Software implementation of the conjugate gradient method for shared and distributed memory multiprocessor systems

T S Rein and S N Karabtsev¹

UNESCO of ICT chair, Kemerovo State University, Kemerovo, Russian Federation

¹Email: skarab@kemsu.ru

Abstract. The article examines different approaches to parallelizing the conjugate gradient algorithms for shared and distributed memory multiprocessor systems. We use the standards of MPI and OpenMP libraries of the C programming language for parallel programming. Testing the algorithms was carried out on the HP BLc3000 supercomputer, which contains 392 Intel cores and the InfiniBand network. The article presents the test results – acceleration and efficiency of parallel implementation for each type of multiprocessor systems depending on the dimension of a system of linear equations and the number of parallel processes. We present short descriptions of parallelization methods and excerpts of the code.

1. Introduction

Solving a system of linear equations (SLE) is extremely important as it underlies a great number of complex practical tasks. Solving SLEs accounts for around 75% of all computational tasks. Although this task is comparatively seldom of interest for various applications in and of itself, mathematical physical processes modeling often depends on the ability to solve such systems efficiently, since a considerable part of numerical methods of solving different tasks involves solving SLEs as an elementary step of a corresponding algorithm.

It is necessary to solve SLEs when dealing with multi-dimensional boundary value problems, in computational fluid dynamics, electric circuit theory, conservation or balance equations, hydraulics, etc. Simultaneously, SLE matrices are often ill-conditioned. Therefore, common SLE solution methods, such as Gaussian elimination, prove to be inefficient.

Difficulties of computing SLE’s solutions are primarily due to large sizes of the matrices, resulting in the vast amount of computation required, which, in turn, entails significant expenditure of time, especially in three-dimensional tasks.

One way to substantially increase the speed of computation is parallelization of algorithms. The increase in speed, achieved through parallel computing, is only limited by the number of processors (CPUs) and autonomous computers.

The article examines parallel algorithms for the iterative method of solving an SLE – the conjugate gradient method for shared and distributed memory multiprocessor systems. Matrices and vectors act as data in these algorithms. We parallelize the algorithms through data decomposition.

2. Sequential algorithm of the conjugate gradient method

Let the coefficient matrix $A$ SLE $Ax = b$ be symmetric and positive definite, then the function $q(x) = 0.5x^T \cdot Ax - x^T b + c$ has its only minimum value at $x^*$, corresponding to the solution of the
initial system of linear equations. The conjugate gradient method [1] is one of many iterative algorithms which allow solving the initial system’s equation by minimizing the function $q(x)$.

Iteration of the conjugate gradient method involves calculating the next approximation to the exact solution in accordance with the rule: $x^k = x^{k-1} + s^k p^k$. Thus, a new approximation value $x^k$ is calculated taking into account the following: the approximation, calculated in the previous step $x^{k-1}$, scalar step $s^k$ and direction vector $p^k$. Prior to the first iteration, vectors $x^0$ and $p^0$ are taken to be equal to the zero vector, and vector $g^0$ is defined as $-b$. Then, in order to calculate the next $x^k$ approximation value, each iteration in the loop is carried out in five steps:

1. calculate the residual $g^k = A \cdot x^{k-1} - b$;
2. calculate the descent direction $p^k = -g^k + \beta_k p^{(k-1)}$, where $\beta_k = \frac{(g^k)^T \cdot g^k}{(g^{k-1})^T \cdot g^{k-1}}$;
3. calculate the bias value on the selected direction $s^k = \frac{(g^k)^T \cdot g^k}{(p^k)^T \cdot A \cdot p^k}$;
4. calculate the new approximation $x^k = x^{k-1} + s^k p^k$;
5. check the breakpoint condition $\|x^k - x^{k-1}\| \leq \varepsilon$.

Convergence rate of iterative methods depends on spectral properties of the matrix. Of great practical interest is such transformation of the initial SLE matrix when the solution vector remains the same, whilst the spectral properties of the new matrix improve. The transformations are carried out through multiplying the initial system of equations by the matrix $M^{-1}$ in such a way that $M^{-1}A x = M^{-1}b$, where $M$ is a non-degenerate matrix. Preferably, the matrix $M$ should be easily calculable and invertible.

Let the matrix $A$ be representable as a product of lower and upper triangular matrices: $A = LU$. It allows us to solve $Ax = b$ SLEs easily through forward path for the lower triangular $Ly = b$ and then the backward path for the upper triangular system $Ux = y$. Then the matrix $M$, approximate to the initial matrix can be described as: $M = LU + R$, where $R$ is the error matrix with the values of non-zero elements of the decomposition matrix $A$, with $a_{ij} = 0$ [1, 2].

A typical $LU$ -factorization of a sparse matrix leads to phase portrait filling of the matrix, i.e. non-zero elements in matrices $L$ and $U$ appear in such positions $(i, j)$ for which $a_{ij} = 0$. Here, we define the phase portrait $P_A$ of the matrix $A$ as a set of index pairs $(k, m)$ with $a_{km} \neq 0$.

### 3. Parallel implementation of the conjugate gradient method for shared memory systems

In this research, we use data parallelism for algorithms (loop-level parallelism) [3]. Each running parallel process handles a certain part of given data. We parallelize the outer loop, whose variable points at the rows of the initial matrix $A$ and the elements of solution vectors $x$ and residual $g^k$, as well as the descent direction $p^k$. Thus, the data is divided into $p$ horizontal rows, where $p$ is the number of parallel processes [4]. Figure 1 shows a parallel code fragment of the five main steps of the conjugate gradient method, written in C++ using the OpenMP library. The remaining fragments were omitted to save space in the printed work.

Table 1 shows acceleration and efficiency of the parallel algorithm for the conjugate gradient method for shared memory systems. In the table, $p$ denotes the number of threads on a processor (CPU), $N$ - the size of the SLE matrix. For test purposes, a matrix was generated as follows: the elements on the main diagonal where selected at random in a range from $N$ to $2N$, while the remaining elements were given in a range from 0 to 1 and input symmetrically outside the main diagonal. Residual value $\varepsilon = 10^{-6}$ was chosen as the breakpoint criterion of the iterative process.
do /  
Spz=0; Spr=0; Spr1=0;  
#pragma omp parallel private (size,i) // Beginning of parallel program block  
size=omp_get_num_threads();  
Nump=M/size;  
//Parallelization of the cycle computing the numerator and denominator of $s^k$  
#pragma omp for schedule (status,Nump) reduction(+:Spz,Spr)  
for(i = 0; i < M; i++)  
  for (Sz[i] = 0, j = 0; j < M; j++) Sz[i] += A[i][j]*Zk[i];  
  Spz += Sz[i]*Zk[i]; Spr += Rz[i]*Rk[i];  
#pragma omp end for  
Spz=0;  
#pragma omp parallel private (size,i)  
beginning of parallel program block  
size=omp_get_num_threads();  
for(Sppr = 0, i = 0; i < M; i++)  
  Xk[i] += alfa*Zk[i];  
#pragma omp end parallel  
alfa=Spr/Spz;  
#pragma omp for schedule (status,Nump) reduction(+:Spz,Spr)  
for(i = 0; i < M; i++)  
  Xk[i] += alfa*Zk[i];  
  Rk[i] += alfa*Sz[i];  
  Spr1 += Rk[i]*Rk[i];  
  Spr += Rz[i]*Rk[i];  
#pragma omp end for  
alfa=Spr1/Spz;  
#pragma omp critical  
betta=Spr1/Spz;  
#pragma omp end critical;  
while(sqrt(Spr1) > Eps);  
#pragma omp end for  

Figure 1. Parallel code of the conjugate gradient method.

Furthermore, the test was also carried out on the matrices, obtained through numerical simulation of fluid dynamics using the Natural Element Method [5]. In this case, the matrices were sparse, positive definite and took up relatively little storage space [6].

Table 1. Acceleration and efficiency of parallel algorithm.

| N    | p=2   | p=4   | p=8   | p=2   | p=4   | p=8   |
|------|-------|-------|-------|-------|-------|-------|
| 1000 | 0.743 | 0.562 | 0.478 | 0.371 | 0.140 | 0.060 |
| 4000 | 1.503 | 1.816 | 1.175 | 0.751 | 0.454 | 0.147 |
| 8000 | 1.100 | 1.544 | 1.790 | 0.550 | 0.386 | 0.224 |
| 15000| 0.983 | 1.619 | 1.643 | 0.492 | 0.405 | 0.205 |
| 20000| 1.179 | 1.701 | 2.098 | 0.589 | 0.425 | 0.262 |
| 25000| 1.119 | 1.587 | 1.849 | 0.560 | 0.397 | 0.231 |
| 30000| 1.065 | 1.477 | 1.768 | 0.532 | 0.369 | 0.221 |
| 35000| 1.197 | 1.653 | 2.125 | 0.599 | 0.413 | 0.266 |
| 40000| 1.240 | 2.030 | 2.544 | 0.620 | 0.508 | 0.318 |

As the table suggests, the best acceleration is achieved when using 8 cores and the size of the matrix is $N > 15000$. For each $N$, efficiency decreases as the number of threads increases, for instance, if $N = 40000$, the efficiency for 2 cores is $\approx 0.62$, and twice as little for 8 cores. If the size of the matrix is lower than 10000, using OpenMP directives for parallelization yields practically no saving in time.

4. Parallel implementation of the conjugate gradient method for distributed memory multiprocessor systems

While developing the conjugate gradient method algorithm for distributed memory multiprocessor systems [7], it is worth noting that iteration of the method is carried out sequentially, hence the most effective approach is to parallelize the computations inside the iterations. If one uses the
preconditioner matrix, then the algorithm involves two steps. In the first step, we use the parallel LU factorization of the initial matrix, while the obtained SLE with a precondition matrix is solved through parallel Gaussian elimination. During the second step, we parallelize the conjugate gradient method itself through one-dimensional decomposition of the computational domain column by column.

Iterations of the method involve multiplying the matrix by vector, multiplying out the vectors and calculating the dot product of vectors. Obviously, these operations are to be parallelized. During parallel implementation, the data is distributed among the processors as follows: the system matrix \( A \), solution vector \( x \) and right-hand member \( b \), residual vector \( g \) and direction \( p \) are stored in blocks on each processor unit. Simultaneously, the data exchange between processors is required only for the direction vector (figure 2).

```plaintext
Rkp; Xk; Rk[i]=B[i]-Sr[i]; // Each parallel branch defines
Zk[i]=Rk[i]; // initial value for x_k, r_k, z_k
Sr[i]=A[i][j]*Rkp[i];
Pk[i]=A[i][j]*Rkp[i];

do
{ Spp=0; Sppr=0;
  for (i = 1; i < N; i++) { Spp+=Pk[i]*Pk[i]; Sppr+= Pk[i]*Rk[i] }
  MPI_AllReduce(&Spp, &Spp, 1, MPI_DOUBLE, MPI_SUM, comm_gr);
  MPI_AllReduce(&Sppr, &Sppr,1, MPI_DOUBLE, MPI_SUM, comm_gr);
  Alfa = Sppr/Sppp; Spr1=0;
  for ( i =0; i < N; i++) Xk[i]+=alfa*Zk[i]; Rk[i]-=alfa*Pk[i]; Spr1+=Rk[i]*Rk[i];
  MPI_AllReduce (&Spr1, &Spr1p, 1, MPI_DOUBLE, MPI_SUM, comm_gr);
  for(Sppr = 0, i = 0; i < N; i++) Sr[i]+=A[i][j]*Rkp[j];
  MPI_AllReduce(&Sppr, &Sppr,1, MPI_DOUBLE, MPI_SUM, comm_gr);
  betta=Spppr/Sppp;
  for( i = 0; i < N; i++) { Zk[i]=Rk[i]+betta*Zk[i]; Pk[i]=Sr[i]+betta*Pk[i];}
} while (Spr1p/mfp > Eps*Eps);
```

**Figure 2.** Parallel code distributed memory multiprocessor systems.

Both the parallel procedure of multiplying the matrix by vector and the dot product of vectors involve the following steps: calculating local product of vectors on each processor and acquiring of the global output vector through MPI_Allreduce and MPI_SUM.

Table 2 shows the results of acceleration and efficiency of the parallel algorithm for the conjugate gradient method for distributed memory systems.

**Table 2.** Acceleration and efficiency of the conjugate gradient parallel algorithm.

| N   | Acceleration (p=2) | Acceleration (p=4) | Efficiency (p=2) | Efficiency (p=4) |
|-----|--------------------|--------------------|------------------|------------------|
| 1000 | 1.031              | 2.678              | 0.515            | 0.670            |
| 4000 | 1.879              | 3.694              | 0.940            | 0.924            |
| 8000 | 1.934              | 2.775              | 0.967            | 0.694            |
| 10000| 1.918              | 3.696              | 0.959            | 0.924            |
| 15000| 1.897              | 3.712              | 0.949            | 0.928            |
| 25000| 1.993              | 3.956              | 0.997            | 0.989            |
| 30000| 1.981              | 3.941              | 0.991            | 0.985            |
It was based on a parallel program for shared memory, and acceleration values, presented in tables 3 and 4. In due to delays in carrying out individual steps of the algorithm and synchronization of the parallel acceleration, compared to 8 cores. In case of 4 processors, the data transfer time increases for a number of nodes within the range 2 to 32 do not differ significantly. Having said that, for two processors, as the size of the matrix exceeds 25000, using 6 cores on each processor yields better performance of the algorithm and synchronization of the parallel threads. For this reason, using more than 2-4 threads on each processor does not result in better performance of the algorithm and decrease in time. It is also reflected in the relatively lower efficiency and acceleration values, presented in tables 3 and 4. In the tables, $p$ is the number of processors, while $k$ is the number of cores. When calculating the efficiency, the number of nodes was taken as the product of the number of processors $p$ multiplied by the number of cores $k$ for each processor.

**Table 3.** Acceleration of the conjugate gradient parallel algorithm for hybrid systems.

| N     | $p=2$, $k=2$ | $p=2$, $k=4$ | $p=2$, $k=8$ | $p=4$, $k=2$ | $p=4$, $k=4$ | $p=4$, $k=8$ |
|-------|--------------|--------------|--------------|--------------|--------------|--------------|
| 1000  | 3.234        | 4.491        | 4.621        | 3.979        | 1.664        | 3.998        |
| 4000  | 2.856        | 3.408        | 3.394        | 3.330        | 3.421        | 3.135        |
| 8000  | 2.767        | 3.372        | 2.457        | 3.331        | 3.322        | 3.245        |
| 10000 | 2.822        | 3.392        | 3.255        | 3.355        | 3.167        | 3.152        |
| 15000 | 2.769        | 3.110        | 2.852        | 3.186        | 2.790        | 2.670        |
| 20000 | 2.875        | 3.285        | 2.993        | 3.272        | 3.024        | 2.937        |
| 25000 | 2.876        | 3.300        | 3.049        | 3.290        | 2.967        | 2.933        |
| 30000 | 2.871        | 3.284        | 3.056        | 3.109        | 2.926        | 2.735        |
| 35000 | 2.872        | 3.285        | 3.045        | 3.222        | 2.960        | 2.816        |
| 40000 | 0.576        | 0.657        | 0.555        | 0.634        | 0.548        | 0.536        |

**Table 4.** Efficiency of the conjugate gradient parallel algorithm for hybrid systems.

| N     | $p=2$, $k=2$ | $p=2$, $k=4$ | $p=2$, $k=8$ | $p=4$, $k=2$ | $p=4$, $k=4$ | $p=4$, $k=8$ |
|-------|--------------|--------------|--------------|--------------|--------------|--------------|
| 4000  | 0.714        | 0.426        | 0.212        | 0.416        | 0.214        | 0.098        |
| 8000  | 0.692        | 0.421        | 0.154        | 0.416        | 0.208        | 0.101        |
| 10000 | 0.706        | 0.424        | 0.203        | 0.419        | 0.198        | 0.099        |
| 20000 | 0.719        | 0.411        | 0.187        | 0.409        | 0.189        | 0.092        |

5. Hybrid SLE solution algorithm through the conjugate gradient method for shared and distributed memory multiprocessor systems

In modern supercomputers, each processor has several cores, which allows launching several threads. Hybrid implementation of OpenMP and MPI provides a great opportunity to improve the performance of parallel applications. We also implemented a hybrid algorithm for the conjugate gradient method for shared and distributed memory systems. It was based on a parallel program for shared memory systems. Calculation of the local product of vectors on each processor, the residual, the direction vector, the offset value on given direction were carried out through OpenMP directives [4].

Our research has shown that using the parallel version of the algorithm for hybrid systems gives a substantial advantage over the sequential algorithm. However, the branches of the parallel realizations for a number of nodes within the range from 2 to 32 do not differ significantly. Time to solve an SLE decreases in proportion to the increase in the number of processors. Particularly, if $N = 45000$, the computational time on 1 processor (sequential algorithm) is roughly 80 seconds, and on 2 processors (double the number of processors) – 40 seconds, which is twice lower. Further increase in the number of processors up to 40 reduces the time by two. As the table suggests, if $N > 25000$, the efficiency of the parallel algorithm for both 2 and 4 processors is close to 1.
| N  | p=2, k=2 | p=2, k=4 | p=2, k=8 | p=4, k=2 | p=4, k=4 | p=4, k=8 |
|----|---------|---------|---------|---------|---------|---------|
| 40000 | 0.144   | 0.082   | 0.035   | 0.079   | 0.034   | 0.017   |

The decrease in acceleration at $N > 10000$ is due to insufficient computation load on each process. One can conclude that using the hybrid approach to parallelization is reasonable in case of large dimensionality of the task. If $N > 20000$, it is advisable to use no more than 16 nodes.

6. Conclusion

In this article, we have presented the description and quasi-code of parallel implementation of the conjugate gradient method for shared and distributed memory multiprocessor systems. Having examined the efficiency and acceleration values, we can note that balancing the computation load between the processors, in general, is level enough. What is more, in order to assess the parallel program, we compared the theoretical and practical acceleration of the algorithms (this data is not provided in the article). According to our calculations, theoretical assessment corroborates the data obtained after using the software.

Using the parallel version of the algorithm for hybrid systems provides a significant advantage over the sequential one. However, the branches of the parallel realizations for a number of nodes within the range from 2 to 32 do not differ significantly. For this reason, using more than 2 threads on each processor does not result in better performance of the algorithm and decrease in time. It is also reflected in the relatively lower efficiency and acceleration values of parallel implementation for hybrid systems.

References

[1] Saad Y 2003 *Iterative Methods for Sparse Linear Systems* (Minnesota: University of Minnesota Minneapolis)

[2] Antonov A S, Frolov A V, Kobayashi H, Konshin I N, Teplov A M and Voevodin V V 2016 Parallel processing model for cholesky decomposition algorithm in AlgoWiki project *Supercomputing Frontiers and Innovations* 3(3) 61-70

[3] Quinn M J 2008 *Parallel Programming in C with MPI and OpenMP* (New York: McGraw-Hill Education)

[4] Antonov A S, Voevodin V V and Popova N N 2019 Parallel structure of algorithms and training computational technology specialists *J. of Phys.: Conf. Ser.* 1202(1) 12021

[5] Perminov V A, Rein T S and Karabtcev S N 2015 NEM and MFEM Simulation of Interaction between Time-dependent Waves and Obstacles. *IOP Conf. Ser: Materials Science and Engineering* 81(1) 012099

[6] Pissanetzky S 1984 *Sparse Matrix Technology* (London: Academic Press Inc.)

[7] Karniadakis G and Kirby R 2003 *Parallel Scientific Computing in C++ and MPI* (Cambridge: Cambridge university press)