

Implementation of plasma stability code on a Cell cluster system for constant plasma monitoring

Noriyuki Kushida, and Hiroshi Takemiya

Abstract—We developed a high-speed eigenvalue solver and a matrix generation code that are essential parts of a plasma stability analysis system for nuclear fusion reactors on a Cell cluster system. In order to achieve sustainable operation of the reactors, we must evaluate the state of plasma within the characteristic confinement time of the plasma density and temperature. This is because we can prevent plasma from being disrupted by controlling the confining magnetic field, if we can determine the state of the plasma within the characteristic confinement time. Therefore, we introduced a Cell that has high computational power and high performance/cost, to achieve constant monitoring. Furthermore, we developed a novel eigenvalue solver, which consumes most of the plasma evaluation time. The eigensolver is based on the conjugate gradient (CG) method and was designed by considering hierarchical parallelism of Cell. Moreover, we developed a new CG acceleration method, called locally complete LU that has both better acceleration and parallel performance than the formers. Finally, we succeeded in obtaining our target performance: we were able to create and solve a block tri-diagonal Hermitian matrix containing 1024 diagonal blocks, where the size of each block was 128×128 , within two seconds. Therefore, we have found a suitable candidate for achieving a satisfactory monitoring system.

Keywords—PowerXCell8i, Cell cluster, Block tri-diagonal matrices, Eigensolver, Preconditioned Conjugate Gradient Method, Parallel Computing, Plasma stability analysis, Sustainable plasma operation.

I. INTRODUCTION

IN this study, we developed a high-speed eigenvalue solver and a matrix generation code on a Cell cluster system, that are essential components of a plasma stability analysis system for nuclear fusion reactors like International Thermonuclear Experimental Reactor (ITER)[1]. The Japan Atomic Energy Agency (JAEA) has been developing a plasma stability analysis system, in order to achieve sustainable operation. In Fig. 1, we illustrate an overview of the plasma stability analysis system. The plasma stability analysis system works as follows:

- (1) Obtain the current magnetic field status of the exterior of the reactor.
- (2) Analyze the plasma state using numerical simulation.
- (3) Judge the state of the plasma (Plasma is stable/unstable, when the smallest eigenvalue λ is greater/smaller than zero).
- (4) If the plasma is unstable, the operating conditions are changed, in order to stabilize the plasma.

The main component of the plasma stability analysis system is the plasma simulation program MARG2D [2]. MARG2D consists of roughly two parts: One is the matrix

N. Kushida and H. Takemiya are with the Center for Computational Science and E-systems, Japan Atomic Energy Agency, Ibaraki, Japan e-mail: {kushida.noriyuki,takemiya.hiroshi}@jaea.go.jp

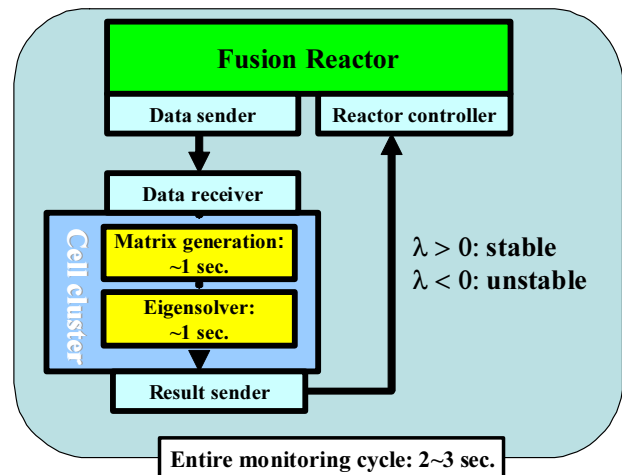


Fig. 1. Illustrative overview of the plasma stability analysis system

generation part; the other is the eigensolver. In order to achieve sustainable operation of nuclear fusion reactors, we must evaluate the state of the plasma every two to three seconds. This is because the characteristic confinement time of the density and temperature in nuclear fusion reactors is from three to five seconds[3], and we can prevent plasma from disruption by controlling the confining magnetic field. Moreover, we estimated that we must determine the state of the plasma within half of the characteristic confinement time, by taking into account the time for data transfer, and other such activities. Since we must solve for the plasma state within a quite short time interval, a high-speed computer is essential.

A massively parallel supercomputer (MPP), which obtains its high calculation speed by connecting many processing units and is the current trend for heavy duty computation, is inadequate for following two reasons.

- (1) We cannot dedicate MPPs to the monitoring system.
- (2) MPPs have a network communication overhead.

We elaborate on the above two points. Firstly, with regard to the first point, when we consider developing the plasma monitoring system, we are required to utilize a computer during the entire nuclear fusion reactors operation. That is because nuclear fusion reactors must be monitored continuously and without delay. For this reason MPPs are inadequate because they are usually shared with a batch job system. Furthermore, using an MPP is unrealistic, because of its high price. Therefore, MPPs could not be dedicated to such a monitoring system. Next, we discuss the latter point. MPPs consist of

many processing units that are connected via a network. The data transfer performance of a network is lower than that of main memory. In addition, there are several overheads that are ascribable to introducing a network, such as the time to synchronize processors, and the time to call communication functions. These overheads are typically from $O(n)$ to $O(n^2)$, where n is the number of processors. Even though the overheads can be substantial with a large number of processors, they are usually negligible for large-scale computing, because the net computational time is quite long[4]. However, the monitoring system is required to terminate within such a short period that network overheads can be dominant. Moreover, the entire time for computation can be longer when the number of processors increases. Thus, we cannot utilize MPPs for the monitoring system.

In order to deal with the above difficulties, we introduced a Cell cluster system into this study. A cell processor is faster than a traditional processor, hence we could obtain sufficient computational power with a small number of processors. Thus, we were able to establish the Cell cluster system at much cheaper cost, and we can dedicate it to monitoring. Moreover, our Cell cluster system requires less network overhead. Therefore, it should be suitable for the monitoring system.

The Cell processor obtains its greater computational power at the cost of more complex programming. Therefore, we also introduce our newly developed eigensolver in the present paper. The details of our Cell cluster system and the eigenvalue solver, are described in the following sections (Sections II and III). The performance is evaluated in Section IV. We also show how the matrix generation part is ported on our Cell cluster system in Section V. The entire performance of our code by using a practical condition is shown in Section VI. Finally, we make conclusions in Section VII.

II. CELL CLUSTER

A. PowerXCell 8i

PowerXCell 8i, which has a faster double precision computational unit than the original version, is a kind of Cell processor[5]. An overview of PowerXCell 8i is shown in Fig. 2. In the figure, PPE denotes a Power PC Processor Element. The PPE has a PPU that is a processing unit equivalent to a Power PC, and also includes a second level cache memory. SPE denotes a Synergetic Processor Element, which consists of a 128 bit single instruction multiple data processing unit (hereinafter referred to as SIMD), In earlier studies [6], the processing unit was called an SPU. together with a local store (LS) and a memory flow controller (MFC), which handles data transfer between LS and main memory. The PPE, SPE, and main memory are connected with an Element Interconnect Bus (EIB). EIB has four buses and its total bandwidth reaches 204.8 Gigabytes per second. Note that the total bandwidth of EIB includes not only the data transfer between the processing unit and the main memory but also data transfer among processing units. Therefore, we usually consider the practical bandwidth of PowerXCell 8i to be 25.6 Gigabytes per second, which is the maximum access speed of main memory.

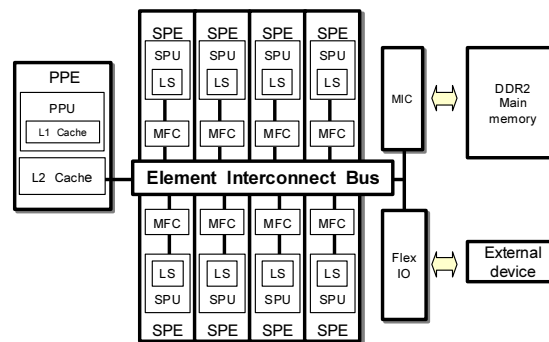


Fig. 2. Overview of the PowerXCell 8i processor

B. Clustering of QS22

For this study, we constructed a Cell cluster system using QS22[7] blades, (developed by IBM), together with the Mpich2 library[8]. We illustrate an overview of our Cell cluster system in Fig. 3. QS22 contains two Cell processors and both can access a common memory space; thus in total, sixteen SPEs are available in one QS22 blade. In addition, two QS22s are connected by a gigabit Ethernet. The Message passing interface (MPI) specification is the standard for the communication interface for distributed memory parallel computing and the Mpich2 library is one of the most well known implementations of MPI on commodity off the shelf clusters. Originally, the MPI specification was developed for a computer system with one processing unit and one main storage unit. This model is simple but not suitable for a Cell processor, because the SPEs have their own memory and therefore do not recognize a change of data in main memory. Thus, we combined two kinds of parallelization; the first is parallelization among blades using Mpich2, and the second is parallelization among SPEs. We observe, however, that a PPE only communicates to other blades using Mpich2 and SPEs do not relate to communication. Moreover, the SIMD processing unit of SPE itself is a kind of parallel processor. Then we must consider three levels of parallelization, in order to obtain better performance of the Cell cluster:

- (1) MPI parallel
- (2) SPE parallel
- (3) SIMD parallel

III. EIGENSOLVER

Although there are numerous eigenvalue solver algorithms, only two are suitable for our purposes, because only the smallest eigenvalue is required for our plasma stability analysis system. One candidate is the Inverse power method, and the other is The conjugate gradient method (hereafter referred to as CG). The inverse power method is quite simple and easy to implement; however, it requires solving the linear equation at every iteration step, which is usually expensive in terms of time and memory. It is fortunate that the computational cost of lower/upper (LU) factorization and backward/forward (BF) substitution of block tri-diagonal matrices is linear of order n .

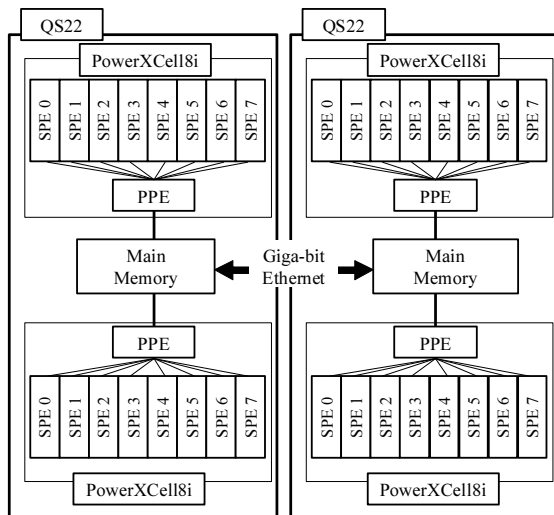


Fig. 3. Overview of our Cell cluster system

However, this is just for the sequential case. We are forced to incur additional computational cost with parallel computing, especially for MPI parallel. According to the articles[9][10], the computational cost of LU factorization increases with a small number of processors and is at least twice as great as the sequential computational cost. In our estimation, such an inflation of computational cost was not acceptable for our system. On the other hand, CG is basically well suited to distributed parallel computing, in that the computational cost for one processor linearly decreases as the number of processors that are actually used, increases. For these reasons, we employ CG as the eigenvalue solver. Details of the conjugate gradient method, including parallelization and the convergence acceleration technique that we developed are described in the following sections.

A. Preconditioned Conjugate Gradient

CG is an optimization method used to minimize the value of a function. If the function is given by

$$\lambda(\mathbf{x}) = \frac{(\mathbf{x}, A\mathbf{x})}{(\mathbf{x}, \mathbf{x})}. \quad (1)$$

the minimum value of $\lambda(\mathbf{x})$ corresponds to the minimum eigenvalue of the standard eigensystem $A\mathbf{x} = \lambda\mathbf{x}$, and the vector \mathbf{x} is an eigenvector associated with the minimum eigenvalue. Here (\cdot, \cdot) denotes the inner product. The CG algorithm, which was originally developed by Knyazev[11] and Yamada and others showed more concrete algorithm in their literature[12],[13] (as shown in Algorithm 1). In the figure, T denotes the preconditioning matrix; the details are introduced later. Several variants of the conjugate gradient algorithm have been developed and have been tested for stability. According to the literature, Knyazev's algorithm achieved quite good stability by employing Ritz method, expressed as the eigenproblem for $S_A\mathbf{v} = \mu S_B\mathbf{v}$, in the algorithm. Yamada's algorithm is equivalent to Knyazev's algorithm; however, it requires only one matrix-vector multiplication, which is one of the most time

consuming steps of the algorithm, whereas Knyazev's original algorithm seems require three such multiplications. Therefore, in the present study, we employ Yamada's algorithm. Let us consider the preconditioning matrix T . The basic idea of preconditioning is to transform the coefficient matrix close to the identity matrix by operating by an inverse of T that approximates the coefficient matrix A in some sense. Even if a higher degree of approximation of T to A provides a higher convergence rate for CG, we usually stop short of achieving $T = A$, because the computational effort can be extremely expensive. Additionally, an inverse of T is not constructed explicitly because the computational effort can also be large. Although the matrix T^{-1} appears in the algorithms, the algorithm only requires solving the linear equation. We usually employ triangular matrices, or some multiples thereof, for T , because we can such a system with Backward/Forward(BF) substitutions. It is fortunate that complete LU factorization for block tri-diagonal matrices can be obtained at reasonable computational cost; we employed complete LU factorization to construct the preconditioning matrix T .

Algorithm 1 Algorithm of the conjugate gradient method as introduced by Yamada et al.

1. Let \mathbf{x}_0 be the initial guess, and $\mathbf{p}_0 := 0$
2. $\mathbf{x}_0 := \mathbf{x}_0 / \|\mathbf{x}_0\|$
3. $\mathbf{X}_0 := A\mathbf{x}_0$
4. $\mu_{-1} := (\mathbf{x}_0, \mathbf{X}_0)$
5. $\mathbf{W}_0 := \mathbf{X}_0 - \mu_{-1}\mathbf{x}_0$
6. **for** $i = 0, 1, 2, 3, \dots$, until convergence **do**
7. $\mathbf{W}_k := A\mathbf{w}_k$
8. $S_A := \{\mathbf{w}_k, \mathbf{x}_k, \mathbf{p}_k\}^T \{\mathbf{W}_k, \mathbf{X}_k, \mathbf{P}_k\}$
9. $S_B := \{\mathbf{w}_k, \mathbf{x}_k, \mathbf{p}_k\}^T \{\mathbf{w}_k, \mathbf{x}_k, \mathbf{p}_k\}$
10. Find the smallest eigenvalue μ and corresponding eigenvector \mathbf{v} of $S_A\mathbf{v} = \mu S_B\mathbf{v}$, $\mathbf{v} = \{v_1, v_2, v_3\}$
11. $\mu_k := (\mu + (\mathbf{x}_k + \mathbf{X}_k)) / 2$
12. $\mathbf{x}_{k+1} := v_1\mathbf{w}_k + v_2\mathbf{x}_k + v_3\mathbf{X}_k$
13. $\mathbf{x}_{k+1} := \mathbf{x}_{k+1} / \|\mathbf{x}_{k+1}\|$
14. $\mathbf{p}_{k+1} := v_1\mathbf{w}_k + v_3\mathbf{p}_k$
15. $\mathbf{p}_{k+1} := \mathbf{p}_{k+1} / \|\mathbf{p}_{k+1}\|$
16. $\mathbf{X}_{k+1} := v_1\mathbf{W}_k + v_2\mathbf{X}_k + v_3\mathbf{P}_k$
17. $\mathbf{X}_{k+1} := \mathbf{X}_{k+1} / \|\mathbf{X}_{k+1}\|$
18. $\mathbf{P}_{k+1} := v_1\mathbf{W}_k + v_3\mathbf{P}_k$
19. $\mathbf{P}_{k+1} := \mathbf{P}_{k+1} / \|\mathbf{P}_{k+1}\|$
20. $\mathbf{w}_k := T^{-1}(\mathbf{X}_{k+1} - \mu_k\mathbf{x}_{k+1})$
21. $\mathbf{w}_{k+1} := \mathbf{w}_{k+1} / \|\mathbf{w}_{k+1}\|$
22. **end for**

B. Parallelization of the conjugate gradient method

Almost the entire algorithm of CG consists of the following three operations:

- (1) Matrix – vector multiplication
- (2) Vector dot product
- (3) Scalar – vector multiplication

Note that the normalization of vectors can be carried out using (2) and (3). Preconditioning is not discussed in this section but is discussed later. As mentioned previously, three levels of parallelization have to be considered for the Cell cluster. These three kinds of parallelization for these three operations are individually discussed in the following sub-sections.

1) *MPI Parallel*: In this sub-section, the parallelization between two QS22 blades is considered. In the present study, this parallelization was achieved using MPI, therefore, we refer to this situation as MPI parallel. By considering the three operations, we assigned the memory and vectors as shown in Fig. 4. We assumed that the matrix has 6×6 blocks. The big square which is shown on the left of the figure is the matrix A and the two rectangles on the right are the vector x . The hatched small squares are non-zero blocks and the white blocks are complete zero blocks. The non-zero blocks have a location indicator. The blocks for matrices are $m \times m$ and for vectors are $1 \times m$. Firstly, we consider matrix – vector multiplication. Since the matrix is used for matrix – vector multiplication, we divided the matrix by rows; namely, the 1st to the 3rd rows of the matrix are stored in Blade 1, and the 4th to the 6th rows are stored in Blade 2. This is also the same for the vector, (1st to 3rd components are in Blade 1 and 4th to 6th in Blade 2), but extra storage area is allocated in order to carry out the matrix – vector operation. The reason for extra storage can be simply explained. When we consider the multiplication of A by x , all the calculations except those for $A(3, 4)$ can be done with $x(1)$, $x(2)$ and $x(3)$ on Blade 1, but then will never terminate if $x(4)$ is absent. Therefore, $x(4)$ must be sent from Blade 2 to Blade 1 before the multiplication. In addition, the same situation occurs for Blade 2. Secondly, we consider the vector dot product. The way to compute it is quite simple; we calculate partial sums locally on blades and exchange them with each other. More precisely, considering vectors x and y , the total product S_{total} can be written as

$$S_{total} = S_{Blade1} + S_{Blade2},$$

where

$$S_{Blade1} = \sum_{i=1}^3 x(i)y(i), S_{Blade2} = \sum_{i=4}^6 x(i)y(i).$$

It is obvious that both S_{Blade1} and S_{Blade2} can be computed on the local blade, and the total can be obtained just by exchanging S_{Blade1} and S_{Blade2} . Scalar – vector multiplication can be performed with no communication.

C. SPE parallel

The QS22 blade has 16 SPEs and each SPE can run in parallel with each other. Originally, each SPE could contain its own instruction set; however, we consider them to have the same instruction set in the present study. In other words, we employ single instruction multiple data parallelization among SPEs. Firstly, we consider matrix – vector multiplication. The computation of matrix – vector multiplication is performed in a block wise manner. In detail, when $y := Ax$, is computed, each SPE computes $y(i) := y(i) + A(i, j) * x(j)$ as an unit.

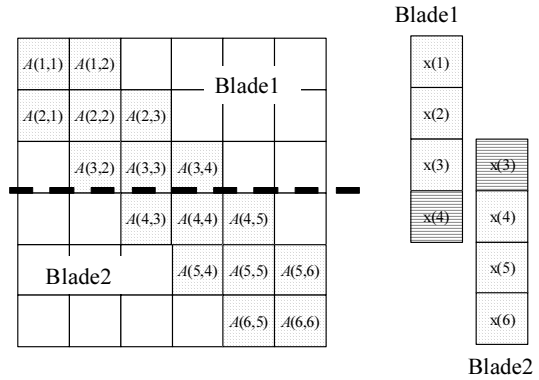


Fig. 4. Memory configuration of matrix and vector on our Cell cluster

We observe that this computation can be done in parallel with different i . Next we consider the vector dot product. The strategy is quite similar to MPI parallel; however, the parallelization is based on SPE. Each SPE computes part of the product and PPE sums the results. Moreover, scalar – vector multiplication can be done just by subdividing the range of computation.

D. SIMD parallel

The SIMD processor is the processing unit that computes two or more floating point values at the same time and SIMD parallel is the smallest parallelization unit in the present study. The SIMD processor provides better computational performance than the traditional processing unit; however, there are two big hindrances to realizing peak performance. One is that the SIMD processor always computes two floating-point values; otherwise we must incur a huge penalty [6]. In other words, we have to consume more time to complete one floating-point calculation than to make two floating-point calculations. In order to explain the penalty, we consider the quite simple operation $c := a + b$. SPE needs around 60 clock cycles (In this case, data load and store operations are included). However, if we apply a technique which enables us to use the SIMD processor, we obtain a result within 30 clock cycles. Therefore, we incur twice as much time cost for one floating-point calculation as for an SIMD calculation.

The other point is that two floating point values that are processed must be arranged in contiguous address spaces. In order to avoid the penalty, we added extra zeros (zero padding) when we cannot compute two floating point values. In Fig. 5, matrix – vector multiplication is used as an example of zero padding. In the figure, the multiplication of a 3×3 block matrix and the corresponding vector is considered. In this study, the block matrix is stored in a row oriented form (the addresses of $C(1, 1)$ and $C(1, 2)$ are contiguous), because the performance of LU factorization of the block matrix, which is required for preconditioning and which dominates the total computational time, was better than using storage in a column oriented form, in our preliminary test. Furthermore, the performance of the entire matrix – vector multiplication is comparable between

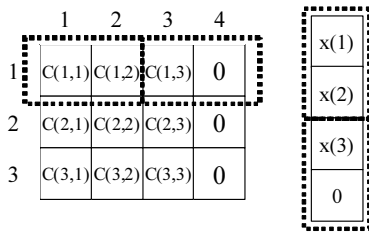


Fig. 5. Memory configuration for SIMD processing

row and column oriented versions. Because the matrix is stored in a row oriented form, the matrix – vector multiplication can be computed as an iteration of vector dot products. Now, consider the computation of the first row and vector \mathbf{x} . The matrix originally has three components in a row, therefore, The first two components can be computed without difficulty. On the other hand, the third component is by itself and extra zeros should be added just after the third component. This zero padding should be applied to the vector not only in the case of matrix – vector multiplication but also for the vector dot product and scalar – vector multiplication. This technique for the usage of the SIMD processor, as described here, could also be used for block-wise matrix – matrix multiplication, LU factorization, and BF substitution, all of which are required for LU factorization of the block tri-diagonal matrix.

E. Complex value handling on the SIMD processor

Usually the real part and the imaginary part of a complex value are stored contiguous address space in traditional programming languages like Fortran. This way is easily comprehensible to human, but is not suitable for the SIMD computation. This is because the SIMD processor applies a same operation to a set of two floating-point values, and therefore, some kinds of complex value operations cannot be performed e.g. we cannot do multiplication. In this study, we employ a structure of array (SoA) strategy, which stores a complex matrix with two arrays i.e. one array is for real part and the other is for imaginary part. By employing SoA strategy, we have to code complex value operations by not compilers but ourselves, but can apply the SIMD parallelism.

F. LU factorization of block tri-diagonal matrices

In this section, parallelization for the LU factorization on one Cell processor is described. Firstly, we introduce the algorithm for sequential block tri-diagonal LU factorization. We consider the linear equation $A\mathbf{x} = \mathbf{b}$, with coefficient matrix

$$A = \begin{bmatrix} A_1 & C_1 & & & \\ B_2 & A_2 & \ddots & & \\ & \ddots & \ddots & C_{n-1} & \\ & & & B_n & A_n \end{bmatrix}$$

Here, A , B , and C are m by m block matrices. Then, nm is the overall dimension of the entire coefficient matrix. We

assume that A can be factorized into L and U , where L is lower triangular and U is upper triangular, respectively. The components of L and U can be expressed as follows.

$$L = \begin{bmatrix} \bar{A}_1 & & & & \\ B_2 & \bar{A}_2 & & & \\ & \ddots & \ddots & & \\ & & & B_n & \bar{A}_n \end{bmatrix},$$

$$U = \begin{bmatrix} I & \bar{C}_1 & & & \\ & I & \ddots & & \\ & & \ddots & \bar{C}_{n-1} & \\ & & & & I \end{bmatrix},$$

where I is the identity matrix. Block matrices with bars are calculated using the following equations.

$$\begin{aligned} \bar{A}_1 &= A \\ \bar{C}_1 &= \bar{A}_1^{-1} C \\ \bar{A}_i &= A_i - B_i \bar{C}_{i-1} \quad (i = 2, \dots, n-1) \\ \bar{C}_i &= \bar{A}_i^{-1} C \quad (i = 2, \dots, n-1) \\ \bar{A}_n &= A_n - B_n \bar{C}_{n-1} \end{aligned}$$

The solution step with BF substitution can be expressed as follows:

$$\begin{aligned} \mathbf{z}_1 &= \bar{A}_1^{-1} \mathbf{y} \\ \mathbf{z}_i &= \bar{A}_i^{-1} (\mathbf{y}_i - B_i \mathbf{z}_{i-1}) \quad (i = 2, \dots, n) \\ \mathbf{x}_n &= \mathbf{z}_n \\ \mathbf{x}_i &= \mathbf{z}_i - \bar{C}_i \mathbf{x}_{i+1} \quad (i = n-1, \dots, 1) \end{aligned}$$

We observe that the algorithms are inherently sequential with respect to the subscript i and that therefore only one SPE can calculate at a time, whereas QS22 has sixteen SPEs. To achieve parallel computation, the cyclic reduction method[14] was employed in this paper. The idea of the cyclic reduction method is to create a group of blocks whose members can be calculated at the same time. In order to create the group, we transform $A\mathbf{x} = \mathbf{b}$ using an orthogonal matrix P as $PAP^T P\mathbf{x} = P\mathbf{b}$. We assume that the transformed matrix PAP^T and its LU factor matrices can be expressed as follows.

$$PAP^T = \begin{bmatrix} A_1 & & & & C_1 & & & & \\ & A_3 & & & B_3 & C_3 & & & \\ & & A_5 & & & B_5 & C_5 & & \\ & & & A_7 & & & B_7 & C_7 & \\ B_2 & C_2 & & & A_2 & & & & \\ & B_4 & C_4 & & & A_4 & & & \\ & & B_6 & C_6 & & & A_6 & & \\ & & & B_8 & & & & A_8 & \end{bmatrix},$$

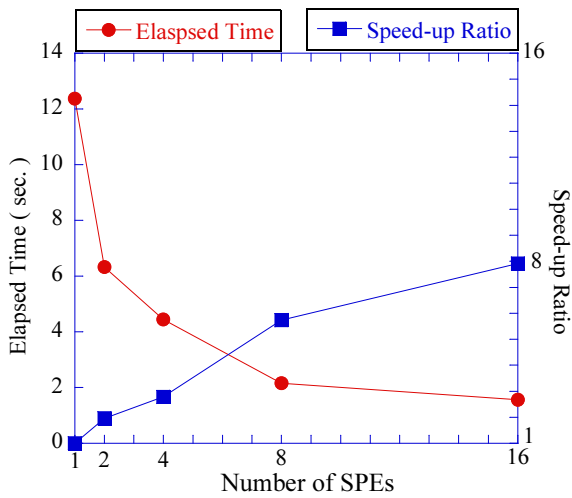


Fig. 6. Computation time to CG convergence and speed-up ratio for various numbers of SPEs

B. MPI parallel

In Fig. 7, the elapsed time and the speed-up ratio for several numbers of QS22s are shown. The elapsed time decreases when the number of QS22s increasing; thus we conclude that our intra-QS22 parallel implementation is successful. Moreover, the speed-up ratio is close to ideal; i.e., we can obtain more speed by adding extra QS22s.

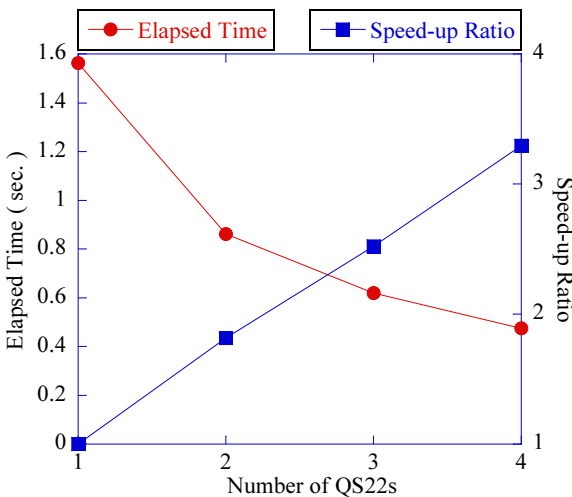


Fig. 7. Computation time to CG convergence and speed-up ratio for various numbers of QS22s

C. Effect of localization

We show the convergence histories of CG, Complete LU preconditioned CG, and Locally complete LU preconditioned CG, in Fig. 8. In this section, locally complete LU preconditioned CG is performed with two QS22s, and the others are performed with one QS22. No obvious difference between complete LU and locally complete LU can be observed.

Moreover, preconditioning accelerates convergence, in fact, making it occur four times as fast as normal.

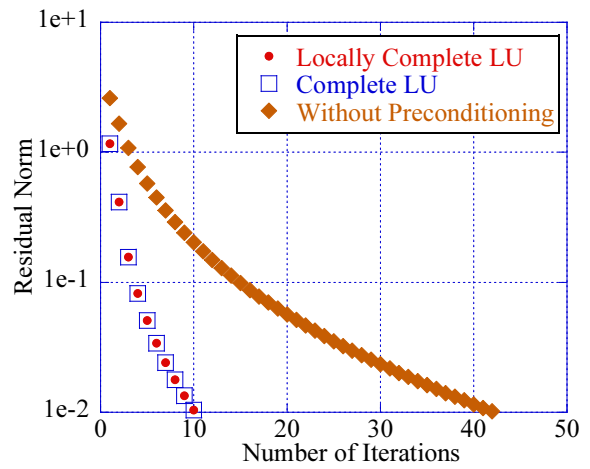


Fig. 8. Convergence history of several CG methods

D. Precision of the estimated eigenvalue

For the purpose of plasma monitoring, the time available for eigenvalue solving is one second. In this section, we discuss the precision of the estimated eigenvalue that can be obtained within one second. In table I, the numbers of iterations which can be performed within one second and the error of the resulting minimum eigenvalue are tabulated. In the table, we solved same problem by changing the number of QS22s. The error is defined as $|\lambda_{true} - \lambda_{estimated}|$. As can be observed in Fig. 8, the convergence behavior of our solver is smooth. Therefore, we can obtain a more precise result if we use more computing power. In the present study, we can compute around 5 more iterations when we add one extra QS22. Moreover, the estimated eigenvalue becomes more precise when we add QS22s. Finally, the error was reduced to $O(10^{-6})$, when we use four QS22s.

TABLE I
NUMBER OF ITERATIONS AND ERROR OF THE ESTIMATED EIGENVALUE OBTAINED WITHIN ONE SECOND.

# of QS22s	# of iterations	Error
1	1	1.71E-03
2	6	4.60E-05
3	10	2.31E-05
4	15	7.54E-06

V. MATRIX GENERATION

When we consider how to make matrices of MARG2D, we have to take account of following two aspects.

- (1) Since we must avoid additional computational time, even if only slightly, we construct matrices as our eigensolver uses.
- (2) How to parallelize.

In this section, we elaborate above aspects and show performance.

A. MPI parallel

As shown in Fig. 4, the entire matrix is divided row-wise manner for MPI parallel. It is fortunate that there is no dependency between upper and lower part of the matrix except the connection part (In the figure, A(3,3), A(3,4), A(4,3), and A(4,4)). These blocks are incomplete only with local data of each PE, therefore, they exchange data with each other.

B. SPE parallel

MARG2D employs combined method of one dimensional finite element method (FEM) and spectral method. The tri-diagonal structure of the matrices originates from FEM and block structure originates from spectral method. SPE parallel should be applied for FEM. However, there are dependencies. In Fig. 8, the construction process of the entire matrix is illustrated. In the figure, small squares show the blocks of the matrix and large squares show the element matrices. When we construct entire matrices, calculate element-wise matrices and then superpose them to the entire. Therefore, if we calculate element-wise matrices at the same time, there are conflicts. In the figure (a), hatched square is a conflicted block between Element 1 and Element 2. In order to avoid dependencies, red-black ordering method was employed. That is, we make two groups that the elements in one group have no conflicts. This idea is explained using Fig. 9 (b) and (c). In this figure, only lower part is used because of simplicity. Elements in “Red” group (red squares) are shown in (b) and they do not share any blocks. Therefore, we can apply parallel computation. On the other hand, Elements in “Black” group (blue squares) have no conflict but they share some blocks with Elements in “Red” group (shared blocks are purple squares). However, the calculation of “Black” group is already done, and the resulted entire matrix is sound.

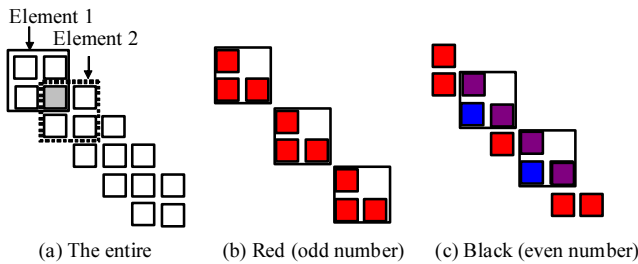


Fig. 9. Construction process of the entire matrix.

C. Hotspot

According to our preliminary calculation time analysis on an Intel Xeon processor, the following three block-wise matrix operations require the longest calculation time in the matrix generation part: (1) Matrix–matrix multiplication (80% of the total), (2) Matrix inversion (5% of the total), and (3) LU factorization (2% of the total). It is fortunate that these three operations also appear in the eigenvalue solver, and therefore, we can reuse them.

D. Parallel performance of matrix generation

In order to investigate the performance of matrix generation, we show the parallel performance with respect to the number of SPEs within one Cell processor (Fig. 10) and the number of QS22s (Fig. 11). Since the total number of SPEs in one Cell processor is 16, we measured calculation time by changing the number of SPEs from one to sixteen. As shown, we always obtained speedup and therefore our implementation works well on our Cell cluster system. Finally, we could complete matrix generation within one second when we use four QS22s.

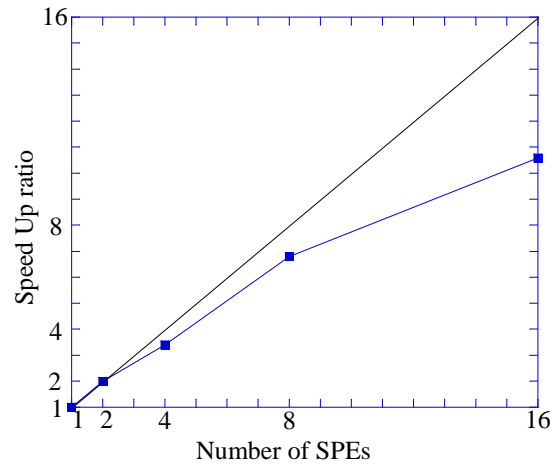


Fig. 10. Parallel performance with respect to the number of SPEs within one QS22.

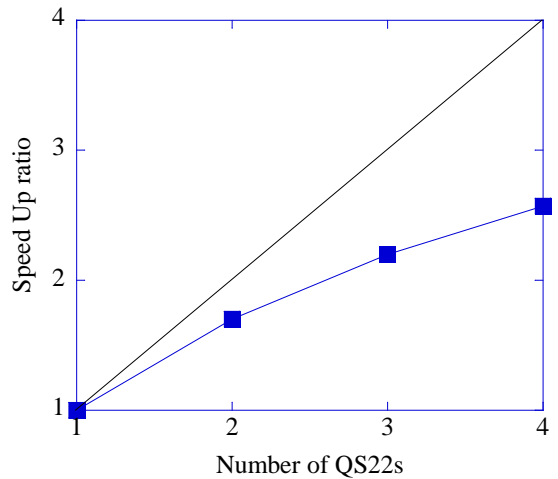


Fig. 11. Parallel performance with respect to the number of QS22s.

VI. PRACTICAL PROBLEM

In this section, we evaluate the performance of our optimized plasma stability analysis code by using a practical condition, which have been used to check the performance of stability analysis codes in JAEA. In this section, the resulted matrix has 1024 diagonal blocks and each block is 100 × 100.

The calculation is carried out on one QS22, and 16 SPEs are used. In order to confirm the effectiveness of LU factorization preconditioning, the convergence history of normal CG (No preconditioning in the figure), and LU preconditioned CG (Full LU in the figure) are compared in Figure 12. In the figure, the convergence histories of each method up to 100 iterations are shown, while we stopped the calculations when the number of iterations reached 10,000. Additionally, the convergence history of block diagonal preconditioned CG (Block diagonal in the figure) is also shown in the figure. Block diagonal preconditioner takes into consideration only diagonal blocks for preconditioning. Consequently, block diagonal preconditioned CG converges faster than normal CG, but slower than LU preconditioned CG. LU preconditioned CG converged with six iterations, while normal CG and block diagonal preconditioned CG did not converge within 10,000 iterations. Based on the result, only LU preconditioned CG is acceptable for our purpose. The time to complete matrix generation was 2.09 seconds for every case. The time to complete for LU preconditioned CG was 0.89 seconds, and normal CG and block diagonal preconditioned CG did not converge after five minutes computation. Therefore, we could complete the entire computation within around 3 seconds. This result shows that our code has sufficient performance.

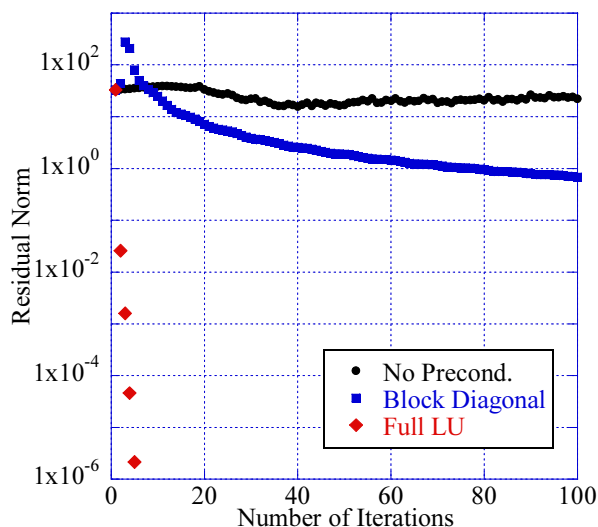


Fig. 12. Convergence history of PCG with several preconditioning on practical plasma state.

VII. CONCLUSIONS

In the present paper, we introduced a high-speed eigenvalue solution and matrix generation system that is required for the plasma stability analysis system of nuclear fusion reactors. We constructed the Cell cluster system using QS22 blades and the Mpich2 library and developed the plasma stability code taking into consideration the architecture of Cell processor. The eigensolver was based on the CG, and locally complete LU factorization was employed for preconditioning. By considering the architecture of the Cell cluster system, we

developed three levels of parallelization: MPI parallel, SPE parallel, and SIMD parallel. By use of these techniques, we achieved solution of the target eigensystem within the target time; we succeeded to solving a system with 1024 diagonal blocks where each block was 128×128 , within one second. We also complete matrix generation part within one second on our Cell cluster system. The performance of the entire plasma stability analysis code is evaluated by using a practical condition, and the result shows that our approach has sufficient performance for plasma monitoring.

ACKNOWLEDGMENT

We are grateful to JSPS for the Grant-in-Aid for Young Scientists (B), No. 21760701. We also thank A. Tomita and K. Fujibayashi @ FIXSTARS Co. for their extensive knowledge of the CELL.

REFERENCES

- [1] ITER project web page, available at <http://www.iter.org/default.aspx>
- [2] S. Tokuda and T. Watanabe, "A new eigenvalue problem associated with the two-dimensional Newcomb equation without continuous spectra", *Physics of Plasmas*, vol. 6, 3012–3026, (1999).
- [3] S. Tokuda, "Development of Eigenvalue Solver for an MHD Stability Analysis Code", 5th Burning Plasma Simulation Initiative Annual Meeting, (2006).
- [4] N. KUSHIDA and H. OKUDA, "Optimization of the Parallel Finite Element Method for the Earth Simulator", *Journal of Computational Science and Technology*, vol.2, No. 1, 81–90, (2008).
- [5] Cell Broadband Engine processor based systems White Paper, available at http://www.irisa.fr/orap/Constructeurs/Cell/cell_be_systems_whitepaper.pdf
- [6] M. Scarpino, "Programming the Cell processor for Games, Graphics, and Computation", Pearson Education, (2009).
- [7] Product information of QS22, available at <http://www-03.ibm.com/systems/info/bladecenter/qs22/>
- [8] MPICH2 web page: <http://www.mcs.anl.gov/research/projects/mpich2/>
- [9] A. van der Ploeg, "Reordering strategies and LU-decomposition of block tridiagonal matrices for parallel processing", technical report NM-R9618, CWI, October, (1996).
- [10] A. van der Ploeg, "Parallelization of a block tridiagonal solver in HPF on an IBM sp2", *Lecture Notes in Computer Science*, vol 1401, pp 242–251, (1998).
- [11] A. V. Knyazev, "Toward the optimal eigensolver: Locally optimal block preconditioned conjugate gradient method", *SIAM J. Sci. Comput.*, 23, 517–541, (2001).
- [12] S. Yamada, T. Imamura, and M. Machida, "Preconditioned Conjugate Gradient Method for Large-scale Eigenvalue Problem of Quantum Problem", *Transactions of JSCES*, Vol. 2006, 20060027, 2006. [In Japanese]
- [13] S. Yamada, T. Imamura, and M. Machida, "16.477 TFLOPS and 159-Billion-dimensional Exact-diagonalization for Trapped Fermion-Hubbard Model on the Earth Simulator", *Proc. of SC2005*, (2005).
- [14] G. Hime, B. Schulze, and D. Marchesin, "Cluster solution of block tridiagonal systems", *Proceedings of CCGrid2007*, URL:<http://ccgrid07.lncc.br/posters.php>, (2007).
- [15] N. Kushida, H. Takemiya, "Optimization of Finite Element Method for the Cell processor" *Transactions of JSCES*, Vol. 2010, 20100002, (2010). [In Japanese]