Computational Geometry in the Parallel External Memory Model

Nodari Sitchinava
Institute for Theoretical Informatics
Karlsruhe Institute of Technology
nodari@ira.uka.de

1 Introduction

Continued advances in VLSI scaling combined with unsustainable power consumption of frequency scaling resulted in parallel processors having become mainstream as demonstrated by modern multicores. Current off-the-shelf processors already contain up to 16 cores [4] and the prototypes boast up to 48 cores [14] on a single chip.

The wide availability of multicores renewed interest in parallel algorithm research for these architectures. While multicores resemble the well-studied PRAM model in the way that the interprocessor communication is performed via writing to and reading from the shared memory, unlike the PRAM model, multicores also contain low-latency caches and load data from shared memory in blocks – cache lines – larger than a single word. Thus, researchers started looking into new ways to address parallelism and memory hierarchies of these processors.

In the sequential setting, the I/O model of Aggarwal and Vitter [1] serves as a simple model to consider spatial locality in algorithm design, which results in algorithms which are also cache-efficient [11]. However, the I/O model is sequential and does not address processor-level parallelism. The only parallel variant of the I/O model considers access to multiple disks in parallel while still processing data using a single processor.

Recently, the *parallel external memory (PEM)* model has been introduced to study the presence of private caches in the multiprocesor environment and a number of problems have been studied in this model [2, 3, 6, 7, 15]. There is a number of fundamental geometric problem that arise in the Geographic Information Systems. In this article we present an overview of the geometric problems that can be solved in parallel and I/O-efficiently in the PEM model.

2 The Model

The PEM model is a simple multiprocessor extension of the I/O model of Aggarwal and Vitter [1]. It consists of P processing elements (PEs), each containing a private cache of size M. All processors share main memory of conceptually unlimited size. Each processor enjoys exclusive access to its own cache. Just as in the sequential I/O model, the data has to be

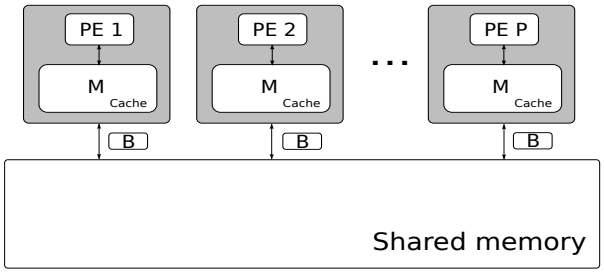

Figure 1: The PEM model.

present in the processor's cache before the processor can perform any operation on it. The transfer between the shared memory and caches is performed via *parallel I/O* operations. In one such operation *each* processor can transfer one block of size B between its cache and the shared memory. The complexity metric of an algorithm – the *parallel I/O complexity* – is the maximum number of I/O operations that any one of the processors performs throughout the computation. Just as in the PRAM model, we can distinguish variants of the model depending on how they handle concurrent accesses to the same block of data in shared memory by multiple processors. The version of the model that we consider in this article allows concurrent reading from the same block, but disallows concurrent writing to the same block even if it is to different addresses within the block. Thus, it is up to the algorithm designer to ensure that no concurrent writes happen to the same block.

While there are a number of other theoretical models that model modern multicores, including their more advanced multi-level memory hierarchies (e.g. see [9, 10, 12]), the PEM model offers the simplest way to study parallelism and cache-efficiency required for efficient computations on modern multicores.

2.1 Fundamental Results

The original paper on the PEM model by Arge et al. [6] studied the fundamental problems of parallel prefix sums and sorting, which they showed can be performed optimally in $\Theta(N/PB+\log P)$ and $\Theta\left(\frac{N}{PB}\log_{M/B}\frac{N}{B}\right) = \text{sort}_P(N)$ parallel I/O complexity, respectively. Note, if the number of processors is at most $\frac{N}{B\log N}$, the I/O complexity of the prefix sums solution reduces to $\Theta(N/PB) = \text{scan}_P(N)$, the I/O complexity of scanning N elements using P processors.

3 Parallel Distribution Sweeping

Distribution sweeping [13] is an algorithmic technique that has been used to solve a number of geometric problems I/O-efficiently. The technique recursively partitions the plane into $\Theta(M/B)$ slabs, each containing roughly an equal number of objects. At each recursive level, a vertical sweep of objects is performed during which a subset of output is reported and the objects are distributed into the appropriate slabs in preparation for the next recursive level. The recursion stops when the objects of each slab fit in internal memory, i.e. after $\Theta(\log_{M/B}(N/M))$ recursive calls.

Ajwani et al. [2] extend the distribution sweeping framework to the PEM model. First, they set the number of vertical slabs at each recursive level to $k = \min\{\sqrt{N/P}, M/B\}$[1] and the recursion proceeds until there are P slabs at which point each slab can be processed I/O-efficiently using a single processor. Next, they address the two main challenges that arise with parallelizing distribution sweeping: (1) performing the sweep in parallel and (2) load balancing the output reporting for output-sensitive algorithms to ensure that no processor spends more than $\text{O}(K/PB)$ I/Os for output of size K.

There are two approaches to deal with these challenges. The first approach delays the reporting to the last recursive level by creating duplicates of the objects within the slabs. The second approach reports intersections immediately at each recursive level just as in the sequential distribution sweeping. However, the second approach requires additional I/O operations to perform load-balancing at each recursive level. We will describe the two approaches using the problem of

[1]The choice of k comes from the k-way PEM mergesort of Arge et al. [6] and under their assumption of $M = B^{\text{O}(1)}$ ensures that $k = \text{O}(N/PB)$, the I/O complexity of a parallel scan of the input.

orthogonal line segment intersection reporting as a running example: given a set of axis-aligned line segments, report all pairs of segments that intersect. The problem has optimal sequential I/O complexity of $\Theta\left((N/B)\log_{M/B}(N/B) + K/B\right)$, i.e., the *sequential* sorting complexity of the input and scanning complexity of the output. We will try to achieve the corresponding *parallel* I/O complexities in the PEM model.

The advantage of the first approach is the relative simplicity of load-balancing of the output reporting, which happens only at the lowest level of the recursion. However, one must be careful which segments to duplicate to make sure that not too many duplicate segments are created, thus, offsetting the benefits of the parallelism. In particular, a horizontal segment is duplicated within a vertical slab iff the segment spans the slab and intersects at least one vertical segment within that slab. The problem of detecting whether a horizontal segment spanning a slab intersects a vertical segment within that slab reduces to the problem of *colored prefix sums* with k colors: given an array of numbers and a color associated with each number, compute for each color the prefix sums among the numbers of that color. This problem has been solved by Arge et al. [6] as long as there are at most M different colors. Since a horizontal segment is duplicated within a slab only if it intersects a vertical segment within that slab, at most K additional horizontal segments might be created. Thus, this approach results in $\mathrm{O}(\mathrm{sort}_P(N+K))$ parallel I/O complexity for the line segment intersection reporting problem. Note that for *large* K this is a logarithmic factor off from optimality.

In the second approach, the intersections between horizontal segments spanning a slab and vertical segments within that slab are reported immediately at each recursive level. This avoids creating extra copies of segments. However, to load balance the output reporting among multiple processors, one needs to compute the number of intersections to be reported for each vertical segment at each recursive level. This problem reduces to the problem of *one-dimensional range counting*: for a set of one-dimensional ranges and points, report how many points lie within each range. The known I/O-efficient solution to this problem is easily parallelizable but requires sorting the entire input. Performing a sorting step at each of the $\mathrm{O}(\log_k P)$ recursive levels adds up to $\mathrm{O}(\mathrm{sort}_P(N)\log_k P + \mathrm{scan}_P(K))$ overall parallel I/O complexity for the line segment intersection reporting problem. Note that this is a logarithmic factor off from optimality for *small* K.

In a later paper [3], Ajwani et al. found a new way to perform one-dimensional range counting I/O-efficiently without sorting if the input is already pre-sorted, which can be performed once at the beginning of the algorithm. While the solution is more complicated, it results in the optimal $\Theta(\mathrm{sort}_P(N) + \mathrm{scan}_P(K))$ parallel I/O complexity for the problem of line segment intersection reporting in the PEM model for all values of N and K.

3.1 Additional Problems

The distribution sweeping framework is the key technique to solving a number of geometric problems in the I/O model. Using the idea of *multislabs* [5], parallel distribution sweeping framework can be used to solve the problem of *2D range reporting* in the PEM model. The problem is defined as follows: given a set of axis-aligned rectangles and a set of points on a plane, report all point-rectangle pairs such that the point lies inside the rectangle. Finally, combining the solutions to line segment intersection reporting and 2D range reporting, one also obtains a PEM solution to the *rectangle intersection reporting* problem (given a set of axis-aligned rectangles, report all pairs of non-disjoint rectangles).

Unfortunately, the improved 1D range counting problem used for load-balancing the reporting step in the optimal PEM solution to the line segment intersection reporting problem does not directly apply to the multislab setting. Thus, the best parallel I/O complexities for these two problems remain $O(\text{sort}_P(N+K))$ or $O(\text{sort}_P(N)\log_k P + \text{scan}_P(K))$ [2]. It remains an open problem if one can achieve the optimal I/O complexity of $\Theta(\text{sort}_P(N) + \text{scan}_P(K))$ for these two problems and, in general, for problems that use the multislab technique.

Using the ideas of multi-way merging in the PEM model [6], one can also obtain solutions to the problems of *weighted dominance counting* of a point set on a plane, computation of the *lower envelop* of a set of non-intersecting line segments on a plane, and the computation of the *three-dimensional maxima* of a point set in $O(\text{sort}_P(N))$ parallel I/O complexity in the PEM model [2].

Finally, a careful analysis of the PRAM algorithm for computing a *convex hull* of a point set on a plane [8] reveals it to be I/O-efficient as well. Thus, a convex hull in two dimensions can also be computed in $O(\text{sort}_P(N))$ parallel I/O complexity in the PEM model [2].

4 Parallel Buffer and Range Trees

Most recently, Sitchinava and Zeh [15] describe how to build a parallel version of the buffer tree and extend it to obtain a parallel range tree data structure in the PEM model.

Buffer tree [5] is an external memory data structure which is efficient in solving offline dynamic problems. In dynamic problems, the queries must be answered correctly in the presence of updates to the data structure, e.g., in the form of insert and delete operations, which are interspersed with the queries. The sequential buffer tree is able to achieve $\Theta\left(\frac{1}{B}\log_{M/B}\frac{N}{B}\right)$ I/O complexity per each of N query and update operations and $\Theta\left(\frac{1}{B}\right)$ I/O complexity per each output point reported. Note, that since these I/O complexities can be less than one I/O, the bounds are amortized over all updates and queries and to be able to achieve these bounds, the problems must be offline, i.e., all updates and queries must be given upfront.

Sequential buffer tree is a balanced $\Theta(M/B)$-way search tree with each internal node augmented with a buffer of size M. The update operations and queries are inserted into the buffer of the root node of the tree and are only processed when the buffer becomes full. The processing of a buffer at a node involves distributing the contents to the buffers of the appropriate children nodes. During this process, answers are reported for any query for which there is sufficient information within the buffer alone; otherwise it is propagated to the appropriate child's buffer for further processing. If the buffer of a child becomes full in the process, it is processed recursively. The recursive buffer processing continues until the leaf nodes. At that time all queries on the root to leaf path will have been answered and the tree is rebalanced if it is necessary.

In the *parallel buffer tree* the buffers are increased to size g each and internal nodes' fanouts are increased to $\Theta(g/B)$. The buffer size parameter g depends on the specific problem being solved. In particular, g is defined to be the smallest size for which the problem can be solved I/O-efficiently using all P processors in the PEM model. Then each buffer can be processed using all P processors in parallel while answering queries and distributing the buffer's content to the appropriate children's buffers without loss in parallel I/O-efficiency. For example, the parallel buffer tree can answer *offline dynamic membership queries* in $\Theta\left(\frac{1}{PB}\log_{M/B}\frac{N}{B}\right)$ amortized parallel I/O complexity per each of N query and update operations by setting $g = \Theta(PB^2)$.

Using similar method as in the sequential setting and by setting the buffer size to $g = \Theta(PB^2 \log P)$,

the parallel buffer tree can be used to implement a *PEM range tree* data structure to answer *offline dynamic range queries* in parallel and I/O-efficiently. The data structure achieves optimal amortized parallel I/O complexity of $\Theta\left(\frac{1}{PB}\log_{M/B}\frac{N}{B}\right)$ per each of N query and update operations and $\Theta\left(\frac{1}{PB}\right)$ I/O complexity per each output point reported.

While sequential buffer tree can also be used to construct an *external segment tree*, it seems more challenging in the parallel setting and the existence of a parallel segment tree in the PEM model remains an open problem.

References

[1] A. Aggarwal and J. S. Vitter. The input/output complexity of sorting and related problems. *Communications of the ACM*, 31(9):1116–1127, 1988.

[2] D. Ajwani, N. Sitchinava, and N. Zeh. Geometric algorithms for private-cache chip multiprocessors. In *ESA*, pages 75–86, 2010.

[3] D. Ajwani, N. Sitchinava, and N. Zeh. I/O-optimal distribution sweeping on private-cache chip multiprocessors. In *IPDPS*, pages 1114–1123, 2011.

[4] AMD Corp. New AMD Opteron processors deliver the ultimate in performance, scalability and efficiency. Press Release: `http://www.amd.com/us/press-releases/Pages/new-amd-opteron-processor-2011nov14.aspx`, Nov. 2011.

[5] L. Arge. The buffer tree: A technique for designing batched external data structures. *Algorithmica*, 37(1):1–24, 2003.

[6] L. Arge, M. T. Goodrich, M. J. Nelson, and N. Sitchinava. Fundamental parallel algorithms for private-cache chip multiprocessors. In *SPAA*, pages 197–206, 2008.

[7] L. Arge, M. T. Goodrich, and N. Sitchinava. Parallel external memory graph algorithms. In *IPDPS*, pages 1–11, Apr. 2010.

[8] M. J. Atallah and M. T. Goodrich. Parallel algorithms for some functions of two convex polygons. *Algorithmica*, 3:535–548, 1988.

[9] G. Blelloch, P. Gibbons, and H. Simhadri. Low depth cache-oblivious algorithms. In *SPAA*, pages 189–199, 2010.

[10] 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*, pages 501–510, 2008.

[11] Y.-J. Chiang. Experiments on the practical I/O efficiency of geometric algorithms: Distribution sweep versus plane sweep. *Comput. Geom.*, 9(4):211–236, 1998.

[12] R. A. Chowdhury, F. Silvestri, B. Blakeley, and V. Ramachandran. Oblivious algorithms for multicores and network of processors. In *IPDPS*, pages 1 –12, 2010.

[13] M. T. Goodrich, J.-J. Tsay, D. E. Vengroff, and J. S. Vitter. External-memory computational geometry. In *FOCS*, pages 714–723, 1993.

[14] Intel Corp. Futuristic Intel chip could reshape how computers are built, consumers interact with their PCs and personal devices. Press Release: `http://www.intel.com/pressroom/archive/releases/2009/20091202comp_sm.htm`, Dec. 2009.

[15] N. Sitchinava and N. Zeh. A parallel buffer tree. In *SPAA*, pages 214–223, 2012.