorithms under D-CMP. A *PDF* schedule assigns to an available processor, the node earliest in the 1DF schedule that is ready to be executed [3, 2], and it often has good performance under S-CMP. The recently proposed *controlled-PDF* scheduler [1] is a refined version of PDF that gives good cache performance for many divide and conquer algorithms under the Multicore caching model.

1.3 Overview of Our Technique

For each of the three classes of problems we consider, we introduce a *tiling sequence* $t[d],\ d\geq 0$, and our CMP algorithm is a parallel recursive algorithm that uses tiling parameter $t[d]$ at level d of the recursion. For each type of CMP (D-CMP, S-CMP, or Multicore) we specify the tiling parameters and then give a parallel schedule that ensures good performance with respect to both parallelism and cache-efficiency. The known cache-oblivious sequential algorithms for the problems (given in [7] for LCS and GEP, in [6] for LDDP, and in [5] for Parenthesis) can be viewed as special cases of our multicore algorithms, where the tiling parameter $t[d]$ is the same constant for all d (and where no parallelism is specified). Table 1 lists our results together with the tiling parameters used. Details are in the following sections.

2. LCS AND LDDP

Local Dependence DP (LDDP) includes a very large group of problems solvable by dynamic programming. The key feature of this class is that it applies to some constant number of dimensions $k\geq 2$, and it updates each position in a k-dimensional table by considering the previously computed values immediately adjacent to the current position, and using exactly one of those values to compute the value at the current position. Several LDDP problems are of practical importance, and we present experimental results in Section 5 for two such problems, *pairwise sequence alignment* and *3-way sequence alignment (or median)*, both with affine gaps. Here, for simplicity, we illustrate the LDDP method with arguably the simplest problem in the class, the two-dimensional LCS (or longest common subsequence) problem (see, e.g., [10]). Our results generalize to general k-dimensional LDDP (see [6] for a definition and sequential cache-oblivious algorithm for this problem.)

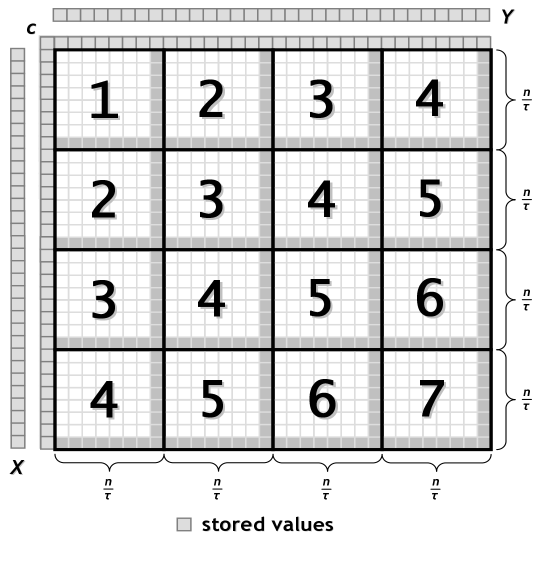
Given two sequences $X=x_1x_2\cdots x_n$ and $Y=y_1y_2\cdots y_n$, (for simplicity, we assume equal-length sequences here), an LCS of X and Y is a sequence of maximum length that is a subsequence of both X and Y. If we define $c[i,j]$, $0\leq i,j\leq n$, to be the length of an LCS of $x_1x_2\cdots x_i$ and $y_1y_2\cdots y_j$ then $c[n,n]$ is the LCS length of X and Y, and the $c[i,j]$'s can be computed by dynamic programming using the following recurrence relation (see, e.g., [10]):

$$c[i,j]=\begin{cases}0 & \text{if } i=0 \text{ or } j=0,\\ c[i-1,j-1]+1 & \text{if } i,j>0\wedge x_i=y_j,\\ \max\{\ c[i,j-1],\ c[i-1,j]\ \} & \text{if } i,j>0\wedge x_i\neq y_j.\end{cases} \tag{2.1}$$

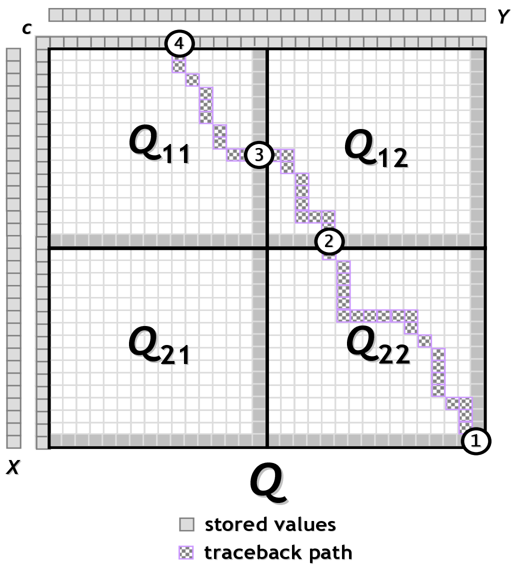
We denote by $parent(i,j)$, an adjacent cell whose c-value determines $c[i,j]$.

Typically, two types of outputs are expected when evaluating this recurrence: (i) the value of $c[n,n]$, and (ii) the traceback path starting from $c[n,n]$. The *traceback path* from any cell $c[i,j]$ is the path following the chain of *parent* cells through c that ends at some $c[i',j']$ with $i'=0\vee j'=0$.

Using equation 2.1 LCS can be solved by a simple $\mathcal{O}\left(n^2\right)$ time dynamic programming algorithm, whose DAG has linear critical path length ($T_\infty(n)=\Theta(n)$). Several refinement of this algorithm that optimize for space and cache-efficiency are known (e.g., [18, 7]), and the sequential algorithm in [7] that gives an $\mathcal{O}(n)$-space cache-oblivious algorithm with $\mathcal{O}\left(n^2/(MB)\right)$ cache misses has the best performance across both space and cache-efficiency.



(a)



(b)

Figure 2: The recursive tiled LCS algorithm: (a) Tiled-Boundary-LCS**: Given $\tau = t[recursion\ depth]$, the LCS table is decomposed into τ^2 sub-squares of equal size. In step $r \in [1, 2\tau-1]$, output boundaries of all sub-squares labelled with r are computed recursively in parallel. Outputs of step r act as inputs for step $r+1$. For each cell u on an output boundary we also compute the position on the input boundary at which the traceback path from u will enter. (b)** Tiled-LCS**: It first computes the output boundaries of Q_{11}, Q_{12}, Q_{21} and Q_{22} by calling** Tiled-Boundary-LCS **in sequence. Now given a location labelled 1 on the output boundary of the current table, it determines in $\mathcal{O}(1)$ time the locations at which the traceback path through location 1 intersects the input boundaries of all (at most three) quadrants (e.g., locations 2, 3 and 4 in the figure). Then it recursively calls itself in parallel on all quadrants hit by the traceback path, and extracts the path fragments from each of them.**

Our tiled LCS algorithm is illustrated in Figure 1.3. The algorithm has two parts, both of which are recursive. Figure 1.3(a) illustrates Tiled-Boundary-LCS, which computes the LCS costs for cells corresponding to the output boundary of the LCS table, i.e., the rightmost column and the bottom row, and additionally, for each such cell u, the cell on the input boundary at which the traceback path from u will enter. The algorithm is supplied the subsequences X' and Y' for the current recursive call, together with a positive integer τ, which is the tiling parameter for the current recursive call. Figure 1.3(a) is shown with $X' = X$ and $Y' = Y$ and with $\tau = 4$. The algorithm proceeds by dividing X and Y into τ equal pieces, thereby tiling the cost table with τ^2 sub-tables. These sub-tables are computed recursively in $2\tau - 1$ parallel steps, where the rth parallel step performs the computation for each sub-table along the rth forward diagonal. In the figure all sub-tables computed in step r, for $1 \le r \le 7$ are labelled with r. The specification of the algorithm includes the *tiling sequence* $t[d]$, $d \ge 0$, which is a sequence of integers, and at recursion level d the algorithm uses tiling parameter $t[d]$. These parameters will be optimized for the three different caching models we consider. Additionally, we obtain the sequential algorithm in [7] by using $t[d] = 2$ for all d. As in that sequential algorithm, the space requirement of our tiled algorithm is linear in the size of the input (even though a quadratic number of intermediate values are computed).

Figure 1.3(b) illustrates Tiled-LCS, which computes the trace-back path from a given cell on the output boundary. Tiled-LCS is again a recursive algorithm, and it also calls Tiled-Boundary-LCS. It takes the same inputs as Tiled-Boundary-LCS, together with one additional input u, which a cell on the output boundary of the current table. In figure 1.3(b), u is the bottom-right position in the table, and is indicated by 1. The algorithm proceeds by calling Tiled-Boundary-LCS on the four sub-tables Q_{11}, Q_{12}, Q_{21} and Q_{22}, which are derived from the current table considering two equal-sized halves of the two input strings. This computation generates for each output boundary cell x in each of these three sub-tables, the entry point on the input boundary of the sub-table of the traceback path from x. After this computation, the algorithm determines the cells 2, 3, and 4 shown in Figure 1.3(b), as we describe below.

Cell 2 in the figure is the position on the input boundary of Q_{22} where the traceback path from cell 1 meets. This cell could be on the output boundary of either Q_{12} or Q_{21}, and the figure has chosen this to be Q_{12} (as a result, the traceback path from cell 1 does not pass through Q_{21}). Cell 3 is the cell on the input boundary of Q_{12} where the traceback path from cell 2 meets; this cell will be either on the output boundary of Q_{11} or on the input boundary of the overall table (in which case there is no cell 4). The figure has illustrated the case when cell 3 is on the output boundary of Q_{11}, and in this case, cell 4 is the cell on the input boundary of Q_{11} where the traceback path from 3 meets. This cell is guaranteed to be in the input boundary of the overall table. Since the earlier computation of Tiled-Boundary-LCS has computed the corresponding traceback cell on the input boundary for each cell on the output boundary, cells 2, 3, and 4 can be determined by at most 3 look-up steps given cell 1, and hence this computation takes constant time and work.

Once the cells 2, 3 and (if needed) 4 of the traceback path are known, TILED-LCS recursively calls itself on the (at most three) subproblems that contain a portion of the traceback path, using the tiling parameter for next level of recursion.

The base case of the recursion occurs when the parent of cell 1 lies on the input boundary of the current table. Each parent cell is computed in such a base case call.

Correctness of this method is straightforward by induction on the input size. A succinct pseudocode for this algorithm is available in [9].

2.1 Performance Analysis

For performance, we tailor the tiling sequence to the CMP models as well as the sequential case as follows.

(1) Sequential algorithm in [7]. If we use $t[d] = 2$ for all $d \geq 0$, a 1-processor implementation of this multicore algorithm is exactly the sequential cache-oblivious algorithm in [7] and hence its I/O complexity is $\mathcal{O}\left(\frac{n^2}{MB}\right)$, where M and B are the cache and block sizes respectively.

(2) p-processor D-CMP. Since this model has a private cache for each core, we use a tiling sequence that gives the largest amount of locality to each processor, so that the private caches can be used most effectively. For this we use $t[0] = p$ and $t[d] = 2$ for $d \geq 1$. Note that this results in a cache-oblivious D-CMP algorithm.

The parallel schedule consists of assigning the ith column of sub-tables (see figure 1.3) to the ith processor at the top level recursion ($d = 0$). Further levels of recursion are executed entirely on the processor that was assigned the subproblem at level 0. On an input of size n and with p processors, the parallel running time for TILED-BOUNDARY-LCS is $T_B(n,p) = \mathcal{O}\left((\frac{n}{p})^2 \cdot (2p-1) + n\right) = \mathcal{O}\left(n^2/p + n\right)$ since there are $2p - 1$ parallel steps, each executing the sequential algorithm on an input of size n/p. There are p^2 subproblems of size n/p, each executing the sequential cache-oblivious algorithm, hence the number of cache misses is $\mathcal{O}\left(\left(\frac{(n/p)^2}{MB} + \frac{n/p}{B} + 1\right) \cdot p^2\right) = \mathcal{O}\left(\frac{n^2}{MB}\right)$ under the natural assumptions that $n \geq pM$ (i.e., the input does not fit within the caches), and block-size $B \leq n/p$ (which will hold since $B \leq M$).

For TILED-LCS, we schedule the (up to) 3 recursive subproblems with $p/3$ processors each, hence the parallel running time $T(n,p)$ is given by $T(n,p) = \mathcal{O}\left(n^2/p + n\right) + T(n/2, p/3)$, which remains $\mathcal{O}\left(n^2/p + n\right)$. By a similar analysis the cache complexity also remains the same as for TILED-BOUNDARY-LCS. Hence we obtain $I_\infty(n) = \Theta(n)$.

(3) p-processor S-CMP. Here all processors share a single cache, hence we choose a tiling schedule that causes the processors to work on parallel tasks that are close to one another in the sequential order. For this, we use $t[d] = 2$ for $d < \log(n/p)$ and $t[\log_2(n/p)] = p$. This is again a cache-oblivious strategy.

The parallel schedule consists of a Brent-type processor allocation (see, e.g., [19]) at recursion level $\log n/p$. At this level of recursion the input has two substrings of length p and hence we are applying parallelism at the finest level of granularity. Since the shared cache can be expected to have size $M \geq p$ this computation will have the same cache-complexity as the sequential case. There are $\Theta\left(\frac{n^2}{p^2}\right)$ subproblems executed, each with parallel time $\Theta(p)$, hence the parallel running time is $\mathcal{O}\left(p \cdot (n^2/p^2) + n\right)$ and hence remains $\mathcal{O}\left(n^2/p + n\right)$. The analysis of TILED-LCS is similar to that for D-CMP, and again we have cache-efficient parallelism I_∞ to be $\Theta(n)$.

(4) Multicore. Here we need to adapt to the the private L_1 caches of size C_1 and the shared L_2 cache of size C_2. Hence we use a strategy that combines our approach for S-CMP and D-CMP. Let $r = \log \frac{n}{C_2}$. We use $t[r] = p$ and $t[d] = 2$ for all $d \neq r$.

As in D-CMP, at recursion level r we assign the ith column of sub-tables to the ith processor, $1 \leq i \leq p$. There are $\frac{n^2}{C_2^2}$ subproblems, each with two input strings of length C_2. Each such subproblem is solved similar to the D-CMP case, hence the parallel time is $\mathcal{O}\left(\frac{n^2}{C_2^2} \cdot (2p-1) \cdot (C_2/p)^2 + n\right) = \mathcal{O}\left(n^2/p + n\right)$. The C_2 cache complexity is clearly the sequential complexity $\mathcal{O}\left(n^2/(C_2 B)\right)$ since all parallelism is exposed at a problem size when the entire input fits into the L_2 cache. Since each subproblem whose input size is C_2 is solved with private caches using the same method as in D-CMP, the L_1 cache complexity is $\mathcal{O}\left(\frac{n_2^2}{C_2^2} \cdot \frac{C_2^2}{C_1 B}\right)$, which is $\mathcal{O}\left(n^2/(C_1 B)\right)$. Hence this give a Multicore LCS algorithm with cache-efficient parallelism $I_\infty(n) = \Theta(n)$.

3. GEP (GAUSSIAN ELIMINATION PARADIGM) PROBLEMS

Let $c[1 \ldots n, 1 \ldots n]$ be an $n \times n$ matrix with entries chosen from an arbitrary set $\mathcal{S}$, and let $f : \mathcal{S} \times \mathcal{S} \times \mathcal{S} \times \mathcal{S} \rightarrow \mathcal{S}$ be an arbitrary function. By GEP (or the *Gaussian Elimination Paradigm*) introduced by the authors in [7], we refer to the computation in Figure 3. Here the algorithm G modifies c by applying a given set of updates of the form $c[i,j] \leftarrow f(c[i,j], c[i,k], c[k,j], c[k,k])$, where $i,j,k \in [1,n]$. We use the notation $\langle i,j,k \rangle$ ($1 \leq i,j,k \leq n$) denotes an update of the form $c[i,j] \leftarrow f(c[i,j], c[i,k], c[k,j], c[k,k])$, and we let Σ_G denote the set of such updates that the algorithm needs to perform.

As noted in [7] many practical problems can be solved using the GEP construct, including all-pairs shortest paths, LU decomposition and Gaussian elimination without pivoting, and matrix multiplication.

An $\mathcal{O}\left(\frac{n^3}{B\sqrt{M}}\right)$ I/O cache-oblivious sequential algorithm for solving some important special cases of GEP including all problems mentioned above was presented by the authors in [7]. Later in [8], this implementation was named I-GEP, extended to C-GEP which solves all instances of GEP within the same performance bounds, and was also parallelized to run in $\mathcal{O}\left(\frac{n^3}{p} + n\log^2 n\right)$ time on p processors with schedulers to match its sequential cache complexity on D-CMP and S-CMP. Recently in [1] we presented a scheduler to run I-GEP and C-GEP on multicores within the same performance bounds. In this section we improve the parallel algorithm presented in [8] to run in $\mathcal{O}\left(\frac{n^3}{p} + n\right)$ parallel time on D-CMP, S-CMP and multicores while matching its sequential cache complexity. In other words, the new algorithm achieves $I_\infty(n) = T_\infty(n) = \Theta(n)$.

```
G(c, n, f, Σ_G)
(Input $c[1 \ldots n, 1 \ldots n]$ is an $n \times n$ matrix, $f(\cdot,\cdot,\cdot,\cdot)$ is an arbitrary problem-specific function, and $\Sigma_G$ is a problem-specific set of triplets such that $c[i,j] \leftarrow f(c[i,j], c[i,k], c[k,j], c[k,k])$ is executed in line 4 if $\langle i,j,k \rangle \in \Sigma_G$.)

1. for $k \leftarrow 1$ to $n$ do
2.     for $i \leftarrow 1$ to $n$ do
3.         for $j \leftarrow 1$ to $n$ do
4.             if $\langle i,j,k \rangle \in \Sigma_G$ then $c[i,j] \leftarrow f(c[i,j], c[i,k], c[k,j], c[k,k])$
```

Figure 3: GEP: Triply nested *for* loops typifying code fragment with structural similarity to the computation in Gaussian elimination without pivoting.

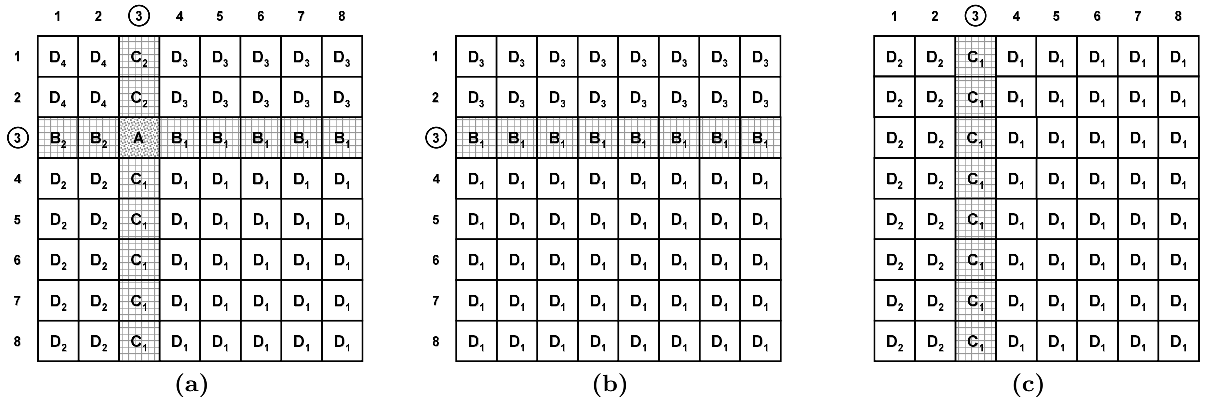


Figure 4: Execution of superstep $k = 3$ of I-GEP functions A (Figure 4(a)), B$_1$ (Figure 4(b)) and C$_1$ (Figure 4(c)). Each cell contains the name of the function used to update the entries in the corresponding submatrix. Submatrices corresponding to dotted cells (if any) are updated first. In the next step all submatrices corresponding to cells with grids are updated in parallel. In the last step submatrices corresponding to white cells are updated simultaneously. Pseudocode for these functions can be found in [9].

3.1 Improved CMP Algorithm

We present an improved parallel implementation of I-GEP which is an extension of the parallel implementation we presented in [8]. We modify each I-GEP function (A, B$_i$, C$_i$, D$_j$, where $i \in [1,2]$ and $j \in [1,4]$) introduced in [8] so that one can control the way the input matrices are subdivided at each level of recursion by specifying a vector t of tiling parameters. At recursion depth $d \geq 0$, an $n \times n$ input matrix is subdivided into $r \times r$ submatrices of size $\frac{n}{r} \times \frac{n}{r}$ each, where $r = t[d]$. We show that by appropriately setting the tiling parameters we can reduce the parallel time complexity of the algorithm from $\mathcal{O}\left(\frac{n^3}{p} + n\log^2 n\right)$ [8] to $\mathcal{O}\left(\frac{n^3}{p} + n\right)$ for p processors. Pseudocode for each of these functions is given in the companion technical report [9].

Each function accepts a (recursion) depth parameter $d \geq 0$, and four (not necessarily distinct) equal size square submatrices X, U, V and W of the input matrix c. We assume that W has a diagonal aligned to the $(1,1)$ to (n,n) diagonal of c, and for each $c[i,j] \in X$ and $c[k,k] \in W$, the entries $c[i,k]$ and $c[k,j]$ can be found in U and V, respectively. Each function updates the entries in X using appropriate entries from U, V and W. The functions differ in the amount of overlap X, U and V have among them. Function A assumes that all three matrices completely overlap, while D$_l$ ($l \in [1,4]$) expects completely non-overlapping matrices. In the intermediate cases, function B$_l$ assumes that only X and V overlap, while C$_l$ assumes overlap only between X and U, where $l \in [1,2]$. Intuitively, the less the overlap among the input matrices the more flexibility the function has in ordering its recursive calls, and thus leading to better parallelism.

The initial call is to function A with $d = 0$ and $X = U = V = W = c$. If X is a 1×1 matrix, then X is updated directly by setting $X \leftarrow f(\ X,\ U,\ V,\ W\ )$. Otherwise each matrix is subdivided into $r \times r$ submatrices of size $\frac{n}{r} \times \frac{n}{r}$ each, where $r = t[d] = t[0]$. The submatrix of X at the i-th position from the top and the j-th position from the left is denoted by $X_{i,j}$. Then the function executes r supersteps. Superstep $k \in [1,r]$ of function A consists of 3 steps (see Figure 4(a)). In step 1, submatrix $X_{k,k}$ is updated recursively by function A. In step 2, the remaining submatrices $X_{i,k}$ and $X_{k,j}$ $(i \neq k, j \neq k)$ are updated in parallel using the entries in $X_{k,k}$ by appropriate calls to functions B_1, B_2, C_1 and C_2 (see Figure 4(a) for details). In step 3, the remaining submatrices of X are updated in parallel using entries computed in step 2 by calling appropriate D$_l$ ($l \in [1,4]$) functions (see Figure 4(a)). Superstep $k \in [1,r]$ of functions B$_l$ and C$_l$ ($l \in [1,2]$) consists of 2 steps. In case of function B$_l$ (C$_l$), step 1 updates all $X_{i,k}$ ($X_{k,j}$, resp.) by parallel calls to B$_l$ (C$_l$, resp.). In step 2, the remaining submatrices of X are updated by parallel calls to appropriate D$_l$ ($l \in [1,4]$) functions (e.g., see Figures 4(b) and 4(c)). Each superstep of function D$_l$ ($l \in [1,4]$) has only one step which updates all submatrices of X in parallel by calling D$_l$ recursively.

We now analyze the performance of the algorithm under different configurations below.

(1) Sequential. We use $t[d] = 2$ for all $d \geq 0$, and execute the entire algorithm on a single processor. We have already analyzed this case in [7, 8], and found the cache complexity of the algorithm to be $\mathcal{O}\left(n^3/(B\sqrt{M}) + n^3/M + n^2/B + 1\right)$ which reduces to $\mathcal{O}\left(n^3/(B\sqrt{M})\right)$ provided the cache is tall (i.e., $M = \Omega\left(B^2\right)$) and the input matrix is too large to fit into the cache (i.e., $n^2 > M$). It is straight-forward to see that the algorithm runs in $\mathcal{O}\left(n^3\right)$ time.

(2) **D-CMP.** We set $t[d] = \sqrt{p}$ for $d = 0$, and $t[d] = 2$ for $d > 0$. Hence at level 0 of recursion, function A will execute $\sqrt{p}$ supersteps, and in each superstep it will make a total of p parallel recursive functions calls with submatrices of size $(n/\sqrt{p}) \times (n/\sqrt{p})$ each. Each such function will be executed sequentially in $\mathcal{O}\left(\left(n/\sqrt{p}\right)^3\right)$ time on a single processor. Thus the parallel running time of the algorithm is $\sqrt{p} \cdot \mathcal{O}\left(\left(n/\sqrt{p}\right)^3\right) = \mathcal{O}\left(n^3/p\right)$. Now let $Q_{\mathrm{A}}(n)$, $Q_{\mathrm{BC}}(n)$ and $Q_{\mathrm{D}}(n)$ denote the cache complexity of functions A, $\mathrm{B_1/B_2/C_1/C_2}$ and $\mathrm{D_1/D_2/D_3/D_4}$, respectively, on an input of size n. Assuming that a submatrix of size $(n/\sqrt{p}) \times (n/\sqrt{p})$ is too large to fit into the cache, and that the cache is tall, the total number of cache misses is $\sqrt{p} \cdot Q_{\mathrm{A}}(n/2) + \sqrt{p} \cdot (2\sqrt{p} - 1) \cdot Q_{\mathrm{BC}}(n/2) + \sqrt{p} \cdot (p - 2\sqrt{p}) \cdot Q_{\mathrm{D}}(n/2) = \mathcal{O}\left(n^3/(B\sqrt{M}) + n^3/M + \sqrt{p} \cdot n^2/B + p\sqrt{p}\right) = \mathcal{O}\left(n^3/(B\sqrt{M})\right)$.

(3) **S-CMP.** We set $t[d] = \sqrt{p}$ if $d \geq \log_2 \frac{n}{\sqrt{p}}$, and $t[d] = 2$ otherwise. We do not make any parallel function calls until we reach a level $d \geq \log_2 \frac{n}{\sqrt{p}}$. Hence, we can use the same recurrence relations as in the sequential case and get the same cache complexity provided $p \leq M$. Now $T_{\mathrm{A}}(n)$, $T_{\mathrm{BC}}(n)$ and $T_{\mathrm{D}}(n)$ denote the parallel running times of functions A, $\mathrm{B_1/B_2/C_1/C_2}$ and $\mathrm{D_1/D_2/D_3/D_4}$, respectively on an input of size n. Then $T_{\mathrm{A}}(n) = T_{\mathrm{BC}}(n) = T_{\mathrm{D}}(n) = \mathcal{O}\left(\sqrt{p}\right)$ if $n \leq \sqrt{p}$. Otherwise,

$$T_{\mathrm{D}}(n) = 8 \cdot T_{\mathrm{D}}(n/2) + \mathcal{O}(1)$$
$$T_{\mathrm{BC}}(n) = 4 \cdot T_{\mathrm{BC}}(n/2) + 4 \cdot T_{\mathrm{D}}(n/2) + \mathcal{O}(1)$$
$$T_{\mathrm{A}}(n) = 2 \cdot T_{\mathrm{A}}(n/2) + 4 \cdot T_{\mathrm{BC}}(n/2) + 2 \cdot T_{\mathrm{D}}(n/2) + \mathcal{O}(1)$$

Solving the recurrences and assuming that $p \leq n^2$, we obtain $T_{\mathrm{A}}(n) = T_{\mathrm{BC}}(n) = T_{\mathrm{D}}(n) = \mathcal{O}\left(n^3/p\right)$.

(4) **Multicore.** We set $t[d] = \sqrt{p}$ if $d = \log_2(n/\sqrt{C_2})$, and $t[d] = 2$ otherwise. All parallel calls are made only at level $d = \log_2(n/\sqrt{C_2})$. Observe that whenever we reach a subproblem of input size $n_1 = \sqrt{C_2/p}$, it is executed entirely on a single processor, and there are $(n/n_1)^3$ such subproblems. Hence, the number of L_1 cache-misses is $(n/n_1)^3 \cdot \mathcal{O}\left({n_1}^3/(B_1\sqrt{C_1}) + {n_1}^3/C_1 + {n_1}^2/B_1 + 1\right)$, which reduces to $\mathcal{O}\left(n^3/(B_1\sqrt{C_1})\right)$ under the assumption that $C_2 > p \cdot C_1$ ($> p \cdot B_1$), and that the L_1 cache is tall (i.e., $C_1 = \Omega\left({B_1}^2\right)$). The number of L_2 cache-misses can be computed using the same recurrence relations as in the sequential case and in **S-CMP**, and thus the L_2 cache complexity is $\mathcal{O}\left(n^3/(B_2\sqrt{C_2})\right)$ provided the L_2 cache is tall (i.e., $C_2 = \Omega\left({B_2}^2\right)$), and the input matrix is too large to fit into that cache (i.e., $n^2 > C_2$). The parallel running time can be computed using the same recurrence relations as in **S-CMP**, but with a different base condition. We use $T_{\mathrm{A}}(n) = T_{\mathrm{BC}}(n) = T_{\mathrm{D}}(n) = \mathcal{O}\left(\sqrt{p} \cdot \left(\sqrt{C_2/p}\right)^3\right)$ for $n \leq \sqrt{C_2}$. Assuming $n^2 > C_2$, we obtain $T_{\mathrm{A}}(n) = T_{\mathrm{BC}}(n) = T_{\mathrm{D}}(n) = \mathcal{O}\left(n^3/p\right)$.

Observe that for all three models we obtain $I_\infty(n) = \Theta(n)$. For D-CMP and S-CMP models, our algorithm is cache-oblvious.

4. THE PARENTHESIS PROBLEM

We consider the *parenthesis problem* [15] which is defined by the following recurrence relation:

$$c[i,j] = \begin{cases} x_j & \text{if } 0 \leq i = j-1 < n, \\ \min_{i<k<j} \left\{ \begin{matrix} (c[i,k] + c[k,j]) \\ +w(i,k,j) \end{matrix} \right\} & \text{if } 0 \leq i < j-1 < n; \end{cases} \tag{4.2}$$

where x_j's are assumed to be given for $j \in [1,n]$. We also assume that $w(\cdot,\cdot,\cdot)$ is a function that can be computed in-core without incurring any cache misses.

The class of problems defined by the recurrence relation above includes RNA secondary structure prediction, optimal matrix chain multiplication, construction of optimal binary search trees, and optimal polygon triangulation. A variant of this recurrence which does not include the $w(i,k,j)$ term and is defined as the *simple dynamic program*, was considered in [5], where an $\mathcal{O}\left(n^3/(B\sqrt{M})\right)$ I/O sequential cache-oblivious algorithm based on Valiant's context-free language recognition algorithm [27] was given for solving the recurrence. A parallel algorithm for solving the parenthesis problem which runs in $\mathcal{O}\left(n^{\frac{3}{4}} \log n\right)$ time and performs optimal $\mathcal{O}\left(n^3\right)$ work was given in [15], but the algorithm is not cache-efficient. A cache-efficient multicore algorithm for the IBM Cyclops64 processor was given in [25].

As in [5], instead of recurrence 4.2 we will use the following slightly augmented version of 4.2 which will considerably simplify the recursive subdivision process in our cache-oblivious algorithm.

$$c[i,j] = \begin{cases} \infty & \text{if } 0 \leq i = j \leq n, \\ x_j & \text{if } 0 \leq i = j-1 < n, \\ \min_{i \leq k \leq j} \left\{ \begin{matrix} (c[i,k] + c[k,j]) \\ +w(i,k,j) \end{matrix} \right\} & \text{if } 0 \leq i < j-1 < n; \end{cases} \tag{4.3}$$

where $w(i,k,j)$ is defined to be ∞ when $k = i$ or $k = j$. It is straight-forward to see that recurrences 4.2 and 4.3 are equivalent, i.e., they compute the same values for any given $c[i,j]$, $0 \leq i < j-1 < n$.

4.1 CMP Algorithm

We assume that the input to the algorithm is an $n \times n$ matrix c that has all $c[i,j]$ with $0 \leq i \leq j \leq i+1 \leq n$ initialized as in recurrence 4.3, and the remaining entries (i.e., all $c[i,j]$ with $0 \leq i < j-1 < n$) initialized to ∞. The algorithm works by recursively subdividing the input matrix, and assumes the existence of a global vector t of tiling parameters which for each level of recursion specifies how the input matrix is subdivided. Pseudocode for the algorithm is given in the companion technical report [9]. Here we give an informal description of the algorithm.

The algorithm (i.e., function E in [9]) splits the input matrix X (which is initially set to c) into $r \times r$ submatrices of size $\frac{n}{r} \times \frac{n}{r}$ each, where $r = t[d]$ and d (≥ 0) is the level (i.e., depth) of recursion. The submatrix of X at the i-th position from the top and the j-th position from the left is denoted by $X_{i,j}$. The k-th diagonal of X includes the submatrices $X_{i,i+k}$, where $k \in [0, n-1]$ and $i \in [1, n-k]$ (see Figure 5). Observe that the submatrices on diagonal 0 of X are smaller instances of the original parenthesis problem defined by X and all of them are independent, and hence are solved recursively by $\frac{n}{r}$ parallel calls to E. The algorithm

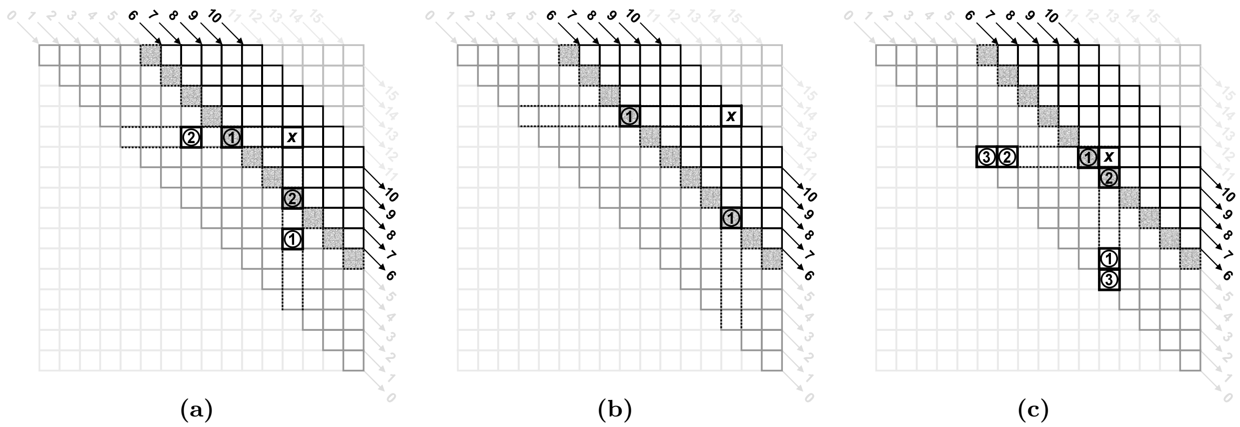


Figure 5: Execution of superstep $k = 6$ of function E for solving the parenthesis problem. Figure 5(a) marks the submatrices used in steps 1 and 2 to update a submatrix x on diagonal $k' \in [k+1, 2k-3]$. Figure 5(b) shows that if x is on diagonal $2k-2$ then it is updated only in step 1, and as shown in Figure 5(c) any submatrix on diagonal k is updated in all three steps (submatrices used in step $r \in [1,3]$ are labelled with r).

then executes $r-1$ supersteps, and maintains the following invariant.

INVARIANT 4.1. *At the start of superstep $k \in [1, r-1]$,*

(i) all entries of c in the submatrices on diagonals 0 to $k-1$ of X have their values finalized, and

(ii) all remaining cells $c[i,j]$ are updated with $(c[i,l] + c[l,j]) + w(i,l,j)$, where both $c[i,l]$ and $c[l,j]$ belong to the submatrices on diagonals 0 to $k-2$ of X.

Superstep k ($1 \le k < r$) consists of up to 3 parallel steps. The first two steps are executed only for $k > 1$. In step 1 each submatrix $X_{i,j}$ on diagonals k to $\min\{2k-2, r-1\}$ is updated using data from the submatrix $X_{i,r+k-i}$ on diagonal $k-1$ that lies to the left of $X_{i,j}$, and the corresponding submatrix $X_{i-k+1,j}$ below $X_{i,j}$. All such $X_{i,j}$'s are updated in parallel using another recursive function H (see the technical report [9] for pseudocode) which is implemented in exactly the same way as the I-GEP function D_3. Similarly, step 2 updates each submatrix $X_{i,j}$ on diagonals k to $\min\{2k-3, r-1\}$ using data from the submatrix $X_{r+k-j,j}$ on diagonal $k-1$ that lies below $X_{i,j}$, and the corresponding submatrix $X_{i,j-k+1}$ to the left of $X_{i,j}$. All such updates are also performed in parallel by calling H. Observe that in step 1, for $X_{i,j}$'s on diagonal $2k-2$ both input submatrices lie on diagonal $k-1$. Hence, submatrices on diagonal $2k-2$ need not be updated in step 2. In step 3 each submatrix $X_{i,j}$ on diagonal k is updated using the two submatrices on diagonal 0 that lie to the left of $X_{i,j}$ and below it, i.e., $X_{i,r-i+1}$ and $X_{r-j+1,j}$, respectively. All these updates are performed in parallel by calling function F. The implementation of this function is similar to that of function E (see the technical report [9] for pseudocode). Since invariant 4.1 was true before the start of superstep k, it is easy to see that after step 3 of this superstep all entries of c inside submatrices on diagonal k will have their values finalized.

Now we analyze the performance of the algorithm under different machine configurations.

(1) Sequential. We use $t[d] = 2$ for all $d \ge 0$, and execute the entire algorithm on a single processor. In this case the algorithm reduces to the sequential algorithm given in [5]. It is straight-forward to see that the running time of each of the three functions (i.e., E, F and H) is $\mathcal{O}(n^3)$. Let the sequential cache-complexity of the three functions on an input of size n be $Q_E(n)$, $Q_F(n)$ and $Q_H(n)$, respectively. Then $Q_E(n) = Q_F(n) = Q_H(n) = \mathcal{O}(n + n/B)$ for $n^2 \le \gamma M$, where γ is a suitable constant. Otherwise,

$$Q_H(n) = 8 \cdot Q_H(n/2), \quad Q_F(n) = 4 \cdot Q_F(n/2) + 4 \cdot Q_H(n/2)$$
$$\text{and} \quad Q_E(n) = 2 \cdot Q_E(n/2) + Q_F(n/2)$$

Solving the recurrences we obtain, $Q_E(n) = Q_F(n) = Q_H(n) = \mathcal{O}\left(n^3/(B\sqrt{M}) + n^3/M + n^2/B + 1\right)$ which reduces to $\mathcal{O}\left(n^3/(B\sqrt{M})\right)$ provided the cache is tall and the input matrix is too large to fit into the cache.

(2) D-CMP. We use the same tiling parameters as in the **D-CMP** case of I-GEP in Section 3.1, and the analyses of performance bounds are also similar. The parallel running time of the algorithm turns out to be $\mathcal{O}(n^3/p)$, and the cache complexity $\mathcal{O}\left(n^3/(B\sqrt{M}) + n^3/M + \sqrt{p} \cdot n^2/B + p\sqrt{p}\right)$, which reduces to $\mathcal{O}\left(n^3/(B\sqrt{M})\right)$ provided the input is too large to fit into the cache (i.e., $n^2 > p \cdot M$) and the cache is tall (i.e., $M = \Omega(B^2)$).

(3) S-CMP. We use the same tiling parameters as in the **S-CMP** case of I-GEP in Section 3.1, and compute the cache complexity similarly. Let $T_E(n)$, $T_F(n)$ and $T_H(n)$ be the parallel running times of E, F and H, respectively, on input size n. Then $T_E(n) = T_F(n) = T_H(n) = \mathcal{O}(\sqrt{p})$ if $n \le \sqrt{p}$. Otherwise,

$$T_H(n) = 8 \cdot T_H(n/2) + \mathcal{O}(1)$$
$$T_F(n) = 4 \cdot T_F(n/2) + 4 \cdot T_H(n/2) + \mathcal{O}(1)$$
$$T_E(n) = 2 \cdot T_E(n/2) + T_F(n/2) + \mathcal{O}(1)$$

Solving the recurrences and assuming that $p \le n^2$, we obtain $T_E(n) = T_F(n) = T_H(n) = \mathcal{O}(n^3/p)$. The cache complexity remains the same as in the sequential case.

(4) Multicore. The analysis is exactly the same (using the same tiling sequence) as in the multicore case of I-GEP in Section 3, and we get $T_E(n) = T_F(n) = T_H(n) = \mathcal{O}(n^3/p)$. The cache complexities for both the L_1 and L_2 caches match its sequential cache complexity.

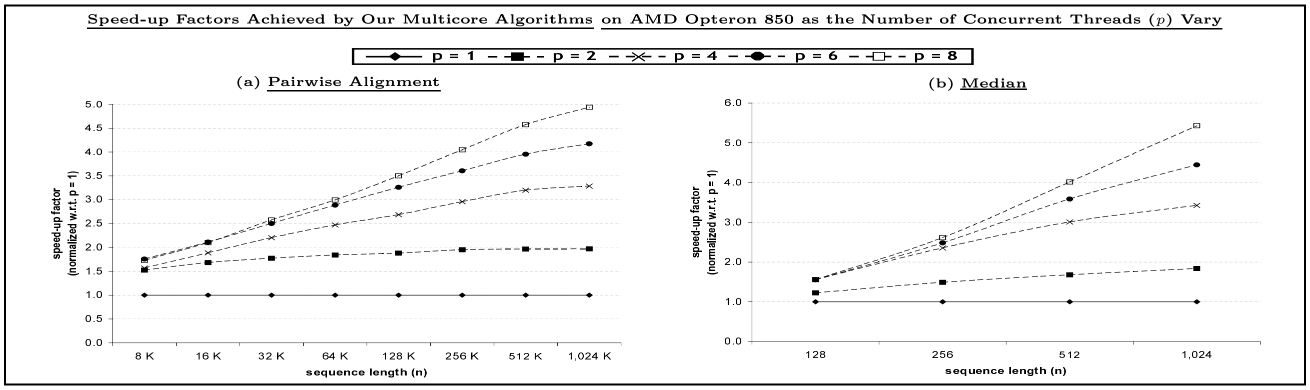


Figure 6: Performance of algorithms for PA and Median on randomly generated sequences over { A, T, G, C }.

Seq. Pair	Lengths ($\times 10^6$)	Cost ($\times 10^3$)	FASTA [22]	Multicore PA				
				1 core	2 cores	4 cores	6 cores	8 cores
human / baboon	1.80|1.51	689	$21h$ $43m$ (5.87)	$17h$ $41m$ (4.79)	$8h$ $58m$ (2.43)	$5h$ $26m$ (1.47)	$4h$ $21m$ (1.17)	$3h$ $42m$ (1.00)
human / chimp	1.80|1.32	585	$20h$ $15m$ (6.34)	$14h$ $28m$ (4.53)	$7h$ $20m$ (2.30)	$4h$ $28m$ (1.40)	$3h$ $32m$ (1.11)	$3h$ $12m$ (1.00)
baboon / chimp	1.51|1.32	574	$17h$ $57m$ (6.22)	$13h$ $0m$ (4.51)	$6h$ $36m$ (2.29)	$4h$ $3m$ (1.40)	$3h$ $16m$ (1.13)	$2h$ $52m$ (1.00)
human / rat	1.80|1.50	1,143	$27h$ $55m$ (6.15)	$21h$ $47m$ (4.80)	$11h$ $2m$ (2.43)	$6h$ $37m$ (1.46)	$5h$ $11m$ (1.14)	$4h$ $32m$ (1.00)
rat / mouse	1.50|1.49	822	$18h$ $39m$ (5.31)	$15h$ $42m$ (4.47)	$7h$ $58m$ (2.27)	$4h$ $55m$ (1.40)	$3h$ $54m$ (1.11)	$3h$ $31m$ (1.00)

Table 2: Performance of pairwise alignment algorithms on 8-core AMD Opteron 850 on CFTR DNA sequences [26]. Parameters used: gap open cost = 2, gap extension cost = 1, mismatch cost = 1. Each number outside parentheses in columns 4–9 is the time for a single run, and the ratio of that running time to the corresponding running time for Multicore PA with 8 cores is given within parentheses.

No	Lengths	Cost	Knudsen [20]	ukk.alloc [23]	ukk.checkp [23]	Multicore Median				
						1 core	2 cores	4 cores	6 cores	8 cores
1	367|387|388	299	1,061 (6.93)	396 (2.59)	548 (3.58)	431 (2.82)	274 (1.79)	181 (1.18)	158 (1.03)	153 (1.00)
2	378|388|403	324	1,136 (7.01)	–	707 (4.36)	460 (2.84)	298 (1.84)	190 (1.17)	173 (1.07)	162 (1.00)
3	342|367|389	339	936 (6.37)	–	795 (5.41)	388 (2.64)	256 (1.74)	166 (1.13)	152 (1.03)	147 (1.00)
4	342|370|474	432	1,154 (7.08)	–	1,595 (9.79)	464 (2.85)	281 (1.72)	195 (1.20)	175 (1.07)	163 (1.00)
5	370|388|447	336	–	–	768 (4.74)	494 (3.05)	307 (1.90)	204 (1.25)	170 (1.05)	162 (1.00)

Table 3: Performance on an 8-core AMD Opteron 850 of Median algorithms on triples of random sequences from 16S bacterial rDNA sequences from the Pseudanabaena group [12]. Parameters used: gap open cost = 2, gap extension cost = 1, mismatch cost = 1. A '–' in a column denotes that the corresponding algorithm could not be run due to high space overhead. Each number outside parentheses in columns 4–11 is the time in seconds for a single run, and the ratio of that running time to the corresponding running time for Multicore Median with 8 cores is given within parentheses.

Observe that similar to the results for I-GEP in Section 3, this algorithm acheives $I_\infty(n) = \mathcal{O}(n)$ for all three CMP models, and is cache-oblivious for D-CMP and S-CMP.

5. EXPERIMENTAL RESULTS

We ran experiments for PA *(pairwise global sequence alignment with affine gap penalty)* and for Median *(median of 3 sequences, again with affine gap penalty)*. Definitions for PA and Median problems as 2- and 3-dimensional LDDP respectively can be found in [9]. Our sequential cache-oblivious algorithms and experimental results for them are given in [6]. We ran our experiments on an 8-core 2.2 GHz AMD Opteron 850, with cache sizes 64KB and 1 MB (8-way) for L1 and L2 caches, 32 GB RAM.

Since we were dealing with only $p = 8$ cores, we used a simple CMP algorithm with tiling parameter $t[d] = 2$ for all d, which is the same tiling sequence used by the sequential algorithm in [7]. This gives rise to parallelism $T_\infty(n) = \mathcal{O}\left(n^{\log_2 3}\right)$ since it satisfies the recurrence $T_\infty(n) = 3 \cdot T_\infty(n/2) + \mathcal{O}(1)$, with $T_\infty(1) = \mathcal{O}(1)$. Using techniques similar to those we have described in section 2 for LCS on D-CMP, S-CMP and Multicore, it is straightforward to obtain schedules that achieve $I_\infty(n) = \mathcal{O}\left(n^{\log_2 3}\right)$ on all three models. However, since our experiments were actual runs on an existing 8-core machine and not simulations, we had no control over the scheduler, so our parallel code was run with the default Linux thread scheduler.

Our algorithms were implemented in C++ (compiled with *g++* 3.3.4) while some software packages we used for comparison were written in C (compiled with *gcc* 3.3.4). Optimization parameter -O3 was used in all cases.

For PA we compared our code (PA-CO) with FASTA *(fasta2)* [22], and for median we compared our code (Median-CO) with Knudsen [21], and with ukk.alloc and ukk.checkp, both from Powell et al. [24]. These are all well-known software, and FASTA especially is very widely used. None of the code we used for comparison were designed for parallelism, but in all cases even the 1-core version of our parallel code ran faster than the all compared code on random triples from the data set.

Experimental Performance. Our experimental results for random input strings in Figure 6 show that both PA-CO and Median-CO achieve good speed-up as the number of processors increases. For example, with 8 processors PA-CO achieves a speed-up factor of about 5 when $n = 1024$ K, and MED-CO achieves speed-up 5.5 when $n = 1,024$.

Tables 2 and 3 present a sample of runs on real data, both on our code and the other software. As seen from the tables, the 1-core runs of our code are faster than the software we compare against. The speed-up for these real data samples as we increase the number of cores is similar to our results for random input strings in Figure 6.

Acknowledgement. We acknowledge the suggestions of the SPAA referees to give informal descriptions of our algorithms and remove all pseudocode.

6. REFERENCES

[1] G. Blelloch, R. Chowdhury, P. Gibbons, V. Ramachandran, S. Chen, and M. Kozuch. Provably good multicore cache performance for divide-and-conquer algorithms. In *Proc. ACM-SIAM SODA*, pages 501–510, 2008.

[2] G. Blelloch and P. Gibbons. Effectively sharing a cache among threads. In *Proc. ACM SPAA*, pages 235–244, 2004.

[3] G. Blelloch, P. Gibbons, and Y. Matias. Provably efficient scheduling for languages with fine-grained parallelism. *JACM*, 46(2):281–321, 1999.

[4] R. Blumofe and C. Leiserson. Scheduling multithreaded computations by work stealing. *JACM*, 46(5):720–748, 1999.

[5] C. Cherng and R. Ladner. Cache efficient simple dynamic programming. In *Proc. Intl Conf Analysis of Algorithms*, pages 49–58, 2005.

[6] R. Chowdhury, H. Le, and V. Ramachandran. Efficient cache-oblivious string algorithms for Bioinformatics. Technical Report TR-07-03, Dept. of Computer Sciences, UT-Austin, 2007.

[7] R. Chowdhury and V. Ramachandran. Cache-oblivious dynamic programming. In *Proc. ACM-SIAM SODA*, pages 591–600, 2006.

[8] R. Chowdhury and V. Ramachandran. The cache-oblivious gaussian elimination paradigm: Theoretical framework, parallelization and experimental evaluation. In *Proc. ACM SPAA*, pages 71–80, 2007.

[9] R. Chowdhury and V. Ramachandran. Cache-efficient dynamic programming algorithms for multicores. Technical Report TR-08-16, Dept. of Computer Sciences, UT-Austin, 2008.

[10] T. Cormen, C. Leiserson, R. Rivest, and C. Stein. *Introduction to Algorithms*. The MIT Press, second edition, 2001.

[11] D. Culler, R. Karp, D. Patterson, A. Sahay, K. Schauser, S. E., R. Subramonian, and T. von Eicken. Logp: Toward a realistic model of parallel computation. In *Proc. 4th SIGPLAN Symp. Principles Practices of Parallel Programming*, pages 1–12, 1993.

[12] T. DeSantis, I. Dubosarskiy, S. Murray, and G. Andersen. Comprehensive aligned sequence construction for automated design of effective probes (CASCADE-P) using 16S rDNA. *Bioinformatics*, 19:1461–1468, 2003. url: http://greengenes.llnl.gov/16S/.

[13] M. Frigo, C. Leiserson, H. Prokop, and S. Ramachandran. Cache-oblivious algorithms. In *Proc. IEEE FOCS*, pages 285–297, 1999.

[14] M. Frigo and V. Strumpen. The cache complexity of multithreaded cache oblivious algorithms. In *Proc ACM SPAA*, pages 271–280, 2006.

[15] Z. Galil and K. Park. Parallel algorithms for dynamic programming recurrences with more than $o(1)$ dependency. *JPDC*, 21:213–222, 1994.

[16] P. Gibbons, Y. Matias, and V. Ramachandran. Can shared-memory model serve as a bridging model for parallel computation? In *Proc. ACM SPAA*, pages 72–83, 1997.

[17] J. L. Hennessy and D. A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann, third edition, 2002.

[18] D. Hirschberg. A linear space algorithm for computing maximal common subsequences. *CACM*, 18(6):341–343, 1975.

[19] R. Karp and V. Ramachandran. Parallel algorithms for shared memory machines. In *Handbook of Theor Comp Sci*, pages 869–941. Elsevier, 1990.

[20] B. Knudsen. Multiple parsimony alignment with "affalign". Software package `multalign.tar`.

[21] B. Knudsen. Optimal multiple parsimony alignment with affine gap cost using a phylogenetic tree. In *Proc. Workshop Algs in Bioinf.*, pages 433–446, 2003.

[22] W. Pearson and D. Lipman. Improved tools for biological sequence comparison. In *Proc. Natl Acad. Sciences*, volume 85, pages 2444–2448, 1988.

[23] D. Powell. Software package `align3str_checkp.tar.gz`.

[24] D. Powell, L. Allison, and T. Dix. Fast, optimal alignment of three sequences using linear gap cost. *Journal of Theoretical Biology*, 207(3):325–336, 2000.

[25] G. Tan, N. Sun, and G. R. Gao. A parallel dynamic programming algorithm on a multi-core architecture. In *ACM SPAA*, pages 135–144, 2007.

[26] J. Thomas et al. Comparative analyses of multi-species sequences from targeted genomic regions. *Nature*, 424:788–793, 2003.

[27] L. Valiant. General context-free recognition in less than cubic time. *JCSS*, 10:308–315, 1975.

[28] L. Valiant. A bridging model for parallel computation. *CACM*, 33(8):103–111, 1990.