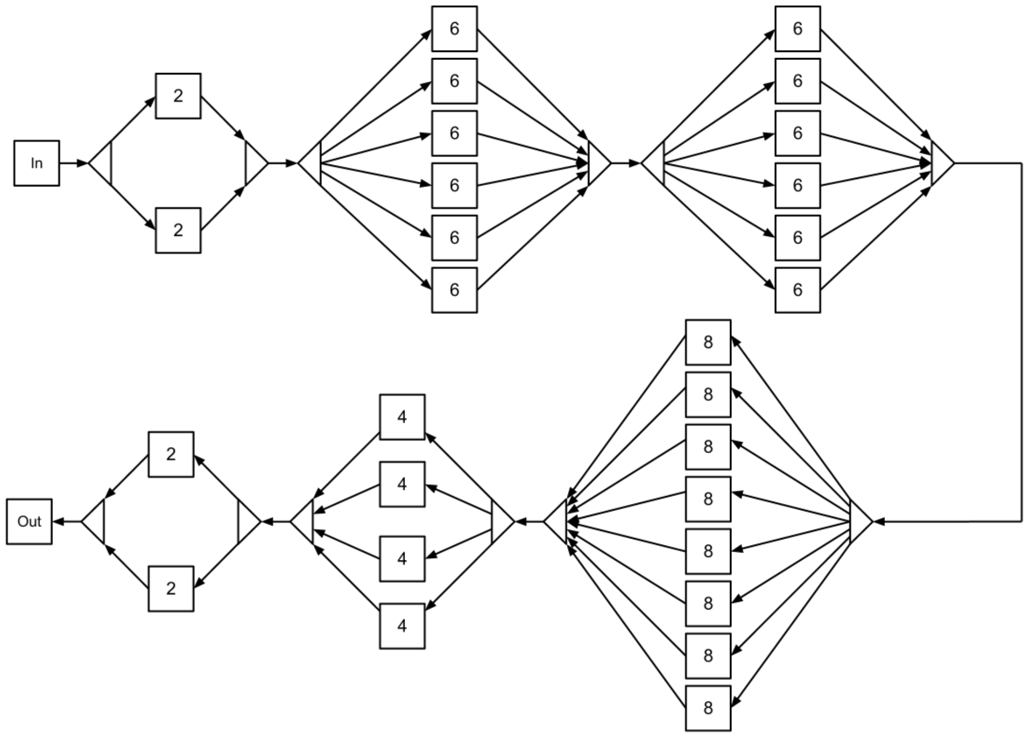
CS 789 Parallel Programming
Assignment 4
Pipeline and Data Parallel Problems

Lucas Bang



Contents

1 Optimizing Pipeline Computations

Consider the pipeline shown in Figure 1.

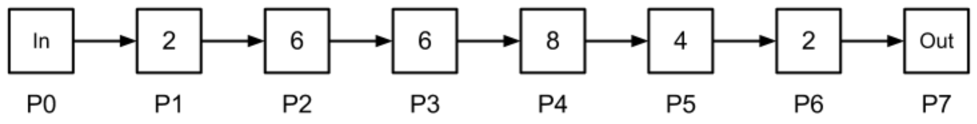


Figure 1: The original pipeline. Process times are indicated in each box representing a processing step.

If we implemented the pipeline as given, the throughput would be 1 data item every 8 time units. We wish to optimize the pipeline so that the throughput is 1 data item every 2 time units. We can accomplish this by using the pipeline shown shown in Figure 2.

The MPI code that implements this pipeline is given in Listing 1 and the output verifying that the program works as desired follows.

In order to increase the throughput to 1 data item every 1 unit of time, we could implement the pipeline shown in Figure 3, doubling the number of data processors compared to the previous pipeline.

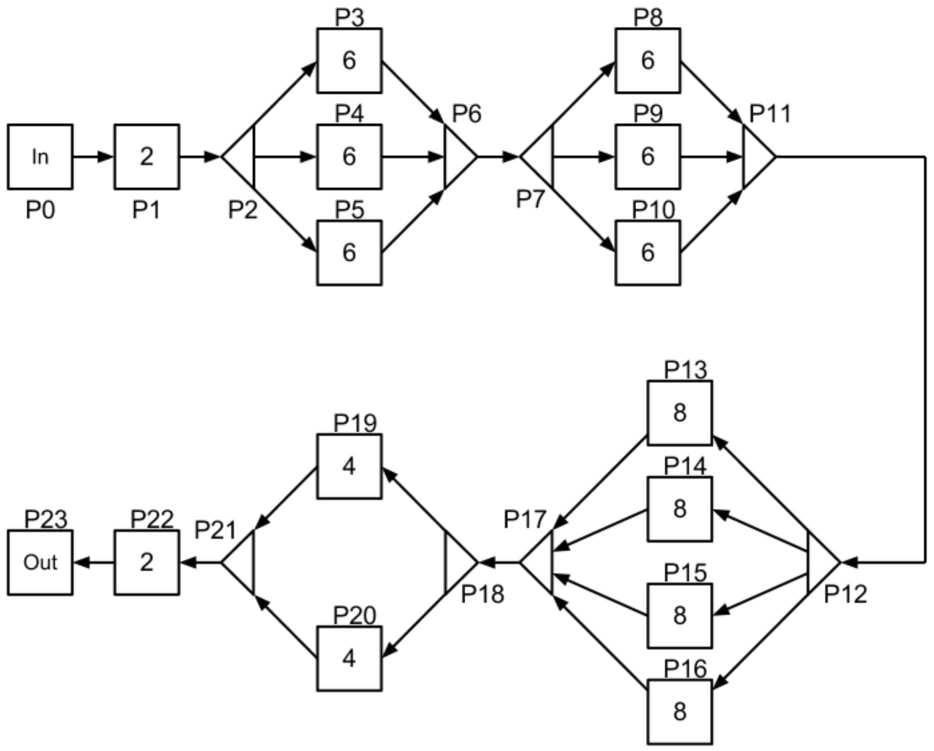


Figure 2: A pipeline which admits a throughput of 1 data item every 2 time units.

Listing 1: Pipeline Code

```c
#include <stdio.h>
#include <mpi.h>
#include <sys/time.h>

void disp(int noProc, int from) {
  MPI_Status status;
  int rank;
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  int value,i;
  while (1) {
    for (i = rank+1;  i <= rank+noProc; i++) {
      MPI_Recv(&value, 1, MPI_INT, from , MPI_ANY_TAG, MPI_COMM_WORLD, &status);
      MPI_Send(&value, 1, MPI_INT, i, 1, MPI_COMM_WORLD);
    }
  }
}

void coll(int noProc, int to) {
  MPI_Status status;
  int value, i;
  int rank;
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  while (1) {
    for (i = rank-noProc; i < rank; i++) {
      MPI_Recv(&value, 1, MPI_INT, i, MPI_ANY_TAG, MPI_COMM_WORLD, &status);
      MPI_Send(&value, 1, MPI_INT, to, 1, MPI_COMM_WORLD);
    }
  }
}


void proc(int delay, int from, int to) {
  MPI_Status status;
  struct timeval t1, t2;
  int a, b, c;
  int value;
  int rank;
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  while (1) {
    MPI_Recv(&value, 1, MPI_INT, from, MPI_ANY_TAG, MPI_COMM_WORLD, &status);
    gettimeofday(&t1,NULL);
    gettimeofday(&t2,NULL);
    while(t2.tv_sec - t1.tv_sec < delay){
      srand(time(NULL));
      a = rand() % 43;
      b = rand() % 31;
      c = 23*a % (b+1);
      gettimeofday(&t2, NULL);
    }
    MPI_Send(&value, 1, MPI_INT, to, 1, MPI_COMM_WORLD);
  }
}


void input_proc(int to) {
  MPI_Status status;
  struct timeval t1, t2;
  //int a, b, c;
  int i;
  int values[100];
  for (i = 0; i<100; i++)
    values[i] = 100+i;

  int num_values = 20;
  int value;
  for(i = 0; i < num_values; i++){
    value = values[i];
    MPI_Send(&value, 1, MPI_INT, to, 1, MPI_COMM_WORLD);
    printf("Input process sending value = %d  through the pipeline.\n", value);
  }
}


void output_proc(int from) {
  MPI_Status status;
  struct timeval t1, t2;
  gettimeofday(&t1, NULL);
  printf("Output process start timer at 0 seconds \n");
  int i;
  int value;
  while(1){
    MPI_Recv(&value, 1, MPI_INT, from, MPI_ANY_TAG, MPI_COMM_WORLD, &status);
    gettimeofday(&t2, NULL);
    printf("Output process received value = %d after approximately %ld seconds. \n", ...
            value, t2.tv_sec - t1.tv_sec);
  }
}

main(int argc, char** argv) {

  int rank, size, source, dest;
  MPI_Init(&argc, &argv);
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  MPI_Comm_size(MPI_COMM_WORLD, &size);

  switch (rank) {
    case 0:
      input_proc(1);    break;
    case 1:
      proc(2, 0, 2);    break;
    case 2:
      disp(3, 1);       break;
    case 3: case 4: case 5:
      proc(6, 2, 6);    break;
    case 6:
      coll(3, 7);       break;
    case 7:
      disp(3, 6);       break;
    case 8: case 9: case 10:
      proc(6, 7, 11);   break;
    case 11:
      coll(3, 12);      break;
    case 12:
      disp(4,   11);    break;
    case 13: case 14: case 15: case 16:
      proc(8, 12, 17);  break;
    case 17:
      coll(4, 18);      break;
    case 18:
      disp(2, 17);      break;
    case 19: case 20:
      proc(4, 18, 21);  break;
    case 21:
      coll(2, 22);      break;
    case 22:
      proc(2, 21, 23);  break;
    case 23:
      output_proc(22);  break;
  }
  MPI_Finalize();
}
```

The output of the code is shown here:

```
Input process sending value = 100 through the pipeline.
Input process sending value = 101 through the pipeline.
Input process sending value = 102 through the pipeline.
Input process sending value = 103 through the pipeline.
Input process sending value = 104 through the pipeline.
Input process sending value = 105 through the pipeline.
Input process sending value = 106 through the pipeline.
Input process sending value = 107 through the pipeline.
Input process sending value = 108 through the pipeline.
Input process sending value = 109 through the pipeline.
Input process sending value = 110 through the pipeline.
Input process sending value = 111 through the pipeline.
Input process sending value = 112 through the pipeline.
Input process sending value = 113 through the pipeline.
Input process sending value = 114 through the pipeline.
Input process sending value = 115 through the pipeline.
Output process start timer at 0 seconds.
Output process received value = 100 after 12 seconds.
Output process received value = 101 after 14 seconds.
Output process received value = 102 after 16 seconds.
Output process received value = 103 after 18 seconds.
Output process received value = 104 after 20 seconds.
Output process received value = 105 after 22 seconds.
Output process received value = 106 after 24 seconds.
Output process received value = 107 after 26 seconds.
Output process received value = 108 after 28 seconds.
Output process received value = 109 after 30 seconds.
Output process received value = 110 after 32 seconds.
Output process received value = 111 after 34 seconds.
Output process received value = 112 after 36 seconds.
Output process received value = 113 after 38 seconds.
Output process received value = 114 after 40 seconds.
Output process received value = 115 after 42 seconds.
```

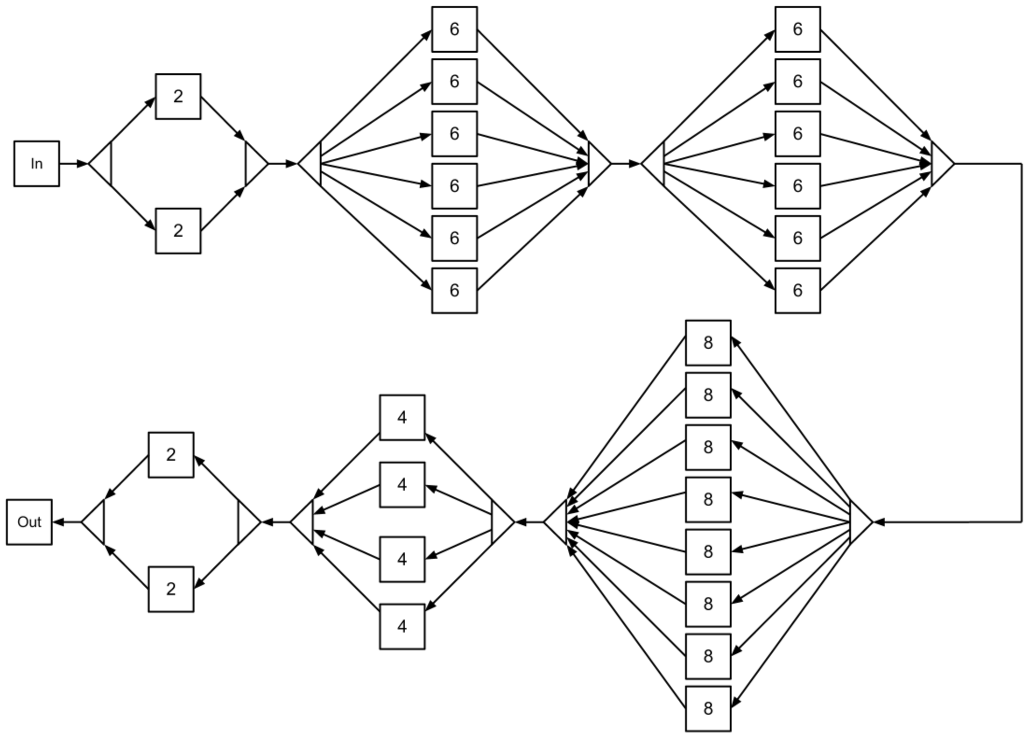


Figure 3: A pipeline which admits a throughput of 1 data item for every 1 unit of time.

2 Solving a Lower Triangular System of Equations

Consider the problem of solving the matrix equation $Ax = b$ where A is a lower triangular matrix.
A simple sequential program to compute the values of the vector x is:

```
    x[0] = b[0]/a[0][0];
    for (i = 0; i < n; i++) {
        sum = 0;
        for (j=0; j < i; j++)
            sum = sum + a[i][j] * x[j];
        x[i] = (b[i] - sum) / a[i][i];
}
```

A type 2 pipeline parallel algorithm can be implemented like this:

```
    sum = 0;
    for (j = 0; j < i; j++) {
        recv(&x[j], i-1);
        send(&x[j], i+1);
        sum = sum + a[i][j] * x[j];
    }
    x[i] = (b[i] - sum) / a[i][i];
    send(&x[i], i+1);
```

Consider solving a single instance of an $n \times n$ triangular system of equations sequentially. By the above pseudo code, the solution for each x_i requires (approximately) 1 division, 1 subtraction, i additions, and i multiplications. Let $t_+, t_\times, t_-$, and $t_\div$ represent the time to add, multiply subtract, and divide two floating point numbers, respectively. The total time to solve for all x_i is then

$$t_{seq1} = \sum_{i=1}^{n} (t_\div + t_- + i \cdot (t_\times + t_+)) = n \cdot (t_\div + t_-) + \frac{n(n+1)}{2} \cdot (t_\times + t_+)$$

Now consider the parallel pipeline for the above pseudo code.

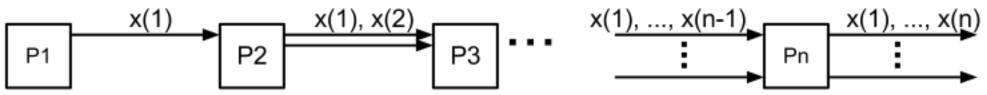


Figure 4: Parallel pipeline for solving triangular system of equations.

The bottleneck of this pipeline is the last processor which must perform1 division, 1 subtraction, n additions, n multiplications, and n communications before it can start outputting the result. Similar to the analysis of the sequential version, filling the pipeline requires n divisions, n subtractions, $\frac{n(n+1)}{2}$ additions, $\frac{n(n+1)}{2}$ multiplications. However, filling the pipeline also requires $\frac{n(n+1)}{2}$ communications! We would expect the communications to slow everything down, even for repeatedly solving multiple instances of $n \times n$ equations, regardless of n. We will see this is the case both in theory and practice.

Now consider solving m instances of the above described system sequentially. It is simple to compute the total time – we just multiply by n.

$$t_{seq} = m \cdot \left(n \cdot t_{\div} + n \cdot t_{-} + \frac{n(n+1)}{2} \cdot t_{+} + \frac{n(n+1)}{2} \cdot t_{\times} \right)$$

This gives us a throughput of 1 $n \times 1$ vector every t_{seq} units of time.

On the other hand, once the parallel pipeline is full, we will have a throughput of 1 $n \times 1$ solution vector every $t_{bottleneck}$ time units. We may therefore compute $t_{par} = t_{fill} + t_{bottlenck}$, where

$$t_{fill} = n \cdot t_{\div} + n \cdot t_{-} + \frac{n(n+1)}{2} \cdot t_{+} + \frac{n(n+1)}{2} \cdot t_{\times} + \frac{n(n+1)}{2} \cdot t_{comm}$$

$$t_{bottlneck} = m \cdot (t_{\div} + t_{-} + n \cdot (t_{+} + t_{\times}) + n \cdot t_{comm})$$

The speedup is given by

$$speedup = \frac{t_{seq}}{t_{par}}$$

$$= \frac{m \cdot \left(n \cdot t_{\div} + n \cdot t_{-} + \frac{n(n+1)}{2} \cdot t_{+} + \frac{n(n+1)}{2} \cdot t_{\times}\right)}{n \cdot t_{\div} + n \cdot t_{-} + \frac{n(n+1)}{2} \cdot t_{+} + \frac{n(n+1)}{2} \cdot t_{\times} + \frac{n(n+1)}{2} \cdot t_{comm} + m \cdot (t_{\div} + t_{-} + n \cdot (t_{+} + t_{\times}) + n \cdot t_{comm})}$$

Now, as $m \to \infty$,

$$speedup \to \frac{n \cdot t_{\div} + n \cdot t_{-} + \frac{n(n+1)}{2} \cdot t_{+} + \frac{n(n+1)}{2} \cdot t_{\times}}{t_{\div} + t_{-} + n \cdot (t_{+} + t_{\times}) + n \cdot t_{comm}}$$

Plugging in $n = 10$, doing a TON of algebra, and canceling lower order terms physicist style:

$$speedup \approx \frac{11}{2} \cdot \left(1 - \frac{t_{comm}}{t_{+} + t_{\times} + t_{comm}}\right)$$

The speedup depends then on the relative values of t_{comm} and the computation times (duh! we knew that already). However, we can see that the speedup has a trivial upper bound of 5.5 for the system of 10 equations. Thus, we should not expect a speedup greater than 5.5. Even that is overly optimistic.

If we experiment to find out the computation times and communication times we find that $t_{comm} \approx 87\mu s$, $t_{+} \approx 1\mu s$, and $t_{\times} \approx 2\mu s$. The communication time estimate was measured by doing only a single send and receive, and so does not include waiting time. This will still yield an overly optimistic result, since we cannot truly communicate in parallel. Plugging in these values we get that

$$speedup \approx 0.183$$

This is actually a slow down. But we kind of expected that. It's good that the theory and our intuitions match up.

When we actually run and compare the sequential and parallel versions on 1,000,000 instances, we find that $t_{seq} = 12.111\ s$ and $t_{par} = 201.508\ s$. This gives us $speedup = 0.0601$, which is lower than our theoretical prediction by a factor of 3.

It is pretty clear that parallelization does not help us in the problem by simply pipelining as described. We need a smarter data distribution scheme to solve a large number of very large systems with a number of processors that is much smaller than the size of the matrix.

Listing 2: Sequential Triangular System Solver

```c
#include <stdio.h>
#include <sys/time.h>

main(int argc, char** argv) {
  int n = 10;
  float A[10][10];
  float b[10];
  float x[10];
  int i, j, sum, k;

  for (i = 0; i < n; i++)
    for (j = 0; j < n; j++)
      A[i][j] = 0.0;

  srand(1);
  for (i = 0; i < n; i++)
    for (j = 0; j <= i; j++)
      A[i][j] = 1.0 + rand()%3;

  for (i = 0; i < n; i++)
    b[i] = i;

  for(k = 0; k<10000; k++){
    for (i = 0; i < n; i++)
      b[i] = i +rand()%10;

    x[0] = b[0] / A[0][0];
    for(i = 0; i < n; i++){
      sum = 0;
      for(j = 0; j < i; j++){
        sum = sum + A[i][j] * x[j];
      }
      x[i] = (b[i] - sum) / A[i][i];
    }
  }
}
```

Listing 3: Sequential Triangular System Solver

```c
#include <stdio.h>
#include <mpi.h>
#include <sys/time.h>

main(int argc, char** argv) {
  int rank, size,  i, j, k;
  int n = 10;
  float A[10][10];
  float b[10];
  float sum;
  float x[10];

  MPI_Init(&argc, &argv);
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  MPI_Comm_size(MPI_COMM_WORLD, &size);
  MPI_Status status;

  for (i = 0; i < n; i++)
    for (j = 0; j < n; j++)
      A[i][j] = 0.0;

  srand(1);
  for (i = 0; i < n; i++)
    for (j = 0; j <= i; j++)
      A[i][j] = 1.0 + rand()%3;

  for (i = 0; i < n; i++)
    b[i] = i;

  for(k = 0; k< 100000; k++){
    if(rank < 10){
      sum = 0;
      for(j = 0; j < rank; j++){
        MPI_Recv(&x[j], 1, MPI_FLOAT, rank-1, MPI_ANY_TAG, MPI_COMM_WORLD, &status);
        MPI_Send(&x[j], 1, MPI_FLOAT, rank+1, 1, MPI_COMM_WORLD);
        sum = sum + A[rank][j] * x[j];
      }
      x[rank] = (b[rank] - sum) / A[rank][rank];
      MPI_Send(&x[rank], 1, MPI_FLOAT, rank+1, 1, MPI_COMM_WORLD);
     }
    else{
      for(j = 0; j < rank; j++){
        MPI_Recv(&x[j], 1, MPI_FLOAT, rank-1, MPI_ANY_TAG, MPI_COMM_WORLD, &status);
      }
    }
  }
  MPI_Finalize();
}
```

3 Partial Sum & Efficient Data Distribution

We wish to write a program in MPI that consists of one leader and $n = 2^k - 1$ $(k > 0)$ worker processes, with the following behavior:

leader (rank = 0):

1. Initialize a list of n values (This is the ONLY operation in this program that may take time $O(n)$).
2. Distribute the list such that worker i gets the ith value of the original list held by the leader. This must be done in time $O(\log n)$.

worker (rank = $1, ..., n$):

1. Participate in the distribution initiated by the leader. The workers should not copy or rewrite or even traverse the list received before sending it on. Again, time should be $O(\log n)$. The sum of the number of data items in all messages used for this distribution may not exceed $\frac{n^2}{2}$.
2. Once processor i has the ith element from the original list, perform a partial sum calculation in time $O(\log n)$ such that processor i holds the value $\sum_{j=1}^{i} x_j$.
3. Distribute the partial sum held by processor n to every worker. Do this in time $O(\log n)$.

If the original array the leader holds is $X = (x_1, x_2, ..., x_n)$ then processor i should hold the two values: Partial sum $= \sum_{j=1}^{i} x_j$ and Total sum $= \sum_{j=1}^{n} x_j$ after the program finishes.

The distribution of the initial list is accomplished using a modification of the tree algorithm presented in the text book. For this data distribution, every process must know the number of items it will receive during the distribution. Keeping in mind that this is a tree, and trees are just begging for recursion, we can compute that number with a simple recursive function:

```
**function** num_items($n$)
    **if** ($n == 2^k$ for some $k$) **return** $n$;
    **else return** num_items($n - 2^{\log_2 n}$)
```

where $\log_2 n$ is the integer valued base 2 logarithm.

For example, if processor 28 wants to know how many items it will receive:

```
num_items(28) = num_items(12) = num_items(4) = 4
```

Note that the number of items received by any processor is always a power of 2. Now, suppose that processor i receives m items to distribute. Process i will keep the first and break the remaining items into $\log m$ groups whose sizes are increasing powers of 2, sending them to to appropriate processor. See the following code and figure for an example of how this works.

leader:

```
  **for** ($i = 1; i < n; i = i * 2$)
      **Send**($\&A[i-1], i, i, ...$)
```

workers:

```
  my_n = num_items(rank)
  **Recv**(A[0], my_n)
  my_val = A[0]
  **for** ($i = 1; i < my\_n; i = *22$)
      **Send**($\&A[i], i, ...$)
```

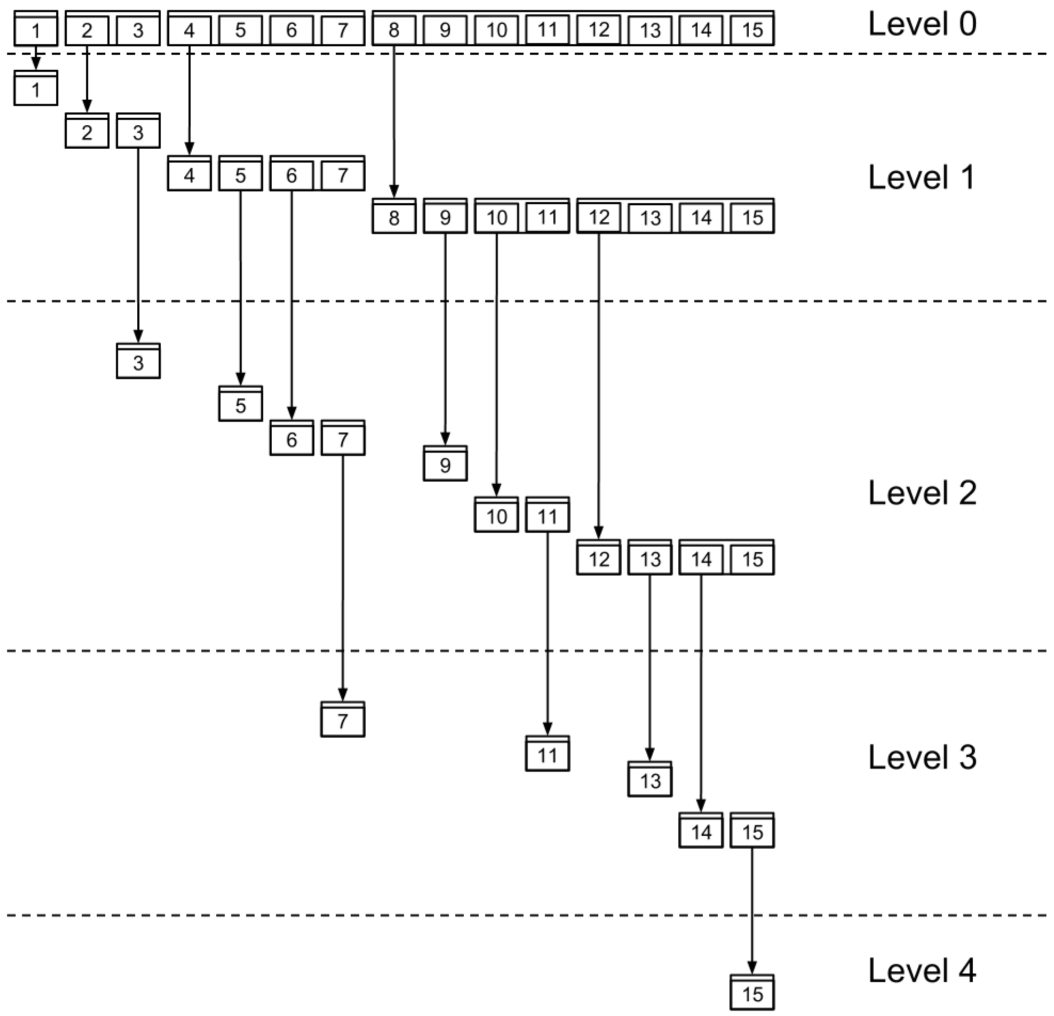


Figure 5: Example distribution of an array of 15 items.

The computation of the partial sum is accomplished by each worker executing the following code. See the figure for an example.

```
partial_sum = my_val
for ($i = 1; i < n; i = *2$)
    if ($i + rank \leq n$)
        Send(partialsum, $rank + i$)
    if ($rank - i \geq 1$)
        Recv (val, $rank - i$)
        partial_sum = partial_sum + val;
```

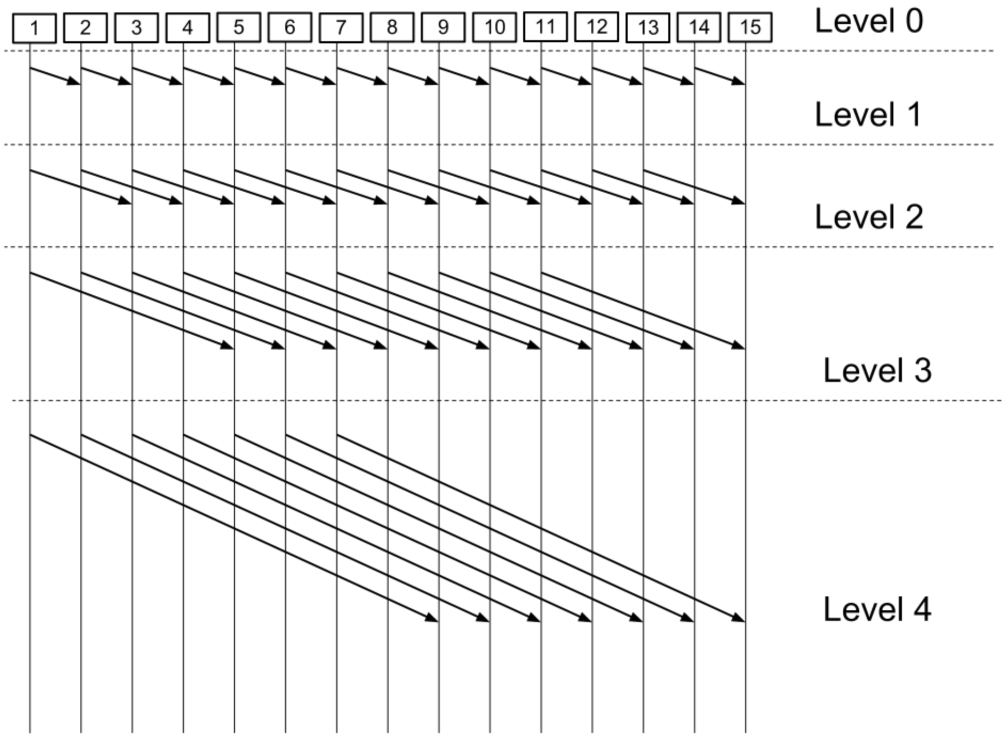


Figure 6: Example partial sum communication diagram for 15 processes.

The distribution of the total sum is accomplished by using an in-order traversal of a binary tree. Each process computes its children based on its own rank. For a process with rank i, its children will be the processes with ranks $2i + 1$ and $2i + 2$.

The last process already has the total sum and initiates by sending the total sum to the leader:

```
**Send**(total_sum, 0)
```

All other process compute their children and forward the total sum as follows:

```
**Recv**(total_sum, ANY_SOURCE)
**if**($rank < \frac{n}{2}$)
**Send**(total_sum, $2 * rank + 1$)
**Send**(total_sum, $2 * rank + 2$)
```

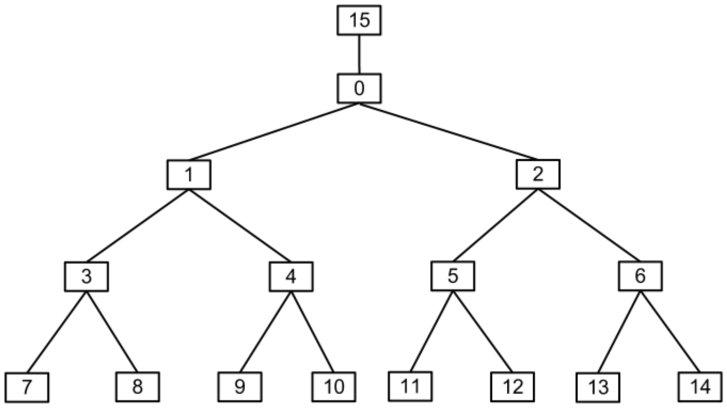


Figure 7: The tree used to distribute the total sum for 15 worker processes.

The entire partial sum program and sample output are included at the end of this section.

Listing 4: Partial Sum Code

```c
#include <stdio.h>
#include <mpi.h>
#include <sys/time.h>

int pow2(int n){
  if (n == 0)
    return 1;
  else
    return (2*pow2(n-1));
}
int isPow2(int n) { return (n - pow2(log2(n)) == 0); }
int num_items(int n){
  if (isPow2(n)) return n;
  else return num_items(n - pow2(log2(n)));
}

main(int argc, char** argv) {
  int rank, size; i, j;
  int n = atoi(argv[1]);
  int total_sum, sum;
  MPI_Init(&argc, &argv);
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  MPI_Comm_size(MPI_COMM_WORLD, &size);
  MPI_Status status;
  if(rank == 0){
    int *A;
    A = (int *) calloc(n, sizeof(int));
    for (i = 0; i < n; i++)
        A[i]= (i+1)*(i+1);
    for(i = 1; i < n; i*=2){
        MPI_Send(&A[i-1], i, MPI_INT, i, 1, MPI_COMM_WORLD);
    }
    MPI_Barrier(MPI_COMM_WORLD);
    MPI_Recv(&total_sum, 1, MPI_INT, MPI_ANY_SOURCE, MPI_ANY_TAG, MPI_COMM_WORLD, &status);
    MPI_Send(&total_sum, 1, MPI_INT, 1, 1, MPI_COMM_WORLD);
    MPI_Send(&total_sum, 1, MPI_INT, 2, 1, MPI_COMM_WORLD);
  }
  else{
    int my_val, val, partial_sum;
    int my_n = num_items(rank);
    int * A;
    A = (int *) calloc(n, sizeof(int));
    MPI_Recv(&A[0], my_n, MPI_INT, MPI_ANY_SOURCE, MPI_ANY_TAG, MPI_COMM_WORLD, &status);
    my_val = A[0];

    for(i = 1; i < my_n; i*=2){
      MPI_Send(&A[i], i, MPI_INT, rank + i, 1, MPI_COMM_WORLD);
    }
    MPI_Barrier(MPI_COMM_WORLD);
    partial_sum = my_val;
    for (i=1; i < n; i*=2){
      if(i + rank <= n){
        MPI_Send(&partial_sum, 1, MPI_INT, rank + i, 1, MPI_COMM_WORLD);
      }
      if(rank - i >= 1){
        MPI_Recv(&val, 1, MPI_INT, rank - i, MPI_ANY_TAG, MPI_COMM_WORLD, &status);
        partial_sum += val;
      }
    }
    if(rank == n){
      total_sum = partial_sum;
      MPI_Send(&partial_sum, 1, MPI_INT, 0, 1, MPI_COMM_WORLD);
    }
    else{
      MPI_Recv(&total_sum, 1, MPI_INT, MPI_ANY_SOURCE, MPI_ANY_TAG, MPI_COMM_WORLD, &status);
      if(rank < n/2){
        MPI_Send(&total_sum, 1, MPI_INT, 2*rank + 1, 1, MPI_COMM_WORLD);
        MPI_Send(&total_sum, 1, MPI_INT, 2*rank + 2, 1, MPI_COMM_WORLD);
      }
    }
    printf("%d:\t\tMy value = %d\t\tMy partial sum = %d\t\tTotal sum = %d \n", rank, ...
              my_val, partial_sum, total_sum);
  }
  MPI_Finalize();
}
```

Sample output of above code:

```
31:  value = 961   partial sum = 10416    Total sum = 10416
2 :  value = 4     partial sum = 5        Total sum = 10416
1 :  value = 1     partial sum = 1        Total sum = 10416
5 :  value = 25    partial sum = 55       Total sum = 10416
6 :  value = 36    partial sum = 91       Total sum = 10416
3 :  value = 9     partial sum = 14       Total sum = 10416
4 :  value = 16    partial sum = 30       Total sum = 10416
14:  value = 196   partial sum = 1015     Total sum = 10416
7 :  value = 49    partial sum = 140      Total sum = 10416
11:  value = 121   partial sum = 506      Total sum = 10416
12:  value = 144   partial sum = 650      Total sum = 10416
13:  value = 169   partial sum = 819      Total sum = 10416
15:  value = 225   partial sum = 1240     Total sum = 10416
9 :  value = 81    partial sum = 285      Total sum = 10416
10:  value = 100   partial sum = 385      Total sum = 10416
30:  value = 900   partial sum = 9455     Total sum = 10416
29:  value = 841   partial sum = 8555     Total sum = 10416
8 :  value = 64    partial sum = 204      Total sum = 10416
16:  value = 256   partial sum = 1496     Total sum = 10416
17:  value = 289   partial sum = 1785     Total sum = 10416
18:  value = 324   partial sum = 2109     Total sum = 10416
19:  value = 361   partial sum = 2470     Total sum = 10416
20:  value = 400   partial sum = 2870     Total sum = 10416
21:  value = 441   partial sum = 3311     Total sum = 10416
22:  value = 484   partial sum = 3795     Total sum = 10416
23:  value = 529   partial sum = 4324     Total sum = 10416
24:  value = 576   partial sum = 4900     Total sum = 10416
25:  value = 625   partial sum = 5525     Total sum = 10416
26:  value = 676   partial sum = 6201     Total sum = 10416
27:  value = 729   partial sum = 6930     Total sum = 10416
28:  value = 784   partial sum = 7714     Total sum = 10416
```