49. Oklobdzija VG (1998) Architectural tradeoffs for low power. In: Proceedings of the ISCA Workshop on Power-Driven Microarchitectures, Barcelona
50. Reed P et al (1997) 250 MHz 5 W RISC microprocessor with on-chip L2 cache controller. Dig Tech Pap IEEE Int Solid St Circ Conf 40:412
51. Sakurai T, Newton R (1990) Alpha-power law MOSFET model and its applications to CMOS inverter delay and other formulas. IEEE JSSC 25(2):584–594
52. Sanchez H et al (1997) Thermal management system for high performance PowerPC microprocessors. In: Digest of papers, IEEE COMPCON, p 325
53. Seng JS, Tullsen DM, Cai G (2000) The power efficiency of multithreaded architectures. In: Invited talk presented at: ISCA Workshop on Complexity-Effective Design (WCED), Vancouver
54. Singhal R (2008) Inside intel core microarchitecture (Nehalem). Presented at Hot Chips-20, Stanford
55. Sohi G, Breach SE, Vijaykumar TN (1995) Multiscalar Processors. In: Proceedings of the 22nd annual international symposium on computer architecture, IEEE CS Press, Los Alamitos, pp 414–425
56. Srinivasan V, Brooks D, Gschwind M, Bose P, Zyuban V, Strenski PN, Emma PG (2002) Optimizing pipelines for power and performance. In: Proceedings of the 35th annual IEEE/ACM symposium on microarchitecture (MICRO-35), ACM/IEEE, Istanbul
57. Talks presented at the ISCA workshops on complexity effective design (WCED-2000 through WCED-2006), http://www.csl.cornell.edu/~albonesi/wced.html
58. Talks presented at the kool chips workshops (1998) http://www.cs.colorado.edu/~grunwald/LowPowerWorkshop
59. Talks presented at the power aware computer systems (PACS) Workshops; e.g. the 2004 offering: http://www.ece.cmu.edu/~pacs04/
60. Tendler JM, Dodson JS, Fields JS Jr, Le H, Sinharoy B (2002) POWER4 system microarchitecture. IBM J Res Dev 46:(1):1–116
61. The International Technology Roadmap for Semiconductors. http://www.itrs.net/reports.html
62. Theme issue (1997) The future of processors. IEEE Comput 30(9):97–93
63. Tiwari V et al (1998) Reducing power in high-performance microprocessors. In: Proceedings of the IEEE/ACM Design Automation Conference. ACM, New York, pp 732–737
64. Tullsen DM, Eggers SJ, Levy HM (1995) Simultaneous Multithreading: Maximizing On-Chip parallelism. In: Proceedings of the 22nd annual internatonal symposium on computer architecture, Santa Margherita Ligure, pp 292–403
65. Yen D (2005) Chip multithreading processors enable reliable high throughput computing. Keynote speech at international symposium on reliability physics (IRPS)
66. Zyuban V (2000) Inherently lower-power high performance super scalar architectures. PhD thesis, Department of Computer Science and Engineering, University of Notre Dame
67. Papers in the special issue of IBM Journal of Research and Development, March/May 2002
68. Zyuban V, Brooks D, Srinivasan V, Gschwind M, Bose P, Strenski P, Emma P (2004) Integrated analysis of power and performance for pipelined microprocessors. IEEE T Comput 53(8):1004–1016
69. Zyuban V, Kogge P (2000) Optimization of high-performance superscalar architectures for energy efficiency. In: Proceedings of the IEEE symposium on low power electronics and design, ACM, New York

PRAM (Parallel Random Access Machines)

Joseph F. JaJa
University of Maryland, College Park, MD, USA

Definition

The Parallel Random Access Machine (PRAM) is an abstract model for parallel computation which assumes that all the processors operate synchronously under a single clock and are able to randomly access a large shared memory. In particular, a processor can execute an arithmetic, logic, or memory access operation within a single clock cycle.

Discussion

Introduction

Parallel Random Access Machines (PRAMs) were introduced in the late 1970s as a natural generalization to parallel computation of the *Random Access Machine* (RAM) model. The RAM model is widely used as the basis for designing and analyzing sequential algorithms. The PRAM model assumes the presence of a number of processors, each identified by a unique id, which have access to a single unbounded shared memory. The processors operate synchronously under a single clock such that each processor can execute an arithmetic or logic operation or a memory access operation within a single clock cycle. In general, each processor under the PRAM model can execute its own program, which can be stored in some type of a private program memory. However, almost all the known PRAM algorithms are of the SPMD (Single Program Multiple Data) type in which a single program is executed by all the processors such that an instruction can refer to the id of the processor, which is responsible for executing the corresponding instruction. A processor may or may not be active during any given clock cycle. In fact, most of the

PRAM algorithms are of the SIMD (Single Instruction Multiple Data) type in which, during each cycle, a processor is either idle or executing the same operation as the remaining active processors.

As a simple example of a PRAM algorithm, consider the computation of the sum S of the elements of an array $A[1:n]$ using an n-processor PRAM, where the processors are indexed by $pid = 1, 2, \ldots, n$. A PRAM algorithm can be organized as a balanced binary tree with n leaves such that each internal node represents the sum computation applied to the values available at the children. The PRAM algorithm proceeds level by level, executing all the computations at each level in a single parallel step. Hence the processors complete the process in $\lceil \log n \rceil$ parallel steps. The algorithm is illustrated in Fig. 1.

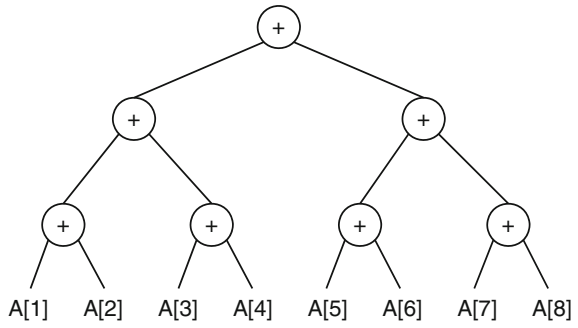
The PRAM algorithm written in the SPMD style is given next where for simplicity $n = 2^k$ for some positive integer k. The array $B[1 : n]$ is used to store the intermediate results.

Algorithm 1 PRAM SUM Algorithm

Input: An array A of size $n = 2^k$ stored in shared memory.
Output: The sum of the elements of A.
begin
  $B[pid] = A[pid]$
  $d = n$
  for $h = 1$ to k **do**
    $d = \frac{d}{2}$
    if $pid \leq d$ **then**
      $B[pid] = B[2pid - 1] + B[2pid]$
    end if
    end for
  return$(B[1])$
end

Given that the processors can simultaneously access the shared memory within the same clock cycle, several variants of the PRAM model exist depending on the assumptions made for simultaneous access to the **same** memory location.

- **Exclusive Read Exclusive Write (EREW) PRAM**. No simultaneous accesses to the same location are allowed under the EREW PRAM model.



PRAM (Parallel Random Access Machines). Fig. 1 A Balanced Binary Tree for the Sum Computation. The PRAM algorithm proceeds from the leaves to the root while executing the operations at each level within a single clock cycle

- **Concurrent Read Exclusive Write (CREW) PRAM**. Simultaneous read accesses to the same location are allowed but no concurrent write operations to the same location are permitted.
- **Concurrent Read Concurrent Write (CRCW) PRAM**. Simultaneous read or write operations to the same memory location are allowed. This model has several subversions depending on how the concurrent write operations to the same location are resolved. The **Common CRCW PRAM** model assumes that the processors are writing the same value, while the **Arbitrary CRCW PRAM** model assumes that one of the processors attempting a concurrent write into the same location succeeds. The **Priority CRCW** assumes that the indices of the processors are linearly ordered, and allows the processor with the highest priority to succeed in case of a concurrent write.

The different assumptions on the concurrent accesses to a single location lead to parallel algorithms that can differ significantly in performance. Consider, for example, the problem of determining whether all the entries of a binary array $M[1:n]$ are all equal to 1. The output A of this computation is in fact the logical AND of all the entries in M. An n-processor Common CRCW PRAM algorithm can perform this computation as follows. Initially, A is set equal to 1. For all $1 \leq pid \leq n$, processor pid tests whether the entry $M[pid] = 0$, and in the affirmative, executes the operation $A = 0$. Clearly this algorithm correctly computes the AND of the n elements

of M in constant time on the Common CRCW PRAM. However, it can be shown, under a very general model, that a CREW PRAM will require $\Omega(\log n)$ parallel steps regardless of the number of processors available. On the other hand, the computational gap between the various versions of the PRAM model is limited in the sense that the strongest p-processor PRAM model, namely the priority CRCW from our list above, can be simulated by a p-processor EREW (weakest PRAM model) with a slowdown of a factor of $O(\log p)$.

From a theoretical perspective, the design of a PRAM algorithm amounts to the development of a strategy that achieves the highest level of parallelism, that is, the smallest number of parallel steps, using a "reasonable" number of processors. Note that the number of processors can depend on the input size. In particular, the class of "well-parallelizable" problems under the PRAM model can be defined as the class of the problems that can be solved in polylogarithmic number of parallel steps (i.e., $O(\log^k n)$ for some fixed constant k), using a polynomial number of processors. Such a class is referred to as the NC complexity class, which has been related to the circuits model used in traditional computational complexity. See the bibliographic notes for references to related work.

Complexity Measures and Work–Time Framework

A PRAM algorithm can be evaluated by several complexity measures, where a complexity measure is a worst case asymptotic estimate of the performance as a function of the input length n. Given a p-processor PRAM algorithm to solve a problem with input size n, let $T_P(p,n)$ be the number of parallel steps used by the algorithm. The optimal (or often the best known) sequential complexity is denoted by $T_S(n)$. The **speedup** is defined to be

$$\frac{T_S(n)}{T_P(p,n)},$$

which represents the speedup of the parallel algorithm *relative to the best sequential algorithm*. Clearly the best possible speedup is $\Theta(p)$. On the other hand, the **relative speedup** is defined to be

$$\frac{T_P(1,n)}{T_P(p,n)},$$

which refers to the speedup achieved by the algorithm with p processors relative to the same algorithm running with one processor.

Another related measure is the **relative efficiency** of a PRAM algorithm defined as the ratio

$$\frac{T_P(1,n)}{pT_P(p,n)}.$$

The two PRAM algorithms presented earlier assume the presence of n processors, where n is the input size. To derive the complexity measures defined above, these algorithms can be mapped into p-processor ($p < n$) PRAM algorithms by distributing the concurrent operations carried out at each parallel step as evenly among the p processors as possible. In particular, the SUM algorithm can be executed in

$$\left\lceil \frac{n}{p} \right\rceil + \left\lceil \frac{n/2}{p} \right\rceil + \cdots + 1 = O\left(\frac{n}{p} + \log n\right)$$

parallel time on a p-processor PRAM algorithm. Hence the speedup is $\Theta(p)$ whenever $p = O(n/\log n)$. However, writing the corresponding algorithm for each processor (using processor ids) is in general a tedious process that distracts from the simplicity of the PRAM model.

An alternative is to describe a PRAM algorithm in the so-called **Work–Time** or simply **WT** framework, which is also related to the **data parallel** paradigm. The algorithm does not assume any specific number of processors, nor does it refer to any processor index, but rather it is expressed as a sequence of steps, where each step is either a typical sequential operation or a set of concurrent operations that can be executed in parallel. A set of parallel operations is specified by the pseudo-instruction **pardo** or **forall**. For example, the previous PRAM SUM algorithm can be expressed in this framework as shown in Algorithm 2.

Under the WT framework, the complexity of a PRAM algorithm can be measured by two parameters – **work** $W(n)$ and **time** $T_P(n)$. The work $W(n)$ is the *total* number of operations required by the algorithm as a function of n and the time $T_P(n)$ is defined as the number of parallel steps needed by the algorithm assuming an unlimited number of processors. The corresponding functions for the SUM algorithm are $W(n) = O(n)$ and $T_P(n) = O(\log n)$, respectively.

Algorithm 2 Parallel SUM Algorithm in the WT Framework

Input: An array A of size $n = 2^k$ residing in memory.
Output: The sum of the elements of A.
begin
  $d = n$
  for all $1 \le i \le n$ **pardo**
    $B[i] = A[i]$
  end for
  for $h = 1$ to k **do**
    $d = \frac{d}{2}$
    for all $1 \le i \le d$ **pardo**
      $B[i] = B[2i-1] + B[2i]$
    end for
  end for
  return$(B[1])$
end

In general, a WT-parallel algorithm with complexity $W(n)$ and $T_P(n)$ can be simulated on a p-processor PRAM using Brent's scheduling principle (see bibliographic notes) in time

$$T_P(p,n) = O\left(\frac{W(n)}{p} + T_P(n)\right)$$

The algorithm is called **work optimal** if $W(n) = \Theta(T_S(n))$. In this case, the corresponding p-processor PRAM algorithm achieves optimal speedup as long as $p = O\left(\frac{T_S(n)}{T_P(n)}\right)$.

Under the WT framework, a typical goal is to develop an algorithm that achieves the fastest parallel time among the work-optimal algorithms. That is, the main goal to develop a work-optimal algorithm that is as fast as possible.

Basic PRAM Techniques

Basic PRAM techniques are illustrated through the description of PRAM algorithms for the following computations: matrix multiplication, prefix sums or scan, list ranking, fractional independent set, and computing the maximum. The WT framework is used to express these algorithms.

Matrix Multiplication

Given two matrices A and B of dimensions $m \times n$ and $n \times q$, respectively, the product $C = AB$ is a matrix of dimension $m \times q$ such that

$$C[i,j] = \sum_{k=1}^{n} A[i,k]B[k,j],\ 1 \le i \le m,\ 1 \le j \le q.$$

Computing the matrix C on the PRAM model is quite simple since both matrices A and B are initially stored in shared memory, and their entries can each be accessed randomly within a single clock cycle. The algorithm in the WT framework amounts to computing in parallel all the products $A[i,k]B[k,j]$, for $1 \le i \le m$, $1 \le k \le n$, and $1 \le j \le q$. The PRAM SUM algorithm can then be used to compute all the entries $C[i,j]$ in parallel, $1 \le i \le m$ and $1 \le j \le q$. Thus, the parallel complexity of the resulting algorithm is $T_P = O(\log n)$ and the total work is $W = O(nmq)$.

This algorithm is based on the standard sequential algorithm. Should the initial sequential algorithm be one of the faster sequential matrix multiplication algorithm, the corresponding PRAM algorithm will then run in logarithmic parallel time using the same number of operations as the initial algorithm.

Prefix Sums or Scan

Given a set of elements stored in an array $A[1:n]$ and a binary associative operation $\otimes$, the prefix sums of A consist of the n partial sums defined by

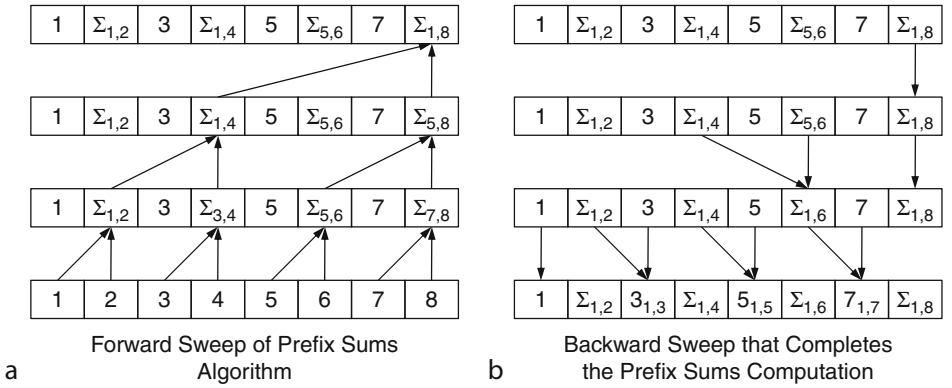
$$PS[i] = A[1] \otimes A[2] \otimes \cdots \otimes A[i], 1 \le i \le n$$

The straightforward sequential algorithm computes $PS[i] = PS[i-1] \otimes A[i]$ for $2 \le i \le n$ starting with $PS[1] = A[1]$, and hence the sequential complexity is $T_S(n) = \Theta(n)$.

A simple fast PRAM algorithm can be designed using a balanced binary tree built on top of the n elements of A. The algorithm consists of a forward sweep through the tree in a way similar to that carried out by the SUM algorithm. A backward sweep will then compute the prefix sums of the set of elements available at each level. A recursive version is described in Algorithm 3.

The algorithm is illustrated in Fig. 2, where the left part illustrates the forward sweep and the right part illustrates the backward sweep.

Since the algorithm involves a forward and a backward sweep through a balanced binary tree of height $O(\log n)$, the parallel complexity of the algorithm satisfies $T_P(n) = O(\log n)$. Also, since the number of



PRAM (Parallel Random Access Machines). Fig. 2 Computation of prefix sums through a forward sweep and a backward sweep of the balanced binary tree algorithm. The notation $\sum_{i,j}$ represents the sum of elements from $A[i]$ through $A[j]$, and at each level an entry with two arrows pointing to it represent a sum operation

Algorithm 3 Prefix Sums Algorithm

Input: An array A of size $n = 2^k$ stored in shared memory.
Output: The prefix sums of A stored in the array PS.
begin
  if $n = 1$ **then**
    return$(PS[1] = A[1])$
  end if
  for all $1 \le i \le n/2$ **pardo**
    $Y[i] = A[2i-1] \otimes A[2i]$
end for
  Recursively compute the prefix sums of $Y[1 : n/2]$ in place
  for all $1 \le i \le n$ **pardo**
    i even: set $PS[i] = Y[i/2]$
    $i = 1$: set $PS[1] = A[1]$
    else: set $PS[i] = Y[(i-1)/2] \otimes A[i]$
  end for
end

internal nodes of a binary tree on n leaves is $n - 1$, the total work is clearly $W(n) = O(n)$, and hence this algorithm is work optimal. It follows that a p processor PRAM can compute the prefix sums of n elements in time $T_P(p,n) = O\left(\frac{n}{p} + \log n\right)$.

List Ranking

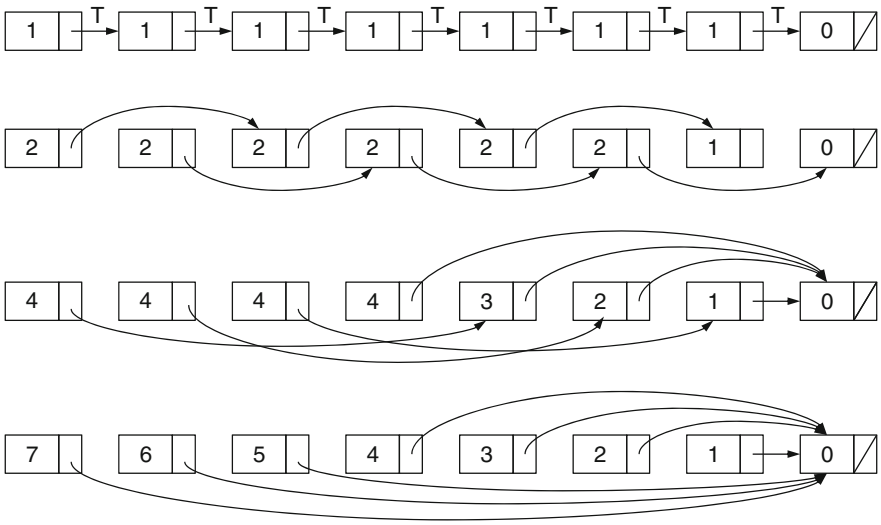
Consider a linked list L of n nodes whose successor relationship is represented by an array $S[1 : n]$ such that $S[i]$ is equal to the index of the successor node of i. For the last node k, its successor is denoted by $S[k] = 0$. The list ranking problem is to determine for each node i the distance $R[i]$ of the node i to the end of the list. While an optimal sequential algorithm is relatively straightforward – start by inverting the list and proceed to compute the ranks incrementally following the predecessor links – the problem may seem at first sight to be quite difficult to parallelize. It turns out that a fast PRAM algorithm can be developed through the introduction of the *pointer jumping* technique as illustrated in Algorithm 4. Note that T is used for the intermediate manipulation of the pointers, while the initial pointers stored in S remain intact.

The list ranking algorithm is illustrated in Fig. 3, where the pointer jumping technique is applied on each node i satisfying $T[i] \neq 0$ and $T[T[i]] \neq 0$.

Since the execution of the last parallel loop involves replacing the successor by its successor, the distance between a node and its successor doubles after each iteration and hence the T pointer of each node will point to the last node after $\lceil \log n \rceil$ iterations. Therefore, the algorithm terminates correctly after $O(\log n)$ iterations, and each iteration involves $O(n)$ operations. The algorithm has parallel complexity $O(\log n)$ using $O(n \log n)$ total number of operations. The work can be made optimal but the corresponding PRAM algorithm is more complex.

Fractional Independent Set

The purpose of introducing the next problem is to illustrate the use of randomization in the design of PRAM



PRAM (Parallel Random Access Machines). Fig. 3 Application of the list ranking algorithm on a list with eight elements

Algorithm 4 List Ranking Algorithm

Input: An array S of size n representing the successor relationship of a linked list.

Output: The distance $R[i]$ of node i to the end of the list, $1 \le i \le n$.

begin
  for all $1 \le i \le n$ **pardo**
  if $S[i] \neq 0$ **then** $R[i] = 1$ **else** $R[i] = 0$
end for
  for all $1 \le i \le n$ **pardo**
  $T[i] = S[i]$
end for
  repeat $\lceil \log n \rceil$ **times do**
  {
  for all $1 \le i \le n$ **pardo**
  if $T[i] \neq 0$ **and** $T[T[i]] \neq 0$ **then**
  $\{ R[i] = R[i] + R[T[i]]; T[i] = T[T[i]] \}$
end for
  }
end

algorithms to break symmetry. Given a directed cycle $C =< v_1, v_2, \ldots v_n >$, a fractional independent set is a subset U of the vertices such that: (i) U is an independent set, that is, no two vertices in U are connected by a directed edge; and (ii) the size of U is a constant fraction of the size of V. The fractional independent set problem is to determine such a set.

It is trivial to develop a sequential algorithm to solve this problem. Starting from any vertex, place every other vertex in U. A simple and fast PRAM algorithm can be designed using randomization as follows. Each vertex, in parallel, is randomly assigned a label 1 or 0 with equal probability. Clearly, the expected number of vertices of each label is $n/2$. However, this does not necessarily solve the problem since there is no guarantee that the vertices with the same label form an independent set. To ensure that this is indeed the case, another parallel step is carried out which involve changing the label of a vertex to 0 if the labels of this vertex and its successor are both equal to 1. The remaining vertices of label 1 are now guaranteed to form an independent set. It can be shown that, with high probability, the size of such an independent set is a constant fraction of the size of the original vertex set V.

On the PRAM model, this algorithm runs in $O(1)$ parallel time using $O(n)$ operations. It is worth noticing that this algorithm can in particular be used to make the list ranking algorithm more efficient (i.e., using a total number of operations which is asymptotically less than $n \log n$).

Superfast Maximum Algorithm

The algorithm presented in this section illustrates an important PRAM technique, called *accelerated cascading*, in combining two strategies. The first amounts to a work optimal algorithm (possibly a sequential

Algorithm 5 Randomized Fractional Independent Set Algorithm

Input: A directed cycle with a vertex set V whose arcs are specified by an array $S[1:n]$, i.e., $< i, S[i] >$ is an arc.
Output: A fractional independent set $U \subset V$.
begin
  for all $v \in V$ **pardo**
  Randomly assign $label(v) = 1$ or 0 with equal probability
  if $label(v) = 1$ and $lavel(S[v]) = 1$ **then** $label(v) = 0$
end for
  return$(U = \{v | label(v) = 1\})$
end

algorithm) to reduce the size of the problem below a certain threshold. The second strategy uses a very fast PRAM algorithm that involves a nonoptimal number of operations. The maximum algorithm will be used to illustrate such a technique and to also demonstrate the extra power of the PRAM model when concurrent write operations are allowed.

The PRAM strategy introduced earlier to compute the sum of n elements can be used to compute the maximum of n elements, resulting in the parallel time complexity $T_P(n) = O(\log n)$ and total work of $W(n) = O(n)$. However, it is possible to develop a *constant time* algorithm on the Common CRCW PRAM model as illustrated in **Algorithm 6**.

Algorithm 6 Constant Time Maximum Algorithm

Input: An array $A[1:n]$ consisting of n distinct elements
Output: A Boolean array $M[1:n]$ such that $M[i] = 1$ if, and only if, $A[i]$ is the maximum element
begin
  for all $1 \le i, j \le n$ **pardo**
  if $A[i] \ge A[j]$ **then** $B[i,j] = 1$ **else** $B[i,j] = 0$
end for
  for all $1 \le i \le n$ **pardo**
  $M[i] = \bigwedge_{j=1}^{n} B[i,j]$
end for
end

The Boolean AND of n binary variables can be performed in $O(1)$ parallel steps using $O(n)$ operations on the Common CRCW PRAM. Therefore, the above algorithm achieves constant parallel time but uses $O(n^2)$ operations, and thus it is extremely inefficient. To remedy this problem, and still achieve faster than $O(\log n)$ parallel time, a doubly logarithmic depth tree is used instead of the balanced binary tree. Essentially, the root of a doubly logarithmic depth tree has $\Omega\left(\sqrt{n}\right)$ children, and each subtree is defined similarly. It is not hard to see that such a tree has height $O(\log\log n)$, where n is the number of leaves. Using the doubly logarithmic depth tree and the constant time maximum algorithm at each node of the tree, the maximum can be computed in $O(\log\log n)$ time using $O(n \log\log n)$ operations.

The accelerated cascading strategy is now used to turn this fast but nonoptimal work algorithm into a fast and work optimal algorithm as follows:

- Partition the array A into $n/\log\log n$ blocks, such that each block contains approximately $\log\log n$ elements and use the sequential algorithm to compute the maximum element in each block.
- Use the doubly logarithmic depth tree on the maxima computed in the first step.

The resulting algorithm has a parallel time complexity of $O(\log\log n)$ using only $O(n)$ operations, and hence is work optimal. Therefore, the maximum can be computed in $O(\log\log n)$ parallel time using a work optimal strategy on the Common CRCW PRAM.

Bibliographic Notes and Further Reading

The PRAM model was initially introduced through a number of papers, most notably in the papers by Fortune and Wyllie [2], Goldschlager [3], and Ladner and Fisher [9]. The Work–Time framework, which ties Brent's scheduling principle [1] and the PRAM model, was first observed by Shiloach and Vishkin [11] and used extensively in the book by JaJa [6]. Related data parallel algorithms were introduced by Hillis and Steele [5]. Substantial work dealing with the design and analysis of PRAM algorithms and the theoretical underpinnings of the model has been carried out in the 1980s and 1990s. An early survey is the paper by Karp and Ramachandran [7], and an extensive coverage of the topic can be found in JaJa's book [6].

An intriguing relationship exists between the PRAM model and the circuits model used in traditional computational complexity.The NC class was formally introduced by Pippenger [10] for circuits. An important related theoretical direction is based on the notion of P-completness, which tries to shed light on problems that do not seem to be highly parallelizable under the PRAM model with polynomial numbers of processors. Interested reader can consult the reference [4] for a good overview of this topic.

Some recent efforts have been devoted to advocate the PRAM as a practical parallel computation model, both in terms of developing prototype hardware that supports the model and software that enables the writing of PRAM programs. The book of Keller, Kessler, and Traff [8] and the recent paper by Wen and Vishkin [12] are illustrative of such efforts.

Bibliography

1. Brent R (1974) The parallel evaluation of general arithmetic expressions. JACM 21(2):201–208
2. Fortune S, Wyllie J (1978) Parallelism in random access machines. In: Proceedings of the tenth ACM symposium on theory of computing. San Diego, CA, pp 114–118
3. Goldschlager L (1978) A unified approach to models of synchronous parallel machines. In: Proceedings of the tenth ACM symposium on theory of computing. San Diego, CA, pp 89–94
4. Greenlaw R, Hoover HJ, Ruzzo WL (1995) Limits to Parallel Computation: P-Completeness Theory. In: Topics in parallel computation. Oxford University Press, Oxford
5. Hillis WD, Steele GL (1986) Data parallel algorithms. Commun ACM 29(12):1170–1183
6. JaJa J (1992) An introduction to parallel algorithms. Addison Wesley Publishing Co., Reading, MA
7. Karp RM, Ramachandran V (1990) Parallel algorithms for shared-memory machines. In: van Leeuwen J (ed) Handbook of theoretical computer science, North Holland, Amsterdam, The Netherlands, Chapter 17, pp 869–942
8. Keller J, Kessler C, Traff J (2001) Practical PRAM programming. Wiley, New York
9. Ladner R, Fisher M (1980) Parallel prefix computations. JACM 27(4):831–838
10. Pippenger N (1979) On simultaneous resource bounds. In: Proceedings twentieth annual IEEE symposium on foundations of computer science. San Juan, Puerto Rico, pp 307–311
11. Shiloach Y, Vishkin U (1982) An $O(n2 \log n)$ parallel max-flow algorithm. J Algorithms 3(2):128–146
12. Wen X, Vishkin U (2008) FPGA-based prototype of a PRAM-on-chip processor. In: Proceedings of the 2008 ACM conference on computing frontiers. Ischia, Italy, pp 55–66

Preconditioners for Sparse Iterative Methods

Anshul Gupta
IBM T.J. Watson Research Center, Yorktown Heights, NY, USA

Synonyms

Linear equations solvers

Definition

Iterative methods for solving sparse systems of linear equations are potentially less memory and computation intensive than direct methods, but often experience slow convergence or fail to converge at all. The robustness and the speed of Krylov subspace iterative methods is improved, often dramatically, by *preconditioning*. Preconditioning is a technique for transforming the original system of equations into one with an improved distribution (clustering) of eigenvalues so that the transformed system can be solved in fewer iterations. A key step in preconditioning a linear system $Ax = b$ is to find a nonsingular *preconditioner* matrix M such that the inverse of M is as close to the inverse of A as possible and solving a system of the form $Mz = r$ is significantly less expensive than solving $Ax = b$. The system is then solved by solving $(M^{-1}A)x = M^{-1}b$. This particular example shows what is known as *left preconditioning*. There are two other formulations, known as *right preconditioning* and *split preconditioning*. The basic concept, however, is the same. Other practical requirements for successful preconditioning are that the cost of computing M itself must be low and the memory required to compute and apply M must be significantly less than that for solving $Ax = b$ via direct factorization.

Discussion

Preconditioning methods are being actively researched and have been for a number of years. There are several classes of preconditioners; some are more amenable to being computed and applied in parallel than others. This chapter gives an overview of the generation (i.e., computing M in parallel) and application (i.e., solving a system of the form $Mz = r$ in parallel) of the most commonly used parallel preconditioners.