PDAC: A Data Parallel Algorithm for the Performance Analysis of Closed Queueing Networks*

F. B. Hanson, Jing-Dong Mei, Charles Tier and Huihuang Xu
Laboratory for Advanced Computing
University of Illinois at Chicago
P. O. Box 4348; M/C 249
Chicago, IL 60680
E-mail: hanson@math.uic.edu

March 2, 2013

Abstract. A parallel distribution analysis by chain algorithm (PDAC) is presented for the performance analysis of closed, multiple class queueing networks. The PDAC algorithm uses data parallel computation of the summation indices needed to compute the joint queue length probabilities. The computational cost of the PDAC algorithm is shown to be of polynomial order with a lower degree than the cost of the serial implementation of the DAC algorithm. Examples are presented comparing the PDAC algorithm with the DAC algorithm to illustrate its advantages and limitations.

Keywords. Data parallel algorithms, Closed queueing networks, Performance analysis.

*This work was supported by the National Science Foundation under grants DMS-88-06099, DMS-89-22988 and DMS-91-02343, by Argonne National Laboratory's Advanced Computing Research Facility, by the Los Alamos National Laboratory's Advanced Computing Laboratory, by the UIC Computing center, by the UIC Software Technology Research Center, by the University of Illinois at Urbana's National Center for Supercomputing Applications.

1 Introduction

Closed, multiple class queueing networks are widely used to analyze the performance of large computer systems. A typical model consisting of N service nodes, M terminals, and R job classes is shown in Figure 1. For a fairly general class of networks [1, 9], there exists an

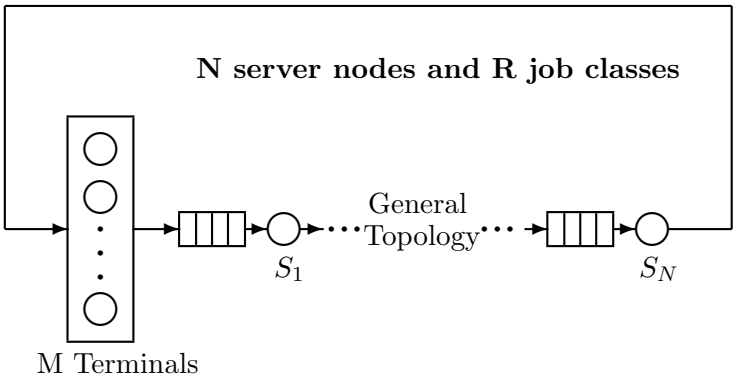

Figure 1: Multiple-Chain Closed Queueing Network System.

exact expression for the stationary queue length distribution as a product form solution. Performance measures, such as marginal queue-length distributions, mean queue lengths, throughputs and mean response times and delays, can be computed directly from the product form solution. However, for networks with a large number of job classes, terminals or nodes, the direct computation of the performance from the product solution is not feasible since the computational cost, depending on the algorithms used, grows exponentially either with the number of chains or with the number of nodes in the network [6, 9].

Several algorithms have been developed to efficiently compute the performance measures. These algorithms essentially belong to two classes, either they require computing the normalization constant for the product form solution or compute performance measures directly. Typical examples of former class are the Convolution algorithm due to Buzen [3] and the RECAL algorithm developed by Conway and Georganas [7]. Algorithms of the later class include the Mean Value Analysis (MVA) algorithm discovered by Reiser and Lavenberg[15] and Mean Value Analysis by Chain (MVAC) algorithm by Conway, de Souza e Silva and

Lavenberg[8]. The Convolution algorithm is a recursion based on building the network one node at a time. On the other hand, MVA builds the network by adding one job at a time. The Convolution algorithm and MVA algorithm require approximately the same amount of computations but the computation costs of both algorithms grow exponentially with the number of closed chains in the network, i.e., the computational costs are $O(c^R)$ as $R \to \infty$ with N fixed and where c is some constant. In RECAL, each job chain that has more than one job is converted into the same number of identical chains. Then each chain contains a single job and number of chains in the network equals the number of jobs. The recursion of the RECAL algorithm is based on adding one chain at a time. The computational cost of the RECAL algorithm grows as a polynomial function of the number of closed chains, but grows exponentially with the number of nodes in the network, i.e., the computational cost is $O(R^N)$ for $R \to \infty$ with N fixed. It has significant computational advantages over the Convolution algorithm for networks with many closed chains and few service centers. The MVAC is similar in form and computational requirements to the RECAL algorithm, but does not require the computation of the normalization constant.

Recently, de Souza e Silva and Lavenberg[16] presented the distribution analysis by chain algorithm (DAC). They have shown that the DAC algorithm is more efficient than the Convolution and the RECAL algorithms for computing the joint queues length probabilities and is also more efficient than the MVAC algorithm for computing mean values. The Convolution algorithm and the RECAL algorithm both require the computation of the normalization constant before computing the joint queue-length probabilities. However, the DAC algorithm involves calculating of the unnormalized joint queue-length probabilities, and has a smaller cost to evaluate joint queue-length probabilities than the Convolution algorithm and the RECAL algorithm. Also, the DAC algorithm computes marginal queue-length probabilities, mean queue length and throughputs more efficiently than the MVAC

algorithm. The basic step of the MVAC algorithm only computes mean queue lengths and throughputs, while DAC algorithm besides computing these measures computes all joint queue-length probabilities using almost same cost of the basic step of the MVAC algorithm. Currently, the DAC is one of the most efficient serial algorithms for computing performance measures in closed multiple class queueing networks with more chains than nodes in which case its computational cost is $O(R^N)$, i.e., grows as a polynomial in the number of closed chains, but exponentially with the number of nodes.

Even though these computational algorithms are improvements over direct calculation methods, each has disadvantages and often require a large number of calculations. For example, the costs of Convolution algorithm and the MVA are of exponential order on the number of chains, and costs of MVAC, RECAL and DAC are of high polynomial order on the number of chains if these algorithms are implemented via a uniprocessor system. This limitation, as we will see in the paper, is a bottleneck for further improvement.

Recently, Greenberg and McKenna[4] implemented a parallel version of the RECAL algorithm on a shared memory MIMD (Multiple Instruction, Multiple Data) multiprocessor. Their implementation resulted in an improvement over the serial implementation of the RECAL algorithm. Techniques such as indexing, parallel overwriting with buffer storage were used. In the SIMD the matrix data is processed in parallel. Greenberg and Mitrani [5] also improved the one-step and multi-step convolution of the Convolution algorithm for a different SIMD processor, using the fast Fourier transform and parallel prefix problem to accelerate the computation.

In this paper, we describe a parallel implementation of the DAC algorithm using a different type of parallel architecture, data parallel. Data parallel computation methods are based on the use of massively parallel machines such as the CM-2 which is the largest commercially available system of this type [17, 2, 12]. Recent applications [11, 13, 10] have shown

the promise of using data parallel computation. Our data parallel version of the DAC algorithm, called PDAC, is implementable and efficient. PDAC can compute joint queue-length distributions and the mean performance measures in computational complexity $O(R^2 \log R)$ for fixed N and large R with the computations spread over $O(R^{N-1})$ processors. The main difference, aside from the differences between the Convolution and DAC algorithms, between the implementation of Greenberg and McKenna [4] and the current parallel implementation of DAC is that they use an MIMD implementation that requires much processor and memory synchronization, especially in their overwriting scheme, while our implementation is for an SIMD (Single Instruction, Multiple Data) or distributed memory processor.

The paper is organized as follows: in Section 2 we present the Distribution Analysis by Chain (DAC) algorithm and discuss its advantages and disadvantages; in Section 3 we develop several rules and the parallel index-generation algorithms which are used to support PDAC, and also give the details for the PDAC algorithm; in Section 4 we describe the data parallel Connection machine CM-2; in Section 5 we analyze the computational requirements for PDAC and present some examples comparing DAC with PDAC.

2 Distribution Analysis by Chain (DAC)

We consider a closed product-form queueing network with multiple chains. Every chain that has more than one customer is converted into identical single customer chains [16]. The DAC algorithm uses recursive computation to compute the joint distribution of queue lengths at all service centers from which, the chain throughputs and then marginal queue-length distributions are computed. We use the following notation to describe the DAC algorithm:

- N: **number of service nodes** in the network.
- R: **total number of chains** in the network.

- $\mu_j(k_j)$: **service rate** of server node j when there are k_j jobs at node $j = 1, ..., R$.
- T_r : **throughput** of r chains (or jobs).
- w_{jr} : **total work load** of r chains on node j, where $r = 1, ..., R$.
- $\mathbf{k} = (k_1, k_2, ..., k_N)$: **joint queue length**; k_j is the number of jobs at node j.
- $\mathbf{1}_j$: N**-dimensional unit vector** of form $(0, \cdots, 0, 1, 0, \cdots, 0)$, such that the j^{th} element is one and other elements are zeros.
- $S^r = \{\mathbf{k} | \sum_{j=1}^{N} k_j = r\}$: **state space** of possible queue lengths at distinct nodes with r chains (jobs).
- $P^r(\mathbf{k})$: **steady-state probability** with joint queue-length $\mathbf{k} \in S^r$.
- $p_j^r(k_j)$: **marginal state probabilities** with k_j jobs at node j.
- G^r : **normalization constant** or **partition function** of the network with single-job chains $1, ..., r$ present, where $G^r = G^{r-1}/T_r$ is satisfied.

The cost of calculating the normalization constant G^r is approximately $O(N2^R)$ exponential order by the Convolution algorithm and $O(R^N)$ polynomial of high degree order by the RECAL algorithm[9]. However, the normalization constant G^r is not explicitly calculated in the DAC algorithm. In addition, by not using G^r explicitly, floating point overflow and underflow problems in the performance measures can be reduced.

In following discussion, the variables $N, R, k_j, \mu_j(k_j)$ and w_{jr} are assumed to be given. The DAC algorithm consists of the three basic equations:

$$P^r(\mathbf{k}) = T_r \sum_{j=1}^{N} w_{jr} k_j P^{r-1}(\mathbf{k} - \mathbf{1_j})/\mu_j(k_j) \tag{1}$$

$$T_r = \left[\sum_{j=1}^{N} w_{jr} \sum_{x=1}^{r} x p_j^{r-1}(x-1)/\mu_j(x) \right]^{-1} \tag{2}$$

$$p_j^{r-1}(x) = \sum_{\substack{\mathbf{k}\in S^{r-1} \\ k_j=x}} P^{r-1}(\mathbf{k}) \tag{3}$$

With above definitions and equations, the DAC algorithm can be written as:

1. Set $r = 1$ and $p_j^1(1) = w_{j1}/\sum_{k=1}^{N} w_{k1}$, for $j = 1, 2, ..., N$ nodes.

2. For $r = 2, ..., R$ chains :

   (a) Use Equation (2) to calculate T_r.

   (b) Use Equation (1) to calculate $P^r(\mathbf{k})$ for all $\mathbf{k} \in S^r$.

   (c) Use Equation (3) to calculate p_j^r for all $j = 1, 2, ..., N$ nodes and $k = 0, 1, ..., r$ jobs.

We note that in the DAC algorithm step (2b) and step (2c) are executed over all the index vectors $\mathbf{k} \in S^r$. In serial implementation of the DAC algorithm, however, the computation of the steady-state probability $P^r(\mathbf{k})$ must be executed one at a time for each of the index vectors $\mathbf{k} \in S^r$ in the sequence:

$$(r, 0, \cdots, 0), (r-1, 1, \cdots, 0), ..., (0, 0, \cdots, r).$$

Since the total number of index vectors in S^r is

$$\begin{pmatrix} N+r-1 \\ N-1 \end{pmatrix} = O(r^{N-1}) \ , \tag{4}$$

for $r \gg 1$ and N fixed, the computation cost is at least $O(r^{N-1})$ (i.e., polynomial order for the large number of chains (jobs) limit with the number of nodes fixed). This computation requirement, $O(r^{N-1})$, is a lower bound for the serial implementation of the DAC algorithm, and also a bottleneck for further improvement of the implementation on uniprocessor systems. Using a parallel implementation, we can overcome this bottleneck by computing $P(\mathbf{k})$ for all $\mathbf{k} \in S^r$ simultaneously. However, this requires that the generation of all indices in S^r be executed in parallel.

3 Parallel Distribution Analysis by Chain (PDAC)

We now present a data parallel computation method for DAC algorithm based on the Connection Machine model, which is discussed in section 4.

The generation of indices in serial implementation is a *data dependent* [14] problem since the generation of k^{th} index vector in the sequence is dependent on the contents of previous $(k-1)^{th}$ index vector generations for $k = 1$ to r. Since the output of the $(k-1)^{th}$ index vector is input or data to the calculation of the k^{th} index vector, we say that the k^{th} is data dependent on the $(k-1)^{th}$ index vector. This means that the parallelization of the index vector is restricted to the calculation of each k^{th} index vector, unless the calculation is restructed, such as in the data parallel method discussed below.

Each time a new index vector generated, it must be guaranteed that the new index vector is not a duplicate. The parallel computation is based on the idea of generating the set of index vectors simultaneously instead of generating them one at a time. We define an **index array** as two dimensional matrix, whose rows consist of index vectors. As we see later, the initial index array is of relative small size. Our data parallel algorithm starts with the initial index array and constructs a new larger index array. The new index arrays are constructed one at a time until a final index array, namely an array consisting of all the necessary index vectors, is constructed. In the process of computation, one index array corresponds to an array of single processors with each processor possessing one element of the index vectors. Thus, *array computation* here signifies the computation by a group of processors, i.e., data parallel computation.

By efficient use of the massively data parallel processors, the costs for steps (2b) and (2c) are reduced to polynomial order of low degree. Whether a group of massive parallel processors can perform calculations cooperatively for DAC algorithm, specifically for the index generation problem bottleneck, is another topic studied in this section.

3.1 PDAC Index Generation Algorithms

In this section, the formal structure of *array computation* is discussed. We first introduce some definitions:

- An **index vector** of parameter (N, R) is represented as $\mathbf{k} = (k_1, ..., k_N)$ with nonnegative components, $k_j \geq 0$ and $j = 1, ..., N$, which satisfy the equation $k_1 + k_2 + \cdots + k_N = R$ (i.e., $\mathbf{k} \in \mathbf{S^R}$, where $S^R = S^r$ when $r = R$).

- A **vector of decreasing order**: "$\gg$" (**vector of increasing order**: "$\ll$") is a partial order between index vectors. Let $\mathbf{k} = (k_1, k_2, ..., k_N)$ and $\mathbf{k'} = (k'_1, k'_2, ..., k'_N)$ be index vectors, then $\mathbf{k} \gg (\ll)\ \mathbf{k'}$ implies there exists an integer i, for $1 \leq i \leq N$, such that $k_i > (<) k'_i$ and $k_j = k'_j$ for all $1 \leq j \leq i - 1$.

- An **index vector array** $M(N|R)$ is a matrix having $\begin{pmatrix} N+R-1 \\ N-1 \end{pmatrix}$ rows and N columns. The rows are index vectors, $\mathbf{k} \in S^R$. We assume that the rows in $M(N|R)$ are listed in vector of increasing order.

- **Fill operation** is a binary operation, which adds one column of a given number of integer elements k to the leftmost side of array $M(i|R)$ to give a new index vector array $M(i+1|R)$.

  EX. If $M(2|1) = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}$, then filling "1" into $M(2|1)$ gives $\begin{pmatrix} 1 & 1 & 0 \\ 1 & 0 & 1 \end{pmatrix}$

- **Merge operation** is a binary operation which vertically merges two index vector arrays of same column size.

  EX. Merge $M(2|1) = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}$ with $M(2|2) = \begin{pmatrix} 2 & 0 \\ 1 & 1 \\ 0 & 2 \end{pmatrix}$ results in $\begin{pmatrix} 1 & 0 \\ 0 & 1 \\ 2 & 0 \\ 1 & 1 \\ 0 & 2 \end{pmatrix}$

- $\mathbf{M(i|R) - \bar{n}_j}$ is the operation to reduce all the elements on j^{th} column of $M(i|R)$ by constant n.

  EX. if $M(2|2) = \begin{pmatrix} 2 & 0 \\ 1 & 1 \\ 0 & 2 \end{pmatrix}$ then $M(2|2) - \bar{\mathbf{1}}_\mathbf{1} = \begin{pmatrix} 1 & 0 \\ 0 & 1 \\ -1 & 2 \end{pmatrix}$.

- **Cut-Off operation** is an operation used to cut off the rows of $M(i|R)$ whose leftmost side elements are negative.

  EX. If $M(2|2) - \bar{\mathbf{1}}_\mathbf{1} = \begin{pmatrix} 1 & 0 \\ 0 & 1 \\ -1 & 2 \end{pmatrix}$ then Cut-Off $(M(2|2) - \bar{\mathbf{1}}_\mathbf{1}) = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}$

We construct **Algorithm (1)** to generate a large index array based on the input of several small index arrays. The resulting large index array has one more column than all these small index arrays, and number of rows in the large index array is the sum of the number of rows of all small index arrays.

Algorithm (1) : Merge small index vector arrays into one large index vector array.

Input : $M(i|0)$, $M(i|1)$,..., $M(i|R)$
Output : $M(i+1|R)$
Begin
    $M \longleftarrow$ empty block
    For $k = 0$ to R do
        Begin
            $l = R - k$
            Fill "l" into $M(i|k) \longrightarrow ML(i|k)$
            $M \longleftarrow$ Merge M with $ML(i|k)$
        End
    Return $M(i+1|R)$
End

EX. For the case of $i = 2$ and $R = 2$, we have :

$$\text{Input:} \quad \begin{pmatrix} 0 & 0 \end{pmatrix}, \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}, \begin{pmatrix} 2 & 0 \\ 1 & 1 \\ 0 & 2 \end{pmatrix}$$

$$\Downarrow \textbf{Algorithm (1)}$$

$$\text{Output:} \quad \begin{pmatrix} 2 & 0 & 0 \\ 1 & 1 & 0 \\ 1 & 0 & 1 \\ 0 & 2 & 0 \\ 0 & 1 & 1 \\ 0 & 0 & 2 \end{pmatrix}.$$

Algorithm (2) works opposite to **Algorithm (1)**. The input is one large index array, and output are several small index arrays. These small index arrays have the same number of columns as the large index array but each has less rows than the large one.

Algorithm (2) : Decompose one large index vector array into small index vector arrays.

Input : $M(i+1|R)$
Output : $M(i+1|R-1)$,...,$M(i+1|0)$
Begin
    $M \longleftarrow M(i+1|R)$
    For $k = 1$ to R do
        Begin
            Minus all elements in first column
                of M by "$+k$"
            $M(i+1|R-k) \longleftarrow$ Cut-Off Block (M)
        End
    Return $M(i+1|R-1)$, $M(i+1|R-2)$,..., $M(i+1|0)$
End

EX. For the case of $i = 2$ and $R = 2$, we have :

$$\text{Input:} \quad \begin{pmatrix} 2 & 0 & 0 \\ 1 & 1 & 0 \\ 1 & 0 & 1 \\ 0 & 2 & 0 \\ 0 & 1 & 1 \\ 0 & 0 & 2 \end{pmatrix}.$$

$$\Downarrow \textbf{Algorithm (1)}$$

$$\text{Output:} \quad \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}, \begin{pmatrix} 0 & 0 & 0 \end{pmatrix}$$

Using **Algorithm (1)** and **Algorithm (2)**, we can generate an entire array of index vectors rather than one vector at a time. Further, by alternately using **Algorithm (1)** and **Algorithm (2)**, we can generate the array of index vectors, $M(N|R)$, which are needed for the PDAC algorithm. The index array $M(N|R)$ is generated by **Algorithm (3)** below:

Algorithm (3) : Generate index vector array $M(N|R)$.

Input : Parameters : $(N|R)$
Output : $M(N|R)$
Begin
    Construct $M(1|0)$, $M(1|1)$,...,$M(1|R)$
    For $i = 2$ to $N - 1$ do
        Begin
            Use **Algorithm (1)** to get $M(i|R)$
            Use **Algorithm (2)** to get $M(i|R-1)$,..., $M(i|0)$
        End
    Use algorithm (1) to get $M(N|R)$
End

The fact that **Algorithm (3)**, using **Algorithm (1)** and **Algorithm(2)**, generates the desired state space can be proved by induction on the number of nodes n for any fixed number of jobs R in the network, where $n = 1, ..., N$.

Using the Connection Machine computation model, we see that **Algorithm (3)** has time computational cost of $O(NR)$ since the "for loop" in **Algorithm (3)** requires computational cost of $O(N)$, while the "for loops" in **Algorithm (1)** and in **Algorithm (2)** require

computational costs of $O(R)$. The *Cut-Off* operation, the *Fill* operation and the *Merge* operation can be implemented on Connection Machine CM-2 in $O(1)$ computational steps. Therefore, a total of $O(NR)$ computational steps are required.

3.2 PDAC Algorithm

The PDAC algorithm is based on the data parallel index generation method. We know that the index array $M(N|r)$ contains all the index vectors $\mathbf{k} \in S^r$. By using these index vectors, we can compute step (2b) and step (2c) in the DAC algorithm simultaneously or in parallel. At computational step (2) of the DAC algorithm with a loop index r fixed, the PDAC algorithm accesses all the steady-state probabilities $P^{r-1}(\mathbf{k})$ indexed by $\mathbf{k} \in S^{r-1}$ simultaneously. Then, we use expression (1) to compute all steady-state probabilities $P^r(\mathbf{k})$ indexed by $\mathbf{k} \in S^r$ in parallel, and store all the results at the same time for use in next loop iteration $r+1$. This allows us to compute all the marginal state probabilities indexed by k_j and j in parallel over all of these available steady-state probabilities using parallel *sum* function. It should be pointed out the loop at computational step (2) of the DAC algorithm is executed in serial in the PDAC algorithm. Therefore, the terminology "parallel" here means the parallel computation over index vectors not over the loop. Further modification would be needed to reduce the number of iterations. In addition, at each iteration loop, the *Cut-Off* operation is employed to achieve the task of applying the data parallel index generation method to the DAC algorithm. In particular, the PDAC algorithm is:

1. Use **Algorithm (3)** to get $M(N, R)$.
2. Set r=1 and $p_j^1(1) = w_{j1} / \sum_{k=1}^{N} w_{k1}$, for $j = 1, 2, ..., N$ nodes.
3. For $r = 2, 3, ..., R$ chains :
   (a) Use Equation (2) to calculate T_r.
   (b) Generate all vectors $\mathbf{k} \in S^r$:
   Reduce all the elements of first column of $M(N, R)$ by "$R - r$", then all $\mathbf{k}$ are in the Cut-Off of M.

(c) Use Equation (1) to calculate $P^r(\mathbf{k})$ for all $\mathbf{k} \in S^r$ simultaneously.

(d) Use Equation (3) to calculate $p_j^r(k_j)$ for all $j = 1, 2, ..., N$ nodes and $k_j = 0, 1, ..., r$ jobs, over all the vectors $\mathbf{k} \in S^r$.

Steps (1-3) in the DAC algorithm yield the joint queue-length distributions of the network as well as the mean performance measures for chain R. As we show in Section 5, our current implementation of the PDAC algorithm has in computational cost $O(R^2 \log R)$ with CM Fortran release 0.7.

4 Parallel Processor Description

The Connection Machine (CM) is well suited for parallel index generation computation, specifically for large number of chains (jobs). The full Connection machine CM model 2 is a massively parallel single instruction, multiple data (SIMD) machine with up to 64K physical processors connected in a 12-dimensional hypercube network. It consists of computer chips with 16 bit processors connected together in a nearest neighbor network. Each processor typically has eight kilobytes of local memory, the entire primary memory of the CM comprises up to half a Gigabyte. An important aspect that simplifies parallel programming is the concept of virtual processors. Each of the physical processors can simulate a number of virtual processors. The maximum number of virtual processors is limited only by the available memory.

To accelerate scientific computing speeds, the full CM-2 has 2048 floating-point accelerator chips, one for every 32 bit processors. This accelerator has two options: single precision or double precision. Both options support IEEE standard floating-point formats and operations, and increase the rate of floating-point calculations by a factor of more than 20.

In the CM model, each processor is supposed to hold one data problem. In practice, a larger scale problem must be decomposed into data elements over a configuration of processors. The parallel computation using this configuration depends on whether the

decomposition is efficient.

The CM Fortran and Paris languages have been developed on the current Connection machine system for scientific computations. CM Fortran contains many of the extensions of Fortran-90 standard. It provides array constructs and operations. Techniques, such as one-to-many broadcasting, many to many broadcasting, fan-in sum operation and nearest-neighbor NEWS shifting, are basic parallel computation tools. We use CM Fortran release 0.7 to implement the parallel PDAC algorithm on a 32K machine at NCSA. We also have implemented the serial DAC algorithm on IBM 3090 at UIC with Fortran-77.

5 Performance Analysis

In order to compare the PDAC algorithm with the DAC algorithm, we measure the computational cost using the numbers of multiplications and additions as in [16]. We define one CM-multiplication by the multiplication between CM arrays or the multiplication between a scale variable and a CM array. Similarly, one CM-addition is defined by one array addition or by one step fan-in addition in the case of summing n scalar elements. For simplicity, the division operation is assumed to have the same cost as the multiplication operation. The network has number of jobs, $R \gg 1$ with the number of nodes $N = O(1)$. It must be pointed out that our implementation method is not quite optimal. Thus, the following analysis gives an overestimate for the calculation of the computational cost. An optimal implementation is being investigated.

We first observe that at step (2) of the PDAC algorithm, we can compute $P_j^1(1)$ simultaneously for all $j = 1, 2, ..., N$. This requires one CM multiplication and order $\log(N)$ CM additions. (In CM Fortran, the build-in *sum* function is executed in a logarithmic order number of steps by a fan-in addition [18].) The largest number of computational steps in the PDAC algorithm occurs in step (3). In the calculation of T_r at step (3a), the inner summation can be computed in logarithmic order, $\lceil \log_2(r) \rceil = O(\log r)$, for each $r = 2, 3, ..., R$ by the fan-in *sum* function, the outer summation is computed one at a time. Hence, $NO(\log r)$ CM-additions are needed for each $r = 2, 3, ..., R$ and N fixed. For the multiplication op-

erations, the inner part is calculated over all distinct indices x in parallel using two CM-multiplications, then one CM-multiplication is needed to multiple the inner summand with factor w_{jr}, and another is needed to compute the inverse. Thus, the computation of T_r needs $(3N+1)$ CM-multiplications for each $r = 2, 3, ..., R$. One CM-addition is counted at step (3b) to do the reduction operation for each $r = 2, 3, ..., R$, for a total of R CM-additions in step (3b). At step (3c) to get $P^r(\mathbf{k})$, the multiplications, for each $j = 1, 2, ..., N$, can be executed in parallel over all distinct index vectors $\mathbf{k}$ in S^r, and the summation is computed in a linear order of steps by a **DO loop** statement, this gives $(3N+1)$ CM-multiplications, including one multiplication by the factor T_r, and $(N-1)$ CM-additions. For step (3d) to get $p_j^{r-1}(x)$, we have $\left(\begin{smallmatrix} N+r-1 \\ N-1 \end{smallmatrix} \right)$ vectors in S^r, for each given $j = 1, 2, \cdots, N$ and each given $x = 1, 2, \cdots, r$. Thus, the summation operations over each pair of j and x can be computed in

$$\log \left(\begin{array}{c} N+r-1 \\ N-1 \end{array} \right) \sim \log \frac{r^{N-1}}{(N-1)!} \sim (N-1) \log r \tag{5}$$

CM-additions for large r and fixed N. An optimal implementation should allow all $p_j^r(x)$ be computed simultaneously for different pair of j and x with the CM Fortran **forall** statement. However, the currently implemented data structure does not provide the best computational environment for the **forall** statement to successfully operate. More study is needed to see if all $p_j^r(x)$ can be computed in parallel. Right now, we simply implement them by serial execution for the $N \cdot r$ different pairs of j and x, i.e., the total cost for each r of step 3 is asymptotic to

$$N(N-1) r \log r \tag{6}$$

CM additions for large r and fixed N. In the PDAC algorithm, step (3a) - step (3d) are executed in $(R-1)$ steps, so the total number of CM multiplications is:

$$(R-1)(3N+1+3N+1) = (R-1)(6N+2) = O(R), \tag{7}$$

where for fixed N as $R \to \infty$, so that the N dependence is ignored. The total number of

CM additions is approximately

$$\sum_{r=2}^{R} N(N-1)r\log r = O(R^2\log R), \tag{8}$$

as $R \rightarrow \infty$ and for fixed N. It must be noted that step (3b) costs $O(1)$ execution time since both the *Reduce* and *Cut-Off* operations need $O(1)$ computation costs, either by the one to many broadcasting operation or by the subarray copy operation. In the case of $R \rightarrow \infty$ and N is fixed, we find from expressions (7) and (8) that the multiplication operation costs are $O(R)$ and the addition operation costs are $O(R^2\log R)$ respectively.

The performance analysis results for the DAC algorithm were given in [16] and show that the number of multiplications of the DAC algorithm is :

$$(3N+1)\begin{pmatrix} N+R \\ N \end{pmatrix} + NR(R+2) + R - 3N^2 - 6N - 2 = O(R^N). \tag{9}$$

And the number of additions is

$$(N-1)\begin{pmatrix} N+R \\ N \end{pmatrix} + \begin{pmatrix} N-1+R \\ N-1 \end{pmatrix} + \begin{pmatrix} N-2+R \\ N-1 \end{pmatrix} - N^2 - 1 = O(R^N), \tag{10}$$

as $R \rightarrow \infty$ and for N fixed. We conclude that the PDAC algorithm has polynomial computational complexity of lower degree $O(R^2\log R) = o(R^3)$ for large numbers of chains (jobs), namely for $R \gg 1$ and N fixed, while the DAC algorithm has polynomial computational complexity of large degree $O(R^N)$ for large numbers of chains (jobs) R and for a sufficiently large fixed number of nodes, $N > 3$.

However, the number of processors PDAC requires is relatively large. PDAC needs $\begin{pmatrix} N+R-1 \\ N-1 \end{pmatrix} = O(R^{N-1})$ Connection Machine one-bit processors for large R and N fixed. This is due to steps (3a- 3d) in the PDAC algorithm which are executed from $r = 2, \cdots, R$ in $R-1$ iterations. Each iteration requires the results of the previous iteration. Therefore for the r^{th} iteration computation, it is necessary to save the results from $(r-1)^{th}$ iteration in the storage. In addition, we need to save all index vectors $\in S^r$ at r^{th} step in order to compute all $P^r(\mathbf{k})$ for $\mathbf{k} \in S^r$. In the Connection Machine model this can be done in $O(R^{N-1})$ storage elements, or virtual processors.

The analyzed performances can be verified from Figure 2 and Figure 3, where the execution time has been decomposed into the various steps of the PDAC algorithm and displayed as a bar graph. In these two figures we observe that step (3d) contributes most of the computational cost in the PDAC algorithm, and step (3c) contributes the second most of the computational cost, unless the number of chains is small. Further, the time for step (3d) grows the fastest as R gets large. These results agree with our performance analysis. It is noted that there is a jump for the total timing performance when R goes from 6 to 8 in Figure 3 (this also occurs when $N = 4$ about $R = 16$, but is not shown here). This occurs when the VP ratio, the number of virtual processor divided by number of physical processors, changes from 1 to two or more. The change in the VP ratio can affect performance, because the number of virtual processors assigned to a physical processor can affect the scheduling and load balancing of the processor computational resources.

The graphs of the total execution time performances, $exec - time_{PDAC}(R)$, are shown in Figure 4 and Figure 5. In Figure 4, we fix the number of nodes of the network to be 4. In Figure 5, we fix the number of nodes to be 5. A larger number of nodes problem can be executed on the 32K and 64K Connection Machine according to the actual problem size ($R = 23$ for $N = 5$ or $R = 59$ for $N = 4$ can be computed on an ACL 16K CM-2, without dedicated access to the full machine), but can not be further computed on the IBM 3090 due to memory limitations. Thus, we only compare the time performance curves of the DAC and the PDAC by using the problems that IBM 3090 can compute.

Using $O(R^{N-1})$ physical processors has reduced the number of operations in the above performance analysis. Since the execution time is proportional to the number of operations, it also leads to a reduction on the execution time. The execution time of the DAC algorithm, $exec - time_{DAC}(R)$, is $O(R^N)$ for fixed N and $R \gg 1$, while the execution time of the optimal PDAC algorithm, $exec - time_{PDAC}(R)$, should be $O(R^N)/R^{N-1} = O(R)$. However, due to the fact that the operations in the PDAC algorithm are not fully balanced but data dependent, PDAC algorithm has computational costs which are $O(R)$ in multiplications and $O(R^2 \log R)$ in additions according to the analysis. A fully balanced computation is one in

which all the processors have same amount of computational work, and all processors are active at the same time. The multiplication operations and addition operations take different execution time with respect to the same operand length. This causes the actual growth of the PDAC performances are somewhere between $O(R)$ and $O(R^2 \log R)$.

The significant improvements in the time performance are clear. However, the serial version of the DAC algorithm is better for a small number of chains. This is due to the fact networks with a small number of chains has much less computational requirements, and uniprocessor system out performs massively parallel machine in this situation. However, for the large number of chains, massively one-bit processors working together have more computational power than the IBM 3090. As we see from Figure 4 and Figure 5, the PDAC algorithm is faster with a very nearly quadratic growth in the total execution time. The total execution time of the DAC algorithm grows like $O(R^N)$.

6 Summary

We have presented a parallel implementation of the DAC algorithm based on the PDAC index generation algorithm. This new computational method is applied to multiple-chain, closed queue networks. The algorithm provides substantial savings in computational costs over a traditional serial implementation of DAC. The data parallel Connection Machine computation model is new to queueing theory and its applications, and will be useful in other computational problems.

References

[1] F. Baskett, K. M. Chandy, R. R. Muntz and F. G. Palacios, Open, Closed, and Mixed Networks of Queues with Different Classes of Customers, *J. ACM.* **22** (2) (1975) 248-260.

[2] G. E. Blelloch, *Vector Models for Data-Parallel Computing* (MIT Press, Cambridge, MA, 1990).

[3] J. P. Buzen, Computational Algorithms for Closed Queueing Networks with Exponential Servers, *Comm. ACM* **16** (9) (1973) 527-531.

[4] A. G. Greenberg and J. McKenna, Solution of Closed, Product form, Queueing Networks via the RECAL and Tree-RECAL Methods on a Shared Memory Multiprocessor, *ACM SIGMETRICS Performance Evaluation Review*, **17** (1) (May 1989) 127-135.

[5] A. G. Greenberg and I. Mitrani, Massively Parallel Algorithms for Network Partition Functions, in *Proc. 1991 Int. Conf. Par. Processing*, **3** (1991) 134-137.

[6] K. M. Chandy and C. H. Sauer, Computational Algorithms for the Product Form Queueing Networks, *Comm. ACM* **23** (10) (1980) 573-583.

[7] A. E. Conway and N. D. Georganas, RECAL - A New Efficient Algorithm for the Exact Analysis of Multiple-Chain Closed Queuing Networks, *J. ACM* **33** (4) (1986) 768-791.

[8] A. E. Conway, E. de Souza e Silva and S. S. Lavenberg, Mean Value Analysis by Chain of Product Form Queueing Networks, *IEEE Trans. Computers* **38** (3) (1989) 432-442.

[9] S. S. Lavenberg, *Computer Performance Modeling Handbook* (Academic Press, New York, 1983).

[10] K. K. Mathur and S. L. Johnsson, The finite element method on a data parallel computing system, *Int. J. High-Speed Comp.* **1** (1) (1989) 29-44.

[11] O. A. McBryan, The Connection Machine: PDE solution on 65536 processors, *Parallel Computing* **9** (1988) 1-24.

[12] J. J. Modi, *Parallel Algorithms and Matrix Computation* (Clarendon Press, Oxford, 1988).

[13] P. Olsson and S. L. Johnsson, A Dataparallel Implementation of an Explicit Method for the Three-Dimensional Compressible Navier-Stokes Equations, *Parallel Computing* **14** (1990) 1-30.

[14] M. Wolfe, *Optimizing Supercompilers for Supercomputers* (MIT Press, Cambridge, 1989).

[15] M. Reiser and S. S. Lavenberg, Mean-Value Analysis of Closed Multichain Queueing Networks, *J. ACM* **27** (2) (1980) 313-322.

[16] E. de Souza e Silva and S. S. Lavenberg, Calculating Joint Queue-Length Distributions in Product-Form Queueing Networks, *J. ACM* **36** (1) (1989) 194-207.

[17] L. W. Tucker and G. G. Robertson, Architecture and Applications of the Connection Machine, *IEEE Computer* **21** (8) (1988) 26-38.

[18] Thinking Machines Corporation, *CM Fortran Release Notes*, **0.7-f** 1990.