Optimal Speedup on a Low-Degree Multi-Core Parallel Architecture (LoPRAM)

Reza Dorrigiv, Alejandro López-Ortiz and Alejandro Salinger*

Technical Report CS-2007-48
Cheriton School of Computer Science
University of Waterloo

Abstract

Modern microprocessor architectures have gradually incorporated support for parallelism. In the past the degree of parallelism was rather small and as such it could be best modeled as a constant speedup over the traditional RAM model, however, as a consequence of continued growth this assumption might no longer hold. For example, with the introduction of 32- and 64-bit architectures, bit-level parallelism became significant. This led to the introduction of the transdichotomous RAM model, for which many algorithms which are faster in theory and practice have been developed. Similarly, over the last five years, major microprocessor manufacturers have released road maps for the next decade predicting a rapidly increasing number of cores, with upwards of 64 cores per microprocessor by 2015. In such a setting a sequential RAM computer no longer accurately reflects the architecture on which algorithms are being executed. In this paper we propose a model of low degree parallelism (LoPRAM) which builds upon the RAM and PRAM models yet better reflects recent advances in parallel (multi-core) architectures. The model has as a goal a combination of fidelity of the underlying hardware and ease of programming and analysis. When necessary we make tradeoffs between what is achievable in hardware and what is understandable and programmable by humans/compilers. The proposed model supports a high level abstraction that simplifies the design and analysis of parallel programs. Then we show that in many instances this model, though in many ways similar to the classic PRAM, it naturally leads to work-optimal parallel algorithms via simple modifications to sequential algorithms. This is in contrast to the PRAM model, in which the design and analysis and implementation of work-optimal algorithms proved to be one of the biggest challenges in practice for their adoption.

1 Introduction

Modern microprocessor architectures have gradually incorporated support for a certain degree of parallelism. Over the last two decades we have witnessed the introduction of the graphics processor, the multi-pipeline architecture, and vector architectures. More recently major hardware vendors such as Intel, Sun, AMD and IBM have announced multicore architectures in their main line of processors. Currently two, four and eight core processors are generally available. Until recently the degree of parallelism provided by any of these solutions was rather small and as such it was best studied as a constant speedup over the traditional and/or transdichotomous RAM model. However,

*School of Computer Science, University of Waterloo, 200 University Ave. W., N2L 3G1, Waterloo, Ontario, Canada, e-mail: {rdorrigiv,alopez-o,ajsalinger}@uwaterloo.ca.

recent road maps released by major microprocessor manufacturers predict a rapidly increasing number of cores. In particular Intel's road map released in early 2005 predicted quad core by 2007 (with nearly all microprocessors being multi-core by the end of that year) and more importantly 64 to 128 cores per microprocessor by 2015. A newly revised version of "Moore's Law" now states that the number of cores per chip is expected to double every two years. In this scenario, a constant speedup would no longer accurately reflect the amount of resources available.

We propose a new model of low degree parallelism which better reflects recent multicore architectural advances. We argue that in current architectures the number of processors p available can effectively be assumed to be $O(\log n)$. This parallels the development of the transdichotomous-RAM in which the the presence of bit level parallelism went from being subsumed as a small constant speed-up to the $w = O(\log n)$-bit word model in which the speedup is a function of w.

As with the classical RAM model, the LoPRAM supports different degrees of abstraction. Depending on the intended application and the performance parameters required the design and analysis of an algorithm can consider issues such as the memory hierarchy, interprocess communication, low level parallelism, or high level thread based parallelism. Our main focus is on the higher level, thread based parallelism. Naturally the more abstract the model the easier it is to reason on it at the expense of fidelity in the analysis. As we shall see the design and analysis of algorithms at this higher level is often sufficient to achieve optimal speedup. This, of course, does not preclude the use of low level optimizations when necessary. This parallels the classical RAM model in which issues such as caching, secondary storage and other such hardware characteristics can be incorporated or ignored as it may be deemed most appropriate.

We then apply this model to the design and analysis of algorithms for multicore architectures for a sizeable subset of problems and show that we can readily obtain optimal speedups. This is in contrast to the PRAM model, in which even a work-optimal sorting algorithm proved to be a difficult research question [8]. More explicitly, we show that a large class of dynamic programming and divide and conquer algorithms can be parallelized using the high level LoPRAM thread model while achieving optimal speedup. Interestingly, the assumption that there is a logarithmic bound on the degree of parallelism is key in the analysis of the techniques given. As well, communication cost remains modest under the assumption of low-degree parallelism. Indeed with this bound in place a full processor network based on the complete graph is realizable. Paradoxically the main contribution of this paper is not in the difficulty of the proofs or techniques but rather in their simplicity. We believe this will be a key factor in the adoption of new parallel computation models.

2 Previous Work

The dominant model for previous theoretical research on parallel computations is the PRAM model [13], which generally assumed $\Theta(n)$ processors working synchronously with zero communication delay and often with infinite bandwidth among them. If the number of processors available in practice was smaller, the $\Theta(n)$ processor solution could be emulated using Brent's Lemma [6]. The PRAM model, while fruitful from a theoretical perspective, proved unrealistic and various attempts were made to refine it in a way that would better align to what could effectively be achieved in practice. Among the alternatives introduced were, to name a few examples, the LogP model [10, 18], the LogGP model [3], the bulk-synchronous parallel model [24], and the Asynchronous PRAM [16], among others [20, 23, 1, 2, 7]. In practice there were various important drawbacks of the PRAM model, such as the cost of synchronization, the cost of interprocessor communication, the cost-effectiveness of a massively parallel machine and more importantly the enormous difficulty in developing and implementing work-optimal algorithms (i.e. linear speedup) for a computer with

$\Theta(n)$ processors. Even relatively simple tasks such as sorting required considerable thought before a work-optimal PRAM algorithm could be developed [8]. The state of parallel algorithm research consequently entered into a dormant state in the second half of the 1990s. Recent developments in multicore architectures have brought back the possibility of parallel architectures in practice which has revived the study of parallel algorithms.

However, to the best of our knowledge the assumption of a logarithmic level of parallelism as well as its theoretical implications had yet to be noted in the literature. Our informal discussions with classic PRAM algorithmicists as well as modern multicore researchers only confirmed this perception. We note that while PRAM algorithms were by far and large developed with an assumption of as many as $\Theta(n)$ processors being available, there is previous work in the literature considering smaller number of processors for certain specific cases. For example, Munro and Robertson [22] proved in 1979 that a priority queue algorithm with optimal speedup exists so long as $p = O(\log n)$.

Structure of the paper. In Section 3 we introduce the LoPRAM, a formal model for multicore computing. In Section 4 we show that under this model a large class of divide and conquer and dynamic programming algorithms can be made to run optimally in this model. Lastly in Section 5 we present concluding remarks and future directions of research.

3 Model

The core of a LoPRAM is a PRAM with $p = O(\log n)$ processors running in multiple-instruction multiple-data (MIMD) mode. The read and write model, while architecture dependent, can generally be assumed to be Concurrent-Read Exclusive-Write (CREW) [15, 17]. To support this model, semaphores and automatic serialization on shared variables are available—either hardware or software based—in a transparent form to the programmer. If an unserialized variable is concurrently written this has undefined arbitrary behaviour—including suspension of execution.

The model naturally supports a high level abstraction that simplifies the design and analysis of parallel programs. The application benefits from parallelism through the use of threads. We show that in many instances this leads to work-optimal parallel algorithms derived from simple modifications of sequential algorithms.

3.1 Thread Model

Two main types of threads are provided: standard threads and PAL-threads (Parallel ALgorithmic threads). Standard threads are executed simultaneously and independently of the number of cores available; they are executed in parallel if enough cores are available or by using multitasking if the thread count exceeds the degree of parallelism, just as in a regular RAM. PAL-threads on the other hand are executed at a rate determined by the scheduler. If there are any PAL-threads pending, at least one of them must be actively executing, while all others remain at the discretion of the scheduler. They could be assigned resources, if they are available, or they could be made to wait inactive until resources free over. Once a thread has been activated though, it remains active just like a standard thread (this is important to avoid potential deadlock). Pending PAL-threads are activated in a manner consistent with order of creation as resources become available. While primitives are provided for ad-hoc ordering of PAL-threads activation, by default threads are inserted into an ordered tree. The root of the tree is the main thread with new threads attached to the node corresponding to the activating parent-thread. The scheduler activates the nodes in the tree in parent-child order, i.e. first the parent thread is activated, which issues PAL-threads calls

for its children. The parent thread is now in a wait state and the processor is assigned sequentially to the children, in order of creation. If no further children remain pending then control is returned to the parent thread. If no such thread is present then nodes are activated in the order given by the preorder traversal of the tree.

Execution concludes when there are no further threads to activate and the main thread exits. Consider for example the code for a parallel implementation of the classical sequential mergesort written using suitable C extensions for the LoPRAM:

```
void mergeSort(int numbers[], int temp[], int array_size)
{ m_sort(numbers, temp, 0, array_size - 1); }

void m_sort(int numbers[], int temp[], int left, int right) {
  int mid = (right + left) / 2;
  if (right > left) {
    palthreads {
      m_sort(numbers, temp, left, mid);
      m_sort(numbers, temp, mid+1, right);
    }
    merge(numbers, temp, left, mid+1, right);
  }
}
```

The semantics of the primitive `palthreads` are to create a PAL-threads call for each of the function calls within its scope. These threads are created as children of the current thread in the specific order given. Observe that there is an implicit wait at the end of the `palthreads` statement which can be deactivated using a "`palthreads { ... } nowait;`" construct with an explicit thread join later on if so needed. Note that we introduce this syntax only for the purposes of the example and that it is not inherent to the LoPRAM model. An example of the execution of `mergeSort` with an input of size 16 and 4 processors is shown in Figure 1.

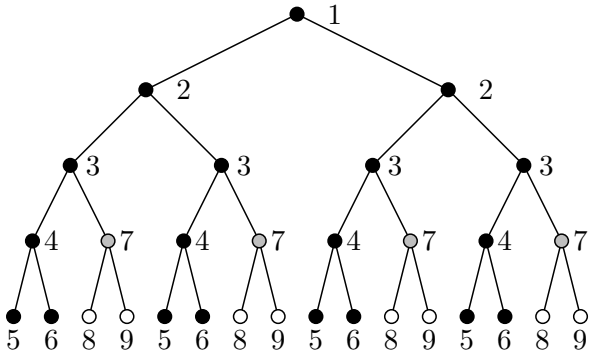

Figure 1: Example of an execution tree for Mergesort with an input of length $n = 16$ and $p = 4$ processors. Black nodes represent active pal-requests, gray nodes represent calls that have been pal-requested but that are not active yet, while white nodes are calls that have not been pal-requested. The number by each node indicates the time step in which the call corresponding to that node is pal-requested. The picture shows the execution at $t = 6$.

3.2 Multiprocessing model

In actuality, the number of cores made available by the operating system may vary as the level of multiprogramming in the system changes. Hence, in the analysis of the algorithm the number of processors available is denoted as p, with the assumption that this number is bounded from

above by $O(\log n)$, i.e., p is $O(\log n)$ but p is not necessarily $\Theta(\log n)$. The algorithm must execute properly for any value of p. The running time is, of course, a function of n and p.

4 Work-Optimal Parallelization

In this section we present two classes of problems which allow for ready parallelization under the LoPRAM model. Note that these same classes were not, in general, readily parallelizable under the classic PRAM model.

4.1 Divide and Conquer

Consider the class of divide-and-conquer algorithms whose time complexity is described by a recurrence which can be resolved using the Master theorem. We show that when these algorithms are executed in a straightforward parallelization on a LoPRAM, their execution time is given by a parallel version of the Master theorem which reports optimal speedup.

Consider a recursive divide-and-conquer sequential algorithm whose time complexity $T(n)$ is a recurrence of the form:

$$T(n) = aT(n/b) + f(n), \tag{1}$$

where $a \geq 1$ and $b > 1$ are constants, and $f(n)$ is a nonnegative function. By the Master theorem, $T(n)$ is such that [9]:

$$T(n) = \begin{cases} \Theta(n^{\log_b a}), & \text{if } f(n) = O(n^{\log_b(a)-\epsilon}) \quad \text{(Case 1)} \\ \Theta(n^{\log_b a} \log n), & \text{if } f(n) = \Theta(n^{\log_b a}) \quad \text{(Case 2)} \\ \Theta(f(n)), & \text{if } f(n) = \Omega(n^{\log_b(a)+\epsilon}) \text{ and } af(n/b) \leq cf(n), \text{ for some } c < 1 \quad \text{(Case 3)} \end{cases} \tag{2}$$

We are interested in the time complexity of such algorithms when p processors are available. We assume that recursive calls can be assigned to different processors, which can execute their instances independently of those of others. All of the processors finish their computation before the results are merged. First, we consider problems for which the merging phase of the algorithm can only be done sequentially in each instance. Multiple processors can still be used to merge subproblems of different instances, but only one processor deals with a particular instance. We denote by $T_p(n)$ the wall-clock time of a parallel algorithm that uses recursion with p processors, and by $T(n) = T_1(n)$ its sequential version.

Sequential Merging We first consider the case when we merge subresults sequentially. The following theorem states the bounds for the wall-clock time of an algorithm whose time is defined by equation (1) when using p processors.

Theorem 1 *Let $T_p(n)$ be the time taken by a recursive algorithm that uses $p = O(\log n)$ processors whose sequential version has time complexity given by $T(n) = aT(n/b) + f(n)$, where $a > 1$ and $b > 1$ are constants, and $f(n)$ is a nonnegative function. Then, the time $T_p(n)$ is a recurrence of the form:*

$$T_p(n) = T\left(\frac{n}{b^{log_a p}}\right) + \sum_{i=0}^{\log_a(p)-1} f\left(\frac{n}{b^i}\right). \tag{3}$$

The bounds for $T_p(n)$ are given by:

$$T_p(n) = \begin{cases} O(T(n)/p), & \text{if } f(n) = O(n^{\log_b(a)-\epsilon}) \quad \textit{(Case 1)} \\ O(T(n)/p), & \text{if } f(n) = \Theta(n^{\log_b a}) \quad \textit{(Case 2)} \\ \Theta(f(n)), & \text{if } f(n) = \Omega(n^{\log_b(a)+\epsilon}) \text{ and } af(n/b) \leq cf(n), \text{ for some } c < 1 \quad \textit{(Case 3)} \end{cases} \tag{4}$$

Proof: Since there are p processors available, some recursive steps of the algorithm can be performed in parallel. However when the number of simultaneous calls exceeds p, we need to solve the rest sequentially. Since the algorithms divides the problem into a subproblems, at the k-th level of the recursion tree we will have a^k subproblems. Thus, when $k = \log_a p$ we will have p subproblems and no more processors are available for the subsequent recursive calls. Then, at this point, as there are no more free cores available, the sequential version of the algorithm is used, with an input of size $n/b^{\log_a p}$ (See Figure 2). Observe that this condition is never explicitly tested for by the scheduling algorithm, rather it is a natural consequence of the proposed order of execution of the parent child threads. Note that it cannot be the case that all the recursive calls are performed in parallel and that there is no sequential component to be executed as it is not hard to see that this would only happen if $b^{\log_a p} \geq n$, which would mean that $p \geq n^{\log_b a} = \omega(\log n)$, but we assume $p = O(\log n)$.

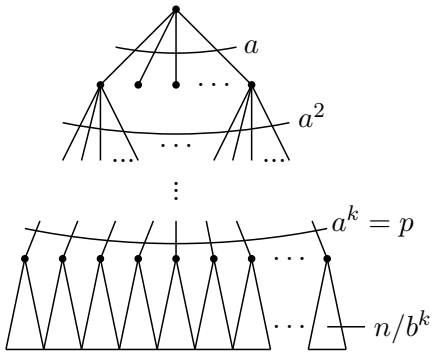

Figure 2: Execution tree of a divide-and-conquer algorithm with p processors: a thread is created for each recursive call until $a^k = p$ calls have been made. Thereafter each thread executes the algorithm sequentially

The cost of merging the solutions of the subproblems using p processors is given by the sum of the cost at each level of the recursion tree, namely: $\sum_{i=0}^{\log_a(p)-1} f\left(n/b^i\right)$. Hence, we can write the time of the parallel algorithm as in Equation (3). Now as with the standard version of the master theorem we prove each case separately.

Case 1 Since $f(n) = O(n^{\log_b(a)-\epsilon})$, by the master theorem, we have that $T(n) = \Theta(n^{\log_b a})$. Substituting in Equation (3) we have

$$T_p(n) \;=\; O\left(\left(\frac{n}{b^{\log_a p}}\right)^{\log_b a} + \sum_{i=0}^{\log_a(p)-1}\left(\frac{n}{b^i}\right)^{log_b(a)-\epsilon}\right) = O\left(\frac{n^{\log_b a}}{p} + \frac{n^{log_b(a)-\epsilon}a}{a-b^\epsilon}\right)$$

Since $T(n) = \Theta(n^{\log_b a})$, the first term of the sum above is $O(T(n)/p)$ while the second term is clearly strictly smaller and hence $T_p(n) = O(T(n)/p)$.

Case 2 Since $f(n) = \Theta(n^{\log_b a})$, we have that $T(n) = \Theta(n^{\log_b a} \log n)$. Then,

$$\begin{aligned} T_p(n) &= O\left(\left(\frac{n}{b^{\log_a p}}\right)^{\log_b a} \log\left(\frac{n}{b^{\log_a p}}\right) + \sum_{i=0}^{\log_a(p)-1} \left(\frac{n}{b^i}\right)^{log_b a}\right) \\ &= O\left(\frac{n^{\log_b a}}{p}(\log n - \log b^{\log_a p}) + n^{\log_b a} \cdot \frac{a}{a-1}\right) \end{aligned}$$

Clearly the first term dominates as $\log n/p = \Omega(1)$. It follows then that $T_p(n) = O(T(n)/p)$.

Case 3 In this case we prove that the total time is dominated by the time that it takes to merge the solutions of the subproblems to produce the final solution in the second level of the recursion tree. Recall that we assume that this process is done sequentially. Thus, no benefit is gained from using p processors in this case. Starting with recurrence (1), we know that $af(n/b) \leq cf(n)$. By a simple induction argument, it can be shown that $f(n/b^i) \leq (c/a)^i f(n),\ 0 \leq i \leq \log_b n$. In addition, we have that $T(n) = \Theta(f(n))$. Hence:

$$T_p(n) \leq O\left(f\left(\frac{n}{b^{log_a p}}\right)\right) + \sum_{i=0}^{\log_a p - 1} \left(\frac{c}{a}\right)^i f(n) \leq O\left(f(n)\left(\frac{c}{a}\right)^{log_a p} + f(n)\right) \leq O(f(n))$$

On the other hand, trivially $T(n) \geq f(n)$ and hence we have $T_p(n) = \Omega(f(n))$. ∎

Parallel Merging We now consider the special case when we can merge the results of subproblems in parallel with optimal speedup. Recall that the time complexity of the sequential algorithm is given by Equation (1). Then we claim that the parallel master theorem for this setting is as before with the exception of case 3 for which we have

$$T_p(n) = \Theta(f(n)/p), \quad \text{if } f(n) = \Omega(n^{\log_b(a)+\epsilon}) \text{ and } af(n/b) \leq cf(n), \text{ for some } c < 1 \tag{5}$$

For the merging phase, at the i-th level of the recursion tree we have to merge a total of a^i solutions of sub-problems of size n/b^i. Each of them takes time $f(n/b^i)$, and since we assume they can be done in parallel, the total time of the merging phase at that level is given by $(a^i/p)f\left(n/b^i\right)$. Thus we obtain an analogous expression to Equation (3) but with a merging cost that is $1/p$ times what it was before. For cases 1 and 2 the dominant term is the first summand of the right hand side of Eq. (3) and hence we obtain the same expression as before. For Case 3, using the derivations as in the previous sequential case but dividing by the speedup factor of p in the merge process we obtain

$$T_p(n) \leq k_1 f(n)\left(\frac{c}{a}\right)^{log_a p} + \frac{f(n)}{p}\frac{1}{1-c/a} \leq o\left(\frac{f(n)}{p}\right) + O\left(\frac{f(n)}{p}\right)$$

and since trivially we have $T_p(n) \geq f(n)/p$ we conclude that $T_p = \Theta(f(n)/p)$. ∎

4.2 Dynamic Programming

Dynamic programming is suitable for solving problems that have optimal substructure as well as overlapping solutions to subproblems. The solutions to such subproblems are then combined into the solution to a larger problem. In many cases these subproblems can be solved in parallel, up to a degree that depends on the problem itself, and hence a certain degree of parallelism is achievable.

In the past parallel versions of certain dynamic programming algorithms have been proposed. In a seminal paper, Apostolico *et al.* [4] studied parallel algorithms for the string editing problem

and other related problems by considering the Directed Acyclic Graph (DAG) corresponding to the problem and computing this graph in parallel. Galil and Park [14] studied various dynamic programming problems, presenting a unified framework for the parallel computation of these problems using the closure methods and the matrix product methods as general tools for developing parallel algorithms. Later, Bradford [5] developed a characterization that models dynamic programming tables by graphs, leading to polylogarithmic time algorithms for optimal matrix chain ordering, the optimal construction of binary trees and the optimal convex polygon triangulation. Bradford shows how to transform these problems to a minimum cost parenthesizing on a weighted semigroup, which is then transformed to a shortest path problem on a weighted directed graph. Most of these studies consider a few dynamic programming problems and provide a parallel algorithm that is specific to one or a few problems. In general they assume a classical PRAM model with $\Theta(n)$ processors, meaning the algorithm is designed so that it can take advantage of that many processors, shall they be available. In our case we restrict ourselves to $p = O(\log n)$ algorithms with the savings in communication, synchronization and complexity of the code.

We show that dynamic programming algorithms can be parallelized by providing a general procedure that, given the specification of the dynamic programing solution to a problem, generates a scheduling strategy to solve it in parallel. The idea is similar to that in the previous work cited above in that we also reduce the original problem to computing the DAG corresponding to the dynamic programming specification of the solution.

A dynamic programming algorithm is an optimization problem whose solution is characterized by (i) a recursive decomposition of the problem into subproblems (ii) a bottom-up computation of the cost of said solution and (iii) recovery of the actual solution from the computed cost together with other ancillary information. In most cases, the partial solutions are stored in a d dimensional table for some integer $d \geq 0$, however this is not an inherent property of dynamic programming algorithms.

Our goal is to compute the solution to the dynamic programming problem with as much parallelism as possible, with a general strategy that works for any problem whose specification is given in an explicit dynamic programming expression of its solution. Hence we assume that we have a solution of the form:

$$M[x] = \begin{cases} f(x) & \text{if } g(x) = 0 \quad \text{(base case)} \\ f(\{M[y_i]\}_{y_i \prec x}, x) & \text{otherwise} \end{cases} \tag{6}$$

For the dynamic programming solution to be effective we require that the object M which stores partial solutions remains of reasonable size, that it can be efficiently indexed using a partial input x as key and that the recursive order $y_i \prec x$ be efficiently constructible in a bottom-up fashion. Alternatively, if the solution cannot be computed efficiently in a bottom-up fashion we can use *memoization* which stores the partial solutions as they are required in the top-down expansion of the recursion. In most cases these two techniques are equivalent, though there are known cases in which the use of one over the other (for either of them) is provably superior.

4.3 Dependency Graph

The recursion described in Equation (6) can be encoded in a graph form. For each partial result $M[x]$ we draw an edge pointing from the solution $M[y_i]$ of subproblems $\{y_i\}$ required in the recursion to the node $M[x]$. This creates a DAG of dependencies, starting in the base cases and with $M[I]$ as the terminal node, where I is the original input. The speedup will be proportional to the amount of parallelism embedded in the graph. To be more precise: we consider the dependencies graph as a partially ordered set (poset) on the subproblems. Then the subproblems in an antichain of

this poset are independent and can be processed at the same time in parallel. We can partition the corresponding dependencies DAG into antichains and then process the subproblems in every antichain in parallel. At each time we find an antichain that does not have any dependencies on the subproblems in the antichains that have not yet processed. Then we process subproblems in that antichain in parallel, and then move on to the next antichain. A dual of Dilworth's theorem states that the size of the largest chain in a poset equals the smallest number of antichains into which the poset may be partitioned [11, 21]. Suppose that $c_1 c_2 \dots c_l$ is a largest chain in the poset. At step i we process c_i together with other elements in its antichain, i.e., elements that are incomparable with c_i.

These antichains capture the degree of parallelism that is readily apparent in the recursive description of the problem. In certain cases, such as one dimensional dynamic programming the DAG is a path and hence there is no speedup possible. In others such as most common examples of two dimensional tables for dynamic programming, there is row, column or diagonal order which allows for a high degree of parallelism.

4.4 Parallel Dynamic Programming

We consider dynamic programming algorithms for which we evaluate the DAG corresponding to the algorithm in parallel. This strategy is divided in three steps: (i) determine the dependencies graph for the cells of the table M, (ii) reverse this graph to obtain a DAG of the problem, and (iii) schedule the computation of the DAG with parallel threads.

For the most common case where the object store is a d dimensional table and the input of the problem has size n, the graph of dependencies will have n^d vertices, where d is the dimension of M. There is an edge from vertices u to v if the computation of the value of the cell corresponding to u needs the value of the cell corresponding to v. Since no edges in the dependencies graph depend on previously determined edges, the dependencies graph can be determined in parallel optimally by all p processors in time $O(mn^d/p)$, where m is the maximum degree of any vertex, i.e. the maximum number of cells that any computation of a cell depends on. The idea is to create an adjacency matrix D to represent the dependencies graph. Each processor is assigned n^d/p vertices for which it has to determine the vertices that they depend on. According to a specification of the dynamic programming solution of the form (6), a unique vertex id is assigned to each cell of table M. Then, in order determine the dependencies of a given vertex v, a processor has to determine the id of the vertex and update D according to the recursive expression given in the specification. The result is a n^d containing the dependency list for node i.

We consider the following scheduling strategy: each vertex v has a counter c_v that indicates, at any time, the number of vertices that v depends on directly and that have not been computed yet. Initially, the counters of all vertices are equal to their in-degree. After a thread computes the value corresponding to a vertex v, it decreases the values of the counters of all the neighbors of v. When these neighbor vertices have their counter equal to 0, i.e. they are ready to be computed, the same thread creates other PAL-threads to compute these vertices and these get executed depending on the availability of processors. Algorithm 1 describes this algorithm in pseudocode.

4.5 Parallel Memoization

Memoization is a strategy to solve problems that have a similar recursive decomposition as in Equation (6), but in which the execution of the algorithm is carried out recursively in a top-down fashion. It differs from usual divide-and-conquer recursive algorithms in that the first time each sub-problem is solved, its result is stored in order to avoid further computations of the same result.

Algorithm 1 parallel_dp(G, S, t)

```
for each u ∈ S do
  PAL-threads { computeVertex(u) } nowait;
end for

computeVertex(u)
  compute u
  for each v such that (u, v) ∈ E do
    v_c ← v_c − 1
    if v_c = 0 then
      PAL-threads { computeVertex(v) } nowait;
    end if
  end for
```

Initially, all the sub-problems in the stored object contain a value that indicates that the solution has not yet been computed. Before a sub-problem is solved, its entry is looked up in the structure. If the entry has a previously computed value, this value is used and no further computation for this sub-problem is carried out. Otherwise, the sub-problem is solved and the solution is stored in the structure.

The parallelization of an algorithm that uses memoization is similar to the one introduced earlier on for divide-and-conquer algorithms: each recursive call is assigned to a different PAL-thread, with the difference that a thread is created only when the value to be computed has not been computed or initiated before. Say that the value $M[x]$ has not been computed so its computation is now assigned to a thread t. Let $M[y_1], \ldots, M[y_r]$ be the values that $M[x]$ depends on. For each of these values $M[y_i]$, thread t checks if they are already available, using this value if this is the case. In contrast to sequential memoization, if the value is not available there are two options: either a new thread is launched to compute it and this is recorded in the object M as "in progress" or if the value is not present but recorded as already in progress by another thread, then the thread registers a notify condition on solution. The thread continues with all other subproblems y_i until all of the subproblems are active or solved. If not all the answers are available the thread enters a wait state until they become available.

Observe that the testing of previous solutions introduces an overhead factor over the sequential version of the same memoization program. If k subproblems require a specific value to proceed we can have as many as $k-1$ probes for the value that do not result in launching a thread nor do they return a value as it is labeled "in progress". If these probes are simultaneous then this access can cause delay depending on the model assumed. For CREW a serialization mechanism is needed to update this value concurrently.This can be done with a $\log p$ overhead using standard techniques for simulating a CRCW with an CREW PRAM [12].

The speedup factor in this case is, as noted by Apostolico et al. [4] heavily dependent on the amount of parallelism imbued in the recursive structure of the solution, which we shall discuss in the next subsection.

4.6 Speedup Factor for DAGs

If $T(n)$ is the time that it takes to compute the solution of the problem, our goal is to compute the solution optimally in time $T_p(n) = O(T(n)/p)$. If the input of a problem has size n, the dynamic programming store is a table of dimension d and the computation of a cell depends at most on m other cells, the time of the sequential solution is $O(mn^d)$. In our parallelization, as argued before, creating the dependencies graph takes time $O(mn^d/p) = O(T(n)/p)$, while reversing the graph

does not need any computation. The time for computing the DAG depends on the extent to which we can parallelize its execution. Depending on each problem, the graph G can be be more or less suitable for parallelization.

It is important to note that updating the counters of the neighbors of a vertex cannot always be done in parallel in a CREW model. Hence we use a standard simulation technique to obtain CRCW behaviour on a CREW PRAM with a $\log p$ slowdown factor.

In general the speedup factor will depend on the amount of parallelism implicit in the DAG (using the antichains argument) as well as the slowdown from the simultaneous lookup of the value associated to a node in the DAG.

Interestingly enough, the same argument can be applied to a general sequential program in which the function call chain can be computed ahead of time. Lokhmotov et al. introduced the concept of sieves which are blocks of code in which all side-effects within them are delayed until the end of the scope and side-effects [19]. Such primitives naturally exploit the parallelism present in the sequential program with minimum effort to the programmer.

5 Conclusions

We introduced a new model for parallel computation, LoPRAM, that is faithful to current architectures, avoids many of the pitfalls of the previous PRAM model, namely difficulty of programming and expensive processor communication infrastructure, and allows for significant classes of problems to be parallelized with little effort. Our model supports a high level abstraction that simplifies the design and analysis of parallel programs. We provided work-optimal LoPRAM parallel algorithms for divide-and-conquer and dynamic programming problems by applying simple modifications of the corresponding sequential algorithms. These serve as starting examples of algorithm design and analysis in the LoPRAM model, and give an insight of the techniques for parallelization that we seek to apply to a wider set of problems.

Future directions of research include refining the model and trying to find other families of algorithms for which we can apply general parallelization techniques to obtain simple work-optimal parallel algorithms.

Acknowledgments We thank Ming-Yang Kao, Phil Gibbons, David Patterson, Prabhakar Ragde and Andrej Brodnik for fruitful discussions in this subject. Key ideas of this paper were first proposed during the 2006 Dagstuhl Workshop on Data Structures, whose participants' comments are gratefully acknowledged.

References

[1] A. Aggarwal, A. K. Chandra, and M. Snir. On communication latency in pram computations. In *SPAA '89: Proceedings of the first annual ACM symposium on Parallel algorithms and architectures*, pages 11–21, New York, NY, USA, 1989. ACM.

[2] Alok Aggarwal, Ashok K. Chandra, and Marc Snir. Communication complexity of prams. *Theor. Comput. Sci.*, 71(1):3–28, 1990.

[3] Albert Alexandrov, Mihai F. Ionescu, Klaus E. Schauser, and Chris Scheiman. Loggp: incorporating long messages into the logp model one step closer towards a realistic model for parallel computation. In *SPAA '95: Proceedings of the seventh annual ACM symposium on Parallel algorithms and architectures*, pages 95–105, New York, NY, USA, 1995. ACM.

[4] Alberto Apostolico, Mikhail J. Atallah, Lawrence L. Larmore, and Scott McFaddin. Efficient parallel algorithms for string editing and related problems. *SIAM J. Comput.*, 19(5):968–988, 1990.

[5] Phillip Gnassi Bradford. Parallel dynamic programming. Technical Report #352, 1994.

[6] Richard P. Brent. The parallel evaluation of general arithmetic expressions. *J. ACM*, 21(2):201–206, 1974.

[7] T. Cheatham, A. Fahmy, D. Stefanescu, and L. Valiant. Bulk Synchronous Parallel Computing — A Paradigm for Transportable Software. In *Proceedings of the 28th Annual Hawaii Conference on System Sciences*, volume II. IEEE Computer Society Press, January 1995.

[8] Richard Cole. Parallel merge sort. *SIAM J. Comput.*, 17(4):770–785, 1988.

[9] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*. The MIT Press, 2nd edition, 2001.

[10] David Culler, Richard Karp, David Patterson, Abhijit Sahay, Klaus Erik Schauser, Eunice Santos, Ramesh Subramonian, and Thorsten von Eicken. Logp: towards a realistic model of parallel computation. In *PPOPP '93: Proceedings of the fourth ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 1–12, New York, NY, USA, 1993. ACM Press.

[11] R. P. Dilworth. A decomposition theorem for partially ordered sets. *Annals of Mathematics*, 51:161–166, 1950.

[12] Faith E. Fich, Prabhakar Ragde, and Avi Wigderson. Relations between concurrent-write models of parallel computation. *SIAM J. Comput.*, 17(3):606–627, 1988.

[13] Steven Fortune and James Wyllie. Parallelism in random access machines. In *STOC '78: Proceedings of the tenth annual ACM symposium on Theory of computing*, pages 114–118, New York, NY, USA, 1978. ACM.

[14] Z. Galil and K. Park. Parallel dynamic programming. Technical Report CUCS-040-91, 1991.

[15] Alan Gibbons and Wojciech Rytter. *Efficient parallel algorithms*. Cambridge University Press, New York, NY, USA, 1988.

[16] P. B. Gibbons. A more practical pram model. In *SPAA '89: Proceedings of the first annual ACM symposium on Parallel algorithms and architectures*, pages 158–168, New York, NY, USA, 1989. ACM Press.

[17] Joseph JáJá. *An introduction to parallel algorithms*. Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA, 1992.

[18] Richard M. Karp, Michael Luby, and Friedhelm Meyer auf der Heide. Efficient PRAM simulation on a distributed memory machine. pages 318–326, 1992.

[19] Anton Lokhmotov, Alan Mycroft, and Andrew Richards. Delayed side-effects ease multi-core programming. In Anne-Marie Kermarrec, Luc Bougé, and Thierry Priol, editors, *Euro-Par*, volume 4641 of *Lecture Notes in Computer Science*, pages 641–650. Springer, 2007.

[20] Kurt Mehlhorn and Uzi Vishkin. Randomized and deterministic simulations of prams by parallel machines with restricted granularity of parallel memories. *Acta Informatica*, 21:339–374, 1984.

[21] L. Mirsky. A dual of dilworth's decomposition theorem. *The American Mathematical Monthly*, 78(8):876–877, 1971.

[22] J. Ian Munro and Edward L. Robertson. Parallel algorithms and serial data structures. In *Proceedings of the 17th Annual Allerton Conference con Communication, Control and Computing*, pages 21–26, 1979.

[23] Christos Papadimitriou and Mihalis Yannakakis. Towards an architecture-independent analysis of parallel algorithms. In *STOC '88: Proceedings of the twentieth annual ACM symposium on Theory of computing*, pages 510–513, New York, NY, USA, 1988. ACM.

[24] Leslie G. Valiant. A bridging model for parallel computation. *Commun. ACM*, 33(8):103–111, 1990.