INTERNATIONAL JOURNAL OF
NUMERICAL ANALYSIS AND MODELING, *SERIES B*
Volume 5, Number 1-2, Pages 123–135

RESEARCH OF MULTICORE-BASED PARALLEL GABP ALGORITHM WITH DYNAMIC LOAD-BALANCE

HANYUAN ZHENG, ANPING SONG, ZHIXIANG LIU, LEI XU, MINCHAO WANG, AND WU ZHANG*

Abstract. Based on Gaussian Belief Propagation(GaBP) algorithm for solving sparse symmetric linear equations, an iterative acceleration optimization method of GaBP is studied and a corresponding optimized storage scheme is proposed. We explore the parallelism and load balancing features of this algorithm and present a multicore-based parallel GaBP algorithm with dynamic load-balance. The numerical results indicate that this algorithm can solve large scale sparse symmetric linear equations with good results and high parallel efficiency.

Key words. multicore-based parallelization, GaBP algorithm, load balance, linear equations.

1. Introduction

Solving linear equations $Ax = b$ is the fundamental problems in various scientific and engineering computing. Numerical methods, such as finite element method, finite difference method, spectral method, and finite volume method [1, 2, 10, 3], convert the actual problem into the problem of solving sparse linear equations. With the increase of the scale and complexity of problems, how to effectively solve large scale sparse linear equations has been a hot area [4].

For solving sparse linear equations, iterative method is mainly used. The iterative method includes classical iterative method, such as Jacobi method, SOR method, Krylov subspace method which is very popular in recent years [5, 6]. In 2008, Ori Shental et al proposed an iterative method for symmetric diagonally dominant linear equations Gaussian Belief Propagation(GaBP) [7]. GaBP algorithm converts the problem of solving linear system into solving the problem of Probability and information dissemination which differs from the classical iterative method and Krylov subspace method. For symmetric diagonally dominant linear equations, GaBP algorithm has a good convergence, and is essentially equivalent to the classic Gauss elimination method.

The main objective of this paper is how to efficiently solve large scale sparse symmetric linear equations. We study the iterative acceleration method to optimize the GaBP algorithm based on its classical GaBP counterpart. By exploring the parallelism and features of GaBP algorithm, we present a multicore-based parallel GaBP algorithm with the feature of dynamic load-balance to solve large-scale sparse linear equations. Numerical experiment of solving large scale are fulfilled and results are compared with other algorithms.

The rest of the paper is organized as follows. In Section 2, GaBP Algorithm will be described in detail. the iterative acceleration optimization method of GaBP and a corresponding optimized storage scheme is shown in Section3. We discuss the experimental results in section 4. Finally, Section 5 concludes the paper.

Received by the editors January 1, 2014 and, in revised form, March 24, 2014.
2000 *Mathematics Subject Classification.* 65F10, 65F50, 68W10.

*Corresponding author.

2. GaBP Algorithm

In this section, we will review the classical GaBP algorithm [7, 8, 9]. For symmetric diagonally dominant linear equations

$$Ax = b, A \in \Re^{n\times n}, x, b \in \Re^n, \tag{1}$$

the coefficient matrix A is a nonsingular symmetric diagonally dominant matrix.

2.1. Symmetric Linear Equations and Its Probability Inference Model. First, We connect undirected graph with symmetric linear equations. Given an undirected graph $G = (V, E)$, where V is a set of all vertices in G corresponding to variables x in linear equations and E is the set of all edges associated with the non-zero elements in matrix A.

Now, we define the following joint Gaussian probability density function based on the coefficient matrix A and the observation vector b

$$p(x) \sim \exp(-\frac{1}{2}x^TAx + b^Tx), \tag{2}$$

and its corresponding graph G consisting of edge potentials ('compatibility functions') ψ_{ij} and self potentials ('evidence') ϕ_i. These graph potentials are simply determined according to the following pairwise factorization of the Gaussian function (2)

$$p(\mathbf{x}) \propto \prod_{i=1}^{n} \phi_i(x_i) \prod_{\{i,j\}} \psi_{ij}(x_i, x_j). \tag{3}$$

where $\psi_{ij}(x_i, x_j) \triangleq \exp(-x_iA_{ij}x_j)$ and $\phi_i(x_i) \triangleq \exp(-\frac{1}{2}A_{ii}x_i^2 + b_ix_i)$.

Proposition 1. *([7] Proposition 1 Solution and inference). The computation of the solution vector x^* is identical to the inference of the vector of marginal means $\mu = \mu_1, \cdots, \mu_n$ over the graph G with the associated joint Gaussian probability density function $p(x) \sim N(\mu \triangleq A^{-1}b, A^{-1})$.*

According to Proposition 1, we can translate the problem of solving the linear system (1) from the algebraic domain to the domain of probabilistic inference, see Figure 1.

Next, we will introduce the BP(Belief Propagation) algorithm. The set of graph nodes $N(i)$ denotes the set of all the nodes neighboring the ith node. The set $N(i) \setminus j$ excludes the node j from $N(i)$.

2.2. BP Algorithm. Belief propagation (BP) is equivalent to applying Pearls local message-passing algorithm [11], originally derived for exact inference in trees, to a general graph even if it contains cycles (loops). The excellent performance of BP in these applications may be attributed to the sparsity of the graphs.

The BP algorithm functions by passing real-valued messages across edges in the graph and consists of two computational rules, namely the $'$sum-product rule$'$ and the $'$product rule$'$. For a graph G composed of potentials ψ_{ij} and ϕ_i as previously defined, the conventional sum-product rule becomes an integral-product rule and the message $m_{ij}(x_j)$ [12], sent from node i to node j over their shared edge on the graph, is given by

$$m_{ij}(x_j) \propto \int_{x_i} \psi_{ij}(x_i, x_j)\phi_i(x_i) \prod_{k\in N(i)\rfloor} m_{ki}(x_i)dx_i. \tag{4}$$

$$
\begin{gathered}
Ax = b \\
\Updownarrow \\
Ax - b = 0 \\
\Updownarrow \\
\min_x \left(\frac{1}{2}x^T A x - b^T x\right) \\
\Updownarrow \\
\max_x \left(-\frac{1}{2}x^T A x + b^T x\right) \\
\Updownarrow \\
\max_x \exp\left(-\frac{1}{2}x^T A x + b^T x\right)
\end{gathered}
$$

FIGURE 1. From the algebraic domain to the domain of probabilistic inference

The marginals are computed according to the product rule

$$p(x_i) = \alpha \phi_i(x_i) \prod_{k \in N(i)} m_{ki}(x_i), \tag{5}$$

where the α is a normalization constant.

2.3. The GaBP Algorithm. GaBP is a special case of continuous BP, where the underlying distribution is Gaussian. Now, we derive the Gaussian BP update rules by substituting Gaussian distributions into the continuous BP update equations (4) and (5). Figure 2. plots a portion of a certain graph, describing the neighborhood of node i. Each node is associated with a variable and self potential ϕ, which is a function of this variable, while edges are identified with the pairwise potentials ψ. Messages propagate along the edges in both directions (only the messages relevant for the computation of m_{ij} are shown in Figure 2.).

Looking at the right hand side of the integral-product rule (4), we first introduce Lemma 1.

Lemma 2.1. *([7] Lemma 2 Product of Gaussians) Let $f_1(x)$ and $f_2(x)$ be the probability density functions of a Gaussian random variable with two possible densities $N(\mu_1, P_1^{-1})$ and $N(\mu_2, P_2^{-1})$, respectively. Then their product, $f(x) = f_1(x)f_2(x)$ is, up to a constant factor, the probability density function of a Gaussian random variable with distribution $N(\mu, P^{-1})$, where*

$$P^{-1} = (P_1 + P_2)^{-1}, \quad \mu = P^{-1}(P_1\mu_1 + P_2\mu_2).$$

Let $\phi_i(x_i) \prod_{k \in N(i) \setminus j} m_{ki}(x_i) \sim N(\mu_{i \setminus j}, P_{i \setminus j}^{-1})$, By formula (3), we can know $\phi_i(x_i) \propto N(mu_{ii}, P_{ii}^{-1})$, $m_{ki}(x_i) \propto N(mu_{ki}, P_{ki}^{-1})$. According to Lemma 1, we

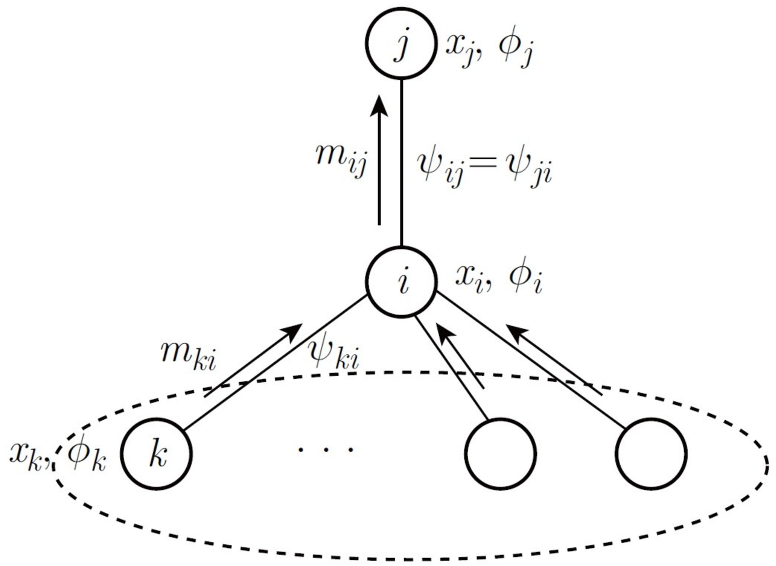

FIGURE 2. The neighborhood of node i.

have

$$P_{i\backslash j} = P_{ii} + \sum_{k\in N(i)\backslash j} P_{ki}, \tag{6}$$

and

$$\mu_{i\backslash j} = P^{-1}_{i\backslash j}(P_{ii}\mu_{ii} + \sum_{k\in N(i)\backslash j} P_{ki}\mu_{ki}), \tag{7}$$

where $P_{ii} \triangleq A_{ii}$, $\mu_{ii} \triangleq b_i/A_{ii}$.

Using the Gaussian integral $\int_{\infty}^{\infty} \exp(-ax^2+bx)dx = \sqrt(\pi/a)\exp(b^2/4a)$ and (4), we can find that the messages $m_{ij}(x_j)$ are proportional to a normal distribution with precision and mean

$$P_{ij} = -A^2_{ij}P^{-1}_{i\backslash j}, \tag{8}$$

$$\mu_{ij} = -P^{-1}_{ij}A_{ij}\mu_{i\backslash j}. \tag{9}$$

Computing the product rule (5) is similar to the calculation of the previous product and the resulting mean (9) and precision(8), Thus, the marginal are found to be Gaussian probability density functions $N(\mu_i, P_i^{-1})$ with precision and mean

$$P_i = P_{ii} + \sum_{k\in N(i)} P_{ki}, \tag{10}$$

$$\mu_i = P^{-1}_{ij}(P_{ii}\mu_{ii} + \sum_{k\in N(i)} P_{ki}\mu_{ki}). \tag{11}$$

For a dense matrix, the number of messages passed on the graph G can be reduced from $\mathcal{O}(n^2)$ down to $\mathcal{O}(n^2)$ messages per iteration round by using a similar construction to Bickson et al. [13]: Instead of sending a unique message composed of the pair of μ_{ij} and P_{ij} from node i to node j, a node broadcasts aggregated sums to all its neighbors, and consequently each node can retrieve locally $P_{i\jmath}$ and $\mu_{i\jmath}$ from the aggregated sums

$$\widetilde{P}_i = P_{ii} + \sum_{k \in N(i)} P_{ki}, \tag{12}$$

$$\widetilde{\mu}_i = \widetilde{P}_i^{-1}(P_{ii}\mu_{ii} + \sum_{k \in N(i)} P_{ki}\mu_{ki}). \tag{13}$$

By means of a subtraction,

$$P_{i\backslash j} = \widetilde{P}_i - P_{ji}, \quad \mu_{i\backslash j} = \widetilde{\mu}_i - P_{i\backslash j}^{-1} P_{ji}\mu_{ji}.$$

The number of messages are reduced from $\mathcal{O}(n^2)$ down to $\mathcal{O}(n^2)$.

to sum up, GaBP algorithm cotains initialization and iteration, and the iteration contains four sub parts. For the initialization part

$$P_{ij} = 0, \mu_{ij} = 0 \ (i \neq j),$$

and

$$P_{ii} \triangleq A_{ii}, \mu_{ii} \triangleq b_i.$$

There are three major computation parts in the iteration part, as listed below.
(1) The accumulation of message:

$$\widetilde{P}_i = P_{ii} + \sum_{k \in N(i)} P_{ki}, \quad \widetilde{\mu}_i = \mu_{ii} + \sum_{k \in N(i)} \mu_{ki},$$

(2) The broadcast and update of message

$$P_{i\backslash j} = \widetilde{P}_i - P_{ji}, \quad \mu_{i\backslash j} = \widetilde{\mu}_i - \mu_{ji}.$$

$$P_{ij} = -A_{ij}^2 P_{i\backslash j}^{-1}, \quad \mu_{ij} = -P_{i\backslash j}^{-1} A_{ij}\mu_{i\backslash j}.$$

(3) The Solving of the vector

$$x_i = \widetilde{\mu}_i / \widetilde{P}_i$$

The GaBP pseudo-code can see Algorithm 1 in [7].

3. Multicore-based parallel GaBP algorithm with dynamic load-balance

3.1. Data structure of sparse symmetric matrix. According to the special features of GaBP algorithm, we establish the data structure and data search method for this algorithm. The original left-hand side matrix is stored as follow.

$$A = \begin{pmatrix} 6 & 1 & 0 & 2 & 0 \\ 1 & 8 & 3 & 0 & 0 \\ 0 & 3 & 9 & 4 & 1 \\ 2 & 0 & 4 & 12 & 5 \\ 0 & 0 & 1 & 5 & 11 \end{pmatrix} = L + D + U =$$

$$\begin{pmatrix} & & & & \\ 1 & & & & \\ 0 & 3 & & & \\ 2 & 0 & 4 & & \\ 0 & 0 & 1 & 5 & \end{pmatrix} + \begin{pmatrix} 6 & & & & \\ & 8 & & & \\ & & 9 & & \\ & & & 12 & \\ & & & & 11 \end{pmatrix} + \begin{pmatrix} & 1 & 0 & 2 & 0 \\ & & 3 & 0 & 0 \\ & & & 4 & 1 \\ & & & & 5 \\ & & & & \end{pmatrix}$$

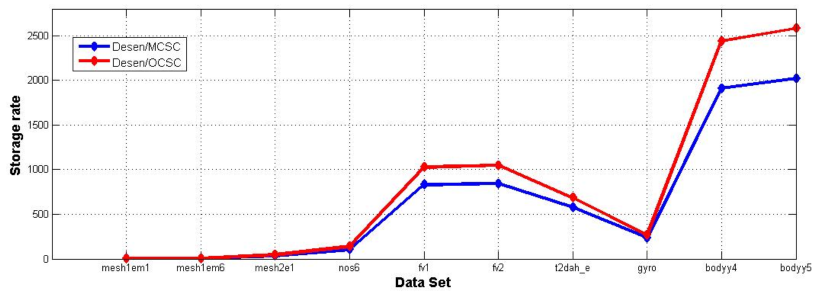

FIGURE 3. Storage ratio

$$AD = \begin{array}{|ccccc|}\hline 6. & 8. & 9. & 12. & 11. \\ \hline\end{array}$$

$$AL = \begin{array}{|cc|c|cc|c|c|}\hline 1. & 2. & 3. & 4. & 1. & 5. & * \\ \hline\end{array}$$

$$AU = \begin{array}{|c|c|c|cc|cc|}\hline * & 1. & 3. & 2 & 4. & 1. & 5. \\ \hline\end{array}$$

$$NBL = \begin{array}{|cc|c|cc|c|c|}\hline 2 & 4 & 3 & 4 & 5 & 5 & * \\ \hline\end{array}$$

$$NBJL = \begin{array}{|cccccc|}\hline 1 & 3 & 4 & 6 & 7 & 7 \\ \hline\end{array}$$

$$NBU = \begin{array}{|c|c|c|cc|cc|}\hline * & 1 & 2 & 1 & 3 & 4 & 5 \\ \hline\end{array}$$

$$NBJU = \begin{array}{|cccccc|}\hline 0 & 1 & 2 & 4 & 6 & 6 \\ \hline\end{array}$$

Let n be the size of matrix and nnz be the number of non-zero elements.

A real array AD contains the Values a_{ii},form column 1 to n, The length is n.

A real array AL contains the real Values $a_{ij}(i > j)$ stored column by column from column 1 to n, The length of AL is $\frac{nnz-n}{2}$.

A real array AU contains the real Values $a_{ij}(i < j)$ stored column by column,form column 1 to n, The length of AU is $\frac{nnz-n}{2}$.

An integer array NBL contains the row indices of elements $a_{ij}(i > j)$ as stored in the array AL . The length of NBL is $\frac{nnz-n}{2}$.

An integer array NBU contains the row indices of elements $a_{ij}(i < j)$ as stored in the array AU . The length of NBU is $\frac{nnz-n}{2}$.

An integer array $NBJL$ contains the pointers to beginning of each row in the arrays NBL and AL.

An integer array $NBJU$ contains the pointers to beginning of each row in the arrays NBU and AU.

We store AD,AL,NBL,$NBJL$,AU,NBU,$NBJU$ If A is not symmetric. This storage format is referred to as the Modified compressed spare column(**MCSC**).

If A is symmetric, the lower triangle matrix L and the diagonal matrix D of A should be stored. i.e. AD,AL,NBL,$NBJL$ are stored. This storage format is referred to as the Optimized compressed spare column (**OCSC**).

From Figure 3 we can see that ratio of density storage to MCSC and the ratio of density to OCSC increase with the order and the sparsity of matrix. Therefore, the MCSC and OCSC format need far less storage space than that of the density storage, and the OCSC format needs less storage space than that of the MCSC format. We use OCSC format to save the sparse matrix. If OCSC format is used to

Algorithm 1 Get a_{ij}

```
if $j = i$ then
    $a_{ij} = AD[i]$
end if
if $j < i$ then
    $i1 = j; j1 = i;$
else
    $i1 = i; j1 = j;$
end if
$tp = NBJL[j1]$
$NBJ = NBJL[j1 + 1] - NBJL[j1]$
for all $k \in 0..NBL$ do
    $tpk = NB[tp + k] - 1$
    if $tpk = i1$ then
        $a_{ij} = AL[tp + k]$
    end if
end for
```

save the sparse matrix, only the non-zero elements are stored. Another advantage of this format is that the diagonal elements can be directly accessed. For non-zero elements that are not located in the diagonal, only the sequence search algorithm is given in algorithm 3.1 We can use binary search to replace it in real implementation.

From GaBP algorithm we know that the sparsity of P μ and left-hand-side matrix A are equal. Both are symmetric in shape. P and A are symmetric in value of elements. Therefore, if **OCSC** format is used to save P and A, NBL,$NBJL$ and AL,AD are the same for P and A. Furthermore, the algorithm to obtain P_{ij} is identical to obtain a_{ij} in algorithm 3.1.

Because μ_{ij} is only symmetric in shape, the elements in upper triangle matrix should also be saved. μD,μL,μNBL,$\mu NBJL$,μU,μNBU,$\mu NBJU$ are saved if M-CSC format is used to save μ_{ij}.

3.2. Multicore-based parallel GaBP algorithm with dynamic load-balance. The parallel algorithm to solve sparse linear equations differs from the partition format of sparse matrix, which could be in row block partition format or the column block partition format. The block partition format contains the shutter format and block sequence partition format. We use the column block partition format to save the sparse matrix. In our algorithm, we assign each process the same number of columns in the column block. Since the number of zero elements in each column is not necessarily the same, the zero elements in each column block to be processed by different process are not necessarily the same. Obviously, the processing of different number of zero elements will lead to load imbalance. To solve this problem, we need to assign as equal the number of zero elements as we can to each process.

The partition algorithm using column block format is listed in algorithm 3.2. The input parameter of this algorithm is $NBsum$, n, p, with array $seg[\,]$ as the return value. In algorithm 3.2, $NB\_sum[j]$ is the number of zero elements in column j, p is the number of threads. The result of this algorithm is, for $q = 0, \cdots, p-1$, the columns ranging from seg_q to $seg_{q+1} - 1$ are assigned to thread q.

Algorithm 2 Partition(seg,NBsum,n,p)

```
1: $k = 0,\ q = 0,\ l = 0;\ seg[0] = 0$
2: for all $j \in 0..n$ do
3:     $l = l + NBsum[j]$
4: end for
5: for all $j \in 0..n$ do
6:     $k = k + NBsum[j]$
7:     if $k > (q+1) * l/p$ then
8:         $seg[q+1] = j;\ q = q + 1$
9:     end if
10:    $seg[p] = n + 1$
11: end for
```

In GaBP algorithm, some nodes will reach the status of dynamic balance after every round of iteration. In this case, it is not necessary to process these nodes. Therefore, the number of nodes in each column block will decrease and result in new load imbalance. To make each process get the same number of zero elements, it is necessary to re-partition the column block and re-assign each process new column block before the next iteration to obtain dynamic load-balance. Here we re-partition the column block after $DEFINE\_LOOP$ iterations. The efficiency is achieved by control the iteration step and the dynamic load-balance.

There are three major computation parts in the iteration part, which are the accumulation of message, the broadcast and update of message, and the solving of the vector. The multicore-based GaBP algorithm is implemented by putting these 3 parts into the parallel pragma context. The detailed implementation is illustrated in Algorithm3.2The input parameter of this algorithm is A, b, n and the output is the solution vector x.
In algorithm 3.2, we add iteration step control before function $Partition()$. The column block is re-partitioned after every $DEFINE\_LOOP$, which can be a self-defined fixed value, or a dynamic varied value. The value of $DEFINE\_LOOP$ is related to the number of columns in the column block in real implementation.

In algorithm 3.2, when x_i satisfies the per-defined convergence precision, we set the value T_i of this node to 1. This node will jump out of the iteration loop in the next iteration, and value $NBsum[j]$ of this node is set to 0.

When the value $NBsum[j]$ of this node equals to 0, this node will be removed from the calculation of its neighboring nodes in the implementation of dynamic load-balance in the next round. Because the computing load in this node is reduced, it is necessary to re-partition the column block to reach new dynamic load-balance.

The re-partition of the new column block before the next iteration will decrease the computation load in general. The computation load is relatively balanced in each thread so that many threads will not wait a long time before thread synchronization. Therefore, the iteration computation can be accelerated hereafter.

4. Numerical results

4.1. Environment of experiments. The hardware of the experiment is a server with 2 Intel Xeon E5-2650(20M cache, 8 cores, 16 Threads, 2.00GHZ) and 96G memory. The operating system is Redhat Linux 6.3X64, the compiler is gcc-4.81.

Algorithm 3 GaBP-MP algorithm

```
Require:
 1: $init()$;
Ensure:
 2: repeat
 3:   if $(iters\ mod\ DEFINE\_LOOP == 0)$ then
 4:     $Partition(seg, NBsum, n, Nthreads)$
 5:   end if
 6:   $\#\ pragma\ omp\ parallel\ private(tid, p, j)\ threads(NTHREADS)$
 7:   $\{\ tid = omp\_get\_thread\_num()$
 8:   for $(i = seg[tid]; j < seg[tid+1]\&\&T_i == 0; i++)$ do
 9:     if $(T_i == 0)$ then
10:       $P_i = A_{ii} + \sum_{k \in N(i)} P_{ki};\ \mu_i = b_i + \sum_{k \in N(i)} \mu_{ki}$
11:       for all $j \in N(i) \setminus j$ do
12:         $P_{ij} = -A_{ij}^2/(P_i - P_{ji});\ \mu_{ij} = -A_{ij}(\mu_i - \mu_{ji})/(P_i - P_{ji})$
13:       end for
14:     end if
15:   end for
16:   $\}$
17:   $\#\ pragma\ omp\ parallel\ private(tid, p, j)\ threads(NTHREADS)$
18:   $\{\ tid = omp\_get\_thread\_num()$
19:   for $(i = seg[tid]; i < seg[tid+1]\&\&T_i == 0; i++)$ do
20:     $P_i = A_{ii} + \sum_{k \in N(i)} P_{ki};\ \mu_i = b_i + \sum_{k \in N(i)} \mu_{ki}$
21:     $x_i = \mu_i / P_i$
22:     if $(T_i == 0)$ and $(x_i\ converged)$ then
23:       $T_i = 1;\ NBsum_j = 0$
24:     end if
25:   end for
26:   $\}$
27: until convergence: all $x_i$ converged
28: Output: $x^* = [x_i]$
```

4.2. Data sets and results. We choose UFget, a set of sparse matrice from University of Florida as the test data. The detailed list of the matrice that we used in our experiments are shown in Table 1.

In Figure 4, the two lines from top to the bottom represent the computing time for parallel GaBP algorithm and parallel GaBP with dynamic load-balance. We can see that the computing efficiency of the optimized parallel GaBP algorithm is better than that of parallel GaBP.

In Figure 5, the arrangement of the matrix that we use for the experiment is sorted according to the number of zero elements. We can see that the speedup of these 2 paralel algorithms increase with the number of zero elements.

From Figure 5, we can also see that when the scale of the problem increases, the speedup of the parallel GaBP algorithm with dynamic load-balance is better than the algorithm without dynamic load-balancie, which indicates that the parallel algorithm with dynamic load-balance is more suitable than the parallel GaBP algorithm in solving large-scale sparse linear equations.

In Figure 6, the two lines from top to the bottom represent the speedup for test case bodyy4, bodyy5, t2dah and dyro when different number of cores is used in

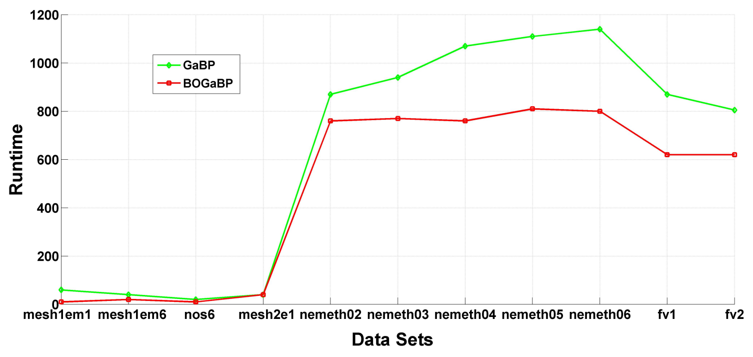

FIGURE 4. The speedup of multicore-based parallel GaBP algorithm

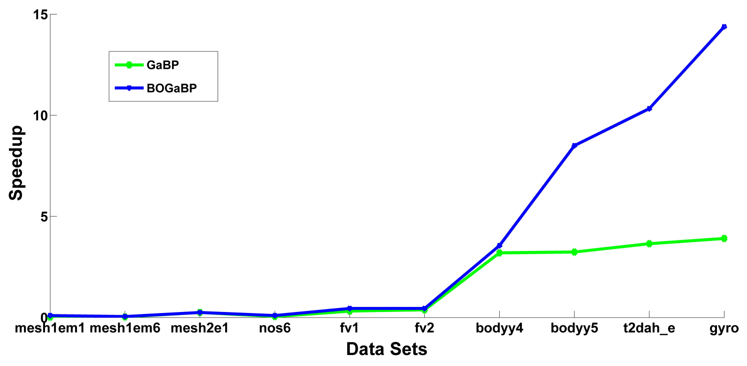

FIGURE 5. Comparison of the speedup of two multicore-based parallel GaBP algorithms

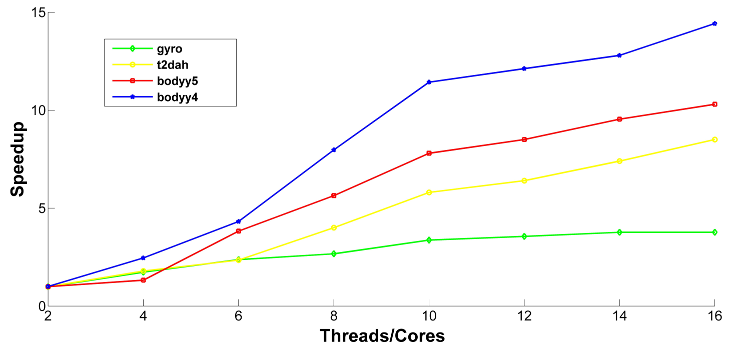

FIGURE 6. The speedup of multicore-based parallel GaBP algorithm using different test cases and different number of cores

Table 1. Data Sets in the Experiments

No	ID	Group	Name	Rows	Non-Zeros
1	873	Pothen	mesh1em1	48	306
2	874	Pothen	mesh1em6	48	306
3	875	Pothen	mesh2e1	306	2018
4	222	HB	nos6	675	3255
5	887	Norris	fv1	9604	85264
6	888	Norris	fv2	9801	87025
7	868	Pothen	bodyy4	17546	121938
8	869	Pothen	bodyy5	18589	129281
9	1205	Oberwolfach	t2dah_e	11445	176117
10	1435	Oberwolfach	gyro	17361	1021159

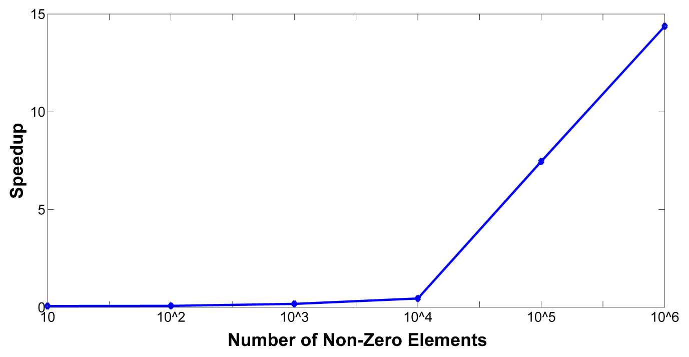

Figure 7. The speedup of multicore-based parallel GaBP algorithm using different order of zero elements

parallel GaBP algorithm with dynamic load-balance. Figure 3 indicates that the speedup of this algorithm increases with the scale of problem.

From Figure 7 we see that the speedup of two multicore-based parallel GaBP algorithms increase with the scale of problem. For small-scale problem where the number of rows in sparse matrix is smaller than 104 and the number of zero elements is smaller than 105, there is hardly any speedup result for multicore-based parallel GaBP algorithm. The time needed in solving small-scale problems is even longer than that of the serial algorithm. The reason for this is that it takes some time to initialize the parallel algorithm, and this initialization time is longer than the execution time of the serial algorithm. However, this should not be a problem for the parallel algorithm as we will not consider parallel computing when the scale of the problem is small and the problem can be easily solved using serial algorithm.

For problems with bigger scale, for example, the number of row reaches the order of 10,000 and the number of zero elements reaches the order of 100,000,there are some acceleration effects for two parallel GaBP algorithms.

For problems with large-scale, for example, the number of row reaches the order of 100,000 and the number of zero elements reaches the order of 10,000,000, both algorithms have good speedup, and the speedup of the multicore-based parallel GaBP algorithm with dynamic load-balancing is very dramatic. Therefore, this algorithm has a significant effect in solving large-scale problems.

5. Conclusion

In this paper, we store large-scale sparse linear equations in limited memory space using an OCSC storage format that is suitable for GaBP algorithm to reduce the space complexity. We explore the parallelism, load balancing features of this algorithm and present a multicore-based parallel GaBP algorithm with dynamic load-balance. The numerical results indicate that the scalability of this algorithm is improved and the iteration speed is accelerated compared with the original GaBP algorithm. The results also suggest that the acceleration efficiency is dramatically improved when the size of the problem increased. Therefore, the multicore-based parallel algorithm with dynamic load balance features higher parallel efficiency and better scalability for large-scale problems.

Acknowledgments

The authors thank all members of the high performance computing group at Shanghai University for their good advice and previous significant research work. This research was supported by the Major Research Plan of NSFC [No. 91330116], Key project of Science and Technology Commission of Shanghai Municipality [No. 11510500300] and the Research Fund for the Doctoral Program of Higher Education [No. 20113108120022].

References

[1] Xu Yan, Vegt van der, Jaap J.W., Bokhove Onno, Discontinuous Hamiltonian finite element method for linear hyperbolic systems, Journal of Scientific Computing, 35(2-3), 2008, 241-265

[2] Yongbin Ge, Fujun Cao, Jun Zhang, A transformation-free HOC scheme and multigrid method for solving the 3D Poisson equation on nonuniform grids, J. Comput. Physics, 234(23), 2013, 199-216

[3] Medi Bijan, Amanullah Mohammad, Application of a finite-volume method in the simulation of chromatographic systems: Effects of flux limiters, Industrial and Engineering Chemistry Research, 50(3), 2011, 1739-1748

[4] Sun X HZhang W, A parallel two-level hybrid method for tridiagonal systems and its application to fast poisson solvers, IEEE Transactions on Parallel and Distributed Systems, 15(2), 2004, 97-106

[5] Thomas J. Ashby, Pieter Ghysels, Wim Heirman, Wim Vanroose, The Impact of Global Communication Latency at Extreme Scales on Krylov Methods, 12th International Conference on Algorithms and Architerctures for Parallel Processing(ICA3PP-12), Fukuoka , Japan 2012, 428-442

[6] Yousef Saad, Iterative methods for sparse linear systems(2nd edition), Science Press, Beijing, 2009

[7] Ori Shental, Paul H. Siegel, Jack K. Wolf, Danny Bickson, Danny Dolev, Gaussian belief propagation solver for systems of linear equations, ISIT 2008, 1863-1867

[8] Yousef El-Kurdi, Warren J. Gross, and Dennis Giannacopoulos, Efficient implementation of Gaussian Belief Propagation Solver for Large Sparse Diagonally Dominant linear systems, IEEE Transactions on magnetics, 48(2), 2012, 471-474

[9] Yousef El-Kurdi, Dennis Giannacopoulos, and Warren J. Gross, Relaxed Gaussian Belief Propagation, 2012 IEEE International Symposium on Information Theory Proceedings(ISIT), Cambridge, MA 2012, 2002-2006

[10] Wang Weiwei, Xu Chuanju,Spectral methods based on new formulations for coupled Stokes and Darcy equations, Journal of Computional Physics, 257(2014), 2014, 126-142

[11] J. Pearl, Probabilistic Reasoning in Intelligent Systems: Networks of Plausible Inference, San Francisco, Morgan Kaufmann, 1988.

[12] Y. Weiss, W. T. Freeman, Correctness of belief propagation in Gaussian graphical models of arbitrary topology, Neural Computation, 13(10), 2001, 2173-2200

[13] D. Bickson, D. Dolev, and Y. Weiss, Modified belief propagation for energy saving in wireless and sensor networks, in Leibniz Center TR-2005-85, School of Computer Science and Engineering, The Hebrew University, 2005. [Online]. Available: http://leibniz.cs.huji.ac.il/tr/842.pdf

[14] A. DAVIS, Y. Hu, The University of Florida Sparse Matrix Collection, ACM Transactions on Mathematical Software, 38(1), 2011, 1-28

Department of Computer, Longyan University, Longyan 364000, China
E-mail: `zhyuan@shu.edu.cn`

School of Computer Science,Shanghai University, Shanghai 200072, China
E-mail: `apsong@shu.edu.cn,zxliu@shu.edu.cn,leixushu@shu.edu.cn,and wzhang@shu.edu.cn`
E-mail: `wzhang@shu.edu.cn`