© 2015 IJSRSET | Volume 1 | Issue 5 | Print ISSN : 2395-1990 | Online ISSN : 2394-4099
Themed Section: Engineering and Technology

Iteration in Asynchronous System

Ajitesh S. Baghel, Rakesh Kumar Katare
Department of Computer Science A. P. S. Univesity, Rewa, Madhya Pradesh, India

ABSTRACT

We present a couple of traditional iterative strategies for unravelling straight comparisons; such routines are broadly utilized, particularly for the arrangement of substantial issues, for example, those emerging from the discrimination of direct fractional differential mathematical statements. We depict the iterative or backhanded systems, which begin from a rough guess to the genuine arrangement and if concurrent, infer a grouping of close estimates the cycle of reckonings being rehashed till the obliged precision is gotten.
Keyword: asynchronous system, P-RAM, MPI, Parallel, Distributed, Interconnection network.

I. INTRODUCTION

In this chapter, we present a couple of traditional iterative strategies for unraveling straight comparisons; such routines are broadly utilized, particularly for the arrangement of substantial issues, for example, those emerging from the discrimination of direct fractional differential mathematical statements. We depict the iterative or backhanded systems, which begin from a rough guess to the genuine arrangement and if concurrent, infer a grouping of close estimates the cycle of reckonings being rehashed till the obliged precision is gotten. It implies that in iterative routines the measure of processing relies on upon the precision obliged and we have additionally talked about JACOBI and Gauss-Seidel calculation with P-RAM and MPI Programming.

II. METHODS AND MATERIAL

A. Method Iteration

Given a distributed algorithm, for each processor, there is a set of times at which the processor executes some computations, some other times at which the processor sends some messages to other processors, and yet some other times at which the processor receives messages from other processors[4]. The algorithm is termed synchronous, in the sense of the Preceding subsection, if it is mathematically equivalent to one for which the times of computation, message transmission, and message reception are fixed and given a priori. We say that the algorithm is asynchronous if these times can vary widely in two different executions of the algorithm with an attendant effect on the results of the computation [4]. The most extreme type of asynchronous algorithm is one that can tolerate changes in the problem data or in the distributed computing system, without restarting itself to some predetermined initials conditions. Iterative methods, also known as trial and error methods, are based on the ideas of successive approximation. They start with one or more initial approximation to the root and obtain a sequence of approximations by repeating a fixed sequence of steps till the solution with reasonable accuracy is obtained. Iterative methods, generally, give one root at a time. Iterative methods are very cumbersome and time-consuming for solving non-linear equations manually. However, they are best suited for use on computers, due to following reasons:

- Iterative methods can be concisely expressed as computational algorithms.
- It is possible to formulate, using trial and error, algorithms which tackle a class of similar problems. For instance a general computational algorithm to solve polynomial equations of order n (where n is an integer) may be written.
- Routing errors are negligible in trial and error procedures compared to procedures based on closed form solutions.

In computational mathematics an iterative method is a mathematical procedure that generates a sequence of improving approximate solutions for a class of problems. A specific implementation of an iterative method, including the termination criteria, is an algorithm of the iterative method. An iterative method is called convergent if the corresponding sequence converges for given initial approximations. A mathematically rigorous convergence analysis of an iterative method is usually performed; however, heuristic-based iterative methods are also common.

A method uses iteration if it yields successive approximations to a required value by repetition of a certain procedure.

B. Iteration Steps

An "iterative" process can be explained by the flowchart given in Fig. 1. There are four parts in the process, namely, initialization, decision, computation and update. The functions of the four parts are as follows: [7]

1. **Initialization:** The parameters of the function and a decision parameter in this part are set to their initial values. The decision parameter is used to determine when to exit from the loop.
2. **Computation:** The required computation is performed in this part.
3. **Decision:** The decision parameter is to determine whether to remain in the loop.
4. **Update:** The decision parameter is updated, and a transfer to the next iteration results.

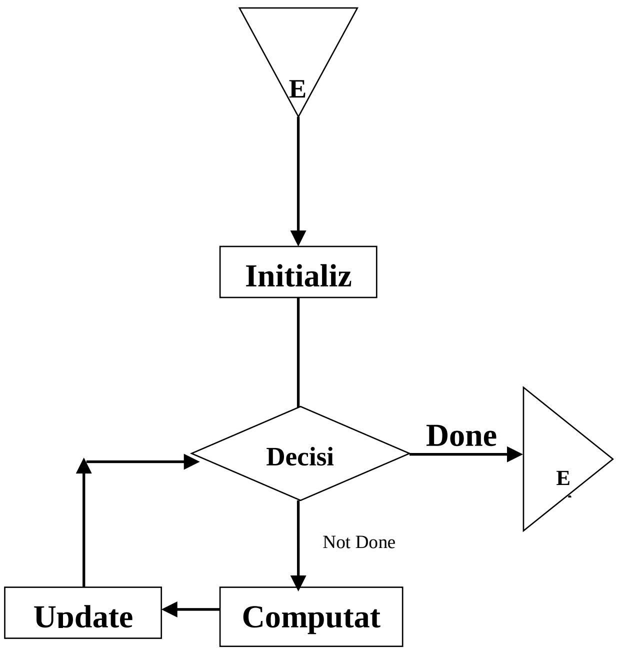

Figure: 1.1 Iteration Explain by flow chart [6 &7]

Lemma 1: Iteration is geometry of nature and its represent to geometry progration.

Proof:
```
for i = 1 to x
        for j= 1 to x
        y  = $x^2$
```
Where, x^2 is representation of geometry progration.

C. P-Ram Model

The P-RAM model allows parallel algorithm designers to treated processing power as an unlimited resource, much as programmers of computers with virtual memory are allowed to treat memory as an unlimited resource. The P-RAM model is unrealistically simple; it ignores the complexity of interprocessor communication. Because communication complexity is not an issue, the designer of P-RAM algorithms can focus on the parallelism inherent in a particular computation.

A P-RAM model consists of a control Unit, global memory, and an unbounded set of Processors, each with its own private memory (Fortune and Wyllie 1978) [25] (see figure: 1.1). Although active processors execute identical instructions, every processor has a unique index, and the value of a processor's index can be used to enable or disable the processor or influence which memory location it accesses.

A P-RAM computation begins with the input stored in global memory and a single active processing element. During each step of the computation an active, enabled processor may read a value from a single private or global memory location, perform a single RAM operation, and write into one local or global memory location. All active, enabled processors must execute the same instruction, albeit on different memory location. The computation terminates when the last processors halts.

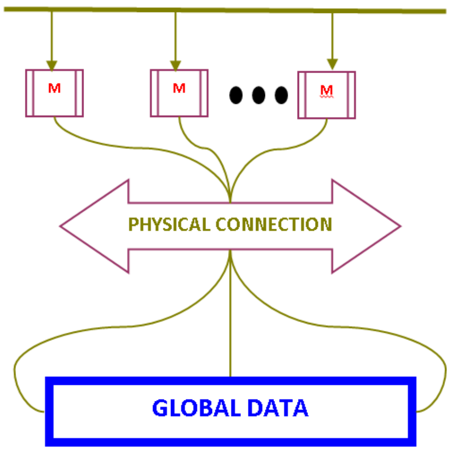

Figure: 1.2 Advance Representation of PRAM Model

Explanation: Let x_l, x_p, x_l be subsets of Euclidean spaces $R^n{}_i$,…., R^{np} respectively. Let $n = n_l + ….. + n_p$, and let $x \subset R^n$ be the Cartesian product $X = \Pi^p{}_{i=1}X_i$. Accordingly, any $x \in R^n$ is decomposed in the from $x = (x_i,…., x^p)$, we write each x_i belonging to $R^n{}_i$. For I 1…..p, let $f_i : X \rightarrow x_i$ be a given function and let $f : X \rightarrow X$ be the function defined by $f(x) = (f_i(x),……,f_p(x))$ for every $x \in X$. We want to solve the fixed - point problem $x = f(x)$. To this end we will consider the iteration
$x: = f(x)$

We will also consider the more general iteration-

$$x_1 = \begin{cases} f_i & if i \in l \\ x_i & \text{otherwise} \end{cases} \qquad (1.1)$$

Where i is a subset of the component index set {l,….., p}, which may change from one iteration to the next.
Let the system be given by

$$a_{ll}x_l + a_{l2}x_2 + a_{13}x_3 + …………+ a_{ln}x_n = b_l$$
$$a_{2l}x_l + a_{22}x_2 + a_{23}x_3 + ………..+ a_{2n}x_n = b_2$$
$$a_{3l}x_l + a_{32}x_2 + a_{33}x_3 + …………..+ a_{3n}x_n = b_3$$
$$…………..$$
$$…………..$$
$$a_{nl}x_l + a_{n2}x_2 + a_{n3}x_3 + ………….. + a_{nn}x_n \; b_n \qquad (1.2)$$

In which the diagonal elements a_{ij} do not vanish, if this is not the case, then the equation should be rearranged so that this condition is satisfied.
Now we can rewrite the above systems as follow -

$$x_2 = \frac{b_2}{a_{22}} - \frac{b_{21}}{a_{22}}x_1 - \frac{a_{23}}{a_{22}}x_3 - ……….….- \frac{a_{2n}}{a_{22}}x_n$$

$$x_3 = \frac{b_3}{a_{33}} - \frac{b_{3l}}{a_{33}}x_2 - \frac{a_{32}}{a_{33}}x_2 - ………..….- \frac{a_{3n}}{a_{33}}x_n$$

$$…………………………………………………………$$
$$…………………………………………………………$$
$$…………………………………………………………$$

$$x_n = \frac{b_n}{a_{nn}} - \frac{b_{nl}}{a_{nn}}x_2 - \frac{a_{n2}}{a_{nn}}x_2 - ………..….- \frac{a_{nn-1}}{a_{nn}}x_{n\text{-}1} \qquad (1.3)$$

Now, we can write the above equation in the form of matrix. Let A be a n*n matrix, let b be a vector in R^n, and consider the system of linear equations-
$Ax = b$

Where, x is an unknown vector to be determined. We assume that A is invertible, so that $Ax = b$ has a unique solution. We write the i^th equation of the systems $Ax = b$ as

$$\sum_{j=i}^{n} a_{ij}x_j b_i$$

Where a_{ij} are the entries of A; also, x_j and bi are the components of x and b, respectively, we assume that $a_{ii} \neq 0$ and solve for xi to obtain –

$$x_i = \frac{1}{a_{ij}}\left[\sum_{j=i} a_{ij}x_j - b_i\right] \qquad (1.4)$$

If all the components x_j ,$j \neq i$, of the solution of $Ax = b$ are known, the remaining component x_i can be determined from Eq.(1.4). If instead some approximate estimates for the components x_j, $j \neq i$, are available, then we can use Eq. (1.4) to obtain an estimate of x_i. This can be done for each component of x simultaneously, leading to the following algorithm:

- **Iteration in Jacobi Algorithm**

In this, we start with some initial vector $x(0) \in R^n$, evaluate x(t), t = 1,2, ….. using the iteration -

$$x_i(t+1) = -\frac{1}{a_{ii}}\left[\sum_{j=i} a_{ij}x_j(t) - b_i\right] \qquad (4.5)$$

The Jacobi algorithm produces an infinite sequence {x(t)} of elements of R^n. If this sequence converges to a limit x, then by taking the limit of both sides of Eq. (1.5) as i tend to infinity, we see that x satisfies Eq. (1.4) for each i, which is equivalent to x being a solution of Ax = b of course; it is possible that the algorithm diverges.

In the above algorithm, each component of x(t + 1) was evaluated based on Eq. (1.4) and the estimate x(t) of the solution. If this algorithm is executed on a serial computer, by the time that x_i (t + 1) is evaluated, we already have available some new estimates x_j (t + 1) for the components of x with index j smaller than i. It may be preferable to employ these new estimates of x_j, j < i when updating x_i. This leads to the next (gauss-seidel) algorithm. The above Jacobi method can be explained easily with this following example:

Example: Find the solution, using Jacobi method to three decimals, of systems.

83x + 11y - 4z = 95 (1)

7x + 52y + 13z =104 (2)

3x + 8y + 29z =7 (3)

The above equation (1) can be written as follows;

83x =95 - 11y + 4z

$$x = \frac{95}{83} - \left(\frac{11}{83}\right) + (4/83)z$$

$$x_{n+1} = \frac{1}{83\{95 - 11y_n + 4z_n\}} \quad (4)$$

The equation (2) also may be written as follows:

52y = 104 - 7x - 13z}

$$y_{n+1} = \frac{1}{52\{104 - 7x_n - 13z_n\}} \quad (5)$$

The equation (3) may be written are as follows:

29z =71 - 3x - 8y

$$z_{n+1} = \frac{1}{29\{71 - 3x_n - 8y_n\}} \quad (6)$$

Now we take initial values of x, y and z, so take initial values;

$x_0 = y_0 = z_0 = 0$

Now we calculate the first iteration:

Iteration – I:

$$x_{n+1} = \frac{1}{83\{95 - 11y_n + 4z_n\}}$$

(4)

Here n = 0, so

$$x_l = \frac{1}{83\{95 - 11*0 + 4*0\}}$$

$$= \frac{95}{83}$$

x_l =1.1445783

y_l = 1/52{104 - 7 * 0 - 13 * 0}

$$= \frac{104}{52}$$

= 2

$$z_1 = \frac{1}{29\{71 - 3*0 - 8*0\}}$$

= 71/29

= 2.4482758

Now we calculate the second iteration, in this we use the recent value of x_l, y_l and z_l.

Iteration – II:

Then here n = 1, so

x_2 = 1/83{95 – 11y_l + 4z_1}

= 1/83{95 - 11 * 2 + 4 * 2.4482758}

x_2 =.9975072

y_2 =1/52{104 – 7$x_1$13z_1}

$$= \frac{1}{52\{104 - 7*1.1445783 - 13*2.4482758\}}$$

= 1.2338532

z_2 =1/29{71 – 3x_l – 8y_l}

=1/29{71 - 3 * 1.1445783 - 8 * 2}

=1.77814707

Iteration – III:

Here n =2

x_3 =1/83{95 – 11y_2 + 4z_2}

$$= \frac{1}{83\{95 - 11*1.2338532 + 4*1.77814707\}}$$

= 1.0667494

y_3 =1/52{104 - 7x_z - 13z_z}

= 1/52{104 - 7 * .9975072 - 13 * 1.77814707}

= 1.4211834

z_3 =1/29{71 - 3xz - 8yz}

=1/29{71 - 3 * .9975072 - 8 * 1.2338532}

=2.0047121

Now, the value of three x, y and z are repeated so we may stop.

So, the value of x, y and z are as follows:

x =1.057

y =1.367

z =1.961

Iteration – IV:
Here n=3
x_4 =1/83{95 – 11y_3 + 4z_3}
=1/83{95 - 11 * 1.4211834 + 4 * 2.0047121}
= 1.0528413
y_4 = 1/52{104 – 7x_3 - 13z_3}
= 1/52{104 - 7 * 1.0667494 - 13 * 2.0047121}
= 1.35522109
z_4 = 1/29{71 – 3x_3 – 8y_3}
=1/29{71 - 3 * 1.0667494 - 8 * 1.4211834}
= 1.9458718

Iteration – V:
Now n = 4
x_s =1/83{95 – 11y_4 + 4z_4}
=1/83{95 - 11 * 1.35522109 + 4 * L9458718}
=1.0587476
y_5 =1/52{l04 – 7x_4 - 13z_4}
= 1/52{104 - 7 * 1.0528413 - 13 * 1.9458718}
= 1.3718034
z_5 = 1/29{71 - 3x_4 – 8y_4}
=1/29{71 - 3 * 1.0528413 - 8 * 1.35522109}
= 1.9655071

The ***pseudo-code*** of sequential Jacobi algorithm is as follows: [M.J. Quinn, 1994]

```
Input

n {size of linear system}
∈ {convergence criterion}
a[l ...n][l…..n] {coefficient of linear equation}
b[l…..n] {constant associated with equation}
Output
x[l...n]{old Estimate of solution vector}
Global
new x[l...n] {new estimate of solution vector}
diff{maximum change of any element of solution}
i,j {loop indices}
Begin
{Estimate values of elements of x}
for i ← 1 to n do
x[j] ← $\frac{b[1]}{a[i][i]}$
end for
{Refine estimates of x until value converge}
do
diff ← a
for i ← 1 to n do
new x[i] ← b[i]
for j ← 1 to n do
if j ≠ then
new x[i] ← newx[i] - a[i] [j] * x[j]
endif
endfor
newx[i] ← $\frac{newx[i]}{a[i][j]}$
endfor
for i ← 1 to n do
diff ← max (diff, [x[i] - newx[i])
x[i] ←newx[i]
endfor
while diff > ∈
end
```

Hence, it is a sequential implementation of the Jacobi Algorithm.

○ **Iteration in Gauss-Seidel Algorithm:**

Starting with some initial vector xe(0) ∈ R^n, evaluate x(t), t = 1,2, ... using the iteration-

$$x_i(t+1) = -\frac{1}{aii}\left[\sum_{j<i} a_{ii}x_j(t+1) + \sum_{j>i} a_{ii}x_j(t) - b_i\right] \quad (3)$$

In above equation, we first update x_l, then x_2, etc. It is equally meaningful to start by updating x_n, then x_{n-1} and proceed backwards, with x_l being updated last. Any other order of updating is possible. Different orders of updating may produce substantially different results for the same system of equation.

The ***pseudo-code*** of Sequential Gauss Seidel method is as follow:

```
Sequential_GS
input A, b, $x^{(0)}$, tolerance
for k = 0 to k_max do the following
for i =1….., n
        sum = 0
for j = 1,2,.... i - 1
        sum = sum + $a_{ij}$ $x_j^k$
end j
```

```
        $x_i^{(k+l)} = (b_i - sum)/a_{ii}$
    end i
    if $||x^{(k+l)} - x^{(k)}|| <$ tolerance then output the solution, stop
end k
end Sequential_GS
```

Hence, it is a sequential implementation of Gauss-Seidel method.

D. Message Passing

It is a concept from computer science, i.e. used extensively in the design and implementation of modern software applications. This concept is used with software and hardware both. Generally, message passing is the indication of passes message from n different nodes, by wired or wireless medium. Another words, it is a way of invoking behavior through some intermediary service or infrastructure of process. According to the concepts of this, when more then to autonomous machines, which are intermediary connected with each others, and passes bundles and packets through established channel or link this think is known as *"Message Passing"*.

E. Discrete Vs Continuous Massege Passing

We explain the discrete and continuous message passing, as follows:

- **Discrete Message Passing:** It is possible for the receiving object to be busy or not running when the requesting object sends the message. It requires additional capabilities for storing and retransmitting data for systems that may not run concurrently. In this, all the capabilities that naturally occur when trying to synchronize system and data are handled by an intermediary level of software. With discrete message passing the sending system does not wait for a response. It simply sends the data bus and the buses stored the message and returns the result when it is available.
- **Continuous Message Passing:** In this, message passing occurs between objects that are running at the same time. It based on typically object-oriented programming, such as: JAVA and Smalltalk. **Message Passing:** It is less complex; the sender sends a message and gets a response the same as simply invoking a function or procedure call. Continuous systems require the sender and receiver to wait for each other to transfer the message.

III. RESULTS AND DISCUSSION

A. Message Passing Models

The message passing technologies have various types of modes. Either some are conceptual or some are practical. Here we explain several models of MPI as per my knowledge.

- **Mathematical Model:**

There are two prominent mathematical models of message passing, as:

1. **Actor model:** This model was inspired by physics (include relativity and quantum physics). It was also influenced by the programming languages like as: LISP, Simula63 and Smalltalk[2]. Its development was "motivated by the prospect of highly parallel computing machines consisting of dozens, hundreds or even thousands of independent microprocessors, each with its own local memory and communications processor, communicating via a high-performance communications network [2]. The actor model in computer science is a mathematical model of concurrent computation that treats "actors" as the universal primitives of concurrent computation, in response to a message that it receives; an actor can make local decisions, create more actors, send more messages, and determine how to respond to the next message received. The actor model originated in 1973 [1]. It has been used both as a framework for a theoretical understanding of computation and as the theoretical basis for several practical implementations of concurrent systems. An actor is a computational entity that, in response to a message it receives, can concurrently:
   - ✓ Send a finite number of messages to other actors.
   - ✓ Create a finite number of new actors.
   - ✓ Designate the behavior to be used for the next message it receives.

   There is no assumed sequence to the above actions and they could be carried out in parallel. The Actor model enabling asynchronous communication and control structures as patterns of passing messages [3]. Recipients of messages are identified by address, sometimes called "mailing address". Thus an actor can only communicate with actors whose port addresses it has. The Actor model is characterized by inherent concurrency of computation within and among actors, dynamic creation of actors, inclusion of actor addresses in messages, and interaction only through direct asynchronous message passing with no restriction on message arrival order.

2. **Pi Calculus:** In theoretical computer science, the **π-calculus** (or **pi-calculus**) is a process calculus. The π-calculus allows channel names to be communicated along the channels themselves, and in this way it is able to describe concurrent computations whose network configuration may change during the computation[3]. The π-calculus is elegantly simple clarification is needed yet very expressive[1]. Functional programs can be encoded into the π-calculus, and the encoding emphasizes the dialogue nature of computation, drawing connections with game semantics. Extensions of the π-calculus, such as the **π-calculus** and applied π, have been successful in reasoning about cryptographic protocols.

   Definition: The π-calculus belongs to the family of process calculi, mathematical formalisms for describing and analyzing properties of concurrent computation. In fact, the π-calculus, like the λ-calculus, is so minimal that it does not contain primitives such as numbers, Booleans, data structures, variables, functions, or even the usual control flow statements (such as if-then-else, while).

- **B. Parallel Processing Model For Distributed System**

In here, machine architecture represents the programming model, we explain figure 4.3 in our words.

- ➢ Each processor Pi has its own memory and clock.
- ➢ Local memory is not accessible by anywhere through the other processors.
- ➢ All processors Pi are connected by a special physical medium i.e., either network of communication or other communication device which are available in present scenario.

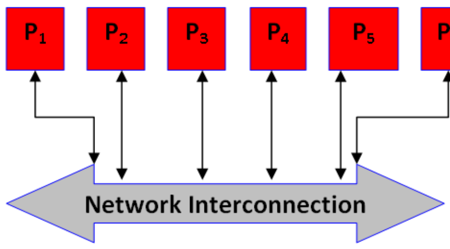

Figure: 1.3 Parallel MPI for Distributed System In architecture

Data and information must be explicitly distributed by the programmer; Communication of processors (i.e., exchanging data's in between processors) is achieved by MPI.

According to above architecture we refers pseudo-code for adding two vector by OPEN MPI:

Source code:

```
#include "mpi.h" /* Include MPI header file */!
int main (int argc, char **argv)
{
int rnk, sz, n, I, info;
double *x, *y, *buff;
n = atoi (argv[1]);  /* Get input size */!
/* Initialize threaded MPI environment */
MPI_Init_thread (&argc, &argv, MPI_THREAD_FUNNELED, &info);
MPI_Comm_size(MPI_COMM_WORLD, &sz); /* Find out how many
MPI procs */
chunk = n / sz; /* Assume sz divides n exactly */
MPI_Scatter(buff,chunk,MPI_DOUBLE,x,chunk,MPI_DOUBLE,0,MPI_
COMM_WORLD);
MPI_Scatter(&buff[n],chunk,MPI_DOUBLE,y,chunk,MPI_DOUBLE,0,M
PI_COMM_WORLD);
#pragma omp parallel for private(i,chunk) shared(x, y)
for (i=0; i<chunk; i++) x[i] = x[i] + y[i];!
MPI_Gather(x,chunk,MPI_DOUBLE,buff,chunk,MPI_DOUBLE,0,MPI_
COMM_WORLD);
MPI_Finalize();
}
```

B. Representation of Array Pattern of Processing Elements (P.E.): [5,8]

Consider a case of three dimensional array patterns with $n^3 = 2^{3q}$ (Processing Elements) PEs [E.D. Dekel, Nassimi, S. Sabni, M.J. Quinn.]. Conceptually this PEs may be regarded as arranged, in n × n × n array pattern. If we assume that the PEs are row major order, the PE (i,j, k) in position (i,j, k) of this array has 2 index $in^2 + jn + k$ (note that array indices are in the range [0, (n - 1)]. Hence, if r_{3q-1} r_0 is the binary representation of the PE position (i,j, k) then $i = r_{3q-1} \ldots r_{zq}$, $j = r_{zq-1} \ldots r_{q,k} = r_{q-1} \ldots r_0$ using A(i,j, k), B(i,j, k) and C(i,j, k) to represent memory locations in P(i,j, k), we can describe the initial condition for matrix multiplication as A(0,j, k) = A_{jk}, B(0,j, k) = B_{jk}, 0 < = j < k,0 <= k < n A_{jk} and B_{jk} are the elements of the two matrices to be multiplied. The desired final configuration is C(0,j, k) = CU, k), 0 <= j < n,0 <= k < n

Where,

$C_{jk} = \sum_{i=0}^{n=1} A_{ij}B_{ik}$ ... ...

The algorithm has three distinct phases. In the first, element of A & B are distributed over the n PEs so that we have A(l,), k) = A and B(l,j, k) = B. In the second phase the products C(l,j, k) = A(l,), k) * B(l,j, k) = A_nB_n are computed. Finally, in third phase the C_{jk} are computed. The details are spelled out in Dekel, Nassimi and Sahni 1981 [E.D. Dekel, Nassimi, S. Sabni,]. In this procedure all PE references are by PE index (Recall that the index of PE (i, j,k) as in + j_n + k).

- **P-RAMf Asyncheonous System: [5,8]**

```
Begin (1)
Repeat log n times do
for all (ordered) pair (i,j, k), 0 < k ≤ n, 0 < i,j, k ≤ n and q = log n in parallel do
a(22q i + 2qj + k) = a(i, j)
a(22qi + zqj + k) = a(i, i)
b(22qi + zqj + k) = b(i)
end for
for all (order)pair (i,), k), i < k < n, i > 0,j > i and q = log n
x(j) = $\frac{b[i]}{a[i][i]}$
end for
[Refine estimates of x untill value converge]
Repeat log n times do
diff = 0
for all (order)pair (i,), k), i < k < n, i > 0,j > i and q = log n
new x[i] =b[i]
for all (order)pair (i,), k), i < k < n, i > 0,j> i and q = log n
if j ≠ i then
new [x] =new x[i] - a[i][j] * x[i]
end if
end for new x [i] = new $\frac{x[i]}{a[i][i]}$
end for
for all (order)pair (i,j, k), i < k < n, i > 0,j > i and q = log n
diff = max(diff, [x[i] - new x[i]])
x[i] = new x[i]
endfor
while diff > ∊
end
```

- **MPI For Asynchronous System:**

Here, we use MPI for asynchronous system as a terms of again matrix multiplication in different manner. The pseudo-code are given below:

```
/* MATRIX MULTIPLICATION PROGRAM USING MPI*/
#include<stdio.h>
#include<conio.h>
#define NUM_ROWS_A 4
#define NUM_COLUMN_A 4
#define NUM_ROWS_B 4
#define NUM_ COLUMN _B 4
#define MASTER_TO_SLAVE_TAG 1
#define SLAVE_TO_MASTER_TAG 4
void make AB ( );
void print Array( );
int rank;
int size;
int i, j;
double mat_a [NUM_ROWS_A] [NUM_COLUMN_A]
double mat_b [NUM_ROWS_B] [NUM_COLUMN_B]
double mat_result [NUM_ROWS_A] [NUM_COLUMN_B]
double start_time;
double end_time;
int low_bound;
int upper_bound;
int portion;
MPI_Status status;
MPI_Request request;
int main (int argc, char *argv[ ])
{
MPI_Init(&argc, &argv);
MPI_comm_rank(MPI_comm_word, &size);
/*MASTER INITIALIZES WORK*/
if (rank == 0)
{
Make AB( );
Start_time = MPI_wtime( );
for(i= 1; i< size; i++)
portion = (NUM_ROWS_A)/(size-1);
low_bound = (i-1) * portion;
if (((i+1) == size) && ((Num_ROWS_A%(size-1)) ! =0))
{
upper_bound = NUM_ROWS_A;
}
else
{
upper_bound = low_bound  + portion;
}
MPI_isend(&low_bound,1,MPI_INT,i,MASTER_TO_SLAVE_TAG,MPI
_COMM_WORLD, &request);
MPI_isend (&mat_a [low_bound] [0], (upper_bound - low_bound) *
NUM_COLUMNS_A,MPI_DOUBLE,i,MASTER_TO_SLAVE_TAG+2,M
PI_COMM_WORLD, &request);
}
}
MPI_Bcast(& mat_b, NUM_ROWS_B * NUM_COLUMN_B,
MPI_DOUBLE,0,MPI_COMM_WORLD);
if (rank>o)
{
MPI_Recv(&low_bound,1,MPI_INT,i,MASTER_TO_SLAVE_TAG,MPI
_COMM_WORLD, &status);
MPI_Recv(&upper_bound,1,MPI_INT,i,MASTER_TO_SLAVE_TAG,M
PI_COMM_WORLD, &status);
MPI_isend (&mat_a [low_bound] [0], (upper_bound - low_bound) *
NUM_COLUMNS_A,MPI_DOUBLE,0,MASTER_TO_SLAVE_TAG+2,
MPI_COMM_WORLD, &status);
for(i= low_bound; i< upper_bound; i++)
{
for(j= 0; j< NUM_COLUMNS_B; j++)
mat_result[i][j]+= (mat_a [i][j] * mat_b [i][j]  );
}
}
}
MPI_isend(&low_bound,1,MPI_INT,0,SLAVE_TO_MASTER_TAG,MPI
_COMM_WORLD, &request);
MPI_isend(&upper_bound,1,MPI_INT,0,SLAVE_TO_MASTER_TAG+
1,MPI_COMM_WORLD, &request);
MPI_isend (&mat_result [low_bound] [0], (upper_bound - low_bound) *
NUM_COLUMNS_B,MPI_DOUBLE,0,SLAVE_TO_MASTER_TAG+2,
MPI_COMM_WORLD, &request);
}
/*MASTER GATHERS PROCESSED WORK*/
if (rank == 0)
{
for(i= 1; i< size; i++)
MPI_Recv(&low_bound,1,MPI_INT,i,SLAVE_TO_MASTER_TAG+1,M
PI_COMM_WORLD, &status);
MPI_Recv(&upper_bound,1,MPI_INT,i,SLAVE_TO_MASTER_TAG+1,
MPI_COMM_WORLD, &status);
MPI_Recv (&mat_result [low_bound] [0], (upper_bound - low_bound) *
NUM_COLUMNS_B,MPI_DOUBLE,i,SLAVE_TO_MASTER_TAG+2,M
PI_COMM_WORLD, &request);
}
end_time – MPI_wtime( );
printf(“\n Runing Time = %f\n\n”, end_time_start_time);
print Arrray( );
}
MPI_Finalize( );
return 0;
}
void make AB ( );
{
for(i= 0; i< NUM_ROWS_A; i++)
{
for(j= 0; j< NUM_COLUMNS_A; j++)
{
Mat_a[i][j] = i+j;
}
}
for(i= 0; i< NUM_ROWS_B; i++)
{
for(j= 0; j< NUM_COLUMNS_B; j++)
{
Mat_b[i][j] = i+j;
}
}
}
void printarray( );
{
for(i= 0; i< NUM_ROWS_A; i++)
{
printf(“\n”);
for(j= 0; j< NUM_COLUMNS_A; j++)
printf(“%8.2f”, mat_a[i][j]);
}
printf(“\n\n\n”);
for(j= 0; j< NUM_COLUMNS_B; j++)
printf(“%8.2f”, mat_b[i][j]);
}
printf(“\n\n\n”);
for(i= 0; i< NUM_ROWS_A; i++)
{
printf(“\n”);
for(j= 0; j< NUM_COLUMNS_B; j++)
printf(“%8.2f”, mat_result[i][j]);
}
printf(“\n\n”);
}
```

IV. CONCLUSION

In this, we examine circulated calculation, for every processor, there is a situated of times at which the processor executes a few processing’s, some different times at which the processor sends a few messages to different processors, but some different times at which the processor gets messages from different processors. And, also represented message passing between gobal host to local host through interconnected physical medium.

V. REFERENCES

[1] Carl Hewitt; Peter Bishop; Richard Steiger (1973). "A Universal Modular Actor Formalism for Artificial Intelligence". IJCAI.

[2] William Clinger (June 1981). "Foundations of Actor Semantics". Mathematics Doctoral Dissertation. MIT.

[3] Carl Hewitt. Viewing Control Structures as Patterns of Passing Messages Journal of Artificial Intelligence. June 1977.

[4] A Calculus of Mobile Processes part 1 page 10, by R. Milner, J. Parrow and D. Walker published in Information and Computation 100(1) pp.1-40, Sept 1992.

[5] Katare, R.K., Chaudhari, N.S., “Study of topological property of interconnection network and its mapping to sparse matrix model”.

[6] Tremblay, J.P. and Manohar, R, “Discrete Mathematical Structure with Applications to Computer Science”. 1997.

[7] Fortune,S; and J. Wyllie. 1978 Parallelelism in random access machines, proceedings of the 10thAnnual ACM Symposium on theory of computing, PP, 114-118.

[8] https//en.wikipedia.oer/wiki/could_computing

[9] Katare, R.K., Chaudhari, N.S., “Acomparative study of Hypercube and perfect difference network for Parallel and distributed system and its application to sparse linear system.” Varahmihir journal of computer and information sciences volume 2, Sandipani Academy, Ujjain, (M.P.) India, p. 13-30,2007

[10] Katare, R.K., Chaudhari, N.S., “An attempt to map sparse linear system on perfect difference network for Parallel and distributed system.” Varahmihir journal of computer and information sciences volume 2, Sandipani Academy, Ujjain, (M.P.) India, p. 1-10,2008