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

Reza Dorrigiv
rdorrigiv@uwaterloo.ca

Alejandro López-Ortiz
alopez-o@uwaterloo.ca

Alejandro Salinger
ajsalinger@uwaterloo.ca

School of Computer Science, University of Waterloo
200 University Ave. W., N2L 3G1, Waterloo, Ontario, Canada

ABSTRACT

Over the last five years, major microprocessor manufacturers have released plans for a rapidly increasing number of cores per microprossesor, with upwards of 64 cores by 2015. In this setting, a sequential RAM computer will no longer accurately reflect 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. This model supports a high level of abstraction that simplifies the design and analysis of parallel programs. More importantly we show that in many instances it naturally leads to work-optimal parallel algorithms via simple modifications to sequential algorithms.

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, with two and four core processors being currently 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, recent road maps released by major microprocessor manufacturers predict a rapidly increasing number of cores, reaching 64 to 128 cores 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 (LoPRAM) which better reflects recent multicore architectural advances. We argue that in current architectures the number of processors available is best described as $O(\log n)$ rather than the implicit $O(n)$ of the classic PRAM model. 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 cost, low level parallelism, or high-level thread-based parallelism. Our main focus is on this higher level, thread-based parallelism, shifting the classical view of the SIMD PRAM with finely synchronized parallelism to a higher-level multi-threaded view. 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.

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 [10]. 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 using an automated scheduler. Interestingly, the assumption that there is a logarithmic bound on the degree of parallelism is key in the analysis of the techniques given. We identify that for certain problems there are sharp thresholds in difficulty when the number of processors grows past $\log n$ and $n^{1-\epsilon}$. This paper provides the first formal argument for the observation that it is easier to develop work optimal parallel programs when the degree of parallelism is small. While this observation is widely known and acknowledged by many, it had previously been stated only at an empirical level and not been the subject of formal study.

As such the main contribution of this work is the combination of a series of established facts, as well as new observations and lemmas into a novel model which is simple, effective and better reflects the state of current parallelism.

2. PARALLEL MODELS

The dominant model for previous theoretical research on parallel computations is the PRAM model [15], 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 [7]. 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 (see, for example, [12, 3, 21, 17, 19, 20, 1, 2]). In addition to its lack of fidelity, an important drawback of the PRAM is the enormous difficulty in developing and implementing work-optimal algorithms (i.e. linear speedup) for a computer with $\Theta(n)$ processors.

To the best of our knowledge the assumption of a logarith-

*An earlier version of this work appeared in [13].

mic level of parallelism as well as its theoretical implications had yet to be noted in the literature.

3. MODEL

The core of a LoPRAM is a PRAM with $O(\log n)$ 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) [18]. To support this model, semaphores and automatic serialization on shared variables are made available—either hardware or software based—in a transparent form to the programmer.

The LoPRAM model naturally supports a high level of abstraction that simplifies the design and analysis of parallel programs. The application benefits from parallelism through the use of threads.

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, in a fashion reminiscent of work stealing [8]. 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, and new threads are attached as children of the thread that creates them, in order of creation.

The scheduler executes the nodes in a combination of parallel breadth-first and depth-first order. When a thread issues calls for its children, the calling thread enters a wait state and children threads are executed in order of creation. If there are enough cores available, each children is assigned to a different core. Otherwise, when all available cores are executing a thread, each core executes the subtree rooted at its thread in depth-first order. If at any point cores become available again, threads are assigned to them in the order given by the preorder traversal of the tree. When no further children remain pending control is returned to the parent thread. Execution concludes when there are no further threads to execute and the main thread exits.

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., $p = O(\log n)$ but 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

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 solved using the master theorem. We show that when these algorithms are executed in a straightforward parallel fashion on a LoPRAM, their execution time is given by a parallel version of the master theorem with optimal speedup.

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

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

where $a, b > 1$ are constants, and $f(n)$ is a nonnegative function. By the master theorem, $T(n)$ is such that [11]:

$$T(n) = \begin{cases} \Theta(n^{\log_b a}), & \text{if } f(n) = O(n^{\log_b(a)-\epsilon}) \\ \Theta(n^{\log_b a}\log n), & \text{if } f(n) = \Theta(n^{\log_b a}) \\ \Theta(f(n)), & \text{if } f(n) = \Omega(n^{\log_b(a)+\epsilon}) \\ & \text{and } af(n/b) \le cf(n),\ c < 1 \end{cases}$$

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 computations before the results are merged. We denote by $T_p(n)$ the running time of an algorithm that uses p processors, and by $T(n)$ that of its sequential version.

We first 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.

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 Equation (1) . Then, the time $T_p(n)$ is a recurrence of the form:*

$$T_p(n) = T\left(\frac{n}{b^{\log_a p}}\right) + \sum_{i=0}^{\log_a(p)-1} f\left(\frac{n}{b^i}\right) \tag{2}$$

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

$$T_p(n) = \begin{cases} O(T(n)/p), & \text{if } f(n) = O(n^{\log_b(a)-\epsilon}) \\ O(T(n)/p), & \text{if } f(n) = \Theta(n^{\log_b a}) \\ \Theta(f(n)), & \text{if } f(n) = \Omega(n^{\log_b(a)+\epsilon}) \\ & \text{and } af(n/b) \le cf(n),\ c < 1 \end{cases}$$

Alternatively if we can merge the results of subproblems in parallel with optimal speedup, the parallel master theorem for this setting is as before with the exception of case 3 for which we have:

$$T_p(n) = \Theta(f(n)/p), \quad \begin{array}{l}\text{if } f(n) = \Omega(n^{\log_b(a)+\epsilon}) \\ \text{and } af(n/b) \le cf(n),\ c < 1\end{array}$$

Theorem 1 implies that optimal parallel algorithms can be readily derived for important divide-and-conquer algorithms

such as Mergesort, Matrix multiplication, Delaunay triangulation, Polygon triangulation and Convex hull, among others. Experiments for Mergesort and Matrix multiplication implementations in the LoPRAM model are presented in the Appendix. A similar work [5] studies a cache model for multicore computation for a general class of divide-and-conquer algorithms, however, it assumes that the merging phase can always be done in parallel.

4.2 Dynamic Programming

In the past parallel versions of certain dynamic programming algorithms have been proposed (see, for example, [4], [16], [6], and more recently [9]). Most of these studies provide parallel algorithms that are specific to a few dynamic programming problems, and assume a classical PRAM model with $\Theta(n)$ processors. In our case we restrict ourselves to $p = O(\log n)$ processors.

We describe now a general procedure such that, given the specification of the dynamic programing solution to a problem, generates a scheduling strategy to solve it in parallel.

Let the specification of the solution be of the form:

$$M[x] = \begin{cases} f(x) & \text{if } g(x) = 0 \quad \text{(base case)} \\ f(\{M[y_i]\}_{y_i \prec x}, x) & \text{otherwise} \end{cases} \tag{3}$$

For the dynamic programming solution to be effective we require that the object M which stores partial solutions 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. If this is not the case, 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.

The recursion described in Equation (3) can be modeled by a directed acyclic graph (DAG), where each vertex v_x corresponds to a subproblem x (or equivalently partial result $M[x]$), and there is an edge from the v_x to v_y iff subproblem y depends on subproblem x. The source vertices in this DAG correspond to the base cases. The goal is to compute this DAG in parallel. The speedup will be proportional to the amount of parallelism embedded in the graph. 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 a row, column or diagonal order which allows for a high degree of parallelism.

Given the specification $\mathcal{D}$ of a dynamic programming solution of the form (3) and an input I, we give a parallel algorithm for solving the problem. We do not explicitly create the entire dependency graph in the beginning. Instead, we create the relevant parts of the graph as the computation proceeds. 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. The counters of all vertices are initialized based on $\mathcal{D}$ and I. Initially a PAL-thread is created for each base case vertex. After a thread computes the value corresponding to a vertex v, it determines its outgoing neighbors according to $\mathcal{D}$ and I, decreases the values of their counters, and creates PAL-threads for each of the vertices whose counter becomes zero.

Our goal is to compute the solution optimally in time $T_p(n) = O(T(n)/p)$. The speedup factor will depend on the parallelism embedded in the graph: if most antichains have size smaller than p, then we cannot obtain much parallelism. However, for most dynamic programming algorithms of dimension more than one, antichains are usually large enough to support high parallelism. In addition, the concurrent update of the counter values in the graph cannot always be done in parallel in a CREW model. Hence, a standard simulation technique can be used to obtain CRCW behaviour on a CREW PRAM with a $\log p$ slowdown factor [14].

5. REFERENCES

[1] A. Aggarwal, A. K. Chandra, and M. Snir. On communication latency in pram computations. In *SPAA '89: Proc. 1st ACM Symp. on Parallel Algs. and Architectures*, pages 11–21, 1989.

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

[3] A. Alexandrov, M. F. Ionescu, K. E. Schauser, and C. Scheiman. Loggp: incorporating long messages into the logp model –one step closer towards a realistic model for parallel computation. In *SPAA '95: Proc. 7th ACM Symp. on Parallel Algs. and Architectures*, pages 95–105, 1995.

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

[5] G. E. Blelloch, R. A. Chowdhury, P. B. Gibbons, V. Ramachandran, S. Chen, and M. Kozuch. Provably good multicore cache performance for divide-and-conquer algorithms. In *SODA '08: Proc. of the 19th ACM-SIAM Symp. on Discrete Algs.*, pages 501–510, 2008.

[6] P. G. Bradford. Parallel dynamic programming. Technical Report #352, 1994.

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

[8] F. W. Burton and M. R. Sleep. Executing functional programs on a virtual tree of processors. In *FPCA '81: Proc. 1981 Conf. on Functional programming languages and computer architecture*, pages 187–194, 1981.

[9] R. A. Chowdhury and V. Ramachandran. Cache-efficient dynamic programming algorithms for multicores. In *SPAA '08: Proc. 20th ACM Symp. on Parallelism in Algs. and Architectures*, pages 207–216, 2008.

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

[11] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*. The MIT Press, 2nd edition, 2001.

[12] D. Culler, R. Karp, D. Patterson, A. Sahay, K. E. Schauser, E. Santos, R. Subramonian, and T. von Eicken. Logp: towards a realistic model of parallel computation. In *PPOPP '93: Proc. 4th ACM SIGPLAN Symp. on Principles and practice of parallel programming*, pages 1–12, 1993.

[13] R. Dorrigiv, A. López-Ortiz, and A. Salinger. Optimal speedup on a low-degree multi-core parallel architecture (lopram). In *SPAA '08: Proc. 20th ACM Symp. on Parallelism in Algs. and Architectures*, pages 185–187, 2008.

[14] F. E. Fich, P. Ragde, and A. Wigderson. Relations between concurrent-write models of parallel computation. *SIAM J. Comput.*, 17(3):606–627, 1988.

[15] S. Fortune and J. Wyllie. Parallelism in random access machines. In *STOC '78: Proc. 10th ACM Symp. on Theory of Computing*, pages 114–118, 1978.

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

[17] P. B. Gibbons. A more practical pram model. In *SPAA '89: Proc. 1st ACM Symp. on Parallel Algs. and Architectures*, pages 158–168, 1989.

[18] J. JáJá. *An introduction to parallel algorithms*. Addison Wesley Longman Publishing Co., Inc., 1992.

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

[20] C. Papadimitriou and M. Yannakakis. Towards an architecture-independent analysis of parallel algorithms. In *STOC '88: Proc. 20th ACM Symp. on Theory of Computing*, pages 510–513, 1988.

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

APPENDIX

A. EXPERIMENTS

We implemented a parallel scheduler for the LoPRAM model as described in Section 3.1 and tested it with two implementations of divide-and-conquer algorithms: Mergesort and Strassen's Matrix multiplication. The parallel implementations of these algorithms are essentially the sequential ones with PAL-threads for recursive calls.

The algorithms were implemented in C++ with MFC[1], and the experiments were run on an Intel(R) Core(TM) 2 Extreme CPU Q6850 (4 cores) at 3GHz, with 8Mb Cache and 4Gb RAM running Windows Vista(TM) Business 64-bit. Figures 1 and 2 show the times of the parallel execution of Mergesort and Matrix multiplication, respectively, with 1, 2, 3, and 4 cores, as well as the times of the sequential algorithm. The speedup with respect to the sequential times are shown on the right. The parallel execution with one core is an execution with the LoPRAM scheduler with 1 as the parameter for the number of available processors.

The difference in performance between the two algorithms can be explained in that as the merging phase of Matrix multiplication is more expensive than the one of Mergesort ($\Theta(n^2)$ vs $\Theta(n)$), the overhead due to the parallel scheduling of recursive calls is less significant for the first algorithm, leading to almost optimal speedup in all instances. In addition, the fact that for Mergesort the parallel execution with one core is faster than the sequential execution is explained in that the operating system schedules the sequential execution in all four cores of the system, incurring in an extra cost for context switching.

These preliminary experiments show that straightforward modifications to these sequential algorithms are sufficient to obtain significant, and nearly optimal speedups in the LoPRAM model.

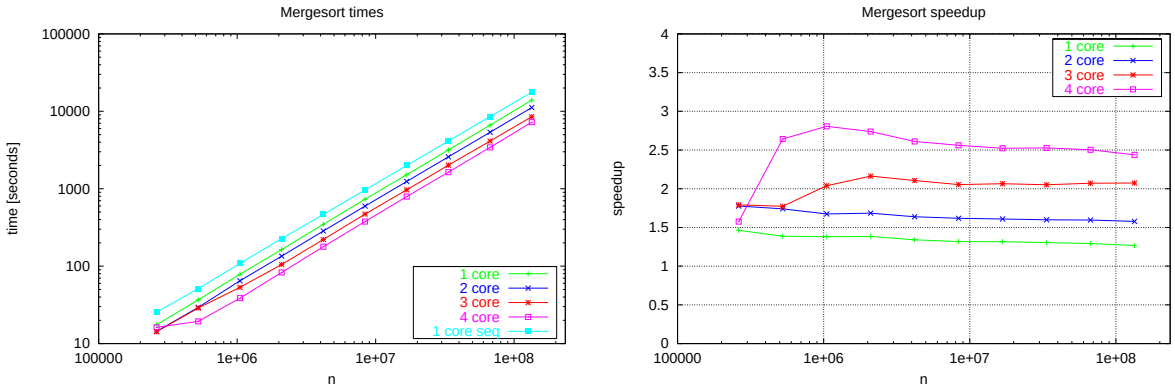

Figure 1: Left: Mergesort times for randomly generated inputs with sizes from 2^{28} to 2^{27}. For each input size, the shown time is the average of 25 runs on different inputs. Right: Speedup with respect to the sequential execution.

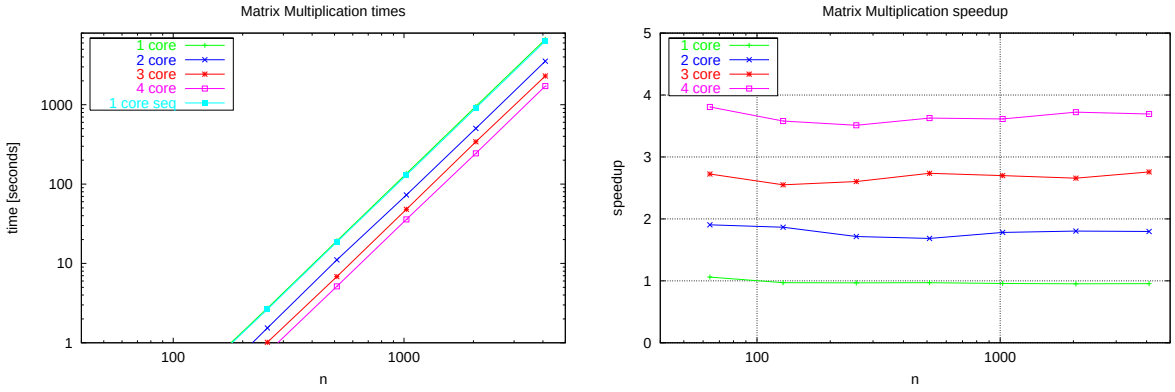

Figure 2: Left: Matrix multiplication times for randomly generated inputs with sizes from 64 to 4096. For each input size, the shown time is the average of 5 runs on different inputs. Right: Speedup with respect to the sequential execution.

[1]Microsoft Foundation Class.