High-Speed Computation of the Kleene Star in Max-Plus Algebraic System Using a Cell Broadband Engine

Hiroyuki GOTO†a), Member

SUMMARY  This research addresses a high-speed computation method for the Kleene star of the weighted adjacency matrix in a max-plus algebraic system. We focus on systems whose precedence constraints are represented by a directed acyclic graph and implement it on a Cell Broadband Engine™ (CBE) processor. Since the resulting matrix gives the longest travel times between two adjacent nodes, it is often utilized in scheduling problem solvers for a class of discrete event systems. This research, in particular, attempts to achieve a speedup by using two approaches: parallelization and SIMDization (Single Instruction, Multiple Data), both of which can be accomplished by a CBE processor. The former refers to a parallel computation using multiple cores, while the latter is a method whereby multiple elements are computed by a single instruction. Using the implementation on a Sony PlayStation 3™ equipped with a CBE processor, we found that the SIMDization is effective regardless of the system’s size and the number of processor cores used. We also found that the scalability of using multiple cores is remarkable especially for systems with a large number of nodes. In a numerical experiment where the number of nodes is 2000, we achieved a speedup of 20 times compared with the method without the above techniques.

key words: directed acyclic graph, max-plus algebra, scheduling, parallel processing, vector processing, Cell Broadband Engine

1. Introduction

This research aims to develop a fast computation method for the representation matrix in a state-space representation in max-plus algebra [1], [2]. This approach is often used in solvers for a class of scheduling problems. Specifically, we focus on quickly calculating the transition matrix, which requires a high computation load using this approach. For the computation, we use a processor called the Cell Broadband Engine (CBE) [3]–[6], which has instructions for vector operations and parallel processing.

The features of the class of focused systems are: (1) parallel execution of multiple tasks, (2) synchronization of multiple tasks, and (3) no-concurrency between the previous or succeeding job. Such systems can also be explained as repetitive discrete event systems in which a number of jobs are processed sequentially using the same facilities in FIFO order. We assume that the precedence relationships of the system are described by a Directed Acyclic Graph (DAG). The behavior of this class can be represented by a pair of simple equations in max-plus algebra, called the state-space representation. Since this is similar to the representation in modern control theory, a number of research developments in modern control theory have been applied to various kinds of scheduling problems [7]–[9].

The state-space representation consists of a pair of state and output equations. The bottleneck of computing the equations is in the transition matrix of the state equation. The matrix is obtained by applying the Kleene star [1] operator to the adjacency matrix that describes the precedence relationships of the system. Essentially, this operation corresponds to solving the longest path problem for all pairs of nodes in terms of graph theory.

In applying the state-space representation approach to online scheduling, it is necessary to recalculate the transition matrix whenever any weights of the arcs change. In the light of this, efficient computation algorithms with $O(n \cdot (n + m))$ time complexity have been proposed [10], [11], where $n$ and $m$ represent the system size and the number of precedence constraints, respectively. Though these have the same theoretical time complexity, each has its own advantages and disadvantages, depending on the features of the adjacency matrix, such as the system size and density. In particular, the algorithm proposed in [11] has the following features:

• Partitioning can be carried out by arbitrary column.
• The resulting matrix is obtained row by row.

Thus, if we make several improvements based on distributed processing and parallel computation, the computation time would be significantly reduced. However, the paper only presents a simple implementation using a single processor, and thus the above advantages are not considered.

On the other hand, in [12], they focus on solving the shortest paths problems for all pairs of nodes, where the advantages of using the CBE are demonstrated. As such, this research focuses on computing the Kleene star of the weighted adjacency matrix of a DAG type, and an implementation of this using SIMD instructions [13] and multiple cores in parallel [14], both of which are available on the CBE, is carried out. We make experiments on a Sony PlayStation 3™ (PS3) equipped with a CBE.

2. Mathematical Preliminaries

Relevant mathematical background and other knowledge are briefly reviewed here.

2.1 Max-Plus Algebra

First, define a field $\mathcal{R}_{\text{max}} \equiv \mathcal{R} \cup \{-\infty\}$, where $\mathcal{R}$ repre-
sents the real field. Operators for addition $\oplus$ and multiplication $\otimes$ are defined as: $x \oplus y = \max(x, y)$, $x \otimes y = x + y$, $x^\oplus = x \cdot y$. Denote the unit elements for these by $e$ ($= -\infty$) and $\epsilon$ ($= 0$), respectively. The priority of $\otimes$ is higher than $\oplus$, and it is abbreviated if no confusion is likely to arise. For an operation on multiple numbers, if $m \leq n$, $\oplus_{k=m}^{n}x_k = \max(x_m, x_{m+1}, \ldots, x_n)$. In the representation and operations on matrices, $[X]_{ij}$ represents the $(i, j)$th element of matrix $X$. If $X, Y \in \mathbb{R}_{\text{max}}^{m \times n}$ and $Z \in \mathbb{R}_{\text{max}}^{n \times q}$, $[X \otimes Y]_{ij} = [X]_{ij} \oplus [Y]_{ij}$, $[X \otimes Z]_{ij} = \oplus_{s=1}^{q} [X]_{is} \otimes [Z]_{sj}$. Let the unit matrices for operators $\oplus$ and $\otimes$ be denoted by $e$ and $\epsilon$, respectively. $e$ is a matrix, all elements of which are $e$, whereas $\epsilon$ is a matrix, all diagonal elements of which are $e$ and all off-diagonal elements are $\epsilon$.

2.2 State Equation

The state equation in max-plus algebra and its relationship to a scheduling problem of the focused system are explained, using terms for manufacturing systems to facilitate understanding.

Let the number of facilities be $n$, and the job number be $k$. Then the processing completion times of job $k$ in all facilities is denoted by $x(k)$ and the constraints regarding their minimum values by $u(k)$. Under these notations, the state equation is expressed as follows [1], [2]:

$$x(k) = A_k \otimes [x(k - 1) \oplus u(k)],$$

where $x(k)$ and $u(k)$ are also referred to as the state and input vectors, respectively. Moreover, $A_k$ is known as the system or transition matrix. With the help of Eq. (1), by giving processing completion times for the previous job $x(k - 1)$ and constraints regarding the corresponding job $u(k)$, the earliest times of events for job $k$ can be obtained. We should note that there are several variants for associating the variables with events.

As an example, consider obtaining the earliest processing completion times for job $k$. Let the list of preceding facilities of facility $i$ $(1 \leq i \leq n)$ be $P(i)$, and the processing times of job $k$ be $d(k)$. Suppose the precedence constraints with respect to the processing order relation are given by the following adjacency matrix:

$$[F_0]_{ij} = \begin{cases} 
  e & : \text{if } j \in P(i), \\
  \epsilon & : \text{if } j \notin P(i).
\end{cases}$$

It is known that the system matrix $A_k$ can be represented as $A_k = P_k X_k^\epsilon$, where $P_k = \text{diag}(d(k))$ and $X_k = F_0 P_k$. $X_k$ is also referred to as a weighted adjacency matrix. Operator $\ast$ is called the Kleene star, the details of which are explained later. In terms of graph theory, facilities and precedence constraints correspond to nodes and arcs, respectively.

2.3 Kleene Star

With a calculation of the Kleene star, reachabilities between two arbitrary facilities can be identified and propagation times of events be calculated. Hereafter, let the weighted adjacency matrix be denoted simply by $X$. If the precedence constraints of the system are represented by a DAG, the Kleene star is calculated as:

$$X^* = \bigoplus_{i=0}^{s-1} X^i,$$

where $X^{s-1} \neq \epsilon$, $X^i = \epsilon$ $(1 \leq i \leq n)$, and $s$ depends on the precedence constraints of the system. $X^*$ in Eq. (2) has the following properties:

$$[X^*]_{ij} = \begin{cases} 
  \tau_{ij} : \text{if node } i \text{ is reachable from node } j, \\
  e : \text{if node } i \text{ is not reachable from node } j, \\
  e : \text{if } i = j.
\end{cases}$$

where $\tau_{ij}$ is the sum of the weights of arcs from node $j$ to node $i$. If there are multiple paths between two nodes, it gives the maximum value. Thus, we can interpret that calculating the Kleene star operator is equivalent to solving the longest path problem for all pairs of nodes.

3. Implementation on a CBE

This section first outlines the structure of the CBE, and then mentions several key points for implementing the algorithm proposed in [15].

3.1 Structure of the CBE

An overview of the structure of the CBE installed on a PS3 is illustrated in Fig. 1. In the CBE processor, there is an all-purpose processor called a PPE (PowerPC Processor Element), and eight processors called SPEs (Synergistic Processor Elements), specialized for numerical computation. Of these SPEs, the number available in Linux is six.

The PPE is a 64-bit processor with a PowerPC architecture that drives the operating system, as well as being in control of the SPEs. On the other hand, the SPE is a processor with 128-bit registers that supports SIMD operations, which allow multiple elements to be calculated with a single instruction. In an operation on 32-bit ‘float’ variables, four elements can be calculated, whilst 16 elements be calculated in an operation on 8-bit ‘char’ variables. However, SPEs do
not calculate scalars very efficiently.

For the main memory, the processor is equipped with 256 MB of fast memory, called XDR (eXtreme Data Rate) DRAM. Each SPE has 256 KB internal RAM, known as the LS (Local Store) which can be accessed from the corresponding SPE very quickly. However, data transfer between the LS and XDR must be performed by DMA transfer through a component called the MFC (Memory Flow Controller).

3.2 Outline of the Algorithm

We briefly review the efficient computation algorithm for the Kleene star proposed in [11].

Let the weighted adjacency matrix be \( X \). First, we carry out a topological sort based on the precedence constraints of the nodes by inspecting the elements of \( X \). If node \( j \) is located upstream of node \( i \) physically, we denote this by \( j < i \). Moreover, we denote the index of node \( i \) after the topological sort by \( \hat{i} \). A topological sort is a method for sorting the nodes such that: if \( j < i \), \( j < \hat{i} \). If we adopt the DFS (Depth First Search) algorithm, the time complexity is standard in several languages such as C, but differs slightly in respect of the following:

- A unit block is defined as 16 bytes, which is equal to the size of the registers in the SPE.
- Any non-square matrix is extended to a square matrix.
- The number of rows and columns are rounded up to a multiple of 4.
- The extended verbose elements are initialized to \( \epsilon \).

A maximum of four float variables can be stored in a block. An example of a vector with five elements is depicted in Fig. 2, where \([x]^{\top}\) represents the value of \( x \) rounded-up to a multiple of \( y \). Moreover, the internal structure of the \( m \times n \) matrix is illustrated in Fig. 3. To represent a column vector, we prepare an array for a row vector with the same size, and distinguish its type by attaching an attribution variable.

Fig. 2 Internal structure of a vector.

With regards the second feature above, the structure may not be advantageous for a vertically or horizontally long matrix. However, since both the adjacency matrix \( X \) and its Kleene star \( X^* \) are square, this redundancy is not that critical. Extending rows and columns is analogous to adding dummy nodes, which does not participate in any calculations of the target nodes. By adopting this structure, SIMDization for various operations can be accomplished easily, and a transpose matrix can be computed quickly.

3.4 Allocation of SPEs

Generally, in technical computations that involve matrix calculations, there are basically two objectives in dividing a process into several sub-processes.

- Using multiple processors, the sub-processes can be computed in parallel, and thus increasing the computation speed.
- A large amount of data can be processed sequentially, since a processor cannot handle all the data at one time.

As described earlier, each SPE on the CBE installed on a PS3 has 256 KB memory as the LS. Moreover, the LS is shared between text, stack and data areas. Thus, the available size for the data area is less than 256 KB, which means it would be desirable if the structures are flexible so that \( \oplus \) and \( \otimes \) could also be calculated easily for general matrices.

Hence, as the internal representation of the matrices, this research adopts the method proposed in [15]. We use four-byte float variables for representing values, and a special value ‘-FLT_MAX’ is used to represent \( \epsilon \). The elements of vectors and matrices are stored in a one-dimensional array, and those for matrices are in a row-major order. This structure is standard in several languages such as C, but differs slightly from the following:

- A unit block is defined as 16 bytes, which is equal to the size of the registers in the SPE.
- Any non-square matrix is extended to a square matrix.
- The number of rows and columns are rounded up to a multiple of 4.
- The extended verbose elements are initialized to \( \epsilon \).

A maximum of four float variables can be stored in a block. An example of a vector with five elements is depicted in Fig. 2, where \([x]^{\top}\) represents the value of \( x \) rounded-up to a multiple of \( y \). Moreover, the internal structure of the \( m \times n \) matrix is illustrated in Fig. 3. To represent a column vector, we prepare an array for a row vector with the same size, and distinguish its type by attaching an attribution variable.

With regards the second feature above, the structure may not be advantageous for a vertically or horizontally long matrix. However, since both the adjacency matrix \( X \) and its Kleene star \( X^* \) are square, this redundancy is not that critical. Extending rows and columns is analogous to adding dummy nodes, which does not participate in any calculations of the target nodes. By adopting this structure, SIMDization for various operations can be accomplished easily, and a transpose matrix can be computed quickly.

3.3 Internal Representation of Matrices

We now consider an internal representation for the matrices, particularly the adjacency matrix \( X \) as the input matrix, and \( X^* \) as the resulting matrix.

Though the primary focus of this research is to compute the Kleene star of the adjacency matrix of a DAG type, it would be desirable if the structures are flexible so that \( \oplus \) and \( \otimes \) could also be calculated easily for general matrices. Hence, as the internal representation of the matrices, this research adopts the method proposed in [15]. We use four-byte float variables for representing values, and a special value ‘-FLT_MAX’ is used to represent \( \epsilon \). The elements of vectors and matrices are stored in a one-dimensional array, and those for matrices are in a row-major order. This structure is standard in several languages such as C, but differs slightly with respect of the following:

- A unit block is defined as 16 bytes, which is equal to the size of the registers in the SPE.
- Any non-square matrix is extended to a square matrix.
- The number of rows and columns are rounded up to a multiple of 4.
- The extended verbose elements are initialized to \( \epsilon \).

A maximum of four float variables can be stored in a block. An example of a vector with five elements is depicted in Fig. 2, where \([x]^{\top}\) represents the value of \( x \) rounded-up to a multiple of \( y \). Moreover, the internal structure of the \( m \times n \) matrix is illustrated in Fig. 3. To represent a column vector, we prepare an array for a row vector with the same size, and distinguish its type by attaching an attribution variable.

With regards the second feature above, the structure may not be advantageous for a vertically or horizontally long matrix. However, since both the adjacency matrix \( X \) and its Kleene star \( X^* \) are square, this redundancy is not that critical. Extending rows and columns is analogous to adding dummy nodes, which does not participate in any calculations of the target nodes. By adopting this structure, SIMDization for various operations can be accomplished easily, and a transpose matrix can be computed quickly.
that all the elements of a large-scale matrix cannot be handled by a single SPE at one time. On the other hand, considerable overhead is required for invoking a program in an SPE and for transferring data between the PPE and SPE. Thus, if we repeatedly invoke and terminate the same SPE many times, the computation speed decreases.

Accordingly, this research divides the problem into several blocks in which the number of program calls are reduced. We partition the work matrix depicted in Fig. 3 into several vertically long matrices, where each partition is assigned to one SPE. The details of this partitioning are explained later.

Let the number of nodes and SPEs allocated be $n$ and $p$ ($\leq 6$), respectively. For the maximum number of columns that a single SPE can store in the available LS, let the value truncated to a multiple of 4 be $n_s$. We enforce each SPE to process for $n_0 = \lfloor n/p \rfloor^4$ columns. If $n_0 > n_s$, the SPEs have to compute the resulting partial matrices step by step. The reason for setting $n_s$ and $n_0$ to be multiples of 4 is to allow the relevant calculations to be performed by blocks.

### 3.5 Details of the Algorithm

First, we consider a method for creating a list of arcs for an adjacency matrix $X$, namely an adjacency list, which is required as pre-processing for the topological sort. In the DFS algorithm, since the succeeding nodes for a node are searched hierarchically, it is important to obtain a set of succeeding nodes for a given node. The set of succeeding nodes $S(j)$ for node $j$ is obtained as follows:

$$ S(j) = \{ i \mid |X|_{ij} \neq \epsilon \}. $$

This implies that an efficient framework for seeking the $j$th column vertically is essential. Thus, we partition the input matrix $X$ into vertically long block matrices as shown in Fig. 4. Each of these is allocated to one SPE, where $n_0$ is rounded down to a multiple of 4 to comply with the SIMDization.

To assist the readers’ understanding, an illustrative numerical example is presented. Figure 5 depicts a system of a DAG type with six nodes. The numbers in the circles represent the node number. We assume here that the weights of arcs are equal to the node number of the upstream node. Then, the weighted adjacency matrix $X$ is given by:

$$ X = \begin{bmatrix} S_{PE1} & S_{PE2} \\ \epsilon & \epsilon & \epsilon & \epsilon & \epsilon & \epsilon \\ 1 & \epsilon & \epsilon & 4 & \epsilon & \epsilon \\ \epsilon & 2 & \epsilon & \epsilon & 5 & 6 \\ 1 & \epsilon & \epsilon & 4 & \epsilon & \epsilon \\ \epsilon & \epsilon & \epsilon & 4 & \epsilon & \epsilon \end{bmatrix}. \tag{5} $$

Here we assume that SPE1 processes columns 1 to 4, while SPE2 processes the 5th and 6th columns. Regarding the internal representation of this matrix, it should be recalled that the matrix is expanded to an 8x8 matrix, and both SPE1 and SPE2 have the 7th and 8th rows, all elements of which contain $\epsilon$. Moreover, SPE2 has the 7th and 8th columns, the elements of which contain $\epsilon$.

We should also recall that the elements of the adjacency matrix $X$ are stored based on a row-major order as shown in Fig. 3. Thus, the elements for each SPE in Eq. (5) are not stored in a continuous area on the PPE side. This implies that a DMA command can transfer the data of a single row between the PPE and SPE at any one time. This is repeated for each of the rows. In all transfers, there is an overhead of preparation and confirmation of a completion. However, its effect may be reduced if we use a multiple buffer technique.

For each partitioned block matrix, after the transfer of the $i$th row, we inspect whether or not $|X|_{ij} = \epsilon$ holds for all $1 \leq j \leq n_0$, and repeat this for all $i$ ($1 \leq i \leq n$). This process is equivalent to inspecting whether or not $i$ is included in the set $S(j)$ in Eq. (4). Then we store the inspection results in an array of ‘unsigned short’ type as shown in Fig. 6. In the first element of the $i$th row, the number of elements of $S(j)$ is stored, and the contents of $S(j)$ are stored in the succeeding elements. After the inspection process for all rows, the resulting matrix is transferred from the SPE to the PPE.
On the other hand, the PPE waits until all transfers of the resulting matrices from the SPEs have been completed.

Subsequently, we carry out a topological sort based on the DFS algorithm. Since it is known that parallelizing the DFS requires repetitive communications between processors[17], it would be better to execute the sort either by the PPE or a single SPE. Moreover, recall here that the size of the LS on an SPE is small. This implies that it is impossible for large matrices to complete the sort with an SPE at one data transfer. In view of these, we have built an experimental implementation using only a PPE and measured the computation time of it. Since the time for the sort is much smaller than the time for computing $X^*$, we perform the topological sort using only the PPE. A result of the topological sort for the system in Fig. 5 is depicted in Fig. 7. Denoting the array storing this result by $\mathcal{T}$, we have:

$$\mathcal{T} = [4, 6, 1, 5, 2, 3].$$

Next, for all sets $(i, j)$ that satisfy $[X]_{ij} \neq 0$, we create lists of the succeeding nodes $S(j)$ for node $j$ and the weights of arcs $[X]_{ij}$. These are created sequentially from topologically upstream nodes to downstream nodes with respect to $j$.

Let the lists be denoted by $\mathcal{N}$, $\mathcal{L}$ and $\mathcal{W}$, where the first two lists and the latter one are arrays of ‘unsigned short’ and ‘float’ types, respectively. $\mathcal{N}$ and $\mathcal{L}$ hold a list of the number of succeeding nodes and the corresponding node numbers, respectively. The former list is created from the first elements of each row in Fig. 6 whilst the latter is from the succeeding elements. This is performed in the topological order specified by $\mathcal{T}$. For the case illustrated in Fig. 7, $\mathcal{N}$ and $\mathcal{L}$ are given as:

$$\mathcal{N} = [3, 1, 2, 1, 1, 0],$$

$$\mathcal{L} = [[2, 5, 6], [3], [2, 5], [3], [3], []).$$

Moreover, consider creating $\mathcal{W}$ that holds the weights of the arcs. The weights are obtained from $[X]_{ij}$, where $i$ and $j$ represent the preceding and succeeding node number given by the element of $\mathcal{T}$ and $\mathcal{L}$, respectively. This yields:

$$\mathcal{W} = [[4, 4, 4], [6], [1, 1], [5], [2], [1]]$$

Though all elements of each subset are equal in this example, they may be different if we weigh the arcs in a different manner.

For the performance of creating the above four lists, we have built an experimental implementation using only a PPE. Since the computation time for creating these is much smaller than the overall computation, we calculate this part using the PPE only.

The above lists are passed to all SPEs for use in calculating the partial matrices of $X^*$. Then, the partial matrix of $X^*$ is computed based on Eq. (3). On the SPE side, we dynamically allocate a data area for storing the partial matrix shown in Fig. 4, and initialize the elements to be a partial matrix of the unit matrix $e$. Next, we update the values of the partial matrix based on Eq. (3), where $j$ is executed only for the allocated $n_0$ columns, and not for all $j (1 \leq j \leq n)$. Moreover, if $n > n_1 \cdot p$, the update is carried out for $n_1$ columns at one time, and this is repeated multiple times.

We explain the procedure in SPE1 considering the system shown in Fig. 5. First, prepare a partial unit matrix $Z$ of size $8 \times 4$. Regarding the topmost node, $i = 1$, $\mathcal{T}[1] = 4$, $\mathcal{L}[1] = [2, 5, 6]$ and $\mathcal{W}[1] = [4, 4, 4]$. In accordance with Eq. (3), these can be stated as $\tilde{l} = 4$, $S(\tilde{l}) = [2, 5, 6]$, and $X_{\tilde{l}j} = [4, 4, 4]$. Then, we add the values of the 4th row of $Z$ multiplied by 4 (added to 4 in conventional algebra) to the 2nd, 5th and 6th rows. Next, since we can obtain $l = 2$, $\mathcal{T}[2] = 6$, $\mathcal{L}[2] = [3]$ and $\mathcal{W}[2] = [6]$, we add the values of the 6th row multiplied by 6 to the 3rd row. Then, we repeat the same procedure for all nodes. An outline of this procedure is illustrated in Fig. 8, where the empty cells contain $e$. Since the actual result of calculating $X^*$ is given by:
3.7 Overall Flow

We confirm the overall flow of the proposed method. Figure 9 gives a flowchart of the overall process of calculating \( X^* \) after the weighted adjacency matrix \( X \) has been given, and defines the roles of the PPE and SPEs. We divide the entire flow into four phases. Each SPE is invoked twice in phases (1) and (4). The first time is to create a list of succeeding nodes and to return the result to the PPE. On the PPE side, barrier synchronization is performed whilst receiving the results from all the SPEs. The second invocation is to calculate a partial matrix of \( X^* \), with the result returned to the PPE row by row in non-blocking mode for every process of one row. On the PPE side, barrier synchronization is once again performed to receive the results from all the SPEs. Since the required size of working memory differs between the first and second invocations, the maximum number of columns that a SPE can handle at one time, \( n_s \), may differ.

In calculating the partial matrix of \( X^* \), all SPEs use lists \( T, N, L \), and \( W \). We should pay attention here to that \( T \) and \( N \) have \( n \) elements, whilst \( W \) and \( L \) have \( n \cdot (n - 1)/2 \) elements at the maximum. This implies that all elements of the last two lists cannot be stored in an SPE if \( n \) is large and \( X \) is dense. Thus, we set a maximum size for storing elements of these lists. If the number of arcs exceeds the...

\[
X^* = \begin{bmatrix}
e & e & e & e & e \\
1 & e & e & 4 & e & e \\
6 & 2 & e & 10 & 5 & 6 \\
e & e & e & e & e & e \\
1 & e & e & 4 & e & e \\
e & e & e & e & e & e
\end{bmatrix}
\]

we can confirm that SPE1 calculates the values of columns 1–4.

### 3.6 SIMDization

As described in 3.1, an SPE executes a variety of operations using the 16-byte registers. Thus, using SIMD operations, four float-type variables can be processed in a single instruction, and thus contributing to reducing the computation time. In this research, we use multiple SPEs in the following two procedures:

1. Generating an adjacency list from the weighted adjacency matrix \( X \), equivalent to the set of succeeding nodes \( S \).

2. Calculating a partial matrix of \( X^* \) from lists \( T, N, L \) and \( W \).

With respect to 1., consider the partial matrix of the weighted adjacency matrix \( X \) shown in Fig. 4. In each SPE, the values of \([X]_{ij} (1 \leq j \leq n_0)\) are inspected at every transfer of a row from the PPE. If a value is anything other than \( e \), we increment the \((j, 1)\)th element of the array in Fig. 6, and then append the node number \( i \) to the last position of the corresponding row. We can utilize SIMD instructions for the inspection irrespective of whether or not the elements are \( e \). However, the update in the array in Fig. 6 must be performed for discrete elements. Thus, this update cannot be SIMDized.

In addition, this process inspects the values only of the \( i \)th row of \( X \). Thus, during the above update process, we can read the values of the next row through a non-blocking DMA transfer, and thus contributing to reducing the total computation time. This technique is referred to as a double buffer, which is often used to achieve speedup in computing large scale matrices and vectors.

With respect to 2. above, every SPE has the same lists \( T, N, L \) and \( W \), and each calculates the values of the assigned columns of \( X^* \). According to Eq. (3), the values are obtained sequentially from the topologically upstream nodes to the downstream nodes. The essence of this procedure consists of a simple operation in which the values of the \( i \)th row multiplied by a constant are added to other rows. Thus, a SIMDization of this procedure is straightforward.

Moreover, we pay attention to the feature that, at the start of processing the \( i \)th row, the corresponding row will not be changed any more. This means that the values of the \( i \)th row of \( X^* \) are already fixed before processing the \( i \)th row. Accordingly, if we issue a transfer command to the PPE in non-blocking mode before processing the \( i \)th row, we can reduce the computation time.
size, the remaining elements are transferred from the PPE to SPEs sequentially, and the calculation of the partial matrix continues until all elements of the lists will be processed. Moreover, considerable overhead is incurred in starting and terminating a SPE. This implies that the performance would deteriorate if we were to implement code in which an SPE is invoked at each request. Hence, we adopt the policy where all SPEs are invoked from the PPE in the initial stage, and then each SPE waits until a command is received from the PPE. After the procedure has been completed, the SPE waits for the next calculation or termination command.

4. Implementation and Performance Evaluation

Using an implementation based on the proposed method, the speedup effect of SIMDization and parallelization is examined. The execution environment is a Sony PlayStation 3™, running Fedora Core 10. We use gcc-4.1.1 as the compiler and the CELL Software Development Kit (SDK) [18] Version 3.1. The following commands are used for compiling programs for the PPE and SPEs:

- PPE: ppu-gcc -O3 -m64 -altivec -mabi=altivec
- SPE: spu-gcc -O3

We reserve 128 KB of memory in the LS of an SPE for the data area, which is shared by the partial matrix and the lists for succeeding nodes. For the maximum size of lists \( L \) and \( W \), we limit the maximum elements to 4096, which corresponds to 8 KB and 16 KB, respectively. Thus, the maximum size of the adjacency matrix that can process at one time is determined by \([n]^{4} \cdot n_{0} \leq (128 - 8 - 16) \cdot 1024/4\), where \( n_{0} = \lceil n/p \rceil \). For example, the specific maximum values are \( n = 160 \) and \( 388 \) for \( n = 1 \) and \( 6 \), respectively.

Regarding the system’s size, we consider cases where the number of nodes are \( n = 100, 200, 400, 1000 \) and 2000. Though the algorithm itself can compute for larger number of nodes, it is required to activate the swap space to allocate a sufficient memory for such cases. Thus, we limit the maximum number of nodes to 2000, where all elements of the resulting matrix can be stored in the main memory.

For the adjacency matrix \( F_{0} \), we first attach the constraint \( i \rightarrow j \) with a probability of 0.5 for all pairs of \((i, j)\) that satisfy \( i < j \). Then, we attach the precedence constraint \( i \rightarrow (i + 1) \) for all \( i \ (1 \leq i \leq n - 1) \). This means that the total number of arcs is \( m \approx n \cdot (n - 1)/4 + (n - 1) \). For example, if \( n = 100, m \) is around 2574, and 2500749 if \( n = 1000 \). Then, we sort the indexes of nodes randomly and use the resulting matrix as the adjacency matrix \( F_{0} \). For the diagonal matrix \( P_{k} \) that represents the weights of nodes, suppose the diagonal elements obey the normal uniform distribution \([0, 1] \). We measure the computation time of calculating \( X_{k} \) from the time when \( X_{k} \ (= F_{0}P_{k}) \) is given. This measurement is executed ten times, and the average time is adopted.

Tables 1 and 2 show the computation times of the SIMDized and non-SIMDized functions, respectively. The unit used in all cases is microseconds. The first columns represent the number of SPEs allocated, equivalent to \( p \). Regarding \( p = 1 \), the computation times increase in proportion to \( n^{2} \) roughly. This would because the number of arcs is \( m = n \cdot (n - 1)/4 + (n - 1) \) and thus the computation time complexity is \( O(n^{3}) \). In respect to the effect of using multiple SPEs, the computation times are reduced as \( p \) increases.

We now examine speedup effects of the help of SIMDization and using multiple SPEs. Figures 10 and 11 depict the speedup effects of using multiple SPEs. By defining an index for evaluating the scalability as:

\[
\text{Computation time using an SPE} = \frac{\text{Computation time using } p \text{ SPEs}}{p}
\]

it is calculated for all cases based on Tables 1 and 2. If this index increases linearly in proportion to \( p \), it can be said that the performance is scalable. In Fig. 10, for large scale systems \( n = 1000 \) and \( n = 2000 \), the performance curves increase almost linearly, and the scalability indices for \( p = 6 \) are 4.54 and 5.02, respectively. Thus, the proposed algorithm would be enough scalable for large scale systems. For smaller-scale systems \( n \leq 400 \), the curves increase but not that remarkable compared with the large-scale ones. In addition, they are likely to saturate around \( 4 \leq p \leq 7 \). This would because the relevant overheads regarding the control of the SPEs and the DMA transfers become relatively large.

Figure 11 depicts the speedup effects of the non-SIMDized algorithm. The performance curves for large-scale systems \( n \geq 1000 \) show a similar trend and they appear to be more scalable than Fig. 10. By contrast, the curves for small-scale systems \( n \leq 400 \) behave in a significant different manner. For instance, the curves for \( n = 100 \) and \( n = 400 \) saturate at \( p = 5 \) and \( p = 4 \), respectively. On the other hand, the curve for \( n = 200 \) increases over linearly, and does not saturate in \( 1 \leq p \leq 6 \).

Since the overheads for controlling the SPEs and for the DMA transfer would be almost the same as the SIMDized algorithm, this ‘fluctuation’ may be caused from the difference of the computation times of the adjacency lists and the resulting partial matrices. We should recall here that all numerical computations including scalar variables

| Table 1 | Computation times of the SIMDized function. |
|---------|--------------------------------------------|
| SPEs   | \( n = 100 \) | 200 | 400 | 1000 | 2000 |
| 1      | 713          | 5,732 | 39,761 | 668,816 | 6,957,865 |
| 2      | 493          | 2,769 | 24,470 | 356,795 | 3,640,312 |
| 3      | 417          | 2,213 | 19,338 | 252,142 | 2,513,187 |
| 4      | 381          | 1,954 | 14,446 | 207,170 | 1,949,704 |
| 5      | 349          | 1,748 | 14,298 | 177,075 | 1,627,725 |
| 6      | 341          | 1,678 | 14,284 | 147,201 | 1,385,228 |

| Table 2 | Computation times of the non-SIMDized function. |
|---------|-----------------------------------------------|
| SPEs   | \( n = 100 \) | 200 | 400 | 1000 | 2000 |
| 1      | 3,351        | 33,718 | 225,486 | 2,324,904 | 27,269,638 |
| 2      | 1,847        | 13,249 | 132,751 | 1,652,351 | 13,972,260 |
| 3      | 1,382        | 9,288  | 100,715 | 1,121,208 | 9,420,154  |
| 4      | 1,127        | 7,454  | 68,960  | 894,086  | 7,135,947  |
| 5      | 881          | 5,970  | 69,394  | 740,230  | 5,836,885  |
| 6      | 881          | 5,464  | 69,951  | 585,397  | 4,843,476  |
are performed using 128 registers in the SPEs. Thus, conversions for a scalar variable are needed both before and after computation. Moreover, in the non-SIMDized algorithm, the algorithm uses a larger amount of variables than the SIMDized one. This means that the SPEs may have to preserve, restore and exchange the registers more frequently. These overheads would increase the computation times unpredictably.

Furthermore, we compare the computation times between several processors whether the focused algorithm is suited for other processors. We prepare the following three environments:

(a) PS3: PPE (PowerPC compatible), 3.2 GHz
(b) PC: Pentium 4, 3.2 GHz
(c) PC: Athlon 64 X2, 2.9 GHz

To evaluate objectively, we use the same source code, compiler and compilation option, running the same operation system, Fedora 10. We compile the program using gcc-4.3.2 with ‘-O3’ option. In this experiment, only a single core and no SIMD instruction is used. Thus, all calculations are performed only in the main memory and no DMA transfer is required. Table 3 shows the computation times for processors (a)–(c). As it is often said, it seems that the PPE, processor (a), is not good at floating point calculations. Comparing with Tables 2 and 3, for larger $n \geq 1000$, the computation times by processors (b) and (c) are close to those with one and two SPEs, respectively. In view of this, although the algorithm is originally designed for using an interface of message passing type, it may also be efficient for Intel compatible processors with multiple cores.

In the last, we measure the execution times of the internal phases. Table 4 shows the execution profiles of the non-SIMDized function using an SPE. Phases (1)–(4) correspond to those illustrated in Fig. 9. The total computation time for each case is equivalent to the value of the 1st row in Table 2. We recall here that the computation times in phases (1) and (4) include those by both the PPE and the SPEs, whilst in phases (2) and (3) by only the PPE. For all cases, we can find that the values in phase (4) are the largest. In addition, for smaller $n$, the values in phase (1) are the second largest. Thus, to reduce the computation time further, it would be necessary to tune the code of phase (4) intensively.

5. Conclusion

This research has investigated the speedup effect of calculating the Kleene star for a weighted adjacency matrix of a DAG type in a max-plus algebraic system. By implementing a program adapted for a Cell Broadband Engine processor, the overall results of the numerical experiments indicate that the computation time can be reduced if we use multiple SPEs and perform SIMDization. Generally, the computation time can be reduced as the number of SPEs allocated increases. In terms of scalability, it becomes more scalable as the number of nodes $n$ increases. With the non-SIMDized algorithm, the scalability was less significant and unpredictable compared with the SIMDized one.
Through numerical experiments, we found that the computation time for the Kleene star may be further reduced for large-scale systems if we use a larger number of SPEs than a single CBE has. To accomplish this, a solution is to use multiple CBEs simultaneously, and techniques for cluster computing are required. Developing this kind of efficient algorithm remains our future work.

Acknowledgment

This research has been supported in part by research grants from the Kayamori Foundation of Informational Science and the JSPS Grants-in-Aid for Scientific Research Nos. 20710114 and 20560423.

References

[1] B. Heidergott, G.J. Olsder, and L. Woude, Max Plus at Work: Modeling and Analysis of Synchronized Systems, Princeton Univ. Press, New Jersey, 2006.
[2] F. Baccelli, G. Cohen, G.J. Olsder, and J.P. Quadrat, Synchronization and Linearity, John Wiley & Sons, New York, 1992. (http://maxplus.org)
[3] T. Chen, “Cell broadband engine architecture and its first implementation. available: http://www.ibm.com/developerworks/power/library/pa-cellperf/,” 2005.
[4] J. Kahle, M. Day, H. Hofstee, C. Johns, T. Mauzer, and D. Shippy, “Introduction to the cell multiprocessor,” IBM J. Res. and Development, vol.49, no.4/5, pp.589–604, DOI: 10.1147/rd.494.0589, 2005.
[5] D. Pham, S. Asano, M. Bolliger, M.N. Day, H.P. Hofstee, C. Johns, J. Kahle, A. Kameyama, J. Keaty, Y. Masubuchi, M. Riley, D. Shippy, D. Stasiak, M. Suzukki, M. Wang, J. Warnock, S. Weitzel, D. Wendel, T. Yamazaki, and K. Yazawa, “The design and implementation of a first-generation CELL processor,” ISSCC Digest Tech. Papers, vol.1, pp.184–185, 592, DOI: 10.1109/ISSCC.2005.1493930, 2005.
[6] M. Scarpino, Programming the Cell Processor: for Games, Graphics, and Computation, Prentice Hall, New York, 2008.
[7] G. Schullerus, V. Krebs, B. Schutter, and T. Boom, “Input signal design for identification of max-plus-linear-systems,” Automatica, vol.42, no.6, pp.937–943, DOI: 10.1016/j.automatica.2006.01.025, 2006.
[8] R. Goverde, “Railway timetable stability analysis using max-plus system theory,” Transportation Res. Part B, vol.41, no.2, pp.179–201, 2007.
[9] H. Takahashi, H. Goto, and M. Kasahara, “Application of a critical chain project management based framework on max-plus linear systems,” Int. J. Comput. Sci., vol.3, no.2, pp.117–132, 2009.
[10] H. Goto, “Efficient calculation of the transition matrix in a max-plus linear state-space representation,” IEICE Trans. Fundamentals, vol.E91-A, no.5, pp.1278–1282, May 2008.
[11] H. Goto and H. Takahashi, “Fast computation methods for the Kleene star in max-plus linear systems with a DAG structure,” IEICE Trans. Fundamentals, vol.E92-A, no.11, pp.2794–2799, Nov. 2009.
[12] K. Matsumoto and G. Sedukhin, “A solution of the all-pairs shortest paths problem on the cell broadband engine processor,” IEICE Trans. Inf. & Syst., vol.E92-D, no.6, pp.1225–1231, June 2009.
[13] P. Cockshott and K. Renfrew, SIMD Programming Manual for Linux and Windows, Springer, Heidelberg, 2004.
[14] R. Shonkwiler and L. Lefton, An Introduction to Parallel and Vector Scientific Computing (Cambridge Text in Applied Mathematics), Cambridge Univ. Press, 2006.
[15] H. Goto and A. Kawaminami, “An efficient solver for scheduling problems on a class of discrete event systems using CELL/B.E. processor,” 9th Int. Conf. Intelligent Syst. Design and Appl., pp.725–730, 2009.
[16] T. Cormen and C. Leiserson, Introduction to Algorithms, MIT Press, Massachusetts, 2001.
[17] N. Rao and V. Kumar, “Parallel depth first search. part I. Implementation,” Int. J. Parallel Programming, vol.16, no.6, pp.479–499, 1987.
[18] IBM, CBE Programmer’s Guide Version 3.1, IBM, 2008.

Hiroyuki Goto received his B.S. and M.S. degrees from the University of Tokyo in 1995 and 1997, respectively. He joined Mitsubishi Research Institute, Inc., where he engaged in research on parallel computation and fluid analysis. He was engaged in research on time-series analysis and multivariate analysis at the Japan Research Institute, Ltd. from 2001 to 2005. He received his D.E. degree from Tokyo Metropolitan Institute of Technology in 2004. Currently, he is an associate professor at the Department of Management & Information Systems Science, Nagaoka University of Technology. He is also a member of the Japan Industrial Management Association (JIMA) and the Operations Research Society of Japan (ORSJ).