Gahan M. Saraiya
18MCEC10

DPS Assignment 1

March 19, 2019

1 Write major MPI routine with its application, where to write (i.e.master/slave/any sender processor/any or all receiver) it's syntax and sample mpi statement with description. Use table to describe it and also mention Communication cost

Routine	**Syntax**	**Description**
MPI_Init	MPI_Init (&argc,&argv)	Initializes the MPI execution environment. This function must be called in every MPI program, before any other MPI functions and must be called only once in an MPI program.
MPI_Comm_Size	MPI_Comm_size (comm,size)	Returns the total number of MPI processes in the specified communicator, such as MPI_COMM_WORLD. If the communicator is MPI_COMM_WORLD, then it represents the number of MPI tasks available to your application.
MPI_Comm_rank	MPI_Comm_rank (comm,&rank)	Returns the rank of the calling MPI process within the specified communicator. Initially, each process will be assigned a unique integer rank between 0 and number of tasks - 1 within the communicator MPI_COMM_WORLD.
MPI_Get_processor_name	MPI_Get_processor_name (&name,&resultlength)	Returns the processor name. Also returns the length of the name. The buffer for "name" must be at least MPI_MAX_PROCESSOR_NAME characters in size.
MPI_Wtime	MPI_Wtime()	Returns an elapsed wall clock time in seconds (double precision) on the calling processor.
MPI_Finalize	MPI_Finalize()	Terminates the MPI execution environment.

Table 1: Environment Management Routines

Example of Environment Management Routines

```
// required MPI include file
#include "mpi.h"
#include <stdio.h>

int main(int argc, char *argv[]) {
int  numtasks, rank, len, rc;
char hostname[MPI_MAX_PROCESSOR_NAME];

// initialize MPI
MPI_Init(&argc,&argv);

// get number of tasks
MPI_Comm_size(MPI_COMM_WORLD,&numtasks);

// get my rank
MPI_Comm_rank(MPI_COMM_WORLD,&rank);

// this one is obvious
MPI_Get_processor_name(hostname, &len);
printf ("Number of tasks= %d My rank= %d Running on %s\n",
 ↪  numtasks,rank,hostname);
// do some work with message passing
// done with MPI
MPI_Finalize();
}
```

1.1 Point to Point Communication Routines

Example of Point to Point Communication

```
// Find out rank, size
int world_rank;
```

Routine	**Syntax**	**Description**	**Communication Cost**
MPI_Send	MPI_Send(void* data, int count, MPI_Datatype datatype, int destination, int tag, MPI_Comm communicator)	almost every MPI call uses similar syntax. The first argument is the data buffer. The second and third arguments describe the count and type of elements that reside in the buffer. MPI_Send sends the exact count of elements, and MPI_Recv will receive at most the count of elements (more on this in the next lesson).	$no\_of\_processor$ $\times$ $(Time_{startup} + Time_{data})$
MPI_Recv	MPI_Recv(void* data, int count, MPI_Datatype datatype, int source, int tag, MPI_Comm communicator, MPI_Status* status)	The fourth and fifth arguments specify the rank of the sending/receiving process and the tag of the message. The sixth argument specifies the communicator and the last argument (for MPI_Recv only) provides information about the received message.	$Time_{startup} + Time_{data}$

Table 2: Point to Point Communication Routines

```
MPI_Comm_rank(MPI_COMM_WORLD, &world_rank);
int world_size;
MPI_Comm_size(MPI_COMM_WORLD, &world_size);

int number;
if (world_rank == 0) {
    number = -1;
    MPI_Send(&number, 1, MPI_INT, 1, 0, MPI_COMM_WORLD);
} else if (world_rank == 1) {
    MPI_Recv(&number, 1, MPI_INT, 0, 0, MPI_COMM_WORLD,
             MPI_STATUS_IGNORE);
    printf("Process 1 received number %d from process 0\n",
           number);
}
```

1.2 Collective Communication Routines

Example of MPI_Bcast

```
for (i = 0; i < num_trials; i++) {
    // Time my_bcast
    // Synchronize before starting timing
    MPI_Barrier(MPI_COMM_WORLD);
    total_my_bcast_time -= MPI_Wtime();
    my_bcast(data, num_elements, MPI_INT, 0, MPI_COMM_WORLD);
    // Synchronize again before obtaining final time
    MPI_Barrier(MPI_COMM_WORLD);
    total_my_bcast_time += MPI_Wtime();

    // Time MPI_Bcast
    MPI_Barrier(MPI_COMM_WORLD);
    total_mpi_bcast_time -= MPI_Wtime();
    MPI_Bcast(data, num_elements, MPI_INT, 0, MPI_COMM_WORLD);
    MPI_Barrier(MPI_COMM_WORLD);
```

```
16      total_mpi_bcast_time += MPI_Wtime();
17  }
```

Example of MPI_Reduce

```
1   float *rand_nums = NULL;
2   rand_nums = create_rand_nums(num_elements_per_proc);
3
4   // Sum the numbers locally
5   float local_sum = 0;
6   int i;
7   for (i = 0; i < num_elements_per_proc; i++) {
8     local_sum += rand_nums[i];
9   }
10
11  // Print the random numbers on each process
12  printf("Local sum for process %d - %f, avg = %f\n",
13         world_rank, local_sum, local_sum / num_elements_per_proc);
14
15  // Reduce all of the local sums into the global sum
16  float global_sum;
17  MPI_Reduce(&local_sum, &global_sum, 1, MPI_FLOAT, MPI_SUM, 0,
18             MPI_COMM_WORLD);
19
20  // Print the result
21  if (world_rank == 0) {
22    printf("Total sum = %f, avg = %f\n", global_sum,
23           global_sum / (world_size * num_elements_per_proc));
24  }
```

In the code above, each process creates random numbers and makes a local_sum calculation. The local_sum is then reduced to the root process using MPI_SUM. The global average is then $global\_sum/(world\_size * num\_elements\_per\_proc)$

Routine	**Syntax**	**Description**	**Communication Cost**
MPI_Bcast	MPI_Bcast(void* data, int count, MPI_Datatype datatype, int root, MPI_Comm communicator)	Broadcasts (sends) a message from the process with rank "root" to all other processes in the group	$Time_{startup}$ + ($no\_of\_processor$ × $Time_{data}$)
MPI_Scatter	MPI_Scatter(void* send_data, int send_count, MPI_Datatype send_datatype, void* recv_data, int recv_count, MPI_Datatype recv_datatype, int root, MPI_Comm communicator)	MPI_Scatter is a collective routine that is very similar to MPI_Bcast	$Time_{startup}$ + ($no\_of\_processor$ × $Time_{data}$)
MPI_Gather	MPI_Gather(void* send_data, int send_count, MPI_Datatype send_datatype, void* recv_data, int recv_count, MPI_Datatype recv_datatype, int root, MPI_Comm communicator)	Gathers distinct messages from each task in the group to a single destination task. This routine is the reverse operation of MPI_Scatter	$Time_{startup} + Time_{data}$
MPI_Reduce	MPI_Reduce(void* send_data, void* recv_data, int count, MPI_Datatype datatype, MPI_Op op, int root, MPI_Comm communicator)	Similar to MPI_Gather, MPI_Reduce takes an array of input elements on each process and returns an array of output elements to the root process. The output elements contain the reduced result.	$Time_{startup} + Time_{data}$

Table 3: Collective Communication Routines

2 Code for making ring, pass token from masters to others (ring topology)

```
1  //
   ↪  ------------------------------------------------------------------------
2  // Author: Gahan Saraiya
3  // GiT: http://github.com/gahan9/
4  // StackOverflow: https://stackoverflow.com/users/story/7664524
5  // Website: http://gahan9.github.io/
6  //
   ↪  ------------------------------------------------------------------------
7  // Ring Topology for MPI processes
8  #include "mpi.h"
9  #include <stdio.h>
10
11 int main(int argc, char *argv[]) {
12     int world_size, world_rank;
13
14     MPI_Init( &argc, &argv );
15     MPI_Comm_size( MPI_COMM_WORLD, &world_size );
16     MPI_Comm_rank( MPI_COMM_WORLD, &world_rank );
17
18         int token;
19         if (world_rank != 0) {
20             MPI_Recv(&token, 1, MPI_INT, world_rank - 1, 0, MPI_COMM_WORLD,
                ↪  MPI_STATUS_IGNORE);
21             printf("[process-%d] I received token `%d` from process %d\n",
                ↪  world_rank, token, world_rank-1);
22         }
23
24         else {
25             // Set the token's value if you are process 0
26             token = 2020;
27         }
```

```
28
29          MPI_Send(&token, 1, MPI_INT, (world_rank + 1) % world_size, 0,
            ↪  MPI_COMM_WORLD);
30
31          // Now process 0 can receive from the last process.
32          if (world_rank == 0) {
33              MPI_Recv(&token, 1, MPI_INT, world_size - 1, 0, MPI_COMM_WORLD,
                ↪  MPI_STATUS_IGNORE);
34          printf("[process-%d] I received token `%d` from process %d\n",
            ↪  world_rank, token, world_size-1);
35          }
36
37      MPI_Finalize();
38      return 0;
39  }
```

Output

```
[process-1] I received token `2020` from process 0
[process-2] I received token `2020` from process 1
[process-3] I received token `2020` from process 2
[process-4] I received token `2020` from process 3
[process-5] I received token `2020` from process 4
[process-6] I received token `2020` from process 5
[process-7] I received token `2020` from process 6
[process-8] I received token `2020` from process 7
[process-9] I received token `2020` from process 8
[process-10] I received token `2020` from process 9
[process-11] I received token `2020` from process 10
[process-12] I received token `2020` from process 11
[process-13] I received token `2020` from process 12
[process-14] I received token `2020` from process 13
[process-15] I received token `2020` from process 14
[process-16] I received token `2020` from process 15
```

```
[process-0] I received token `2020` from process 17
[process-17] I received token `2020` from process 16
```

3 Dwarfs of computing

- Dense linear algebra
- Sparse linear algebra
- Spectral methods
- N-body methods
- Structured grids
- Unstructured grids
- Monte Carlo methods

4 Applications for Distributed Parallel Systems

- Internet
- Intranet and peer-to-peer networks
- Email systems
- Cloud file backup/storage systems
- Electronic banking
- Travel reservation systems
- Distributed supercomputers
- Cloud/grid computing
- Mining Pool for Cryptocurrency

5 Network Operating System Vs Distributed Operating System

Parameter	Network Operating System	Distributed Operating System
Definition	The network operating system is the platform to run a system software on a server and allow the server to manage the users, data, groups, security, applications and other networking functions. It is considered as the primary form of an operating system for the distributed architecture. The idea behind the network operating system is to permit resource sharing among two or more computers operating under their own OSs.	The distributed operating system handles a group of independent computers and makes them look like an ordinary centralised operating system. This is achieved by enabling the proper communication between the different computers connected with each other. The main aim of the distributed operating system is the transparency where the use of multiple hardware resources is hidden from the users. The distributed operating system is less autonomous than network operating system as the system has complete control in this environment.
Objective	Provision of local services to the remote client.	Management of hardware resource.
Use	Loosely coupled system employed in heterogeneous computers.	Tightly coupled system used in multiprocessor and homogeneous computers.
Architecture	2-tier client/server architecture.	N-tier client/server architecture.
Level of transparency	Low	High
Basis for communication	Files	Shared memory and messages
Resource Management	Handled at each node.	Global central or distributed management.
Implementation Complexity	Easy	Hard
Scalability	High	Less/moderate
Openness	Open	Closed
Operating system on all nodes	Can be different	Same
Rate of autonomy	High	Low
Fault tolerance	Less	High

Table 4: Network Operating System Vs Distributed Operating System

6 Transperency in Distributed Parallel Systems

A transparency is some aspect of the distributed system that is hidden from the user (programmer, system developer, user or application program). A transparency is provided by including some set of mechanisms in the distributed system at a layer below the interface where the transparency is required. A number of basic transparencies have been defined for a distributed system. It is important to realize that not all of these are appropriate for every system, or are available at the same level of interface. In fact, all transparencies have an associated cost, and it is extremely important for the distributed system implementer to be aware of this. Much of the text of this book describes engineering solutions to archiving these transparencies, and attempts to outline the cost of the solutions. It is a matter for much research how the costs of implementing multiple transparencies interact. This is part of current research in how to reduce operating system and communications stack overheads through such approaches as Application Layer Framing and Integrated Layer Processing. The transparencies are:

- **Access Transparency** There should be no apparent difference between local and remote access methods. In other words, explicit communication may be hidden. For instance, from a user's point of view, access to a remote service such as a printer should be identical with access to a local printer. From a programmers point of view, the access method to a remote object may be identical to access a local object of the same class. This transparency has two parts:
  - Keeping a syntactical or mechanical consistency between distributed and non-distributed access,
  - Keeping the same semantics. Because the semantics of remote access are more complex, particularly failure modes, this means the local access should be a subset. Remote access will not always look like local access in that certain facilities may not be reasonable to support (for example, global exhaustive searching of a distributed system for a single object may be unreasonable in terms of network traffic).
- **Location Transparency** The details of the topology of the system should be of no concern to the user. The location of an object in the system may not be visible to the user or programmer. This differs from access transparency in that both the naming and access methods may be the same. Names may give no hint as to location.
- **Concurrency Transparency** Users and Applications should be able to access shared data or objects without interference between each other. This requires very complex mechanisms in a distributed system, since there exists true concurrency rather than the simulated concurrency of a central system. For example, a distributed printing service must provide the same atomic access per file as a central system so that printout is not randomly interleaved.

- **Replication Transparency** If the system provides replication (for availability or performance reasons) it should not concern the user. As for all transparencies, we include the applications programmer as a user.

- **Fault Transparency** If software or hardware failures occur, these should be hidden from the user. This can be difficult to provide in a distributed system, since partial failure of the communications subsystem is possible, and this may not be reported. As far as possible, fault transparency will be provided by mechanisms that relate to access transparency. However, when the faults are inherent in the distributed nature of the system, then access transparency may not be maintained. The mechanisms that allow a system to hide faults may result in changes to access mechanisms (e.g. access to reliable objects may be different from access to simple objects). In a software system, especially a networked one, it Is often hard to tell the difference between a failed and a slow running process or processor. This distinction is hidden or made visible here.

- **Migration Transparency** If objects (processes or data) migrate (to provide better performance, or reliability, or to hide differences between hosts), this should be hidden from the user.

- **Performance Transparency** The configuration of the system should not be apparent to the user in terms of performance. This may require complex resource management mechanisms. It may not be possible at all in cases where resources are only accessible via low performance networks.

- **Scaling Transparency** A system should be able to grow without affecting application algorithms. Graceful growth and evolution is an important requirement for most enterprises. A system should also be capable of scaling down to small environments where required, and be space and/or time efficient as required.

7 Application of Amdahl's law and Gustafson's law

Amdahl's law

Multiprocessing

The original formulation of Amdahl's law states the impact of inherently sequential portion of a task on the speedup during multiprocessing. Suppose f represents the fraction of the task that is inherently sequential then using N processors the speedup is given by

$$S = \frac{1}{f + (1 - f)N} \tag{1}$$

$$S = \begin{cases} N, & \text{when } f = 0; \text{resulting in an ideal linear speedup} \\ < 5, & \text{when } f = 0.2; \text{independent of N} \\ < 2, & \text{when } f = 0.5; \text{independent of N} \\ \frac{1}{f}, & \text{for large } N; \text{independent of N} \end{cases}$$

This relationship generates pessimism regarding the viability of massively parallel processing especially if we overestimate the value of the fraction f. But, researchers in parallel computation community started suspecting the usefulness and validity of Amdahl's law after observing impressive linear speedups in some large applications.

MEMORY HIERARCHY DESIGN

INSTRUCTION SET AND PROCESSOR DESIGN

Gustafson's law

- beam stress analysis
- surface wave simulation
- unstable fluid flow

8 Write sample MPI code of work pool. Copy code and replace code fragment applying different termination approach. Briefly describe it.

```
1  //
   ↪  ------------------------------------------------------------------------
2  // Author: Gahan Saraiya
3  // GiT: http://github.com/gahan9/
4  // StackOverflow: https://stackoverflow.com/users/story/7664524
5  // Website: http://gahan9.github.io/
6  //
   ↪  ------------------------------------------------------------------------
7  // Work Pool Example MPI
8
9  #include "mpi.h"
10 #include <stdio.h>
11
12 #define f(x)    ((x) * (x))
13
14 int main(int argc, char* argv[]){
15     // MPI variables
16     int dest, no_of_processes, rank;
17     MPI_Status status;
18
19     // problem variables
20     int i, chunk, no_of_chunks, no_of_rects;
21     double area, at, height, lower, width, total, range;
22     double lower_limit, upper_limit;
23
24     // MPI setup
25     MPI_Init(&argc, &argv);
26     MPI_Comm_size(MPI_COMM_WORLD, &no_of_processes);
27     MPI_Comm_rank(MPI_COMM_WORLD, &rank);
28
```

```
29      if (rank == 0) {
30          // Master process
31          // collect parameters
32          fprintf(stderr, "No. of chunk to divide problem into:\n");
33          scanf("%d", &no_of_chunks);
34          fprintf(stderr, "No. of steps per chunk:\n");
35          scanf("%d", &no_of_rects);
36          fprintf(stderr, "Interval's low end:\n");
37          scanf("%lf", &lower_limit);
38          fprintf(stderr, "Interval's high end:\n");
39          scanf("%lf", &upper_limit);
40          printf(
41              "\n============\n"
42              "INPUT STATUS"
43              "\n============\n"
44              "Total Chunks: %d"
45              "\nsteps per chunk: %d"
46              "\nInterval: %lf to %lf"
47              "\n============\n"
48              , no_of_chunks, no_of_rects, lower_limit, upper_limit
49          );
50      }
51
52      MPI_Bcast(&no_of_chunks, 1, MPI_INT, 0, MPI_COMM_WORLD);
53      MPI_Bcast(&no_of_rects, 1, MPI_INT, 0, MPI_COMM_WORLD);
54      MPI_Bcast(&lower_limit, 1, MPI_DOUBLE, 0, MPI_COMM_WORLD);
55      MPI_Bcast(&upper_limit, 1, MPI_DOUBLE, 0, MPI_COMM_WORLD);
56
57      // Collect information and output result
58      if (rank == 0){
59          // if rank is then assign chunk, collect results, print results
60          total = 0.0;
61          if (no_of_processes - 1 < no_of_chunks){
62              chunk = no_of_processes - 1;
63          }
```

```
64          else{
65              chunk = 0;
66          }
67          for(i=1; i <= no_of_chunks; i++){
68              MPI_Recv(&area, 1, MPI_DOUBLE, MPI_ANY_SOURCE, MPI_ANY_TAG,
                ↪ MPI_COMM_WORLD, &status);
69              printf("Area for process %d, is: %f\n", status.MPI_TAG, area);
70              total += area;
71              if (chunk != 0 && chunk < no_of_chunks){
72                  chunk++;
73              }
74              else{
75                  chunk = 0;
76              }
77              MPI_Send(&chunk, 1, MPI_INT, status.MPI_TAG, chunk,
                ↪ MPI_COMM_WORLD);
78          }
79          printf("The area from %f to %f is: %f\n", lower_limit, upper_limit,
            ↪ total);
80      }
81      else {
82          // all other processes, calculate aread for chunk and send results
83          // chunk = rank > no_of_chunks?0:rank;
84          if (rank > no_of_chunks) {
85              chunk = 0;
86          }
87          else {
88              chunk = rank;
89          }
90          while (chunk != 0) {
91              // adjust problem size for subproblem
92              range = (upper_limit - lower_limit) / no_of_chunks;
93              width = range/no_of_rects;
94              lower = lower_limit + range * (chunk - 1);
95              // printf("upper_limit: %lf\n", upper_limit);
```

```
96              // printf("lower_limit: %lf\n", lower_limit);
97              // calculate area for this chunk
98              area = 0.0;
99              for (i=0; i < no_of_rects; i++){
100                 at = lower + i * width + width / 2.0;
101                 // printf("Iteration : %d \t width: %lf\n", i, at);
102                 height = f(at);
103                 area = area + width * height;
104             }
105             // printf("Area ------- %lf", area);
106             // send results and get next chunk
107             dest = 0;
108             MPI_Send(&area, 1, MPI_DOUBLE, dest, rank, MPI_COMM_WORLD);
109             MPI_Recv(&chunk, 1, MPI_INT, 0, MPI_ANY_TAG, MPI_COMM_WORLD,
                ↪ &status);
110         }
111     }
112     // finish
113     MPI_Finalize();
114     return 0;
115 }
```

Output

```
============
INPUT STATUS
============
Total Chunks: 1000
steps per chunk: 20
Interval: 1.000000 to 100.000000
============
Area for process 1, is: 0.109124
Area for process 28, is: 1.371924
Area for process 2, is: 0.130667
```

```
Area for process 8, is: 0.300675
Area for process 10, is: 0.372869
Area for process 16, is: 0.636026
Area for process 20, is: 0.850276
Area for process 24, is: 1.095575
Area for process 26, is: 1.229869
Area for process 20, is: 3.219413
Area for process 26, is: 3.446855
Area for process 20, is: 3.563487
Area for process 26, is: 3.682059
Area for process 26, is: 3.925026
Area for process 26, is: 4.049420
Area for process 26, is: 4.175755
Area for process 26, is: 4.304031
Area for process 26, is: 4.434247
Area for process 26, is: 4.566403
Area for process 26, is: 4.700501
Area for process 26, is: 4.836539
Area for process 26, is: 4.974517
Area for process 20, is: 3.802572
Area for process 26, is: 5.114436
Area for process 20, is: 5.256296
Area for process 26, is: 5.400096
Area for process 20, is: 5.545837
Area for process 26, is: 5.693518
Area for process 20, is: 5.843140
Area for process 26, is: 5.994703
Area for process 26, is: 6.303650
Area for process 26, is: 6.461035
Area for process 26, is: 6.620360
Area for process 26, is: 6.781625
Area for process 26, is: 6.944832
Area for process 26, is: 7.109979
Area for process 26, is: 7.277066
```

```
Area for process 26, is: 7.446094
Area for process 26, is: 7.617063
Area for process 26, is: 7.789972
Area for process 26, is: 7.964822
Area for process 26, is: 8.141612
Area for process 26, is: 8.320343
```

⋮

```
Area for process 26, is: 862.107505
Area for process 26, is: 863.937685
Area for process 26, is: 865.769806
Area for process 26, is: 867.603867
Area for process 26, is: 869.439869
Area for process 26, is: 871.277811
Area for process 26, is: 873.117694
Area for process 26, is: 874.959518
Area for process 26, is: 876.803282
Area for process 26, is: 878.648986
Area for process 26, is: 880.496632
Area for process 22, is: 813.426193
Area for process 24, is: 815.203977
Area for process 26, is: 882.346218
Area for process 26, is: 887.906619
Area for process 26, is: 889.763967
Area for process 26, is: 891.623256
Area for process 26, is: 893.484486
Area for process 26, is: 895.347656
Area for process 26, is: 897.212766
Area for process 26, is: 899.079818
Area for process 26, is: 900.948810
Area for process 26, is: 902.819742
Area for process 26, is: 904.692615
Area for process 26, is: 906.567429
Area for process 26, is: 908.444183
```

```
Area for process 26, is: 910.322878
Area for process 26, is: 912.203514
Area for process 26, is: 914.086090
Area for process 26, is: 915.970606
Area for process 26, is: 917.857064
Area for process 26, is: 919.745461
Area for process 26, is: 921.635800
Area for process 26, is: 923.528079
Area for process 26, is: 925.422299
Area for process 26, is: 927.318459
Area for process 26, is: 929.216560
Area for process 26, is: 931.116601
Area for process 26, is: 933.018583
Area for process 26, is: 934.922506
Area for process 26, is: 936.828369
Area for process 26, is: 938.736173
Area for process 26, is: 940.645918
Area for process 26, is: 942.557603
Area for process 26, is: 944.471228
Area for process 17, is: 781.757927
Area for process 23, is: 783.500780
Area for process 7, is: 785.245574
Area for process 27, is: 786.992308
Area for process 26, is: 946.386795
Area for process 22, is: 884.197744
Area for process 24, is: 886.051211
Area for process 26, is: 955.993735
Area for process 26, is: 961.781186
Area for process 26, is: 963.714218
Area for process 26, is: 965.649190
Area for process 26, is: 967.586103
Area for process 26, is: 969.524956
Area for process 26, is: 971.465750
Area for process 26, is: 973.408485
```

```
Area for process 26, is: 975.353160
Area for process 26, is: 977.299776
Area for process 26, is: 979.248332
Area for process 26, is: 981.198829
Area for process 26, is: 983.151267
Area for process 26, is: 985.105645
Area for process 26, is: 987.061964
Area for process 26, is: 989.020223
Area for process 22, is: 957.920945
Area for process 24, is: 959.850095
Area for process 17, is: 948.304301
Area for process 23, is: 950.223749
Area for process 7, is: 952.145137
Area for process 27, is: 954.068465
Area for process 4, is: 198.763574
Area for process 12, is: 164.383666
Area for process 14, is: 0.540545
Area for process 30, is: 106.693411
Area for process 19, is: 110.589368
Area for process 5, is: 0.206938
Area for process 29, is: 1.445863
Area for process 8, is: 2.892804
Area for process 16, is: 207.643770
Area for process 28, is: 204.959335
Area for process 9, is: 201.407257
Area for process 15, is: 0.587315
Area for process 37, is: 2.107234
Area for process 25, is: 1.161752
Area for process 32, is: 1.679323
Area for process 34, is: 1.844666
Area for process 40, is: 2.387269
Area for process 2, is: 2.787816
Area for process 6, is: 205.852206
Area for process 10, is: 206.747018
```

```
Area for process 20, is: 209.443097
Area for process 1, is: 272.146856
Area for process 41, is: 2.484495
Area for process 39, is: 2.291983
Area for process 36, is: 2.017771
Area for process 33, is: 294.157354
Area for process 13, is: 0.495715
Area for process 21, is: 0.908690
Area for process 18, is: 208.542463
Area for process 3, is: 274.206223
Area for process 38, is: 2.198639
Area for process 11, is: 322.594152
Area for process 35, is: 109.935190
Area for process 31, is: 1.599562
The area from 1.000000 to 100.000000 is: 333332.999798
```