Mandelbrot set – MPI parallelization.

Don Wijesinghe/20164902

1. Static decomposition - Block partitioning.

In task 1, static decomposition was used to solve the Mandelbrot set where the complex plane was divided into horizontal rectangular strips of chunk size,

```
71.     chunk_size = N*N/size;
```

for each processor to do calculations on. The way the Mandelbrot set is calculated remains the same as the serial version where it does calculations by looping over from left to right starting from the bottom of the complex plane to the top using a 1D array. The main change we have implemented is a distributed computation approach where each processor handles a portion of the complex plane. This has been made into a function called ‘mandelbrot’, which takes the parameters rank, chunk size and a pointer to the memory block ‘chunk’ where each processor’s image data is stored to later be combined to form the image.

```
73.     chunk=(float *) malloc(chunk_size*sizeof(float));
```

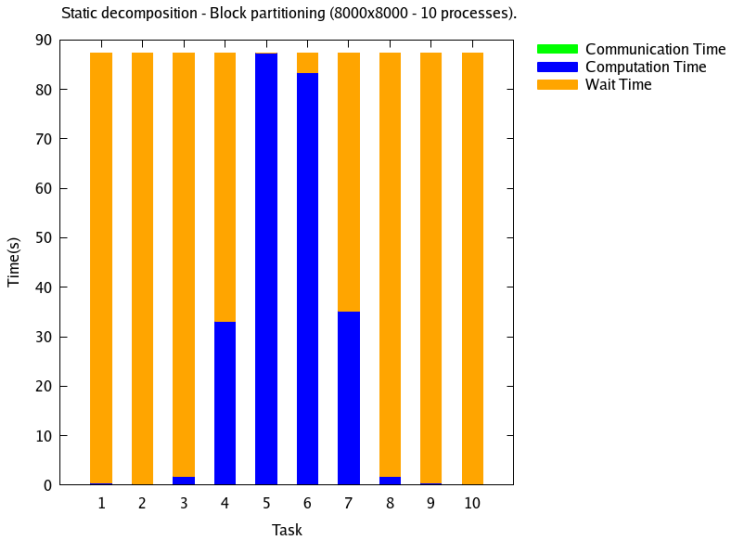
Each process iterates over its assigned portion of the chunk and completes each calculation on points ‘i’ and ‘j’. Line 19 shows that the starting index of calculation for each processor is dependent on its rank.

```
13. void mandelbrot(int rank, int chunk_size, float* chunk){
14.         int i,j,k,l,loop;
15.         float T1,T2;
16.         float complex   z, kappa;
17.
18.         T1 = MPI_Wtime();
19.         loop = rank*chunk_size;
20.         for (l=0; l < chunk_size; l++,loop++) {
21.             i=loop%N;
22.             j=loop/N;
23.
24.             z=kappa= (4.0*(i-N/2))/N + (4.0*(j-N/2))/N * I;
25.
26.             k=1;
27.             while ((cabs(z)<=2) && (k++<MAXITER))
28.                 z= z*z + kappa;
29.
30.             chunk[l]= log((float)k) / log((float)MAXITER);
31.
32.         }
33.         T2 = MPI_Wtime();
34.         float Calc_time = T2-T1;
35.         MPI_Barrier(MPI_COMM_WORLD);
36.         T1 = MPI_Wtime();
37.         float Wait_time = T1-T2;
38.         printf("%d %f %f", rank+1, Wait_time, Calc_time);
39. }
```

The time spent on calculations is calculated by using MPI_Wtime before and after the calculation is finished in each processor and taking the difference. Each processor then enters MPI_Barrier after calculations have finished and the time spent waiting for synchronization is calculated as the wait time.

After completing the Mandelbrot calculations, the image data from each processor is combined into the image array 'x' on the root processor using MPI_Gather. MPI_Gather inherently gathers data in the order of increasing rank numbers, ensuring that the image data falls into memory in the correct order. The order of data gathering is not dependent on which processor finishes its calculations first; rather, it follows the predetermined order of MPI ranks. The image data is then written to the .ppm file same way as the serial version and will be the same for Tasks 2 and 3.

```
57.     mandelbrot(rank,chunk_size, chunk);
58.
59.     if (rank == ROOT){
60.
61.         x = (float *) malloc(N*N*sizeof(float));
62.
63.     }
64.
65.     T1 = MPI_Wtime();
66.     MPI_Gather(chunk,chunk_size,MPI_FLOAT,x,chunk_size,MPI_FLOAT,ROOT,MPI_COMM_WORLD);
67.     T2 = MPI_Wtime();
```



The load balancing analysis reveals a significant imbalance between computation time and wait time. This result arises from the uneven distribution of work across processes due to the Mandelbrot set's characteristics. Processes 4-7, which handle the middle region of the complex plane requiring more iterations, bear most of the workload. Despite this imbalance, communication time across processors remains insignificant. Moreover, static decomposition outperforms the serial version where the speedup is equal to 2.86. In the serial version, calculations on a single processor took 249 seconds, whereas block partitioning across all 10 processors completed calculations in 87 seconds for an 8000x8000 resolution.

2. Dynamic decomposition – Master/worker parallelism

For task 2 we use dynamic decomposition utilising master worker parallelism where the master process coordinates the work distribution and collects results from the worker processes dynamically, unlike static decomposition where tasks are statically assigned to processors before execution begins. In master/worker parallelism the processing units can request new tasks when they become idle, ensuring that computational resources are utilised efficiently and evenly among the workers allowing for load balance. In our implementation of dynamic distribution for solving the Mandelbrot set, the program takes in an arbitrary chunk size value as a command line argument as a parcel of work to be worked on by each worker. The main functions contain the ‘Master’ function which is only ran by the master processor and the ‘Worker’ function ran by the remaining processors.

```
169.     chunk_size = atoi(argv[1]);
170.
171.
172.     if (rank == MASTER){
173.
174.         Master(rank, size, chunk_size);
175.
176.     }
177.     else{
178.
179.         Worker(rank, chunk_size);
180.
181.     }
182.
183.     MPI_Finalize();
184.     return 0;
185. }
```

We first begin with the ‘Master’ function, which takes the parameters rank, size, and chunk size. It first calculates the number of chunks to be worked on by the workers along with any remainder, as shown by line 58-59. The master allocates memory for the image array (‘x’) same as the root processor did in task 1. And it also allocates memory to store the starting indices (‘stored index’) for each worker to begin work from in the image array. Subsequently, the master initiates the workload distribution by sending starting indices (‘next_index’) to worker processes based on the specified chunk size by looping over all the workers. 'next_index’ is incremented by the chunk size each time it loops over to a new worker so that the next portion of the complex plane is sent to be worked on. We can also see that this value is stored in memory in the ‘stored_index’ array so that once the results are retrieved from that specific worker, it can be placed in the correct place in memory inside the image array. Any remaining chunks are evenly distributed among workers during the initial distribution. Adjustments to chunk size are made within the loop, ensuring proper index adjustment as seen by line 71. The master

process then enters a loop to manage task distribution and result collection until all parcels of completed work has been collected.

```
56. void Master(int rank, int size, int chunk_size){
57.         MPI_Status status;
58.         int chunks_left = N*N/chunk_size;
59.         int remainder = N*N % chunk_size;
60.
61.         float* x = (float *) malloc(N*N*sizeof(float));
62.         int* stored_index = (int *) malloc((size-1)*sizeof(int));
63.         int next_index = 0;
64.
65.         /*initial distribution*/
66.         for (int dest=1; dest < size; dest++){
67.
68.                 stored_index[dest-1] = next_index;
69.                 MPI_Send(&next_index, 1, MPI_INT, dest, WORK_TAG, MPI_COMM_WORLD);
70.
71.                 next_index += (dest <= remainder) ? chunk_size +1 : chunk_size;
72.         }
73.
74.         int recv_count, worker;
75.
76.         /*while work remains to be done, send out new work and receive finished work*/
77.         while (chunks_left> 0){
78.
79.                 MPI_Probe(MPI_ANY_SOURCE, WORK_TAG,MPI_COMM_WORLD, &status);
80.                 MPI_Get_count(&status, MPI_FLOAT, &recv_count);
81.                 worker = status.MPI_SOURCE;
82.
83.                 MPI_Recv(&x[stored_index[worker-1]],recv_count,MPI_FLOAT,worker,WORK_TAG,MPI_COMM_WORLD,&status);
84.                 if (chunks_left <= size-1){
85.
86.                         MPI_Send(&next_index,1,MPI_INT,worker,FINISH_TAG,MPI_COMM_WORLD);
87.                 }
88.                 else{
89.                         stored_index[worker-1] = next_index;
90.                         MPI_Send(&next_index, 1, MPI_INT, worker, WORK_TAG, MPI_COMM_WORLD);
91.                         next_index += chunk_size;
92.                 }
93.                 chunks_left--;
94.         }
95.         MPI_Barrier(MPI_COMM_WORLD);
96.         write_to_file(x);
97.         free(x);
98.         free(stored_index);
99. }
```

We utilise MPI_Probe with MPI_ANY_SOURCE to detect incoming completed work and identify the corresponding worker. Since initial work distributions may have varying chunk sizes, MPI_Get_count is employed to determine the size of the received work. With this information, MPI_Recv retrieves the work parcel from the worker, placing it into the image array using the starting index in ‘stored_index’ specific to that worker. Communication utilises distinct tags: WORK_TAG for sending and receiving work between master and workers, and FINISH_TAG to instruct a worker to stop work if the condition in line 84 is met. Otherwise, the master sends the next starting index to the finished worker for new work, updating the 'stored_index' array accordingly and incrementing 'next_index'.

```
3. #define WORK_TAG 1
4. #define FINISH_TAG 2
```

When the remaining number of chunks is less than or equal to the number of workers (size - 1) in line 84, all remaining work parcels have been sent to the workers, and while each worker works on its last parcel, the master sends a message with the FINISH_TAG to tell that worker to stop once they finish with that work and requests for new work. Then the master loops around to receive the remaining work until ‘chunks_left’ is equal to 0. Once the loop ends, the image data is written to the .ppm file.

Now on to the worker function which is responsible for computing the Mandelbrot set for its assigned chunk and sending the results back to the master. it checks for any remaining work from the initial distribution. If this is the case, ‘extra’ is set to 1, accounting for the additional chunk (line 107). Then we allocate memory for the ‘chunk’ array accounting for this extra chunk sent in the intital distribution of work. Subsequently, we enter the while loop where workers receive the starting index from the master to begin computation.

```
101. void Worker(int rank, int chunk_size){
102.         int extra,start_index;
103.         MPI_Status status;
104.
105.         /*initial run can contain remainder of chunks*/
106.         int remainder = N*N%chunk_size;
107.         extra = (rank <= remainder) ? 1 : 0;
108.
109.         double T1, T2, Wait_Time;
110.         double Comp_Time=0; double Comm_Time=0;
111.
112.         float* chunk=(float *) malloc((chunk_size + extra)*sizeof(float));
113.
114.         while(1){
115.
116.                 /*recieve work*/
117.                 T1 = MPI_Wtime();
118.                 MPI_Recv(&start_index,1, MPI_INT,MASTER,MPI_ANY_TAG,MPI_COMM_WORLD,&status);
119.                 T2 = MPI_Wtime(); Comm_Time += T2 - T1;
120.
121.                 if (status.MPI_TAG == FINISH_TAG)
122.                         break;
123.
124.                 /*do work*/
125.                 T1 = MPI_Wtime();
126.                 mandelbrot(chunk_size + extra, start_index, chunk);
127.                 T2 = MPI_Wtime(); Comp_Time += T2 - T1;
128.
129.                 /*send completed work back to master*/
130.                 T1 = MPI_Wtime();
131.                 MPI_Send(chunk,chunk_size + extra, MPI_FLOAT,MASTER,WORK_TAG,MPI_COMM_WORLD)
132.                 T2 = MPI_Wtime(); Comm_Time += T2 - T1;
133.
134.                 extra = 0;
135.         }
```

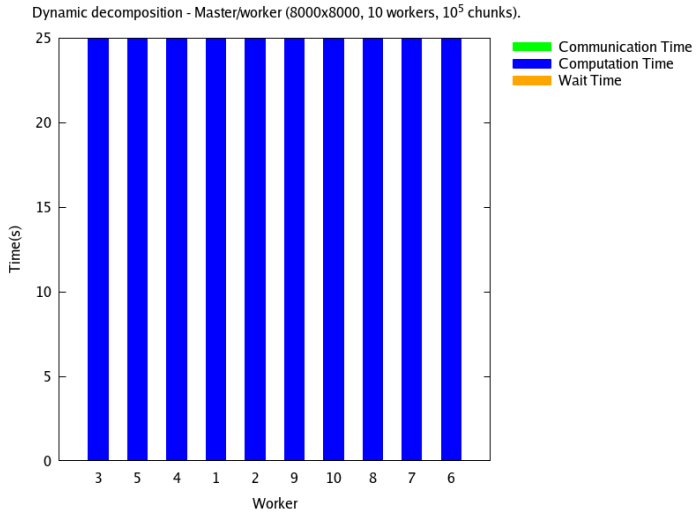
Within the loop, the worker process computes the Mandelbrot set for its assigned chunk using the ‘mandelbrot’ function.

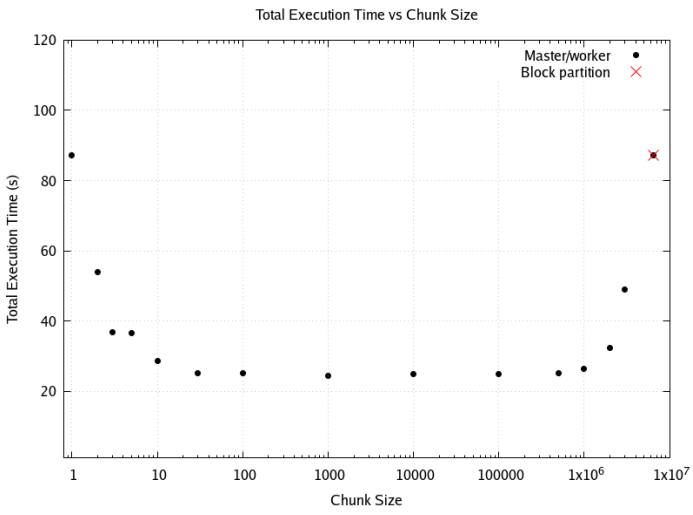
```
16. void mandelbrot(int chunk_sent,int start_index,  float* chunk)
```

This function works the same as in task 1 except that ‘loop’ is now equal to ‘start_index’ and is no longer dependent on the rank. Also, calculation and wait times are no longer computed within the function. After computations, the chunk of work, sized 'chunk_size + extra', is sent back using MPI_Send in line 131. 'extra' is reset to 0 in the first loop since future work sizes match 'chunk_size'. The worker halts its work by breaking the loop upon receiving the 'FINISHED_TAG' in lines 121-122. Time spent on calculation and communication is calculated within the loop, along with wait time for workers in the remaining part of the function, using the same method as task 1.

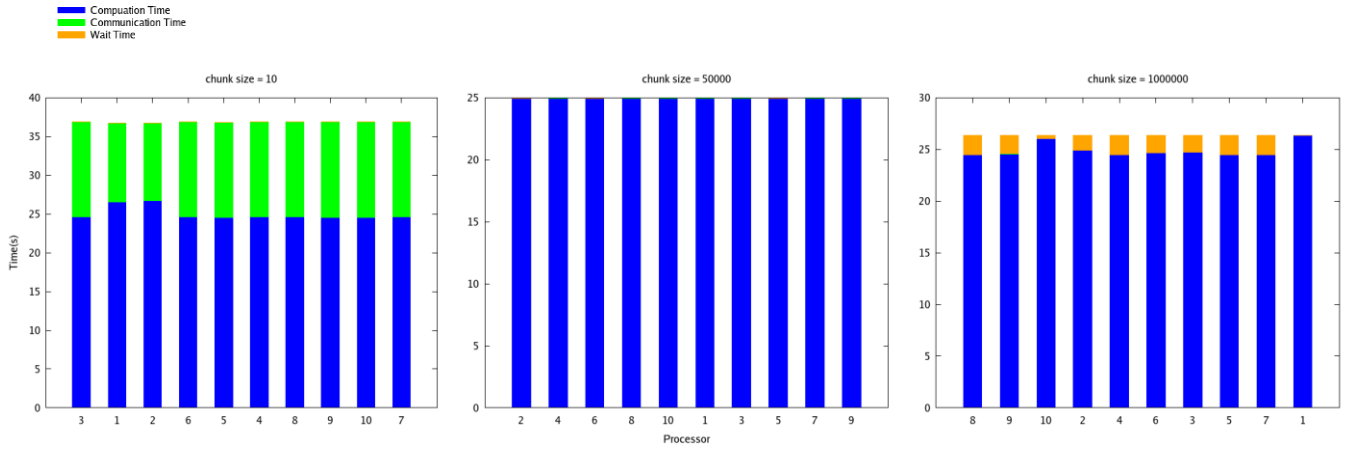
```
137.          T1 = MPI_Wtime();
138.          MPI_Barrier(MPI_COMM_WORLD);
139.          T2 = MPI_Wtime(); Wait_Time = T2 - T1;
140.
141.          printf("%d %f %f %f\n",rank, Comp_Time, Wait_Time, Comm_Time);
142.
143.          free(chunk);
144. }
```

Looking at the load balance for dynamic decomposition from the graph on the next page, with a chunk size of 100,000 and 10 workers, immediate improvements over static decomposition in task 1 are evident. All worker processes exhibit uniform calculation times, with negligible time spent on waiting and communication. Comparing this to the serial version which took 249s for computation, we see even better improvements in computation times at around 25s across all workers resulting in a speed up value of 10 for the given chunk size.





From the graph above of total execution time vs chunk size (excluding writing to file) for the master processor we observe that the optimum chunk size ranges from 100 to 100,000. Within this range, there is a high computation to communication ratio, resulting in optimal load balance and minimised total execution time. For very large values of chunk size (10^6 - $6.4x10^6$ chunks), the execution time increases significantly. This is due to larger chunks lead to uneven distribution of work among workers, where some workers finish their tasks much earlier than others, leading to idle time which leads to load imbalance. We can also see that when the chunksize is 1/10 of total amount of work ($6.4x10^6$ chunks), the total execution time approaches that of Task 1 - 87s as expected.



For very small values of chunk size (1-50), the execution time increases gradually as the chunk size decreases. Also we see that the communication overhead associated with the task distribution and receiving completed work also increases as we decrease the chunk size. With smaller chunk sizes, the number of tasks generated increases proportionally. Each task needs to be distributed from the master process to the worker processes and then the results need to be retrieved. This incurs overhead in terms of communication between processes. Hence the increase in communication times. When the chunk size is very small, each worker process is assigned a smaller portion of the overall workload. With smaller chunks, the computational workload per task decreases. However, workers don't seem to complete its assigned tasks more quickly resulting in shorter calculation times and rather computation time stays at a constant value. This suggests that the program benefits from coarse granularity in its parallelization strategy. Coarse-grained parallelism is well-suited for efficiently solving the Mandelbrot set, as finer granularity does not reduce computation times. Furthermore, we have good load balancing in this range, as computation to wait time ratio gets higher. Below a chunk size of 10, the total execution time rapidly increases due to significant communication overhead.

3. Static decomposition - Cyclic partitioning.

Cyclic partitioning is another static decomposition technique used in parallel computing to distribute computational tasks evenly among multiple processing units. In the case of solving the Mandelbrot set, the complex plane is again divided into rectangular horizontal strips, by the number of processors like we did in task1 of chunk size, N*N/size. this strip is then divided into sub strips for all processors of chunk size,

```
105.      chunk_size = (N*N/size)/size;
```

With cyclic partitioning, the assignment of these strips to processing units follows a round-robin pattern, in a sequential manner. For an example, if we have 4 processors and 16 strips, the first processing unit would be assigned the strips 1,5,9,etc, the second processing unit would be assigned strips 2,6,10,etc and so on until all strips are assigned. This means that each processor obtains 4 strips each on the complex plane. Thus, for an arbitrary number of processors, the number of strips assigned to each processor will be the number of processors itself ('size'). The size of this array is determined by multiplying the chunk size by the number of strips allocated to each processor.

```
108.      chunk=(float *) malloc(chunk_size*size*sizeof(float));
```

Each processing unit independently computes the Mandelbrot set for the assigned strips of the complex plane using the 'mandelbrot' function which has the following changes made to it from task 1.

```
16. void mandelbrot(int rank, int size, int chunk_size, float* chunk){
17.         int i,j,k,l, loop, offset;
18.         double T1,T2;
19.         float complex   z, kappa;
20.
21.         T1 = MPI_Wtime();
22.         offset = 0;
23.         loop = rank*chunk_size + offset;
24.         for (int cycle=1; cycle<=size; cycle++){
25.
26.
27.                 for (l= 0; l < chunk_size; loop++, l++) {
28.                         i=loop%N;
29.                         j=loop/N;
30.
31.                         z=kappa= (4.0*(i-N/2))/N + (4.0*(j-N/2))/N * I;
32.
33.                         k=1;
34.                         while ((cabs(z)<=2) && (k++<MAXITER))
35.                         z= z*z + kappa;
36.
37.                         chunk[l+(cycle-1)*chunk_size]= log((float)k) / log((float)MAXITER
38.
39.                 }
40.
41.                 offset = cycle*chunk_size*size;
42.
43.                 loop = rank*chunk_size + offset;
44.         }
45.
46.         T2 = MPI_Wtime();
47.         double Calc_time = T2-T1;
48.         MPI_Barrier(MPI_COMM_WORLD);
49.         T1 = MPI_Wtime();
50.         double Wait_time = T1-T2;
```

The computation now involves each processor doing the Mandelbrot calculation iteratively through each strip. In each cycle the image index value ('loop') will be offset by the total chunk size of the strips seen in line 41. Note that the 'loop' value is still dependant on the rank of the processor same as in task 1. After calculation for a single strip is complete it is placed in

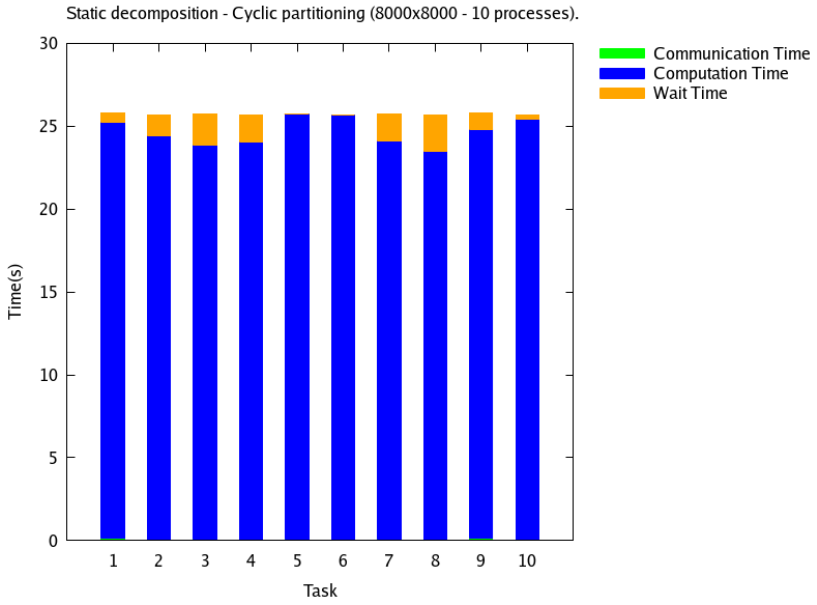
the chunk array. After all calculations are done MPI_Gather is used to send these arrays back to the root processor and placed in the image array.

```
110.     if (rank == ROOT){
111.
112.         x = (float *) malloc(N*N*sizeof(float));
113.
114.     }
115.
116.     mandelbrot(rank, size, chunk_size, chunk);
117.     T1=MPI_Wtime();
118.     MPI_Gather(chunk, chunk_size*size, MPI_FLOAT, x , chunk_size*size, MPI_FLOAT, ROOT, MPI_COMM_WORLD);
119.     T2=MPI_Wtime();
120.     printf("comm time %f\n", T2-T1);
121.     if (rank==ROOT){
122.
123.         sort_image(rank, size, chunk_size, x);
124.         write_to_file(x);
125.         free(x);
126.     }
127.     free(chunk);
128.     MPI_Finalize();
129.     return 0;
```

Since the image data is not ordered properly to form the appropriate image, a sorting algorithm is applied in line 123 before writing to the .ppm file. The sorting algorithm for the image array is shown below.

```
74. void sort_image(int rank, int size, int chunk_size, float* x){
75.         int ncycles = size; /*number of cycles*/
76.         int nsubstrips = size; /*number of chunks per cycle*/
77.
78.         float *temp_x = malloc(N*N*sizeof(float));
79.         int src_index, dest_index;
80.
81.         for (int i=0; i<nsubstrips; i++){
82.                 for (int j=0; j<ncycles; j++){
83.
84.                         src_index = (j*nsubstrips+i)*chunk_size;
85.                         dest_index = (i*ncycles+j)*chunk_size;
86.
87.                         memcpy(&temp_x[dest_index], &x[src_index], chunk_size*sizeof(float));
88.                 }
89.         }
90.         memcpy(x, temp_x, N*N*sizeof(float));
91.         free(temp_x);
92. }
```

It first calculates the number of cycles and blocks per cycle based on the number of processes. Then, it initializes a temporary array (‘temp_x’) to store the sorted data. The function iterates over each block and cycle, determining the source and destination indices for copying data between the original array and the temporary array. The source index represents the starting index of the chunk of the strip data to be copied from ‘x’. while the destination index represents the starting index where the strip of data will be copied to in the temporary array. Using memcpy, it copies chunks of data from ‘x’ to ‘temp_x’ in a specific order determined by the cyclic pattern. Once all data has been rearranged, the sorted data is copied back to the original array ‘x’.



The load balance for the cyclic partitioning is significantly improved from the block partitioning as work is more evenly distributed among processes due to each processor being able to calculate a portion of the middle section of the complex plane where more computation is required. The speed up relative to the serial version is equal to 9.57 and cyclic partitioning is 3.34 times faster than block partitioning.

In summary, dynamic decomposition with master/worker parallelism emerges as the most effective approach for solving the Mandelbrot set. The observed behaviour from the execution time vs chunk size suggests that using an optimal chunk size can ensure the best performance where the benefits of parallelism are maximised while minimising communication overhead. By adapting task allocation based on workload and processor availability, dynamic decomposition minimises idle time for superior load balance. Following closely is cyclic partitioning, which also proves to be an effective strategy for Mandelbrot set computation, offering nearly comparable performance to master/worker parallelism in workload distribution for load balancing. These two strategies results in significant speedup relative to the serial version, unlike block partitioning which suffers significantly from load imbalance due to inefficient workload distribution.