North-Holland

HIGH PERFORMANCE COMPUTER SYSTEMS

E. Gelenbe
Editor

HIGH PERFORMANCE COMPUTER SYSTEMS

Proceedings of the International Symposium on
High Performance Computer Systems
Paris, France, 14–16 December 1987

edited by

Erol GELENBE
Ecole Des Hautes Etudes en Informatique
Université René Descartes
Paris, France

1988

NORTH-HOLLAND
AMSTERDAM · NEW YORK · OXFORD · TOKYO

ISBN: 0 444 70485 x

Publishers:
ELSEVIER SCIENCE PUBLISHERS B.V.
P.O. Box 1991
1000 BZ Amsterdam
The Netherlands

Sole distributors for the U.S.A. and Canada:
ELSEVIER SCIENCE PUBLISHING COMPANY, INC.
52 Vanderbilt Avenue
New York, N.Y. 10017
U.S.A.

LIBRARY OF CONGRESS
Library of Congress Cataloging-in-Publication Data

International Symposium on High Performance Computer Systems (1987 Paris, France)
High performance computer systems : proceedings of the International Symposium on High Performance Computer Systems, Paris, France, 14-16 December 1987 / edited by E. Gelenbe.
p. cm.
ISBN 0-444-70485-X
1. Electronic digital computers--Evaluation--Congresses.
2. Parallel processing (Electronic computers)--Congresses.
3. Computer architecture--Congresses. I. Gelenbe, E., 1945-
II. Title.
QA76.9.E94I575 1987
004'.35--dc19 88-19725
CIP

PRINTED IN THE NETHERLANDS

EDITOR'S PREFACE

The International Symposium on High Performance Computer Systems was held in Paris, France, on December 14 to 16, 1987 in the Conference Center of the French Ministry of Research and Technology.

It received the moral and financial support of the C3 Research Programme (Communication, Concurrency, Cooperation) of the French Research Ministry and of CNRS, and was organized by Ecole des Hautes Etudes en Informatique.

Its purpose was to bring together researchers in the various areas of computer science and engineering which contribute to the usage, development and design of advanced computer systems and of their relevant application algorithms.

Many novel advanced high performance architectures were presented at the symposium, including the RP3, the Teraflop 1, and the LCAP systems from IBM, the Connection Machine built by Thinking Machines Corp. and the Hypercube pioneered by Caltech.

These proceedings contain most of the papers presented at the meeting, while the authors of some other presentations concerning recent industrial projects did not make available written reports on the work they presented for understandable reasons.

The papers in these proceedings are grouped into six subject areas: Parallel Algorithms, Performance Measurement, Performance Analysis, Performance Modelling of Parallel Systems, User Needs and New Architectures, and Distributed Computations and Systems.

All of the papers in the first subject area concern numerical algorithms, and their design and adaptation for general purpose or specific parallel architectures. Since highly parallel large scale computer systems are designed for high performance, it

is important that appropriate methods exist for monitoring and measuring their performance. This issue is examined in the three papers contained in the second subject area, and it is useful to notice that two of these papers emanate from the National Bureau of Standards (USA).

Issues related to the analysis of parallel computer system performance, pertaining in particular to two novel architectures (the Hypercube, and the Connection Machine) are examined in the two papers of the third part.

Theoretical performance models which, in this particular instance are relatively independent of a particular machine, are dealt with in the fourth part of these proceedings.

Three papers dealing with the effect of very large main memories on the performance of data base machines, with a recent commercial parallel architecture (the SCS-40) and with the needs for supercomputing for the academic environment, are discussed in the fifth part. Two papers concerning granularity of distributed computation and reliability issues complete the present volume.

I wish to express my gratitude to J.C. BERMOND, M. COSNARD, G. FAYOLLE, S. LAVENBERG, A. LICHNEWSKY, and D. KUCK for accepting to be on the organizing committee of this meeting and to M. AUGUIN, F. BACCELLI, F. BOERI, J. DONGARA, D. EVANS, J.M. FOURNEAU, G. MAZARE, E. MONTAGNE, J. LENFANT, H. MUHLENBEIN, B. PLATEAU, G. PUJOLLE, P. QUINTON, Y. SAAD, and U. SCHENDEL for their membership of the programme committee.

In addition to these thanks, I would like to express my special gratitude to a number of colleagues and friends who have helped considerably in making this meeting a success. They include Thérèse BRICHETEAU of INRIA, Colette CONNAT, Michel COSNARD, Jean-Michel FOURNEAU, Hector GARCIA-MOLINA, Claude OUANNES, Steve LAVENBERG, Alain LICHNEWSKY, Maurice NIVAT, Jean-Claude SIMON, Satish TRIPATHI, and Jean-Pierre VERJUS, as well as Arjen SEVENSTER of North-Holland for his continued support and encouragement.

Erol GELENBE
Paris, April 1988

CONTENTS

DISTRIBUTED COMPUTATIONS AND SYSTEMS

PARALLEL ALGORITHMS

HIGH PERFORMANCE COMPUTER SYSTEMS
E. Gelenbe, (Editor)

TECHNIQUES FOR PARALLEL MANIPULATION OF SPARSE MATRICES (Extended Abstract)

Clyde P. Kruskal

Computer Science Department
Institute for Advanced Computer Studies
University of Maryland
College Park, Maryland 20742

Larry Rudolph

Department of Computer Science
The Hebrew University of Jerusalem
Jerusalem 91904, Israel

Marc Snir

IBM T.J. Watson Research Center
P.O. Box 218
Yorktown Heights, New York 10598

ABSTRACT

We present new techniques for the manipulation of sparse matrices on parallel MIMD computers. This yields efficient algorithms for the following problems: matrix addition, matrix multiplication, row and column permutation, matrix transpose, matrix vector multiplication, and Gaussian elimination.

1. INTRODUCTION

Many practical problems have computational solutions that involve solving large systems of linear equations as well as other types of manipulations of large matrices. In many cases these matrices are *sparse,* i.e. the number of nonzero entries is small. Consequently, it is desirable to use data representations and algorithms that avoid wasting space and time for zero entries. Sparse matrix algorithms for sequential machines have been extensively studied.

We will consider parallel algorithms that perform the same arithmetic operations that would be performed by a sequential algorithm. This does not preclude the achievement of significant speedups. Simple matrix algorithms contain many independent arithmetic operations that can be executed in parallel. The issue is the overhead required to allocate the arithmetic operations evenly to the processors of a parallel computer and to compute the locations of the operands to these operations. We desire overhead per processor to be proportional to the amount of arithmetic processing done, so that the total execution time is equal, up to a (small) multiplicative constant, to the time required to perform the arithmetic operations, using all available processors. Such algorithms will exhibit optimal speedup.

The algorithms use a compact representation of matrices, where only the nonzero entries are stored. In order to avoid sequential bottlenecks, the entries are not necessarily stored in row or column order, rather as a set of tuples. We show how radix sort and prefix operations can be used to organize such a bag of entries according to the needs of the computation and to distribute work evenly to all processors.

It is fairly easy to obtain a matrix addition algorithm that executes with optimal speedup. The difficulty is to use space proportional only to the sizes of the input and output matrices. Our algorithm achieves this.

For Gaussian elimination we introduce a new technique of "lazy evaluation". A redundant matrix representation is used, which may have several entries with the same row and column indices; the "true" value of the matrix entry is the sum of all these entries. Entries are summed only when the true value is needed, or when there is a sufficient backlog of work to justify a scan and compaction of the entire data structure.

2. FOUNDATIONS

We assume the EREW PRAM computation model with p autonomous processors, all having access to shared memory: At each step each processor performs one operation from its instruction stream. An instruction may involve access to shared memory. Concurrent access to the same memory location by several processors is forbidden.

Let m denote the size of the problem, that is, the number of nonzero entries in all the matrices involved, or, equivalently, the number of nonzero values in the input and in the output. Let $T(m)$ be the sequential time to solve a problem, and $T_p(m)$ be the parallel time with p processors. The *speedup* of a parallel algorithm is $T_1(m)/T_p(m)$ and its *efficiency* is $T_1(m)/pT_p(m)$. An algorithm is *efficient* if its efficiency is bounded away from zero, so that it achieves a speedup proportional to p, for large enough m. We also wish to use an amount of memory that does not exceed the size of the input or the output by more than a constant factor.

Matrices are assumed to be sparse (most entries are zero), but not too sparse. Let d denote the sum of the lengths of sides of the matrices. Then, formally, we assume $m = \Omega(d^{\epsilon})$ for some positive ϵ. In most cases of interest the number of nonzero entries will be at least linear in the dimensions of the matrices, so this assumption is quite reasonable. We also assume that the problem size is somewhat larger than the number of processors used, or, more formally, that $m = \Omega(p^{1+\delta})$ for some positive constant δ. Both assumptions are likely to be fulfilled by problems that are large enough to justify the use of parallel computers.

The *canonical* representation of a sparse matrix stores the matrix as a set of quadruples, one quadruple for each nonzero element. The four components of each quadruple are matrix name, row index, column index, and value. The quadruples in the list can be stored in arbitrary order. The name of the matrix need not be explicitly stored at each element; algorithms that operate on two matrices (e.g. matrix addition or matrix multiplication algorithms) will need an extra bit per element to differentiate the matrices. A *row major* representation of a matrix stores the nonzero elements in row major order. The matrix is represented by a list that contains for each row the number of elements in this row, followed by an ordered list of the nonzero elements in this row, represented by column index and value. This represen-

tation is more space efficient whenever the matrix is not too sparse, i.e. whenever there is at least one entry per row, on the average. The *column major* representation of a matrix is defined similarly.

3. BASIC PARALLEL ROUTINES

The sparse matrix algorithms are easy to explain and understand given several basic routines. They all require time $O(n/p + \log p)$ when using p processors.

Parallel Prefix: Given n numbers $x_1, \ldots, x_n$ stored in consecutive locations in an array and an associative operation $*$, the *parallel prefix problem* is to compute the products $x_1 * \cdots * x_n$, for $i = 1, \ldots, n$. This can be accomplished in time $O(n/p + \log p)$ and space $n + O(p)$ (see [1] or [2]). Parallel prefix can be computed within the same time bounds when the items are stored on a linked list [3,4].

Summing by Groups: Assume that n items are to be summed and are divided into groups; items that belong to the same group are contiguous and the first item in each group is marked. The *summing by groups problem* is to compute the sum within each group. This is handled as a parallel prefix computation, by defining a suitable sum operation that does not carry across set boundaries [2]. The computation yields the initial sums within each subgroup.

Broadcasting and Packing: The *broadcast problem* is to create n copies of an item. This again is parallel prefix, with the product defined as $a*b = a$. Broadcast by groups is executed in a similar manner. Given an array of length n, the *packing problem* is to move the nonempty entries in the array to the head of the array, while preserving their order. Packing can be done by ranking the nonzero entries, and then moving each to the location indicated by its rank. Ranking can be done by assigning nonempty elements the value one and empty elements the value zero, and then applying parallel prefix using addition.

Sorting and Merging: Radix sort can be used to sort n integers in the range 1 to R in time $T_p(n) = O((n/p)(\log R / \log(n/p)))$ and space $O(pn^{\epsilon} + n)$ for any constant $\epsilon > 0$ [5]. The time is $O(n/p)$, whenever the number of distinct key values R is at most polynomially larger than n/p. Two sorted lists of sizes $m \leq n$ can be merged in time $O((m+n)/p + \log n)$ [6].

3.1. Cross Product

Consider the problem of forming the cross product (i.e. Cartesian product) of two sets S and T. The result is to be placed in two arrays $\bar{S}$ and $\bar{T}$, where $\bar{S}[i] \times \bar{T}[i]$ is the ith element of the cross product.

First determine the cardinalities s and t of the two sets. We need t copies of each element of S, and s copies of each element of T. Consider $\bar{S}$ and $\bar{T}$ as being two dimensional arrays of size $s \times t$, stored in row major order. Row i of $\bar{S}$ will consist of t copies of element i in S, and, similarly, column i of $\bar{T}$ will consist of s copies of element i in T.

The following operations will produce $\bar{S}$: Place element i of S into location $i \cdot t$ of array $\bar{S}$. Broadcast each element to the next $t-1$ locations of $\bar{S}$. To produce $\bar{T}$, replace $i \cdot t + j$ by $j \cdot t + i$.

A cross product computation makes use of parallel prefix on arrays of size $s \cdot t$ or smaller; it can be done in time $O(s \cdot t + \log p)$ and space $O(s \cdot t + p)$.

Cross products can also be done by groups: Assume that the sets S_k are packed to the top of array S and the sets T_k are packed to the top of array T. The cross products $S_k \times T_k$ are to be placed in the arrays $\overline{S}$ and $\overline{T}$.

(1) Place the cardinality of each S_k into σ_k of a temporary array σ and the cardinality of each T_k into τ_k of a temporary array τ.

(2) Form the products $\sigma_k \tau_k$ to determine the cardinalities of the cross product sets.

(3) Perform a parallel prefix with addition on these products to determine the starting location of each cross product in the final answer.

(4) Copy the starting location of the k th cross product along with the cardinality of T_k to the first element in each set S_k.

(5) Broadcast the starting of location the k th cross product along with the cardinalities of S_k and T_k to all elements within each set S_k, using broadcast by groups.

(6) Determine the rank of each element within each set (parallel prefix by groups)

(7) Set all locations of $\overline{S}$ to empty.

(8) Place each element i of each set S_k into location $i \cdot |T|$ plus the starting location of the k th cross product of array $\overline{S}$.

(9) Broadcast each element in each set $\overline{S}_k$ to the next $|T_k| - 1$ locations of $\overline{S}$, using broadcast by groups.

Analyses: Recall that we make two assumptions about the problem size: (1) $m = \Omega(d^\epsilon)$ for some positive ϵ; and (2) $m = \Omega(p^{1+\delta})$ for some positive constant δ. Thus, parallel prefix on the m elements of matrix takes time $O(m/p)$. Also, for $n \times n$ matrices, the integer range for sorting is $R = n^2$. Using radix sort, the time to sort is $O(m/p)$. The space is $O(pm^\epsilon + m)$ for any constant $\epsilon > 0$. By assumption (2), there is an ϵ such that the space is $O(m)$ (namely any $\epsilon < \delta$). More generally, for 2-dimensional matrices that are not necessarily square, the length of each side of a matrix is bounded by d. The integer range for the sorting will be at most d^2, so the radix sort time and space remain the same.

3.2. Format Conversion, Matrix Transpose, and Matrix Addition

It is easy to see that a canonical representation of a $q \times r$ matrix can be computed from its full matricial representation in time $O(qr/p)$: Each processor is allocated qr/p entries of the matrix; it creates a list of quadruples representing the nonempty entries in its set. These lists are then packed in time $O(qr/p)$. Conversely, a canonical representation can be converted into a full matricial form in time $O(m/p)$ if the matrix is already initialized to zero, and time $O(qr/p)$ otherwise. A sparse matrix in canonical representation is transposed by inverting in each quadruple the row and column indices, in time $O(m/p)$ and space $O(m)$.

Suppose we are given two sparse matrices A and B, and wish to form their sum C. We need to pair together elements of A and B with the same indices (i.e. in the same row and column) and then replace each such pair by the sum of their values. The pairing of values can easily be done by collecting the A and B quadruples together, and sorting using an element's index as the key. The total time for the algorithm is $O((m/p)(\log d)/\log(m/p))$ and the space is $O(pm^\epsilon + m)$. Under our assumptions, these simplify to time $O(m/p)$ and space $O(m)$. If the two matrices are already sorted the addition can be done using merging rather than sort-

ing, which improves the time and space complexities.

4. MATRIX MULTIPLICATION

We now show how to form the matrix product C of two sparse matrices A and B. Every nonzero element of column k in A must be multiplied by every nonzero element of row k in B. This means that elements of column k in A need to be grouped together with elements of row k in B. Each such group is composed of a set of elements from A and a set of elements from B. Within each group, form the cross product of the two sets and multiply the A and B values forming each pair. The product of the pair a_{ik} and b_{kj} contributes to c_{ij}. Thus, the pairs need to be sorted by the key (i, j), where i is the row of the A element and j is the column of the B element. All products with the same key (i, j) are summed to form the c_{ij}, which are packed to the top of the array.

The algorithm works as follows:

(1) Place the two matrices A and B together into one set of quadruples (matrix name, row index, column index, value).

(2) Sort the quadruples using, for A, the column index and using, for B, the row index.

(3) Within each group of elements with the same index from step (2), form the cross product of the elements from set A with the elements from set B.

(4) Multiply the A value with the B value in each cross product pair, and form a new C element with that value, and with row index from the A element and the column index from the B element.

(5) Sort the C elements by row and column index.

(6) Sum the values of all C elements with the same row and column indices (using summing by groups), and place the value into the first element of the group.

(7) Pack the first element of each group into the final answer C.

Let T be the number of nontrivial terms occurring in the matrix product. The serial matrix multiplication algorithm uses $O(T)$ time and $O(m+M)$ space, where m is the input size and M is the output size. The parallel algorithm presented here requires time $O((T/p)\log(d)/\log(T/p))$ and space $O(p^{1-\epsilon}T^{\epsilon}+T)$, which simplifies under our assumptions to time $O(T/p)$ and space $O(T)$. Although the time for our parallel algorithm is optimal, the space may be significantly larger than optimal. We now refine the parallel algorithm so as to reduce the amount of space.

The main idea is to process the cross products in blocks. A partial result matrix C is kept of the accumulated sums computed so far. Whenever a new block is processed, a matrix of values produced from the cross products in that block is formed. The partial result matrix C is then updated by adding to it the new matrix. After all of the blocks are processed, C will be the desired product matrix. The goal is to choose the block sizes small enough so as not to use extra space, but large enough so that significant time is not wasted updating C.

Assume that the cross products are formed from groups U_i of elements from A and groups V_i of elements from B.

(1) Form array R_i of the cross product sizes $|U_i| \cdot |V_i|$. Assume, wlog, that each $R_i > 0$.

(2) Use parallel prefix to form $S_i = \sum_{j=1}^{i} R_j$ (and set $S_0=0$).

(3) Let the product matrix $C = 0$ (by initializing it to the empty list). Let $i_0 = 0$.

Repeat for $k = 1,2, \cdots$ until all of the blocks are processed.

(4) Let $b = \max(m, |C|)$.

(5) Let $i_k = \min\{i : S_i - S_{i_{k-1}} > b\}$ (if the condition never holds, let i_k = index of the last group).

(6) Form the product matrix using groups U_j and V_j, for $i_{k-1} < j \leq i_k$.

(7) Update C by adding to it the new matrix.

At each iteration, the time to compute the new product matrix is at least as large as the time to add the two matrices, since the number of terms in the block is always chosen to be at least as large as the size of the current product matrix C. Let t_k be the number of products computed at the k th iteration of the above algorithm ($t_k = S_{i_k} - S_{i_{k-1}}$). Then, not counting the time to determine the next block, the total time for the algorithm is

$$\Sigma O\left(\frac{t_k}{p}\frac{\log d}{\log(t_k/p)}\right),$$

where $\Sigma t_k = T$. By our assumptions, this simplifies to $\Sigma O(t_k/p) = O(T/p)$.

The value of i_k can be determined at each iteration by just one processor using unbounded binary search: Conservatively, the processor starts at $i = i_{k-1}+1$ and continues increasing i by doubling the difference $i - i_{k-1}$ until $S_i - S_{i_{k-1}} \geq b$; after that, a traditional binary search (between the last two values of i) will find i_k in total time $O(\log(i_k - i_{k-1}))$. Since $i_k - i_{k-1} \leq t_k$, the time to determine i_k is bounded by $O(\log t_k)$. By our assumptions, $p \leq t_k^{1-\epsilon}$, which implies

$$\frac{t_k}{p} \geq t_k^{\epsilon} = \Omega(\log t_k).$$

The time to broadcast i_k to all of the processors is $O(\log p)$. Similar reasoning to the above shows that $\log p = O(t_k/p)$. Thus the searching and broadcasting time is dominated by the computation time.

Finally, we need to show that the space usage is not large: At each iteration, the total size of the new matrix, not counting the last cross product group in the block, is proportional to the maximum of m and the current size of the product matrix C. This is clearly $O(m + M)$. All of the terms in a given cross product group are associated with a distinct element of the product matrix. Thus, the last group has size $O(M)$. So, the total size of the matrices at each iteration is $O(m + M)$. The sorting algorithm uses space $O(p(m+M)^{\epsilon} + (m+M))$. By our assumptions this is simply $O(m + M)$.

5. GAUSSIAN ELIMINATION

Standard algorithms for the inversion or decomposition of an $n \times n$ (dense) matrix involve a sequence of n stages (one stage for each row of the matrix). Most of the computations done at each stage are independent vector operations; these can be computed efficiently in parallel. The computations done at successive stages are

strongly data dependent. In order to reduce the time required to invert or decompose a matrix in parallel below time $\Omega(n)$ it is necessary to use different algorithms (e.g. [7]). Such algorithms seem to be both numerically unstable and inefficient with respect to the number of operations performed. We shall, therefore, consider parallel versions of the standard serial algorithms. We first show how to do Gaussian elimination assuming that the pivots are known in advance, and then discuss pivot selection.

Let A be a square $n \times n$ (sparse) matrix with m nonzero entries. Assume that $m \geq n$; otherwise the matrix is singular. We shall consider the problem of solving the system of linear equations $Ax = b$ using Gaussian elimination. Computing an **LU** decomposition for the matrix A is done essentially in the same manner, and will not be discussed.

Gaussian elimination consists of an elimination phase and a back substitution phase. The elimination phase consists of a sequence of n stages that modify the entries in the extended matrix $[Ab]$. A pivot element is chosen in this matrix, and suitable multiples of the row containing this element are subtracted from the remaining rows.

Each row of the matrix is stored as a set of pairs, (column, value). The column indices need not be distinct so that the actual value of an element in row r and column c is the sum of the values in the row r set having column index c. Similarly, each column is stored as a set of pairs, (row, value), where the row indices need not be distinct. Thus, the value of an element can be determined from its row or column set. The actual values of the elements in some particular row (column) are *resolved*, i.e., the values of the elements with the same column index (row index) are summed when the new pivot element is from that row (column). Thus, the algorithm uses lazy evaluation to avoid wastefully accessing all the elements of a row or column when only a few elements need to be resolved. In order not to waste space storing each element as the sum of many pairs, all of the row and column sets are resolved whenever the total number of pairs in the row and column sets reaches a certain threshold (which is dynamically set).

The algorithm consists of n stages. At each stage, the next pivot element is selected. The values in its row and column sets are resolved (independently). The values on these two sets are then used to determine the set of values needed to zero out the column of the pivot element while keeping the other matrix values consistent. This set is unioned into the row and column sets.

More formally, let E be the extended matrix $[Ab]$ and let $<i_1,j_1>$, $<i_2,j_2>$, ..., $<i_n,j_n>$ be the successive pivots, then the algorithm works as follows:

Forward Elimination:

let *prev_size* $= |E|$;

for $k := 1$ to n do begin

(1) resolve row i; place it in row vector U;

(2) resolve column j; place it, *without* the pivot $(a_{i_k j_k})$, into row vector V;

(3) let $Q = \frac{-1}{a_{i_k j_k}} U^T \otimes V$; $((U^T \otimes V)_{ij} = U_i * V_j)$

(4) let $E = E \bigcup Q$;