

Scalable Parallel Matrix Multiplication Algorithms with Application to a Maximum Entropy Problem

Xin Wang

September 5, 1997

Abstract

This paper presents some efficient and scalable parallel and distributed implementations of a matrix multiplication operation, ADA' , where A is a very “fat” matrix, D a diagonal matrix and A' is the transpose of A , a very “thin” matrix, and their application to a maximum entropy problem. Theoretical performance results of a time-optimal implementation and a space-optimal implementation are given in the IBM supercomputer SP2 environment.

1 Introduction

In this paper, we consider parallel and distributed computation of a matrix multiplication of the following form:

$$B = ADA'. \quad (1)$$

Here, $A = [a_{ij}]$ is a “fat” matrix of the real numbers of size $m \times n$ with $m \ll n$, $D = \text{diag}\{d_1, \dots, d_n\}$ is a diagonal matrix of the real numbers of size $n \times n$ with diagonal elements d_1, \dots, d_n , and the sign $'$ denotes the matrix transpose operation. The resulting matrix has size $m \times m$, and can be considered as a relatively small matrix.

1.1 Motivation: A Maximum Entropy Problem

Though the computation in (1) has its generality in linear algebra, we are specifically motivated here by its parallelization for a maximum entropy computation studied in [MJJ97], where the computation is repeated within an iteration in the following fashion:

```
...
generate a fat matrix A
for (i = 0; i < SomeNumberOfIterations; i++) {
    ...
    compute a short vector u of size m
    form a long vector v = A' u
    generate a large diagonal matrix D, with D(k,k) = exp{v(k) - 1}
    form a short vector w = A [D(1,1), ..., D(n, n)]'
    compute a small matrix B = A D A'
    ...
}
...
```

Note that the matrix A remains constant within the iteration and the matrix D varies from iteration to iteration. The pseudo-code above only singles out the part of the maximum entropy computation that is computation and memory intensive. The rest of the computation (indicated by ...) does not directly deal with the matrices A , D and B , and takes a very small fraction of computation time and memory space.

In the maximum entropy problem, the size of A and D are determined by some problem and computation parameters. Let q be the number of problem parameters and g the number of computational grid points for each parameter. Then the number n of columns of A is equal to g^q , and the number m of rows of A is the number of constraints posted on the problem, which is equal to either $2q + 1$ when only the means and variances of the parameters are given or $(q^2 + q)/2 + 1$ when the (symmetric) covariance matrix of the parameters is given.

Tables 1 and 2 list resource requirements for the two different cases of m . The requirements listed are memory requirement M (in MegaBytes) on the matrices A and D and calculation requirement F (in MegaFlops – mega floating point arithmetic operations) for computing the matrix B . They are calculated according to the following formulas, with no consideration of all the zero entries in the diagonal matrix D :

$$M = s * g^q (m + 1) / 10^6 \quad \text{and} \quad F = 2 * m^2 * g^q / 10^6,$$

where the number g of grid points for each parameter is taken to be 10 and each floating point number is assumed to consume 4 bytes.

q	4	5	6	7	8	9	10
$m = 2q + 1$	9	11	13	15	17	19	21
$n = g^q$	10^4	10^5	10^6	10^7	10^8	10^9	10^{10}
M (MegaByte)	0.4	4.8	56	640	7,200	80,000	880,000
F (MegaFlop)	1.62	24.2	338	4,500	57,800	722,000	8,820,000

Table 1: Resource requirements for the maximum entropy computation with given means and variances.

q	4	5	6	7	8	9	10
$m = (q^2 + q)/2 + 1$	11	16	22	29	37	46	56
$n = g^q$	10^4	10^5	10^6	10^7	10^8	10^9	10^{10}
M (MegaByte)	0.48	6.8	92	1,200	15,200	188,000	2,280,000
F (MegaFlop)	2.42	51.2	968	16,820	273,800	4,232,000	62,720,000

Table 2: Resource requirements for the maximum entropy computation with a given covariance matrix.

Clearly, as the number of parameters q becomes greater than 6, the memory requirement M alone exceeds capabilities that a single state-of-the-art PC or workstation can offer. Moreover, according to the current serial implementation of the maximum entropy computation used in [MJJ97], 7000 to 20000 PC nodes are needed to make up the required memory space for the $q = 10$ case. Nevertheless, by running in a parallel mode on a parallel and distributed platform, it becomes feasible to solve larger problems that are impractical for single PC or workstation execution.

1.2 Previous Work on Parallel Matrix Multiplications

Various parallel and distributed implementations of general matrix multiplication have been proposed in the literature of parallel numeric computation. In spite of its apparent simplicity and long history, the subject remains very interesting. This is mainly due to evolutionary architectures of parallel and distributed computers such as 1D- and 2D-systolic arrays, 2D-meshes, 2-D tori, and hypercubes, as well as the shared memory model and distributed memory model [Fox88, HJT93]. In addition, activities in standardization of the message passing interface (MPI) [MPI, MPI2] and distributed implementations BLACS [DW97], ScaLAPACK [CDPW92], PUMMA [CDW94], SUMMA [GW97] of linear algebra operations BLAS [DDHD90], LAPACK [LAPACK92] desire more general, simpler and more efficient methods and implementations.

Most work considers matrix computations of the form

$$aAB + bC \tag{2}$$

where a, b are scalars and A, B, C are matrices. Clearly, the computation in (1) can be carried out based on implementations of (2) in two steps: the first is to compute an intermediate matrix $X = DA'$ or $X = AD$, and the second is to compute the desired product AX or XA' . Since the intermediate matrix X has the same size of A or A' , such an implementation requires re-distribution of a vary large amount of data of X during the computation and thus increases substantially communication time among processors involved and ultimately the overall computation time.

1.3 Results

In this paper, we consider a general 2-dimensional block version of the matrix computation in (1), and conduct a comparative analysis on various matrix partitions and their resulting parallel and distributed computations. Based on this analysis, we provide two straightforward, highly efficient implementation using some collective communication primitives, one being optimal in time and the other optimal in memory space. We present a performance and scalability analysis for the two implementations, and their theoretical performance results in the IBM supercomputer SP2 environment. We suggest that the space-optimal implementation is particularly appropriate for a distributed implementation of the maximum entropy computation.

The rest of the paper is organized as follows. Section 2 defines parallel and distributed processing environments upon which the two implementations are based, especially three collective communication primitives: broadcast, gather and reduce. Section 3 describes a general 2-D decomposition of the matrices A and D , with 1-D row and column decompositions in particular. Section 4 presents the two implementations, and Section 5 details their performance and scalability analysis, together with some theoretical performance results on the IBM supercomputer SP2. Section 6 gives an application of the space-optimal implementation to the maximum entropy problem. Finally, Section 7 concludes the paper with some remarks on other possible parallel and distributed implementations, and on the trade-off in applying the time-optimal and space-optimal implementations to the maximum entropy problem.

2 Parallel and Distributed Processing Environments

We now define a parallel and distributed computation environment. Physically, it can be either a tightly integrated supercomputer or a loosely coupled cluster of workstations. As in [WB96, WBPM7], we assume that the environment consists of p homogeneous processing nodes, labeled from 0 to $p - 1$; The node with label (or id) i will be denoted by P_i . The homogeneousness means that all the nodes have a same computing speed and an equal amount of memory space. In addition, we assume that each node contributes a process to the computation. Thus, we do not distinguish nodes and processes in this paper.

For the purposes of illustration and experiment, we will use the IBM SP2 [IBMSPP2] as an example.

2.1 Computation

For computational power, we assume that each single node has enough memory and swap space to carry the computation of a proper size and it takes γ micro-seconds (μs) to perform a floating-point arithmetic operation. By definition, the time γ is equal to the reciprocal of the Megaflops of the node.

2.2 Point-to-Point Communication

For inter-process communication, we assume that between connected nodes are bidirectional communication links, and in the absence of network conflicts, the time (in micro-seconds) required for sending a message of m bytes between any two connected nodes is modeled by

$$\alpha + m\beta,$$

where α is the latency for sending a message, and β is the communication time per byte; usually the reciprocal of β is considered to be the bandwidth of the network links.

2.3 Collective Communication

Collective communication is inter-process communication that involves more than two nodes; in many cases, it involves all the nodes in the environment. Examples that we are interested in this paper are broadcast, gather and combine. These collective primitives are usually implemented in terms of the point-to-point communications. The implementation algorithms given below have minimal startup costs, i.e. the least numbers of messages passing among the nodes.

2.3.1 Broadcast

In a broadcast operation, a source node sends a message to all other nodes. The operation can be implemented on a linear array of nodes using a minimum spanning tree: it starts by dividing the linear array in two (approximately) equal parts and choosing a receiving node in the part that does not contain the source node. The broadcast proceeds recursively by treating each of the nodes that possess the message as a new source for a broadcast within its own half of the previous array. If the message is m -byte long, the total time required for the broadcast is

$$\lceil \log p \rceil (\alpha + m\beta). \quad (3)$$

2.3.2 Gather

In a gather operation, a destination node gathers messages from all nodes, itself included. Note that the gather operation is not merely a reverse operation of the broadcast operation, as the outcome of the broadcast is that every node receives a same message, while the outcome of the gather is that the destination node receives p possibly distinct messages.

The gather can be implemented like the of broadcast in reverse, except at each stage only the message in the other part of the array is sent to the receiving part and gets concatenated with the message on the receiving part. If the messages that all nodes initially possess have an equal length, say m bytes, the cost for the gather is

$$\lceil \log p \rceil \alpha + \frac{p-1}{p} m\beta. \quad (4)$$

2.3.3 Combine

The combine operation combines messages provided by all nodes, using a specified operation, and delivers the combined value to a destination node. It is logically equivalent to a gather operation followed by the operation. The operation is usually a binary one that is associative and commutative. Examples are numeric $+$ and \times , and Boolean OR, AND and XOR. If the operation is $+$ and every node provides a list of numbers of equal length, then the outcome of the combine operation is that the destination node receives the result of summing the lists element-wise.

The combine can be implemented similarly by running the broadcast communication in reverse order and interleaving communication with the combine operation. This requires a total time

$$\lceil \log p \rceil (\alpha + m\beta + \frac{m}{s}\gamma). \quad (5)$$

The last term $m\gamma/s$ in the parenthesis is the calculation time used during each stage to combine the result.

2.4 The IBM Supercomputer SP2

The IBM SP2 [IBMSP2] is a distributed memory system, which fits very well the parallel and distributed model assumed in the previous subsections. The SP2 at the University of Southern California has in total 28 “thin-2” nodes, each of which has 128 MB main memory and about 768MB of /tmp space. Thin-2 nodes have additional paths to memory and to the integer and floating point processing units, doubling these bandwidths, and each have 66 MHz clock speed and 266 peak Megaflops. The peak floating-point arithmetic operation speed on single nodes is $\gamma = 1/266 \approx 0.00376(\mu s)$.

The SP2 networks are bidirectional multistage interconnection networks using the high-performance switch. They provide bi-directional, any-to-any internode connection, allowing all processors to send messages simultaneously. When using communication protocol US, their peak latency and bi-directional bandwidth are $\alpha = 39\mu s$ and $35.7MB/s$. Thus, the communication time per byte is about $\beta = 1/35.7 = 0.028(\mu s)$.

3 Data Decomposition

An important issue in parallel and distributed computation is to distribute data over processors involved. We now consider two-dimensional decompositions of the matrices and their corresponding block formulations of the multiplication in (1). The one dimensional cases will be automatically included in these decompositions by letting either row or column dimension of the two dimensional mesh equal one.

Let the matrix A be partitioned into a two-dimensional mesh of size $u \times v$:

$$A = \begin{bmatrix} A_{11} & A_{12} & \dots & A_{1v} \\ A_{21} & A_{22} & \dots & A_{2v} \\ \vdots & & \ddots & \vdots \\ A_{u1} & A_{u2} & \dots & A_{uv} \end{bmatrix}, \quad (6)$$

and correspondingly the diagonal matrix D into:

$$D = \begin{bmatrix} D_{11} & & & \\ & D_{22} & & \\ & & \ddots & \\ & & & D_{vv} \end{bmatrix}. \quad (7)$$

In general, submatrices A_{st} are of sizes $x_s \times y_t$, and D_{tt} of sizes $y_t \times y_t$, with $\sum_{s=1}^u x_s = u$ and $\sum_{t=1}^v y_t = v$. For simplicity, we assume that all A_{st} have the same size $x \times y$ and D_{tt} have $y \times y$ such that $u * x = m$ and $v * y = n$.

Based on this decomposition, the resulting matrix B has a block form:

$$\begin{aligned} B &= \begin{bmatrix} A_{11} & A_{12} & \dots & A_{1v} \\ A_{21} & A_{22} & \dots & A_{2v} \\ \vdots & & \ddots & \vdots \\ A_{u1} & A_{u2} & \dots & A_{uv} \end{bmatrix} \begin{bmatrix} D_{11} & & & \\ & D_{22} & & \\ & & \ddots & \\ & & & D_{vv} \end{bmatrix} \begin{bmatrix} A'_{11} & A'_{21} & \dots & A'_{v1} \\ A'_{12} & A'_{22} & \dots & A'_{v2} \\ \vdots & & \ddots & \vdots \\ A'_{1u} & A'_{2u} & \dots & A'_{vu} \end{bmatrix} \\ &= \begin{bmatrix} \sum_{t=1}^v A_{1t} D_{tt} A'_{t1} & \sum_{t=1}^v A_{1t} D_{tt} A'_{t2} & \dots & \sum_{t=1}^v A_{1t} D_{tt} A'_{tu} \\ \sum_{t=1}^v A_{2t} D_{tt} A'_{t1} & \sum_{t=1}^v A_{2t} D_{tt} A'_{t2} & \dots & \sum_{t=1}^v A_{2t} D_{tt} A'_{tu} \\ \vdots & & \ddots & \vdots \\ \sum_{t=1}^v A_{ut} D_{tt} A'_{t1} & \sum_{t=1}^v A_{ut} D_{tt} A'_{t2} & \dots & \sum_{t=1}^v A_{ut} D_{tt} A'_{tu} \end{bmatrix} \\ &= \begin{bmatrix} B_{11} & B_{12} & \dots & B_{1u} \\ B_{21} & B_{22} & \dots & B_{2u} \\ \vdots & & \ddots & \vdots \\ B_{u1} & B_{u2} & \dots & B_{uu} \end{bmatrix} \end{aligned} \quad (8)$$

where

$$B_{st} = \sum_{k=1}^v A_{sk} D_{kk} A'_{kt}$$

for $1 \leq s, t \leq u$.

There are two special one-dimensional decompositions of A in (6): the row decomposition and the column decomposition. The former has the column dimension of the mesh equal to one, $v = 1$, while the latter has the row dimension equal to one, $u = 1$. These one-dimensional decompositions lead to special forms of the resulting matrix B .

In the row decomposition, the matrix A is broken into blocks of rows in the form

$$\begin{bmatrix} A_1 \\ A_2 \\ \vdots \\ A_u \end{bmatrix}$$

and the decomposition of D remains unchanged. Consequently, the matrix B is

$$B = \begin{bmatrix} A_1 \\ A_2 \\ \vdots \\ A_u \end{bmatrix} D [A'_1, A'_2, \dots, A'_u] \quad (9)$$

$$= \begin{bmatrix} A_1 D A'_1 & A_1 D A'_2 & \dots & A_1 D A'_u \\ A_2 D A'_1 & A_2 D A'_2 & \dots & A_2 D A'_u \\ \vdots & & \ddots & \vdots \\ A_u D A'_1 & A_u D A'_2 & \dots & A_u D A'_u \end{bmatrix}. \quad (10)$$

In the column decomposition, the matrix A is partitioned into blocks of columns in the form

$$[A_1, A_2, \dots, A_v]$$

and the matrix D is

$$\begin{bmatrix} D_1 & & & \\ & D_2 & & \\ & & \ddots & \\ & & & D_v \end{bmatrix}$$

The matrix B in this case is

$$B = [A_1, A_2, \dots, A_v] \begin{bmatrix} D_1 & & & \\ & D_2 & & \\ & & \ddots & \\ & & & D_v \end{bmatrix} \begin{bmatrix} A'_1 \\ A'_2 \\ \vdots \\ A'_v \end{bmatrix} = \sum_{t=1}^v A_t D_t A'_t. \quad (11)$$

It turns out that the row and column decomposition of A lead to a time-optimal and a space-optimal algorithms for the matrix multiplication in (1).

4 Parallel and Distributed Algorithms

An ultimate requirement for any parallel and distributed computation is to maximize use of available computing resources (e.g., memory and speed of all nodes and bandwidth and topology of the network) in order to minimize overall computation time. As all nodes in the environment are assumed to be homogeneous, the computation deployment needs to satisfy a load-balance requirement as well as an overhead minimization requirement. The load-balance is about even distributions of both the data and the computation over all the nodes, whereas the overhead minimization asks reducing computation and communication overheads due to the parallelization to the minimal.

In Section 1, we mentioned a simple two-step parallel implementation of (1) that uses existing packages such as Scalapack, Summa and Pumma. It does not meet the overhead minimization requirement, because it requires redistribution of an either very fat or very thin intermediate matrix, which incurs a collective communication of very long messages among all the nodes.

In the following, we present two parallel and distributed algorithms for the matrix computation in (1): one is optimal in time and the other is optimal in space. For the sake of I/O consideration and later on the iterative nature of the maximum entropy computation, we require in the both algorithms that the resulting matrix B be put onto a single node, say node P_0 , rather than being distributed over all the nodes. Their performance and scalability analysis will be given in the next section.

4.1 Time-Optimal Algorithm

In this algorithm, each node is responsible for computing an equal number of blocks B_{st} in (8):

$$B_{st} = \sum_{k=1} A_{sk} D_{kk} A'_{kt}$$

and then sends the results to the node P_0 through a gather operation. Note that the above calculation is equivalent to

$$B_{st} = A_s D A'_t$$

where A_s and A_t are row blocks in the row decomposition used in (9). Thus, this algorithm is based on the row decomposition of the matrix A . As the computation of the matrix B is evenly distributed over all the nodes and the communication is minimal in terms of both the number of communications and the length of communicated messages, this algorithm yields the minimal overall computation time in all possible algorithms and hence is optimal in time. However, as computing B_{st} involves row blocks A_s and A_t of A and the entire matrix D , this algorithm is not space-optimal in that same data may occur on different nodes.

A pseudo version of MPI code for the time-optimal algorithm is given as follows for an arbitrary node P_i , $i = 0, \dots, p-1$. It is written in the SPMD (single program and multiple data) style. The only collective call it uses is MPI's gather operation, `MPI_GATHER`.

```

    for (all assigned row indices s and column indices t) do
        compute As D At'
        concatenate the result to a buffer Bsub
    end do
    MPI_GATHER(Bsub, m*m/p, MPI_FLOAT, B, m*m, MPI_FLOAT, 0, MPI_COMM_WORLD)

```

4.2 Space-Optimal Algorithm

Clearly, the time-optimal algorithm is feasible only when each node has enough memory space to hold the both matrices A (actually those row blocks A_s and A_j) and D . To reduce the space requirement to the minimal, we need to consider data distributions that do not yield any overlaps, i.e., no node shares the same data with any other nodes, and at the same time introduce minimal computation and communication overheads. The distribution based on the column decomposition given in (11) is a such one.

For p computing nodes, we decompose the matrices A and D into p column blocks A_t and D_t of approximately equal size, $t = 1, \dots, p$, and let node P_{t-1} compute the $m \times m$ matrix $B_t = A_t D_t A'_t$. Afterwards, we perform a combine operation to sum all these matrices B_t onto the node P_0 . The combine operation ensures that we have the correct result $B = \sum_{t=1}^p B_t$.

This algorithm is optimal in space because it has the minimal requirement on the memory space at each node. But it is not optimal in time as it could introduce more computation and communication times, as we will see in the next section. However, this algorithm has the advantage that it allows larger problems to be solved for a fixed number of nodes.

The following is a pseudo version of MPI code for the space-optimal algorithm for an arbitrary node P_{t-1} , $t = 1, \dots, p$, where the MPI combine call, `MPI_REDUCE`, is used.

```

    compute Bsub = At Dt At'
    MPI_REDUCE(Bsub, m*m, MPI_FLOAT, B, m*m, MPI_FLOAT, 0, MPI_COMM_WORLD)

```

5 Performance and Scalability Analysis

Here we conduct a performance analysis on the time-optimal and space-optimal algorithms, in terms of their computation and communication times, speedup, efficiency, as well as scalability. For simplicity, we assume that all number divisions by the number p of the nodes can be made even. For example, in the space-optimal algorithm based on the column decomposition, we assume that the number of columns n of the matrix A is divisible by p .

5.1 Notions of Speedup and Efficiency

First we need to clarify some concepts. Given a parallel and distributed algorithm parametrized by the number of nodes p , we let T_p denote the overall execution time of the algorithm over the p nodes. Normally, T_p is considered as consisting of two parts, one from calculation T_{cal} and the other from communication T_{comm} . Speedup S of the algorithm is defined as the ratio between its execution time on a single node and its execution time on p nodes:

$$S = \frac{T_1}{T_p}.$$

Efficiency E is defined as the ratio between its execution time on a single node and its total execution time summed over p nodes:

$$E = \frac{T_1}{pT_p} = \frac{S}{p}.$$

5.2 Computation and Memory Complexities on a Single Node

For the matrix computation in (1), it is easy to see that the total computation time (in μs) required on a single node is

$$T_1 = 2m^2n\gamma,$$

where m and n are the numbers of rows and columns of the matrix A , and γ is the computation time per floating point arithmetic operation.

The memory requirements for the matrices A , D and B are

$$mns, \quad ns \quad \text{and} \quad m^2s,$$

respectively, where s is the number of bytes per floating point number.

5.3 Time-Optimal Algorithm

In this case, each node has an even share of the calculation time T_1 . Thus,

$$T_{calc} = \frac{T_1}{p} = \frac{2m^2n\gamma}{p}.$$

At the end of calculation, each node has a message of equal-length m^2s/p bytes that is ready to be gathered onto the node P_0 . According to (4), the communication time needed in gathering the messages is

$$T_{comm} = \lceil \log p \rceil \alpha + \frac{p-1}{p} \frac{m^2s}{p} \beta.$$

Therefore, the total execution time, speedup and efficiency of this algorithm are

$$T_p = T_{calc} + T_{comm} = \frac{2m^2n\gamma}{p} + \lceil \log p \rceil \alpha + \frac{p-1}{p} \frac{m^2s}{p} \beta,$$

$$S = \frac{T_1}{T_p} = p \frac{2m^2n\gamma}{2m^2n\gamma + p\lceil \log p \rceil \alpha + (p-1)m^2s\beta/p},$$

and

$$E = \frac{S}{p} = \frac{2m^2n\gamma}{2m^2n\gamma + p\lceil \log p \rceil \alpha + (p-1)m^2s\beta/p}. \quad (12)$$

5.4 Space-Optimal Algorithm

In this algorithm, each node P_{t-1} , $t = 1, \dots, p$, calculates the matrix $B_t = A_t D_t A_t'$. Thus,

$$T_{calc} = 2m^2 \frac{n}{p} \gamma,$$

which is the calculation time for the matrix B with n being replaced by n/p . Note that B_t is an $m \times m$ matrix. At the end of calculation, each node has a message of equal-length $m^2 s$ bytes that is ready to be combined onto the node P_0 . According to (5), the communication time needed in combining the messages is

$$T_{comm} = \lceil \log p \rceil (\alpha + m^2 s \beta + m^2 \gamma).$$

Therefore, the execution time, speedup and efficiency of the space-optimal algorithm are

$$T_p = T_{calc} + T_{comm} = 2m^2 \frac{n}{p} \gamma + \lceil \log p \rceil (\alpha + m^2 s \beta + m^2 \gamma),$$

$$S = \frac{T_1}{T_p} = p \frac{2m^2 n \gamma}{2m^2 n \gamma + p \lceil \log p \rceil (\alpha + m^2 s \beta + m^2 \gamma)}$$

and

$$E = \frac{S}{p} = \frac{2m^2 n \gamma}{2m^2 n \gamma + p \lceil \log p \rceil (\alpha + m^2 s \beta + m^2 \gamma)}. \quad (13)$$

5.5 Scalability

An important aspect of performance analysis is the scalability study of how algorithm performance varies with parameters such as problem size and the number of processors. An algorithm is considered to be scalable if, in order to keep efficiency constant, the amount of computation as well as memory space needs to increase at a rate that is at most linear with respect to p . In contrast, an algorithm with a quadratic or exponential increasing rate are considered to be poorly scalable.

Clearly, both the time-optimal and space-optimal algorithms have good speedup and efficiency in solving large-size problems on a fixed number of nodes. The larger, the better. But, to a problem of fixed-size, using more nodes may not increase their speedup and efficiency. After a certain number, they start to decrease.

If we ignore the $\log(p)$ term in both (12) and (13), which grows very slowly when p is reasonably large, and simplify the $(p-1)m^2 s \beta / p$ term to $m^2 s \beta$, we observe the following: If we increase p and wish to maintain nondecreasing efficiency, we must increase $m^2 n$ with p in the time-optimal case and n with p in the space-optimal case. Since memory requirements grow with mn , $m \ll n$, and physical memory in the parallel and distributed environment grows linearly with p as more nodes are added, we conclude that the two algorithms are all scalable in memory in the following sense: If we maintain constant memory use per node, these algorithms will maintain nondecreasing efficiency, provided $\log(p)$ is treated as a constant. For a similar reason, the algorithms are also scalable in computation.

Mapping the above scalability analysis to the maximum entropy computation, where the sizes of the matrices A and D , m and n , increase exponentially in the number of parameters q and polynomially in the number of grid points on each parameter g , that is, $m = 2q + 1$ or $(q^2 + q)/2 + 1$ and $n = g^q$, we claim that, with constant speedup and efficiency, more nodes will help to solve problems with more parameters and more grid points on each parameters.

5.6 Theoretical Performance Results on the IBM Supercomputer SP2

Here, we present theoretical performance results of the above two optimal algorithms on the IBM SP2 (at USC that consists of thin-2 nodes).

Tables 5.6 and 5.6 list predicted performances of using 4, 8 and 12 nodes of the IBM SP2 to solve the matrix multiplication in the maximum entropy computation for the two different cases of m . They represent lower-bounds on the execution times T_p and speedups S_p . The parameters used in these tables are $\alpha = 39\mu s$, $\beta = 0.028\mu s$, $\gamma = 0.00376\mu s$, $s = 4$, $g = 10$.

q	4	5	6	7	8	9	10
$m = 2q + 1$	9	11	13	15	17	19	21
$T_1 (s)$	0.0061	0.0910	1.27	16.91	217.29	2714.29	33157.9
$T_4 (s)$	0.00166	0.02294	0.318	4.229	54.32	678.57	8289.5
S_4	3.6793	3.9654	3.996	3.999	4.0	4.0	4.0
E_4	0.9198	0.9914	0.9991	0.9999	1.0	1.0	1.0
$T_8 (s)$	0.000879	0.01155	0.15908	2.1148	27.162	339.29	4144.7
S_8	6.9411	7.8798	7.9878	7.9988	7.9999	8.0	8.0
E_8	0.8676	0.9850	0.9985	0.9998	1.0	1.0	1.0
$T_{12} (s)$	0.00062	0.00774	0.106	1.4099	18.107	226.19	2763.2
S_{12}	9.8920	11.7496	11.9745	11.9974	11.999	12.0	12.0
E_{12}	0.8243	0.9791	0.9979	0.9998	1.0	1.0	1.0

Table 3: Predicted Performance for the time-optimal algorithm

q	4	5	6	7	8	9	10
$m = (q^2 + q)/2 + 1$	11	16	22	29	37	46	56
$T_1 (s)$	0.009	0.192	3.639	63.233	1029.32	15909.77	235789.474
$T_4 (s)$	0.00238	0.04825	0.9099	15.808	25.733	3977.444	58947.369
S_4	3.8218	3.9886	3.9992	3.9999	4.0	4.0	4.0
E_4	0.9555	0.9972	0.9998	1.0	1.0	1.0	1.0
$T_8 (s)$	0.00129	0.02426	0.4551	7.9045	12.866	1988.722	29473.685
S_8	7.0185	7.9321	7.9950	7.9996	8.0	8.0	8.0
E_8	0.8773	0.9915	0.9994	0.9999	1.0	1.0	1.0
$T_{12} (s)$	0.0948	0.01628	0.3036	5.2699	85.777	1325.815	19649.124
S_{12}	9.5948	11.8187	11.9865	11.9989	11.9999	12.0	12.0
E_{12}	0.7814	0.9832	0.9987	0.9999	1.0	1.0	1.0

Table 4: Predicted Performance for the space-optimal algorithm.

6 Application to the Maximum Entropy Computation

We propose to employ a standard master-slave model for a parallel and distributed implementation of the maximum entropy computation. In this model, all the nodes are slaves when they come to performing memory- and calculation-intensive portion of the computation, and the node P_0 is the master who is responsible for conducting the rest of the computation.

In this implementation, we use the space-optimal algorithm for calculating the matrix $B = ADA'$. The reason is as follows. In order to solve problems of more than 6 parameters, we have to prioritize the space concern over the time concern. The space-optimal algorithm is based on the column decomposition, as compared to that the time-optimal algorithm on the row decomposition. Since the matrix A generally has its height m less than 60 (for the number of parameters up to 10) and its width n is much, much greater than 100, the column decomposition of A has many advantages over the row decomposition. First, the latter may generate higher load imbalance among the processors if the size of the matrix B , m^2 , is not a multiple of the number of nodes p . Second, for small- to modest-size problems, the time-optimal algorithm restricts the maximal number of nodes that can be used, For instance, when $q = 7$ and $m = 15$, the maximal number of nodes that can be used is $m^2 = 225$. This limits making full use of possibly available computing resources. and will become a sever problem if more nodes are needed to contribute memory in order to attack the maximum entropy problem with many parameters.

In order to reduce computation and communication overheads, we are considerable in dealing with distribution of matrix data. A simple way of distributing of the matrices A and D onto the nodes is to let one node generate these matrices and then broadcasts (or scatters – see [MPI] for its definition) them to other nodes. This causes computational load-imbalance and extra communication. As the matrices A and D are all dynamically generated in the maximum entropy computation, we will use node-wise data generation instead; That is, we will let each node generate its needed portion of the matrices. Compared with the simple distribution, this not only eliminates unnecessary inter-process communication, but also parallelizes the computation for the matrix generation and reduces memory requirement for storing the full matrices on single nodes.

Recall that A remains constant in each iteration, whereas D , now represented as a vector $[d_1, \dots, d_n]$, is dynamically generated and varies from iteration to iteration. A parallel implementation based on the column partition of the matrix A can be given as follows for an arbitrary node P_{t-1} , $t = 1, \dots, p$:

```

if (id == 0) {
    ...
}
generate the matrix At
for (i = 0; i < SomeNumberOfIterations; i++) {
    compute a short vector u of size m
    v = At' u
    form vector Dt, with entries d(k) = exp{v(k) - 1}
    compute vector wt = At Dt
    compute matrix Bt = At Dt At'
    gather wt to node P_0
    reduce Bt to node P_0 with operation sum
    if (id == 0) {
        ...
    }
}
if (id == 0) {
    ...
}

```

7 Concluding Remarks

Other parallel and distributed algorithms for the matrix computation given in (1) and the entropy computation in [MJJ97] can be obtained by making trade-offs between time and space requirements. However, we consider that the time-optimal algorithm is the best for solving relatively small entropy problems, while the space-optimal algorithm is particularly appropriate for large-sized entropy computations.

The suggested algorithm for the maximum entropy computation is considerably efficient, which has been based on the column decomposition of the data matrices. Its flexibility and feasibility are better than the one based on row decomposition when the memory is a major concern, and competitive when the memory is not. This enables solving the maximum entropy problem with more parameters. We should note that for the small-size maximum entropy computation the time-optimal algorithm may outperform the space-optimal algorithm. But the gain may not be significant. The simplicity and flexibility of the space-optimal algorithm warrants consideration. After all, being able to solve large-sized maximum entropy problems is the motivation for us to consider parallel and distributed computation.

References

[LAPACK92] E. Anderson, Z. Bai, J. Demmel, J. J. Dongarra, J. DuCroz, A. Greenbaum, S. Hammarling, A. McKenney, S. Ostrouchov, and D. Sorensen, *LAPACK Users' Guide*, SIAM, Philadelphia, 1992.

- [CDPW92] J. Choi, J. J. Dongarra, R. Pozo, and D. W. Walker, "SCALAPACK: A Scalable Linear Algebra Library for Distributed Memory Concurrent Computers, Proceedings of the Fourth Symposium on the Frontiers of Massively Parallel Computation. IEEE Comput. Soc. Press, 1992, pp. 120-127.
- [CDW94] J. Choi, J. J. Dongarra, and D. W. Walker, "PUMMA: Parallel Universal Matrix Multiplication Algorithms on distributed memory concurrent computers," *Concurrency: Practice and Experience*, Vol 6(7), 543-570, 1994.
- [DDSV91] J. J. Dongarra, I. S. Duff, D. C. Sorensen, and H. A. van der Vorst, *Solving Linear Systems on Vector and Shared Memory Computers*, SIAM, Philadelphia, 1991.
- [DDHD90] J. J. Dongarra, J. Du Croz, S. Hammarling, and I. Duff, "A Set of Level 3 Basic Linear Algebra Subprograms," *TOMS*, Vol. 16, No. 1, pp. 1-16, 1990.
- [DW97] J. J. Dongarra and R. C. Whaley, "A User's Guide to the BLACS v1.1", May 5, 1997. <http://netlib.org/blacs/lawn94.ps>
- [Fox88] G. C. Fox, M. A. Johnson, G. A. Lyzenga, S. W. Otto, J. K. Salmon, and D. W. Walker, *Solving Problems on Concurrent Processors*, Vol. 1, Prentice Hall, Englewood Cliffs, N.J., 1988.
- [MPI] W. Gropp, E. Lusk, A. Skjellum, *Using MPI: Portable Programming with the Message-Passing Interface*, The MIT Press, 1994.
- [IBMSP2] International Business Machines Corporation, IBM SP Hardware and Software Overview, <http://www.mhpcc.edu/training/workshop/html/ibmhsw/ibmhsw.html>.
- [MPI2] Message Passing Interface Forum, "MPI: A Message-Passing Interface Standard", Version 1.1 (June 12, 1995). <http://www.mcs.anl.gov/mpi/mpi-report-1.1/mpi-report.html>.
- [HJT93] S. Huss-Lederman, E. Jacobson, A. Tsao, "Comparison of Scalable Parallel Matrix Multiplication Libraries," in Proceedings of the Scalable Parallel Libraries Conference, Starkville, MS, Oct. 1993.
- [GW97] R. van de Geijn and J. Watts, "SUMMA: Scalable Universal Matrix Multiplication Algorithm," *Concurrency: Practice and Experience*, Vol. 9 (4), pp. 255-274 (April 1997)
- [MJJ97] M. Milman, F. Jiang, and R. W. Jelliffe. Creating Discrete Joint Densities from Continuous Ones: The Moment Matching-Maximum Entropy Approach. LAPK Technical Report 97-2. Submitted for publication. 1997.
- [WB96] X. Wang, E.K. Blum, "Parallel Execution of Iterative Algorithms on Workstation Clusters", *Journal of Parallel and Distributed Computing*, Vol. 34, No. 2, pp. 218-226, 1996.
- [WBPM7] X. Wang, E.K. Blum, D.S. Parker, D. Massey, "The Dance Party Problem and its Application to Collective Communication in Computer-Networks", *Parallel Computing*, Vol. 23, No. 8, 1141-1156, 1997.