the messages can be the same or different. The safe version of our earlier example using MPI_Sendrecv is as follows.

```
1    int a[10], b[10], npes, myrank;
2    MPI_Status status;
3    ...
4     MPI_Comm_size(MPI_COMM_WORLD, &npes);
5     MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
6    MPI_SendRecv(a, 10, MPI_INT, (myrank+1)%npes, 1,
7                  b, 10, MPI_INT, (myrank-1+npes)%npes, 1,
8                 MPI_COMM_WORLD, &status);
9    ...
```

In many programs, the requirement for the send and receive buffers of MPI_Sendrecv be disjoint may force us to use a temporary buffer. This increases the amount of memory required by the program and also increases the overall run time due to the extra copy. This problem can be solved by using that MPI_Sendrecv_replace MPI function. This function performs a blocking send and receive, but it uses a single buffer for both the send and receive operation. That is, the received data replaces the data that was sent out of the buffer. The calling sequence of this function is the following:

```
int MPI_Sendrecv_replace(void *buf, int count,
        MPI_Datatype datatype, int dest, int sendtag,
        int source, int recvtag, MPI_Comm comm,
        MPI_Status *status)
```

Note that both the send and receive operations must transfer data of the same datatype.

6.3.5 Example: Odd-Even Sort

We will now use the MPI functions described in the previous sections to write a complete message-passing program that will sort a list of numbers using the odd-even sorting algorithm. Recall from Section 9.3.1 that the odd-even sorting algorithm sorts a sequence of n elements using p processes in a total of p phases. During each of these phases, the odd-or even-numbered processes perform a compare-split step with their right neighbors. The MPI program for performing the odd-even sort in parallel is shown in Program 6.1 . To simplify the presentation, this program assumes that n is divisible by p .

Program 6.1 Odd-Even Sorting

[View full width]

```
 1  #include <stdlib.h>
 2  #include <mpi.h> /* Include MPI's header file */
 3
 4  main(int argc, char *argv[])
 5  {
 6    int n;         /* The total number of elements to be sorted */
 7    int npes;      /* The total number of processes */
 8    int myrank;    /* The rank of the calling process */
 9    int nlocal;    /* The local number of elements, and the array that stores th
10    int *elmnts;   /* The array that stores the local elements */
11    int *relmnts;  /* The array that stores the received elements */
```

```
12    int oddrank;   /* The rank of the process during odd-phase communication */
13    int evenrank;  /* The rank of the process during even-phase communication */
14    int *wspace;   /* Working space during the compare-split operation */
15    int i;
16    MPI_Status status;
17
18    /* Initialize MPI and get system information */
19    MPI_Init(&argc, &argv);
20    MPI_Comm_size(MPI_COMM_WORLD, &npes);
21    MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
22
23    n = atoi(argv[1]);
24    nlocal = n/npes; /* Compute the number of elements to be stored locally. */
25
26    /* Allocate memory for the various arrays */
27    elmnts  = (int *)malloc(nlocal*sizeof(int));
28    relmnts = (int *)malloc(nlocal*sizeof(int));
29    wspace  = (int *)malloc(nlocal*sizeof(int));
30
31    /* Fill-in the elmnts array with random elements */
32    srandom(myrank);
33    for (i=0; i<nlocal; i++)
34      elmnts[i] = random();
35
36    /* Sort the local elements using the built-in quicksort routine */
37    qsort(elmnts, nlocal, sizeof(int), IncOrder);
38
39    /* Determine the rank of the processors that myrank needs to communicate dur
➥ the */
40    /* odd and even phases of the algorithm */
41    if (myrank%2 == 0) {
42      oddrank  = myrank-1;
43      evenrank = myrank+1;
44    }
45    else {
46      oddrank  = myrank+1;
47      evenrank = myrank-1;
48    }
49
50    /* Set the ranks of the processors at the end of the linear */
51    if (oddrank == -1 || oddrank == npes)
52      oddrank = MPI_PROC_NULL;
53    if (evenrank == -1 || evenrank == npes)
54      evenrank = MPI_PROC_NULL;
55
56    /* Get into the main loop of the odd-even sorting algorithm */
57    for (i=0; i<npes-1; i++) {
58      if (i%2 == 1) /* Odd phase */
59        MPI_Sendrecv(elmnts, nlocal, MPI_INT, oddrank, 1, relmnts,
60            nlocal, MPI_INT, oddrank, 1, MPI_COMM_WORLD, &status);
61      else /* Even phase */
62        MPI_Sendrecv(elmnts, nlocal, MPI_INT, evenrank, 1, relmnts,
63            nlocal, MPI_INT, evenrank, 1, MPI_COMM_WORLD, &status);
64
65      CompareSplit(nlocal, elmnts, relmnts, wspace,
```

```
66                    myrank < status.MPI_SOURCE);
67    }
68
69     free(elmnts); free(relmnts); free(wspace);
70    MPI_Finalize();
71  }
72
73  /* This is the CompareSplit function */
74   CompareSplit(int nlocal, int *elmnts, int *relmnts, int *wspace,
75                int keepsmall)
76  {
77    int i, j, k;
78
79    for (i=0; i<nlocal; i++)
80      wspace[i] = elmnts[i]; /* Copy the elmnts array into the wspace array */
81
82    if (keepsmall) { /* Keep the nlocal smaller elements */
83      for (i=j=k=0; k<nlocal; k++) {
84        if (j == nlocal || (i < nlocal && wspace[i] < relmnts[j]))
85          elmnts[k] = wspace[i++];
86        else
87          elmnts[k] = relmnts[j++];
88      }
89    }
90    else { /* Keep the nlocal larger elements */
91       for (i=k=nlocal-1, j=nlocal-1; k>=0; k--) {
92         if (j == 0 || (i >= 0 && wspace[i] >= relmnts[j]))
93           elmnts[k] = wspace[i--];
94         else
95           elmnts[k] = relmnts[j--];
96      }
97    }
98  }
99
100 /* The IncOrder function that is called by qsort is defined as follows */
101 int IncOrder(const void *e1, const void *e2)
102 {
103   return (*((int *)e1) - *((int *)e2));
104 }
```


PREVIOUS NEXT

```
14
15      nlocal = n/npes;
16
17      /* Allocate memory for arrays storing intermediate results. */
18      px = (double *)malloc(n*sizeof(double));
19      fx = (double *)malloc(n*sizeof(double));
20
21      /* Compute the partial-dot products that correspond to the local columns of
22      for (i=0; i<n; i++) {
23        px[i] = 0.0;
24        for (j=0; j<nlocal; j++)
25          px[i] += a[i*nlocal+j]*b[j];
26      }
27
28      /* Sum-up the results by performing an element-wise reduction operation */
29      MPI_Reduce(px, fx, n, MPI_DOUBLE, MPI_SUM, 0, comm);
30
31      /* Redistribute fx in a fashion similar to that of vector b */
32      MPI_Scatter(fx, nlocal, MPI_DOUBLE, x, nlocal, MPI_DOUBLE, 0,
33          comm);
34
35      free(px); free(fx);
36    }
```

Comparing these two programs for performing matrix-vector multiplication we see that the row-wise version needs to perform only a `MPI_Allgather` operation whereas the column-wise program needs to perform a `MPI_Reduce` and a `MPI_Scatter` operation. In general, a row-wise distribution is preferable as it leads to small communication overhead (see Problem 6.6). However, many times, an application needs to compute not only Ax but also $A^T x$. In that case, the row-wise distribution can be used to compute Ax , but the computation of $A^T x$ requires the column-wise distribution (a row-wise distribution of A is a column-wise distribution of its transpose A^T). It is much cheaper to use the program for the column-wise distribution than to transpose the matrix and then use the row-wise program. We must also note that using a dual of the all-gather operation, it is possible to develop a parallel formulation for column-wise distribution that is as fast as the program using row-wise distribution (see Problem 6.7). However, this dual operation is not available in MPI.

6.6.9 Example: Single-Source Shortest-Path

Our second message-passing program that uses collective communication operations computes the shortest paths from a source-vertex s to all the other vertices in a graph using Dijkstra's single-source shortest-path algorithm described in Section 10.3 . This program is shown in Program 6.6.

The parameter `n` stores the total number of vertices in the graph, and the parameter `source` stores the vertex from which we want to compute the single-source shortest path. The parameter `wgt` points to the locally stored portion of the weighted adjacency matrix of the graph. The parameter `lengths` points to a vector that will store the length of the shortest paths from `source` to the locally stored vertices. Finally, the parameter `comm` is the communicator to be used by the MPI routines. Note that this routine assumes that the number of vertices is a multiple of the number of processors.

Program 6.6 Dijkstra's Single-Source Shortest-Path

[View full width]

```
 1   SingleSource(int n, int source, int *wgt, int *lengths, MPI_Comm comm)
 2   {
 3     int i, j;
 4     int nlocal;  /* The number of vertices stored locally */
 5     int *marker;  /* Used to mark the vertices belonging to V_o */
 6     int firstvtx;  /* The index number of the first vertex that is stored locall
 7     int lastvtx;  /* The index number of the last vertex that is stored locally
 8     int u, udist;
 9     int lminpair[2], gminpair[2];
10     int npes, myrank;
11     MPI_Status status;
12
13     MPI_Comm_size(comm, &npes);
14     MPI_Comm_rank(comm, &myrank);
15
16     nlocal   = n/npes;
17     firstvtx = myrank*nlocal;
18     lastvtx  = firstvtx+nlocal-1;
19
20     /* Set the initial distances from source to all the other vertices */
21     for (j=0; j<nlocal; j++)
22       lengths[j] = wgt[source*nlocal + j];
23
24     /* This array is used to indicate if the shortest part to a vertex has been
   or not. */
25     /* if marker [v] is one, then the shortest path to v has been found, otherwi
   has not. */
26     marker = (int *)malloc(nlocal*sizeof(int));
27     for (j=0; j<nlocal; j++)
28       marker[j] = 1;
29
30     /* The process that stores the source vertex, marks it as being seen */
31     if (source >= firstvtx && source <= lastvtx)
32       marker[source-firstvtx] = 0;
33
34     /* The main loop of Dijkstra's algorithm */
35     for (i=1; i<n; i++) {
36       /* Step 1: Find the local vertex that is at the smallest distance from sou
37       lminpair[0] = MAXINT; /* set it to an architecture dependent large number
38       lminpair[1] = -1;
39       for (j=0; j<nlocal; j++) {
40         if (marker[j] && lengths[j] < lminpair[0]) {
41           lminpair[0] = lengths[j];
42           lminpair[1] = firstvtx+j;
43         }
44       }
45
46       /* Step 2: Compute the global minimum vertex, and insert it into V_c */
47        MPI_Allreduce(lminpair, gminpair, 1, MPI_2INT, MPI_MINLOC,
48           comm);
49       udist = gminpair[0];
50       u = gminpair[1];
51
52       /* The process that stores the minimum vertex, marks it as being seen */
```

```
53        if (u == lminpair[1])
54          marker[u-firstvtx] = 0;
55
56        /* Step 3: Update the distances given that u got inserted */
57        for (j=0; j<nlocal; j++) {
58          if (marker[j] && udist + wgt[u*nlocal+j] < lengths[j])
59            lengths[j] = udist + wgt[u*nlocal+j];
60        }
61      }
62
63      free(marker);
64    }
```

The main computational loop of Dijkstra's parallel single-source shortest path algorithm performs three steps. First, each process finds the locally stored vertex in V_o that has the smallest distance from the source. Second, the vertex that has the smallest distance over all processes is determined, and it is included in V_c . Third, all processes update their distance arrays to reflect the inclusion of the new vertex in V_c .

The first step is performed by scanning the locally stored vertices in V_o and determining the one vertex *v* with the smaller *lengths* [*v*] value. The result of this computation is stored in the array *lminpair* . In particular, *lminpair* [0] stores the distance of the vertex, and *lminpair* [1] stores the vertex itself. The reason for using this storage scheme will become clear when we consider the next step, in which we must compute the vertex that has the smallest overall distance from the source. We can find the overall shortest distance by performing a min-reduction on the distance values stored in *lminpair* [0]. However, in addition to the shortest distance, we also need to know the vertex that is at that shortest distance. For this reason, the appropriate reduction operation is the `MPI_MINLOC` which returns both the minimum as well as an index value associated with that minimum. Because of `MPI_MINLOC` we use the two-element array *lminpair* to store the distance as well as the vertex that achieves this distance. Also, because the result of the reduction operation is needed by all the processes to perform the third step, we use the `MPI_Allreduce` operation to perform the reduction. The result of the reduction operation is returned in the *gminpair* array. The third and final step during each iteration is performed by scanning the local vertices that belong in V_o and updating their shortest distances from the source vertex.

Avoiding Load Imbalances Program 6.6 assigns *n* /*p* consecutive columns of *W* to each processor and in each iteration it uses the `MPI_MINLOC` reduction operation to select the vertex v to be included in V_c . Recall that the `MPI_MINLOC` operation for the pairs (*a* , *i*) and (*a* , *j*) will return the one that has the smaller index (since both of them have the same value). Consequently, among the vertices that are equally close to the source vertex, it favors the smaller numbered vertices. This may lead to load imbalances, because vertices stored in lower-ranked processes will tend to be included in V_c faster than vertices in higher-ranked processes (especially when many vertices in V_o are at the same minimum distance from the source). Consequently, the size of the set V_o will be larger in higher-ranked processes, dominating the overall runtime.

One way of correcting this problem is to distribute the columns of *W* using a cyclic distribution. In this distribution process *i* gets every *p* th vertex starting from vertex *i* . This scheme also assigns *n* /*p* vertices to each process but these vertices have indices that span almost the entire graph. Consequently, the preference given to lower-numbered vertices by `MPI_MINLOC` does not lead to load-imbalance problems.

6.6.10 Example: Sample Sort

The last problem requiring collective communications that we will consider is that of sorting a

8.2 Matrix-Matrix Multiplication

This section discusses parallel algorithms for multiplying two n x n dense, square matrices A and B to yield the product matrix $C = A$ x B. All parallel matrix multiplication algorithms in this chapter are based on the conventional serial algorithm shown in Algorithm 8.2. If we assume that an addition and multiplication pair (line 8) takes unit time, then the sequential run time of this algorithm is n^3. Matrix multiplication algorithms with better asymptotic sequential complexities are available, for example Strassen's algorithm. However, for the sake of simplicity, in this book we assume that the conventional algorithm is the best available serial algorithm. Problem 8.5 explores the performance of parallel matrix multiplication regarding Strassen's method as the base algorithm.

Algorithm 8.2 The conventional serial algorithm for multiplication of two n x n matrices.

```
1.   procedure MAT_MULT (A, B, C)
2.   begin
3.      for i := 0 to n - 1 do
4.          for j := 0 to n - 1 do
5.              begin
6.                 C[i, j] := 0;
7.                 for k := 0 to n - 1 do
8.                     C[i, j] := C[i, j] + A[i, k] x B[k, j];
9.              endfor;
10.  end MAT_MULT
```

Algorithm 8.3 The block matrix multiplication algorithm for n x n matrices with a block size of (n/q) x (n/q).

```
1.   procedure BLOCK_MAT_MULT (A, B, C)
2.   begin
3.      for i := 0 to q - 1 do
4.          for j := 0 to q - 1 do
5.              begin
6.                 Initialize all elements of C_{i,j} to zero;
7.                 for k := 0 to q - 1 do
8.                     C_{i,j} := C_{i,j} + A_{i,k} x B_{k,j};
9.              endfor;
10.  end BLOCK_MAT_MULT
```

A concept that is useful in matrix multiplication as well as in a variety of other matrix algorithms is that of block matrix operations. We can often express a matrix computation involving scalar algebraic operations on all its elements in terms of identical matrix algebraic operations on blocks or submatrices of the original matrix. Such algebraic operations on the submatrices are called *block matrix operations*. For example, an n x n matrix A can be regarded as a q x q array of blocks $A_{i,j}$ ($0 \leq i, j < q$) such that each block is an (n/q) x (n/q)

submatrix. The matrix multiplication algorithm in Algorithm 8.2 can then be rewritten as Algorithm 8.3, in which the multiplication and addition operations on line 8 are matrix multiplication and matrix addition, respectively. Not only are the final results of Algorithm 8.2 and 8.3 identical, but so are the total numbers of scalar additions and multiplications performed by each. Algorithm 8.2 performs n^3 additions and multiplications, and Algorithm 8.3 performs q^3 matrix multiplications, each involving $(n/q) \times (n/q)$ matrices and requiring $(n/q)^3$ additions and multiplications. We can use p processes to implement the block version of matrix multiplication in parallel by choosing $q = \sqrt{p}$ and computing a distinct $C_{i,j}$ block at each process.

In the following sections, we describe a few ways of parallelizing Algorithm 8.3. Each of the following parallel matrix multiplication algorithms uses a block 2-D partitioning of the matrices.

8.2.1 A Simple Parallel Algorithm

Consider two $n \times n$ matrices A and B partitioned into p blocks $A_{i,j}$ and $B_{i,j}$ $0 \leq k < \sqrt{p}$ of size $(n/\sqrt{p}) \times (n/\sqrt{p})$ each. These blocks are mapped onto a $\sqrt{p} \times \sqrt{p}$ logical mesh of processes. The processes are labeled from $P_{0,0}$ to $P_{\sqrt{p}-1,\sqrt{p}-1}$. Process $P_{i,j}$ initially stores $A_{i,j}$ and $B_{i,j}$ and computes block $C_{i,j}$ of the result matrix. Computing submatrix $C_{i,j}$ requires all submatrices $A_{i,k}$ and $B_{k,j}$ for $(0 \leq i, j < \sqrt{p})$. To acquire all the required blocks, an all-to-all broadcast of matrix A's blocks is performed in each row of processes, and an all-to-all broadcast of matrix B's blocks is performed in each column. After $P_{i,j}$ acquires $A_{i,0}, A_{i,1}, \ldots, A_{i,\sqrt{p}-1}$ and $B_{0,j}, B_{1,j}, \ldots, B_{\sqrt{p}-1,j}$, it performs the submatrix multiplication and addition step of lines 7 and 8 in Algorithm 8.3.

Performance and Scalability Analysis The algorithm requires two all-to-all broadcast steps (each consisting of $\sqrt{p}$ concurrent broadcasts in all rows and columns of the process mesh) among groups of $\sqrt{p}$ processes. The messages consist of submatrices of n^2/p elements. From Table 4.1, the total communication time is $2(t_s \log(\sqrt{p}) + t_w(n^2/p)(\sqrt{p}-1))$. After the communication step, each process computes a submatrix $C_{i,j}$, which requires $\sqrt{p}$ multiplications of $(n/\sqrt{p}) \times (n/\sqrt{p})$ submatrices (lines 7 and 8 of Algorithm 8.3 with $q = \sqrt{p}$. This takes a total of time $\sqrt{p} \times (n/\sqrt{p})^3 = n^3/p$. Thus, the parallel run time is approximately

Equation 8.14

$$T_P = \frac{n^3}{p} + t_s \log p + 2t_w \frac{n^2}{\sqrt{p}}.$$

The process-time product is $n^3 + t_s p \log p + 2t_w n^2 \sqrt{p}$, and the parallel algorithm is cost-optimal for $p = O(n^2)$.

The isoefficiency functions due to t_s and t_w are $t_s p \log p$ and $8(t_w)^3 p^{3/2}$, respectively. Hence, the overall isoefficiency function due to the communication overhead is $\Theta(p^{3/2})$. This algorithm can use a maximum of n^2 processes; hence, $p \leq n^2$ or $n^3 \geq p^{3/2}$. Therefore, the isoefficiency function due to concurrency is also $\Theta(p^{3/2})$.

A notable drawback of this algorithm is its excessive memory requirements. At the end of the communication phase, each process has $\sqrt{p}$ blocks of both matrices A and B. Since each block

requires $\Theta(n^2/p)$ memory, each process requires $\Theta(n^2/\sqrt{p})$ memory. The total memory requirement over all the processes is $\Theta(n^2/\sqrt{p})$, which is $\sqrt{p}$ times the memory requirement of the sequential algorithm.

8.2.2 Cannon's Algorithm

Cannon's algorithm is a memory-efficient version of the simple algorithm presented in Section 8.2.1. To study this algorithm, we again partition matrices A and B into p square blocks. We label the processes from $P_{0,0}$ to $P_{\sqrt{p}-1,\sqrt{p}-1}$, and initially assign submatrices $A_{i,j}$ and $B_{i,j}$ to process $P_{i,j}$. Although every process in the i th row requires all $\sqrt{p}$ submatrices $A_{i,k}$ $(0 \leq k < \sqrt{p})$, it is possible to schedule the computations of the $\sqrt{p}$ processes of the ith row such that, at any given time, each process is using a different $A_{i,k}$. These blocks can be systematically rotated among the processes after every submatrix multiplication so that every process gets a fresh $A_{i,k}$ after each rotation. If an identical schedule is applied to the columns, then no process holds more than one block of each matrix at any time, and the total memory requirement of the algorithm over all the processes is $\Theta(n^2)$. Cannon's algorithm is based on this idea. The scheduling for the multiplication of submatrices on separate processes in Cannon's algorithm is illustrated in Figure 8.3 for 16 processes.

Figure 8.3. The communication steps in Cannon's algorithm on 16 processes.

IJARCCE

ISSN (Online) 2278-1021
ISSN (Print) 2319 5940

International Journal of Advanced Research in Computer and Communication Engineering
Vol. 5, Issue 5, May 2016

A Naïve Breadth First Search Approach Incorporating Parallel Processing Technique For Optimal Network Traversal

Laxmikant Revdikar[1], Ayush Mittal[2], Anuj Sharma[3], Dr. Sunanda Gupta[4]

Shri Mata Vaishno Devi University, Katra, Jammu & Kashmir, India[1,2,3]

Dept. of Computer Science & Engineering, Shri Mata Vaishno Devi University, Katra, India[4]

Abstract: The authors have modied the existing breadth first search (BFS) technique by incorporating a parallel processing feature to it. A multithreaded implementation of breadth-first search (BFS) of a graph using Open MP. the results of our research reveal that implementing BFS using multiprocessor runs much faster than the standard BFS.

Keywords: Breadth-First Search, Parallel Programming, optimal Network Traversal Open MP.

I. INTRODUCTION

As we know that a graph can be used to represent any network so using graph theory approaches we can traverse the network, there are many graph traversal algorithms [1] such as

1) Depth First Search
2) Breadth First-Search
3) A*
4) Dijkstra
5) Prim
6) Kruskal
7) Floyd Warshall
8) Bellman Ford

Here we will use BFS and change it run for multi-processor systems.

A. BREADTH FIRST-SEARCH

BFS explores the vertices and edges of a graph, beginning from a specified "starting vertex" that we'll call s. It assigns each vertex a "level" number, which is the smallest number of hops in the graph it takes to reach that vertex from s. BFS begins by assigning s itself level 0. It first visits all the "neighbors" of s, which are the vertices that can be reached from s by following one edge, and assigns them level 1.

Then it visits the neighbors of the level-1 vertices: some of those neighbors might already be on level 0 or 1, but any that haven't already been assigned a level get level 2. And so on -- the so-far-unreached neighbors of level-2 vertices get level 3, then 4, and so forth until there are no more unreached[2].

BFS uses FIFO queue to decide what vertices to visit next. The queue starts out with only s on it, with level[s] =0. Then the general step is to take the front vertex v from the queue and visit all its neighbors. Any neighbor that hasn't yet been visited is added to the back of the queue and assigned a level one larger than LEVEL [v].

IT IS USEFUL IN

1) Social Media[3]
2) Logistic
3) E-Commerce[4]
4) Counter Terrorism
5) Fraud Detection[5]

For example, in the following graph (Fig. 1.), we start traversal from vertex 2. When we come to vertex 0, we look for all adjacent vertices of it. 2 is also an adjacent vertex of 0. If we don't mark visited vertices, then 2 will be processed again and it will become a non-terminating process. A Breadth First Traversal of the following graph is 2, 0, 3, 1.

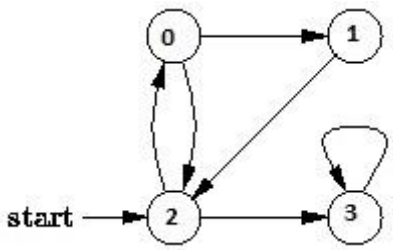

Fig. 1. Example of Breadth First-Search

B. PARALLEL BREADTH FIRST-SEARCH

The idea of doing BFS in parallel is that, in principal, you can process all the vertices on a single level at the same time. That is, once you've found all the level-1 vertices, you can do a parallel loop that explores from each of them to find level-2 vertices. Thus, the parallel code will have an important sequential loop over levels, starting at 0.

In the parallel code, it's possible that when you're processing[6] level i, two vertices v and w will both find the same level-i+1 vertex x as a neighbor. This will cause a data race when they both try to set level[x] =i+1, and also when they each try to set parent[x] to themselves. But if you're careful this is a "benign data race" -- it doesn't actually cause any problem, because there's no

IJARCCE

ISSN (Online) 2278-1021
ISSN (Print) 2319 5940

International Journal of Advanced Research in Computer and Communication Engineering
Vol. 5, Issue 5, May 2016

disagreement about what level[x] should be, and it doesn't matter in the end whether parent[x] turns out to be v or w.

C. OPENMP

OpenMp[7] is a set of compiler directives and library routines for parallel application programs. It is a parallel programming system that aims to be powerful and easy to use, while at the same time allowing the programmer to write high performance programs. Its initial focus was on numerical applications involving loops written in Fortran or C/C++, but it includes the necessary constructs to deal with more kinds of parallel algorithms.

Irregular parallel algorithms involve sub computations whose amount of work is not known in advance, and hence the work can only be distributed at runtime. Important subclasses include algorithms using task pools, as well as speculative algorithms. We are concentrating on the first type, although the problem and solutions we present apply to other types as well. Examples for irregular algorithms are search and sorting algorithms, graph algorithms, and more involved applications like volume rendering.

The expected graph (Fig. 2.) to be of sequential and Parallel BFS is given below:-

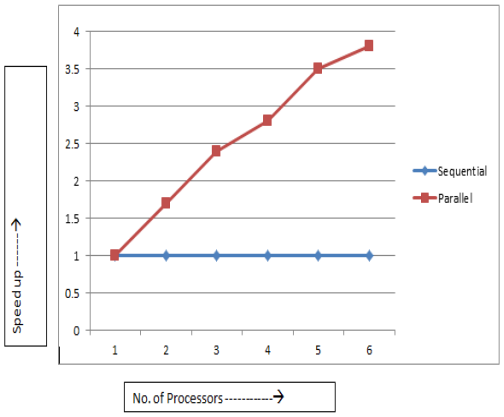

Fig. 2. Expected Speed up using parallel and sequential BFS

II. PURPOSE

Why BFS?

1) Least work/byte of the graph algorithms
2) Building blocks for many other graph algorithms

WHY PARALLEL BFS?

In past history when BFS was formed by scientist for traversing it was meant for single processor but we have opted for the parallel approach because nowadays its rarely practised and also there are systems which have multiple processor.

III. PROBLEM STATEMENT

We Have To Convert Sequential BFS Code into Parallel BFS To Reduce Time Complexity In Traversing Of Graph

IV. GOAL & VISION

Our Goal Is To Reduce Time Complexity Of The Traversing Of The Graph. Our Vision Is To Use Openmp Api In Sequential BFS To Make It Parallel BFS.

V. SEQUENTIAL BFS

A. CODE SNIPPET

```
// author: Laxmikant Revdikar
void simple_bfs()
{
    visited[s]=true;
    q.push(s);
    while(!q.empty()){
        current=q.front();
        q.pop();
        //printf("%d popped\n",current);

        for(i=0;i<nodes[current]->neighbours;i++)
        {
            int next=nodes[current]->next[i]->sn;
            if(!visited[next])
            {
                visited[next]=true;
                q.push(next);
                //printf("%d pushed\n",next);
            }
        }
    }
}
```

Fig. 3. Code Snippet of Sequential BFS in C++

B. EXPLANATION OF CODE

In the above code we have used a queue in which initially we enqueue start node and then change its visited mode and enqueue all its neighbour and then dequeue the start node and now we process the front node of queue and put all its neighbours into the queue and then dequeue it & change its visited mode, we repeat this process until the queue is empty. We created n input containing 99900 edges and then run the above code. The time taken to traverse all the edges was 0.036000.

VI. PARALLEL BFS

A. CODE SNIPPET

```
//author: Laxmikant Revdikar
void parallel_bfs()
{
    visited[start]=true;
    level[start]=0;
    q[0][0]=start;
    l[0]=1;
    current_queue=0;
    while(1)
    {
        next_queue=(current_queue+1)%2;
        l[next_queue]=0;
        omp_set_dynamic(0);
        #pragma omp parallel private(current_node) //num_threads(4)
        {
            #pragma omp for
                for(int i=0;i<l[current_queue];i++)
                {
                    current_node=q[current_queue][i];
                    for(int j=0;j<nodes[current_node]->neighbours;j++)
                    {
                        if(!visited[nodes[current_node]->next[j]])
                        {
                            visited[nodes[current_node]->next[j]] = true;
                            level[nodes[current_node]->next[j]]=level[current_node]+1;
                            #pragma omp critical
                            q[next_queue][l[next_queue]++] = nodes[current_node]->next[j];
                        }
                    }
                }
        }
        if(l[next_queue]==0)
            break;
        else
            current_queue=next_queue;
    }
}
```

Fig. 4. Code Snippet of Parallel BFS in C++

B. EXPLANATION OF CODE

In the parallel version of the BFS we have taken two array, current array and next array. Current array stores the nodes which are processing and the neighbours of all those nodes which have not been visited are kept in next array when

we have visited all nodes of current array then we swap current array and next array and this keeps going on until the no. Of nodes in next array becomes zero and we have kept the section critical where the threads insert the value of nodes into the next array in critical section only one thread at a time will be able to enter that region, so it will prevent overwriting of same values by different threads thus maintaining the synchronization.

We created n input containing 99900 edges and then run the above code and the time taken to traverse the network was 0.01900.

VII. TESTING

Now we have used a same parallel code with different no. Threads for same input. Following is the graph representing time taken vs. No. Of threads

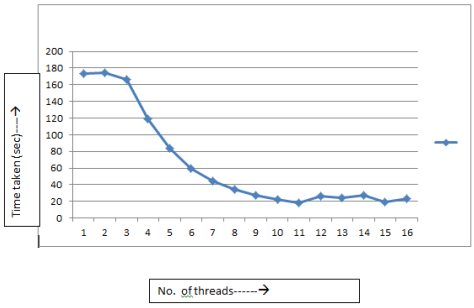

Fig. 5. Plotted Graph of Proposed Parallel BFS

VIII. CONCLUSION

We have studied about graph traversal algorithms one of them is BFS that is breadth first search algorithm and we have implemented a parallel approach to the sequential breadth first search. We experimented with some input graphs and then came to the result that on parallel processing with our proposed algorithm the graph is traversed much faster. We have also studied that by using different no. Of threads, different timings of traversing the graph is noted. To achieve the above we have used openmp with c++.

REFERENCES

[1]. Raynal, M. (2013). Basic Definitions and Network Traversal Algorithms. Distributed Algorithms for Message-Passing Systems, 3-34. doi:10.1007/978-3-642-38123-2_1
[2]. Bundy, A., & Wallen, L. (1984). Breadth-First Search. Catalogue of Artificial Intelligence Tools, 13-13. doi:10.1007/978-3-642-96868-6_25
[3]. Fay, D. (2016). Predictive Partitioning for Efficient BFS Traversal in Social Networks. Studies in Computational Intelligence Complex Networks VII, 11-26. doi:10.1007/978-3-319-30569-1_2
[4]. Bishop, M. (1999). A Breadth-First Strategy for Mating Search. Automated Deduction — CADE-16 Lecture Notes in Computer Science, 359-373. doi:10.1007/3-540-48660-7_32
[5]. Fraud Detection. (2014). Fraud and Fraud Detection A Data Analytics Approach, 7-15. doi:10.1002/9781118936764.ch2
[6]. Parallel Breadth-First Search on Distributed Memory Systems. (2011). doi:10.2172/1050644
[7]. A Quick Reference to OpenMP. (2001). Parallel Programming in OpenMP, 211-216. doi:10.1016/b978-155860671-5/50008-6.

Parallel Programming Unit 4 Ref. 4

https://youtu.be/BjZPuSO8Lvc?si=ToxMTvqJQcck1UDq