18.337 Final Report
Parallel Implementation of Earth Tomography

Xuefeng Shang

May 14, 2008

1. Introduction

Earthquakes are usually disasters to human beings; however, the seismic wave can flash the interior structure of the earth, like lightning in the dark. In order to know the history, the evolution of our planet, we must know current structure of the earth and the seismic wave is the main information resource we can use.

Seismic data contain abundant information about interior earth. The seismogram has direct wave, reflection wave, diffraction wave and so forth. The first step of building up a preliminary earth model is to use the arrival time of direct waves, which is called tomography. Different rocks have different elastic properties; therefore, the travel time of a certain seismic ray depends on the rocks through which the ray path travels. So if we know the path of each ray and the location of sources and receivers, we can invert the elastic velocity model of the earth.

Tomography computation is quite expensive due to massive volume of data. For example, if there are M earthquakes and N stations, MN rays should be considered. In practice, tens of thousands of rays are required to get a reliable result. Therefore, parallel implementation is necessary and it is not very difficult to convert serial codes to parallel ones.

2. Serial Algorithm of Tomography

Assuming the exact locations of earthquakes are available, which is impossible in practice, we can retrace the seismic ray path for each source receiver pair by Snell Law.

The travel time t is

$$t(x,x') = \int_s \frac{ds}{v} \tag{1}$$

where x is source location and x' is receiver location. s is the ray path and v is the elastic velocity of rocks, as shown in figure 1.

In order to understand the structure of interior earth, we need mesh the earth into blocks and assume each block is homogeneous, which means the elastic property of a single block is the same. Equation (1) can be modified as

$$t(x,x') = \sum_j s_j p_j \tag{2}$$

where, $p_j = 1/v_j$ is the slowness. v_j is the elastic velocity of the j-th block; s_j is the length of ray path in j-th block. If one ray does not travel through k-th block, s_k is zero. For a certain source and receiver pair, we can see that vector s is very sparse. Given a medium velocity model P and source receiver location, the ray path can be retrieved by Fermat's principle. By doing ray tracing for each source receiver pair, we finally get the linear equation.

$$Sp = t \tag{3}$$

Where S_{ij} is the length of i-th ray path in j-th block. p_j is the slowness of j-th block. t_i is the travel time of i-th ray.

If we divide the earth into $N \times N \times N$ blocks, and the number of source receiver pairs is M, S is an $M \times N^3$ matrix. p is $N^3 \times 1$ and t is $M \times 1$. Usually, matrix S is large, very sparse and ill conditioned. It turns to be a sparse least square problem. Our aim is to find vector p to minimize $\|t - Sp\|$. The normal equation is

$$S^T Sp = S^T t \tag{4}$$

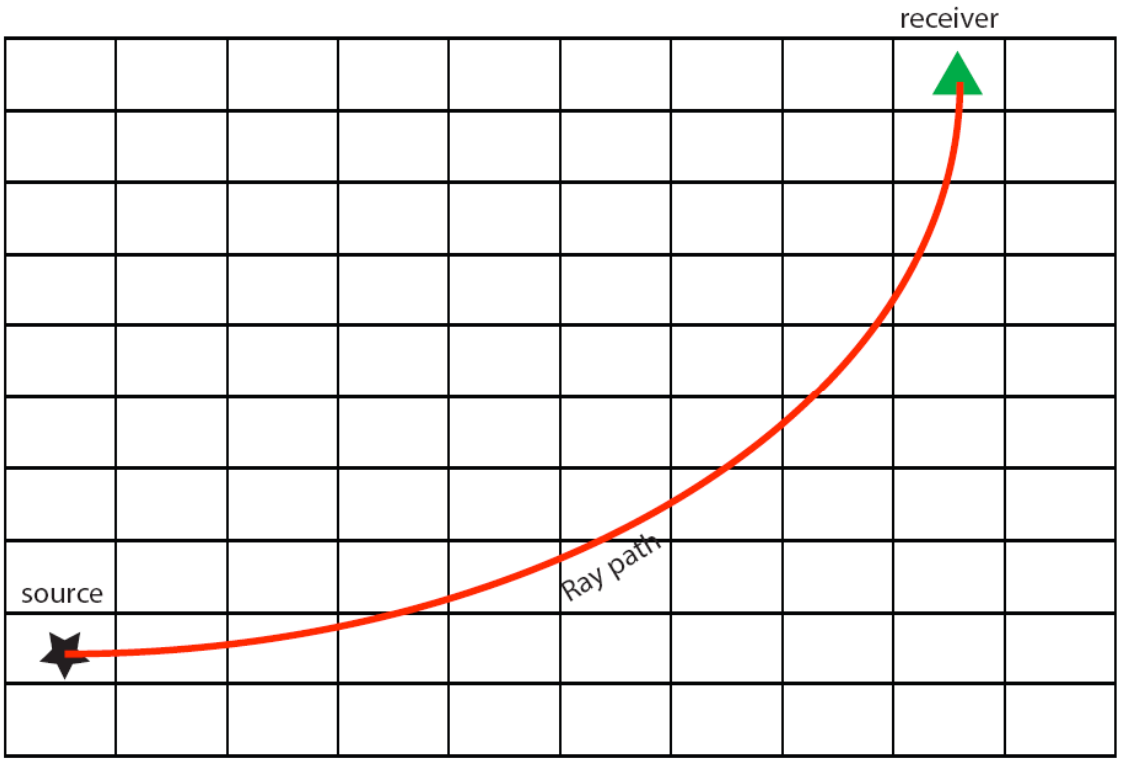

Fig.1 diagram of seismic ray path

.

Serial Tomography Algorithm:

1. Give an initial velocity model $p = p_0$
2. Do ray tracing for each seismic ray based on model p, and derive matrix S
3. Solve normal equation $S^T Sp = S^T t$ and update the value of velocity model p
4. Repeat step 2 and step 3 until meet certain convergent criterion.

3. Parallel Implementation

From the serial algorithm above, we can see that there are two parts in the algorithm: ray tracing and least square problem. Both of them are time consuming and need parallelism.

The first part is ray tracing for each source-receiver pair. For a single ray-tracing task, the input is the location of source (x, y, z) and receiver (x', y', z'), the travel time t as well as the velocity model p. At step 2, tens of thousands of ray-tracing tasks use the same velocity model p. Assuming the number of processors is N_p, and rays is N_R, the communication between nodes is rare. The message received from root node is about $O(7N_R / N_p)$. So it is quite suitable for SIMD parallelism. Such a simple embarrassing parallelism may be not the best choice but it is very practical and easy to implement.

The second one is how to solve a sparse least square problem in parallel. This is a big issue in earth science. One is concerned with storage. In present, most PC can only support up to 4GB memory. In order to get higher resolution, the grid need be divided in very small scale, so the data must be stored distributedly. Another issue is related with the sparse matrix operation. In serial codes, sparse matrix operation can be speed up by virtue of zeros in the sparse matrix. It is not an easy question how efficient the paralleling sparse matrix operation can be. A good parallel algorithm should balance the communication and computation cost well.

In serial world, mainly there are two ways to solve linear problem: direct and iterative methods. Direct methods have two classes: elimination methods and orthogonalization methods. The elimination method is based on Gaussian elimination. The most straightforward method is to compute the Cholesky factorization of normal equation (4).

$$PS^T SP^T = LL^T \tag{5}$$

where P is a permutation matrix. The solution is obtained by solving two triangular systems $Ly = PS^T t$ and $L^T (Pp) = y$. Also we can ignore the symmetric property of the normal equation, and do LU factorization directly. However, in both methods, if S is sparse, there is a fill-in issue during the factorization. In order to keep the sparsity, methods of reordering the pivots are developed, such as Peters-Wilkinson decomposition and dynamic reordering.

Another direct method is based on orthogonalization. The basic idea is that there exists an essentially unique factorization for any $M \times N$ matrix A

$$A = QU \tag{6}$$

where Q is an orthogonal matrix and U is upper triangular. This kind of method has higher stability in ill conditioned cases. Small eigenvalues or singular values can be eliminated during the decomposition. The simplest of this method is the modified Gram-Schmidt (MGS) method. Also Householder transformation or Givens transformation can be used to obtain equation (6).

Most iterative methods used in linear systems is based on conjugate gradient method, such as CGLS, LSQR and so forth. Compared with direct method, iterative methods need less storage and can be better controlled during the iteration. As to sparse matrix, direct methods often have fill-in problems and the index reordering process is relatively complex. So it may not perform as well as on dense systems.

4. LSQR Method

Considering my problem, in which S is very large, very sparse and ill conditioned, the storage and stability are both important. LSQR method, which was proposed by Paige in 1982, is a winner among those methods. I don't need to trace the reordering stuff in parallelized code, which is not quite easy. However, iterative method cost more time on communication, because at step, the new vector need to be updated.

LSQR method is a derivation of conjugate gradient method. Paige first found the neat relationship between two bidiagonalization results from Lanczos process. He also compared LSQR with other methods in his paper and concluded that LSQR method had higher stability. The algorithm of LSQR is below.

LSQR Algorithm:

Giving equation $Ax = b$

1. Initialize
   $\beta_1 u_1 = b, \quad \alpha_1 v_1 = A^T u_1, \quad w_1 = v_1, \quad x_0 = 0$
   $\bar{\phi}_1 = \beta_1, \quad \bar{\rho}_1 = \alpha_1$
   where $\alpha_i, \beta_i > 0$ and $\|v_i\| = 1, \ \|u_i\| = 1$

2. For i=1, 2, 3, … repeat steps 3-6

3. Continue the bidiagonalization
   $\beta_{i+1} u_{i+1} = A v_i - \alpha_i u_i$
   $\alpha_{i+1} v_{i+1} = A^T u_{i+1} - \beta_{i+1} v_i$

4. Construct and apply next orthogonal transformation
   $\rho_i = (\bar{\rho}_i^2 + \beta_{i+1}^2)^{1/2} \qquad c_i = \bar{\rho}_i / \rho_i$
   $s_i = \beta_{i+1} / \rho_i \qquad \theta_{i+1} = s_i \alpha_{i+1}$

$$\bar{\rho}_{i+1} = -c_i \alpha_{i+1} \qquad \phi_i = c_i \bar{\phi}_i$$
$$\bar{\phi}_{i+1} = s_i \bar{\phi}_i$$

5. Update x,w
   $x_i = x_{i-1} + (\phi_i / \rho_i) w_i$
   $w_{i+1} = v_{i+1} - (\theta_{i+1} / \rho_i) w_i$

6. Test for convergence

5. Analysis of Sparse Matrix-Vector multiplication

In tomography equation $Ax = b$, A is very sparse but vectors x,b are dense. In practice, non-zero entries of A distribute quite evenly in rows. On the other hand, the most time-consuming part of LSQR algorithm is step 3 and step 5, which correspond sparse matrix multiplication and vector addition, respectively. Dense vector addition is relatively easy to parallelize, so the key to parallelize LSQR method is to parallelize sparse matrix vector multiplication.

One of storage formats of sparse matrices is Harwell-Boeing format, compressed row storage (CRS), which stores non-zeros row-wise contiguously. A sparse matrix is stored in 3 arrays, shown in Figure 2. One is to store non-zeros, while the others store the column index of each non-zeros and the index of first non-zero in each row, respectively. The total storage is NNZ real numbers and NNZ+N+1 integers, where NNZ is the number of non-zeros and N is the number of rows.

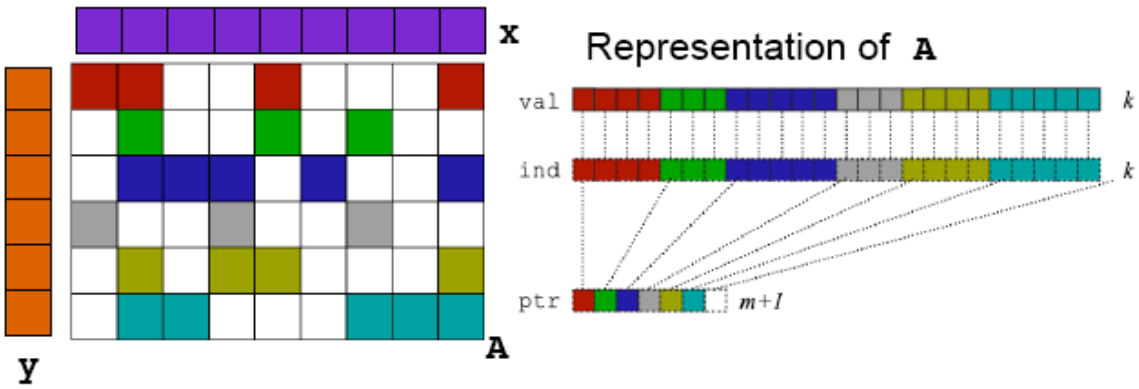

Fig. 2 CRS format of sparse matrices

In CRS format, matrix-vector multiply $y_i = A_{ij} x_j$ can be represented as the following:

```
for each row i
        for k=ptr[i]  to ptr[i+1]-1
                y[i]=y[i]+val[k]*x[ind[k]]
end
```

The matrix-vector multiply $y_i = A_{ij}^T x_j$ in step 3 of LSQR algorithm can be represented as:

```
for each row i
        for k=ptr[i]  to ptr[i+1]-1
                y[ind[k]]=y[ind[k]]+val[k]*x[i]
end
```

In serial algorithm, the performance of SpMV highly depends on machine, kernel and matrix structure. Good data structure may speed it up surprisingly, for instance diagonal matrices. From Fig. 3 we can see that the peak speed of SpMV on scalar tuned machine is only about 20%. Actually the bottleneck of SpMV is due to time of fetching data from memory. In order to improve the performance, for certain matrices, the non-zeros can be stored by block other than individually. The storage can drop down and the speed can increase by 50% ~ 300%. The block size $r \times c$ can be searched in run-time to minimize the time cost. However this method is based on case by case. As to an arbitrary matrix, extra reordering or fill-in cost can be introduced. Back to my problem, the properties of matrix A need to be examined firstly before paralleling implement.

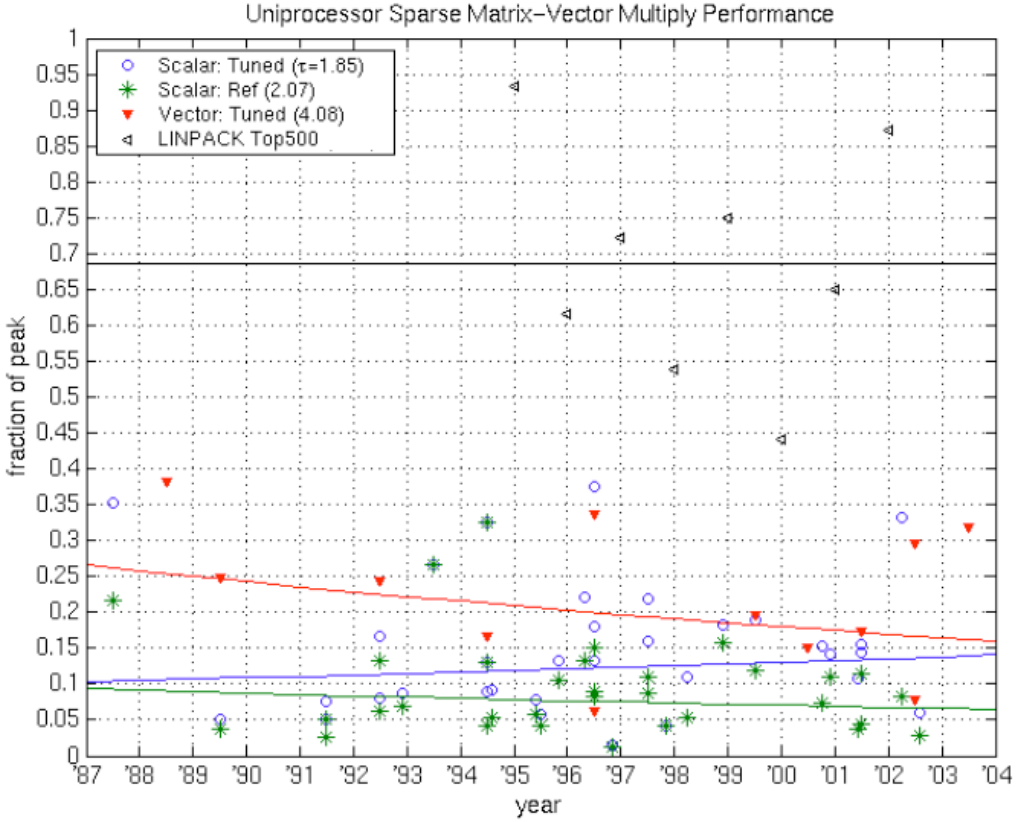

Fig. 3 serial performance of SpMV

6. Parallelism of SpMV

The most important issues of parallelism are locality and load balance. In order to keep load balance, the partitions should be by non-zeros counts, not merely by rows or columns. On the other hand, to keep locality, "owner computes" rule is employed, which means that processor *k* stores $y[i], x[i]$ and row *i* of *A* for all *i* in N_k and computes $y[i] = A_i * x$.

As to the matrix A in our problem, the number of non-zeros in each row is quite the same. So 1D partition by row is a natural way. A bad news is that it is hard to make high locality because the non-zeros distribute very evenly, shown in Fig. 4. The typical sparcity is around 1%. Although it is not as sparse as band diagonal matrices, matrix-vector multiplication can speed up by more than ten times (e.g. Fig. 5).

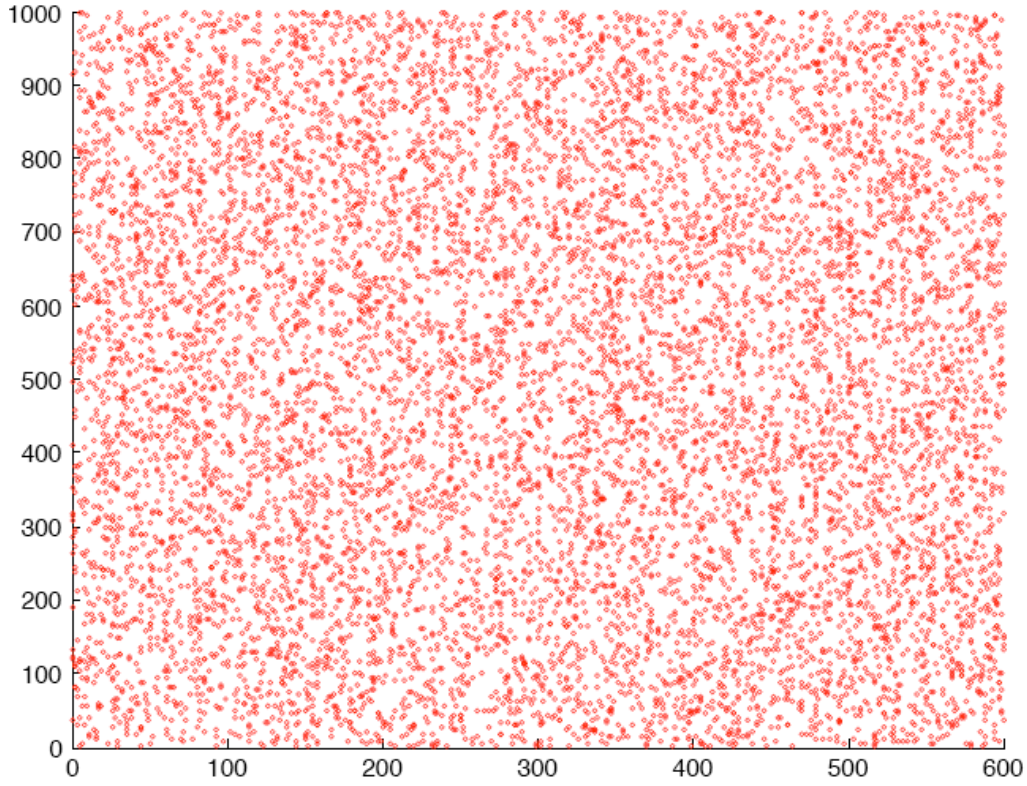

Fig. 4 non-zeros of A. nnz=10020, sparsity=0.0167

Assuming A is $M \times N$, and the sparsity is ρ, then the number of non-zero element in each row is ρN, if non-zeros distribute evenly. If we have n_p processors and divide A by rows (regardless of scale issues), the communication cost of each processor is $O(N/n_p) + O(2\rho MN/n_p)$. Here $O(N/n_p)$ is the cost of sending the value of vector x, and $O(2\rho MN/n_p)$ is the communication cost of matrix A portion. The computation cost of each processor is $O(2\rho MN/n_p)$.

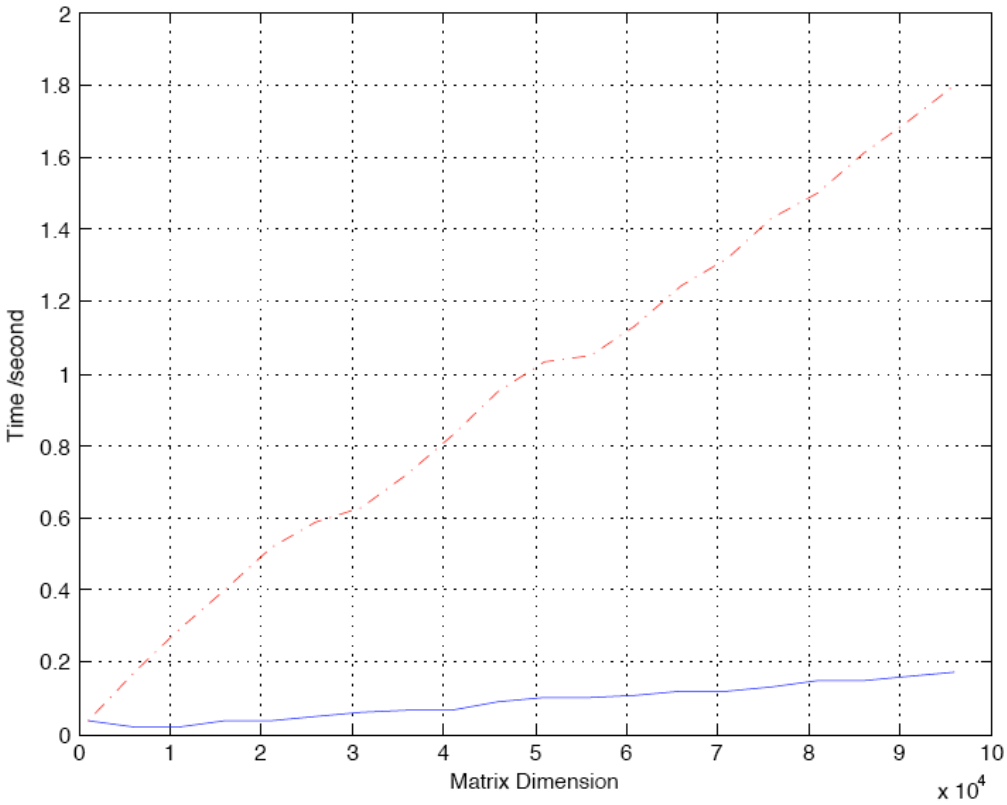

Fig. 5 Time cost of dense matmul and sparse matmul. Red line is dense case and Blue one is sparse. Sparsity is 0.01

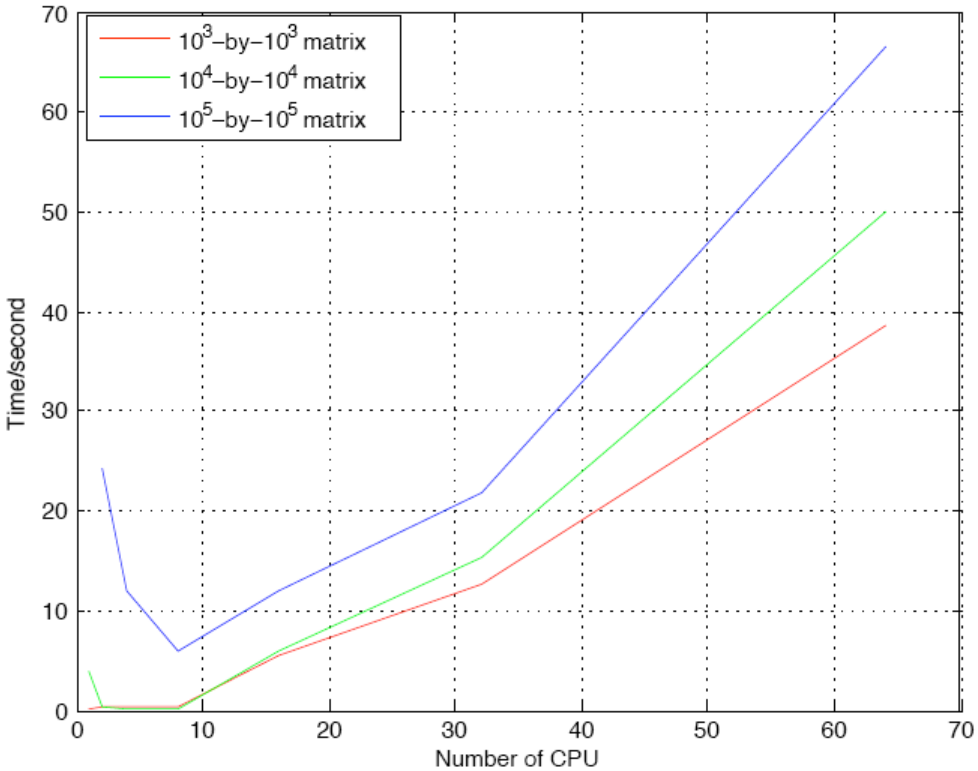

Fig. 6 Time cost of SpMV in parallel. Sparsity of matrices is 0.01.

In this naïve parallel algorithm, communication cost is higher than computation cost. The highest parallel efficiency is no more than 0.5. Figure 6 shows the parallel efficiency with the number of processors. We can see that the time cost does not decrease monotonically as expected. It seems there is a tradeoff between communication cost and computation cost. Index searching of vector x and the network structure effect the computing efficiency remarkably, especially more processors are used.

7. Conclusion

In this report, I analyze the serial algorithm of tomography, and parallelize the codes using different strategies in different steps of the algorithm. In ray-tracing part, a simple embarrassing parallelism is good enough, for it is a typical SIMD case. Another part of tomography is a sparse least square problem. Here I choose LSQR after comparing with other methods. The core of paralleling implementation of LSQR is sparse matrix vector multiplication (SpMV). Here I use 1D row partition given the special structure of the matrix.

From the test results of SpMV, we can see that SpMV parallelism is a quite complex problem. It depends not only the structure of matrices but also the hardware and network issues. It is also a very interesting work to compare direct and iterative methods, or even hybrid methods dealing with sparse least square problems.

Reference

Bjorck A. and Duff I. S., A direct method for the solution of sparse linear least squares problems, *Linear Algebra and Its Applications,* Vol. 34, 1980, 43-67
Paige C. and Saunders M. A., LSQR: an algorithm for sparse linear equations and sparse least squares, *ACM Trans. Math. Software,* Vol 8, No. 1, 1982, 43-71
Golub G. H. and Van Loan C. F, Matrix computations, 1983, The Johns Hopkins University Press, Baltimore, Maryland