ANIRUDH SETH
MAGNUS PIERRAU

Homework Assignment 1

Parallel Computations for Large- Scale Problems

2020-02-09

1. A **processor** is a physical hardware device which is made up of electronic circuits and has the capability of assessing the memory and computing new data values. The CPU can perform the most basic arithmetic, logic and I/O operations. A **process**, on the other hand, is a computational activity that is assigned to a processor. It is an instance of a computer program that is sequentially executed.

2. As per **Amdahl's law** [1], if the fraction of the computation that cannot be divided into concurrent tasks is ***f***, and we assume that there is no overhead when computation is divided into concurrent parts, the time to perform the computation with ***p*** processors can be computed as

$$ft_s + (1-f)t_s/p$$

Here, t_s is the execution time for the fastest possible sequential algorithm on a single processor. The speedup factor can be written as:

$$\begin{aligned} S(p) =& \frac{t_s}{ft_s + (1-f)t_s/p} \\ =& \frac{p}{1+(p-1)f} \\ =& \frac{100}{1+(100-1)\frac{10}{100}} \\ =& 9.174 \end{aligned}$$

The peak performance of this system as measured in Gflops will thus be $9.174 * 2 =$ **18.3486**.

3. The speedup factor, **S(p)** [1] is a measure of relative performance, which can be defined as :

$$S(p) = \frac{t_s}{t_p}$$

where t_s is the execution time using a single processor system (with the best sequential algorithm) and t_p is the execution time using a multiprocessor with **p** processors. The efficiency, ***E*** is defined as:

$$\begin{aligned} E =& \frac{\text{Execution time using one processor}}{\text{Execution time using a multiprocessor} \times \text{number of processors}} \\ =& \frac{t_s}{t_p \times p} \\ =& \frac{S(p)}{p} \end{aligned}$$

The maximum speedup possible is usually **p** with p processors (linear speedup). This would be achieved when the computation can be divided into equal-duration processes,with one process mapped onto an individual processor and there is no overhead involved in the parallel solution.

$$S(p) \leq \frac{t_s}{\frac{t_s}{p}} = p$$

The efficiency becomes **100%** when ***S(p)=p*** as mentioned above. For efficiency to be greater than 100% it must hold that $S(p) \geq p$. This *Super linear speedup* would arise in occasions of using a sub optimal sequential algorithm. If a purely deterministic parallel algorithm were to achieve a speedup better than p, then the parallel algorithm could be emulated on a single processor one parallel part after the another and that would suggest that the original sequential algorithm was not optimal.

Furthermore, the efficiency $\nu_p = S(p)/p$ can be proven to be 1 when the entire program is parallelizable.

If we follow Amdahl's law, we get that

$$\nu_p = \frac{S_p}{p} = \frac{\frac{p}{1+(p-1)f}}{p} = \frac{1}{1+(p-1)f}$$

If we assume the entire program is parallelizable (will give maximum speedup), i.e. the fraction of time spent in the sequential part of the program, f, is 0, then the system efficiency is

$$\nu_p = \frac{1}{1+(p-1)\times 0} = 1.$$

If we instead consider Gustafson's law, and assume that the parallel execution time is fixed, then

$$\nu_p = \frac{S'_p}{p} = \frac{f' + (1-f')p}{p} = \frac{f'}{p} + (1-f').$$

If we yet again assume that the program is fully parallelized, i.e. $f' = 0$, the we get a system efficiency of

$$\nu_p = 0 \times p + 1 - 0 = 1.$$

Thus, we confirm that the maximum system effiency for a system is 100%. This number indicates how well the system is parallelized with respect to the number of processes. If every process does its maximum to speed up the program, then the efficiency will be 100%. Since a processor cannot parallelize more than 100% of the program (unless sequentiall algorithm is not optimal), this is the maximum possible system efficiency.

4. The rank sort in the lectures assumed that we had the same number of processors ***P*** as the number of elements in the list **N**. In this case each rank calculation of a number $n \in [0, N]$ can be assigned to a unique processor **p**. Once $rank_n$ is known, the processor p can put the element at the correct position.In cases when we have $n > p$, a list of n/p numbers can be assigned to each processor p. For example, if $P = 100$ and $N = 1000$ we assign 10 numbers to each processor. Each processor computes the rank of each of its 10 numbers. The figure below shows the architecture for the above

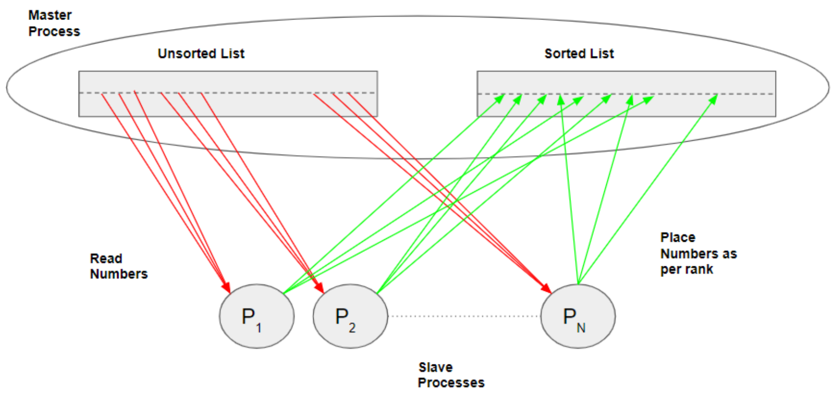


Figure 1: Master-Slave Approach for Parallel Rank Sort with $n > p$

5. (a) To calculate the fraction we need to use **Gustafson's Law** (assuming fixed parallel execution time T_P). If we assume that a fraction f' is spent in the sequential part of the execution, then, with $T_1 = 64s$ as the execution time for the fastest possible sequential algorithm on the given problem, $P = 8$ being the number of processors, and $T_P = 22s$ we can find the fraction by solving for f' in Gustafson's law:

$$\begin{aligned} T_1 &= f'T_P + (1-f')PT_P \\ 64 &= 22f' + 8 \cdot 22(1-f') \\ 64 &= 22f' - 176f' + 176 \\ f' &= \frac{112}{154} \approx 0.727 \end{aligned}$$

We find that our parallel program spends 72.7% of the execution time in the sequential part of the code ($0.727 \times 22 = 15.994$ sec.).

(b) Plugging this into Gustafson's law we get a parallel speedup of

$$S_P = f' + (1 - f')P = 0.727 + 0.273 \cdot 8 = 2.911 \tag{1}$$

(c) For each phase the speedup can be calculated can be calculated as in Q.2

Phase 1

$$\begin{aligned} S(p)_{Phase1} =& \frac{1}{0.2 + 0.8/5} \\ =& 2.78 \end{aligned}$$

Implying that, $t_{s(phase1)} = 2.78t_P$

Phase 2

$$\begin{aligned} S(p)_{Phase2} =& \frac{1}{0.2 + 0.8/10} \\ =& 3.57 \end{aligned}$$

Therefore ,$t_{s(phase2)} = 3.57t_P$

Phase 3

$$\begin{aligned} S(p)_{Phase1} =& \frac{1}{0.6 + 0.4/15} \\ =& 1.59 \end{aligned}$$

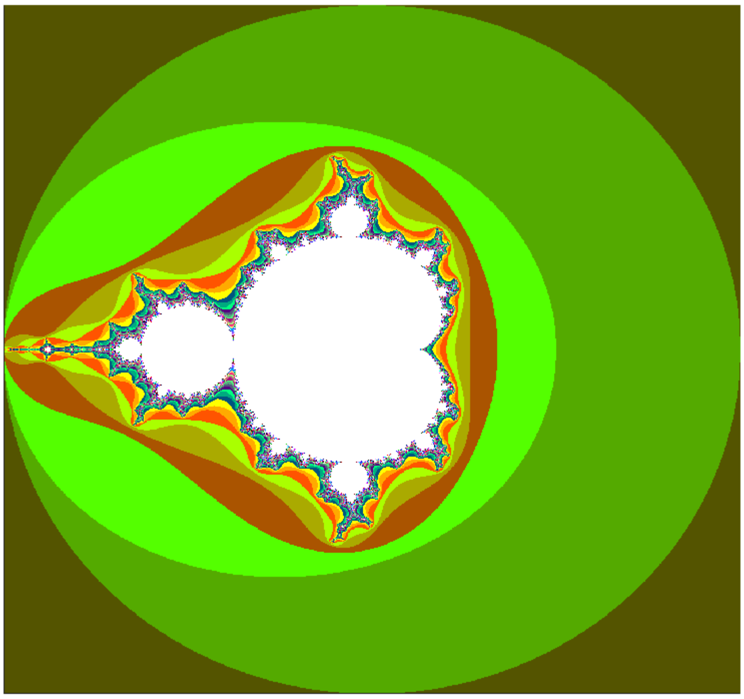
Therefore ,$t_{s(phase3)} = 1.59t_P$

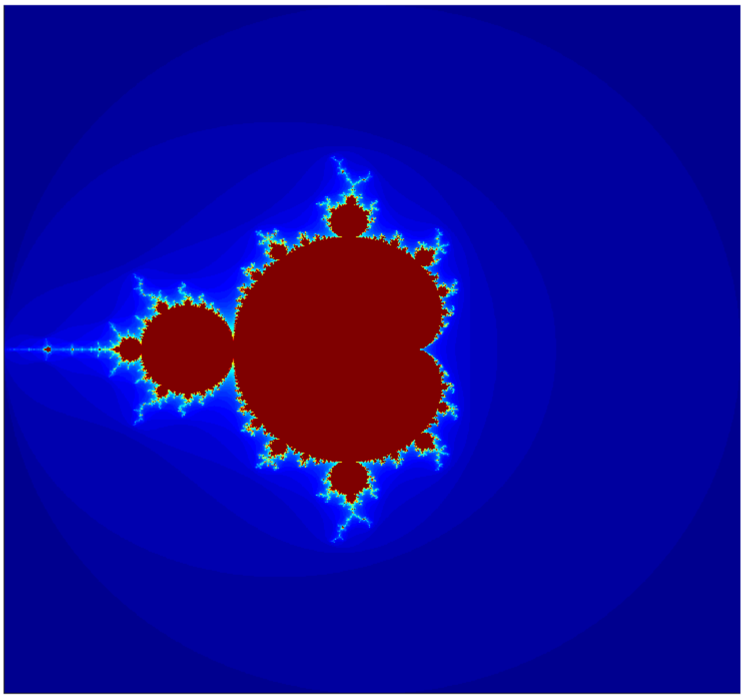
The speedup for the total program can then be calculated as :

$$\begin{aligned} S(p) =& \frac{t_{s(phase1)} + t_{s(phase2)} + t_{s(phase3)}}{t_P} \\ =& \frac{2.78t_P + 3.57t_P + 1.59t_P}{t_P} \\ =& 7.94 \end{aligned}$$

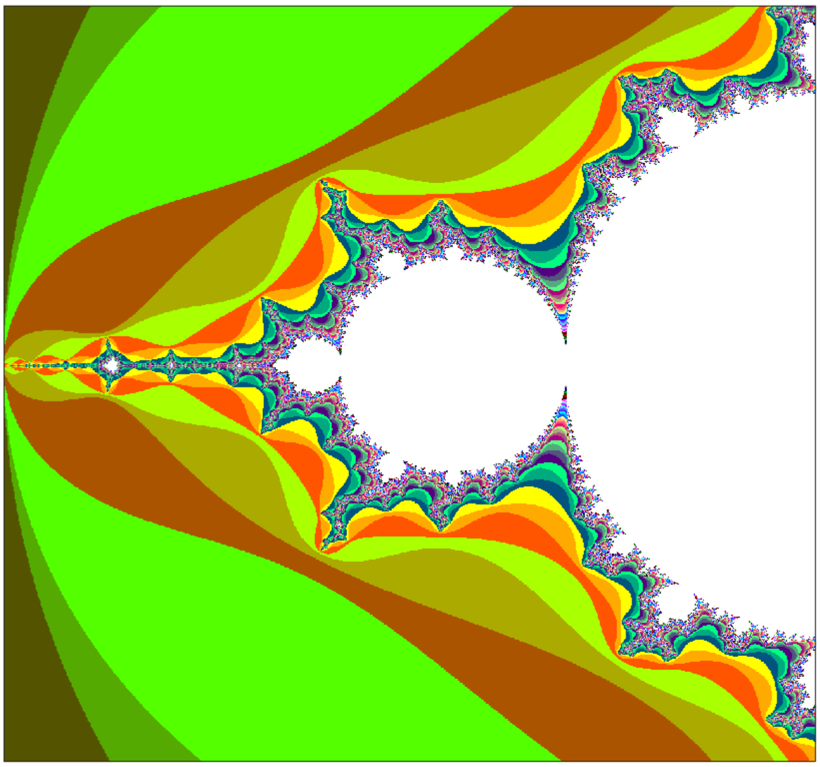
6. (a) The Mandelbrot program was implemented using `C` and the `MPI` paradigm. The source code is available in the Appendix.

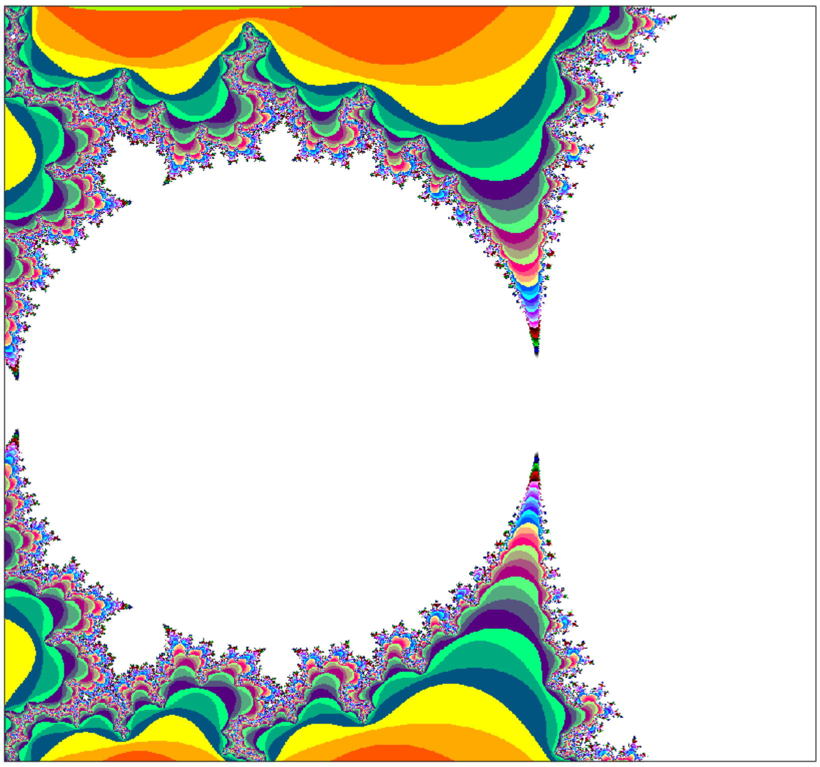
   (b) The figures from the lecture notes have been recreated (colorcube and jet colormaps from MATLAB were used).

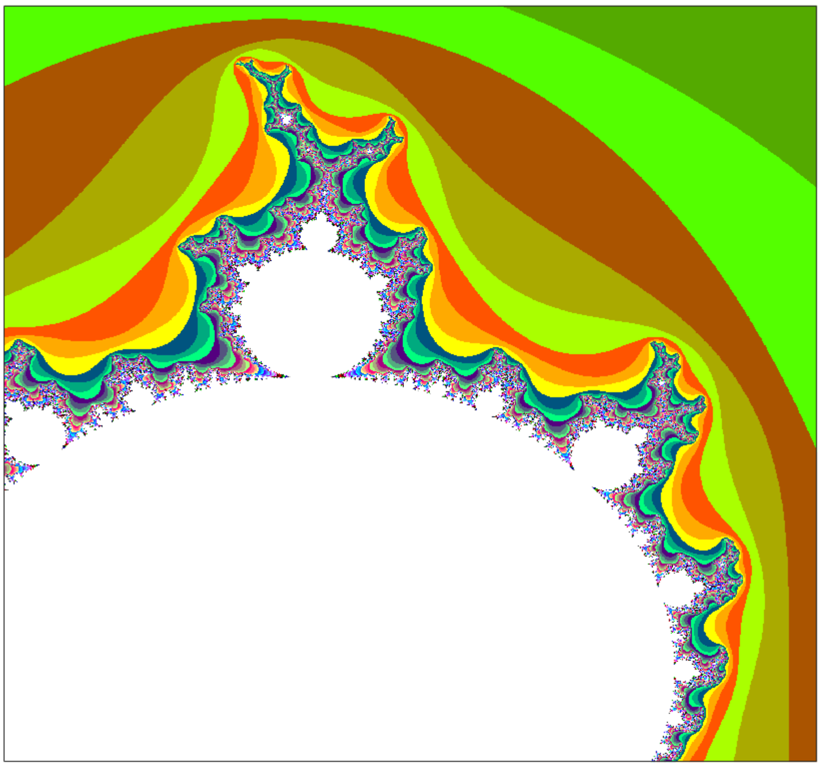


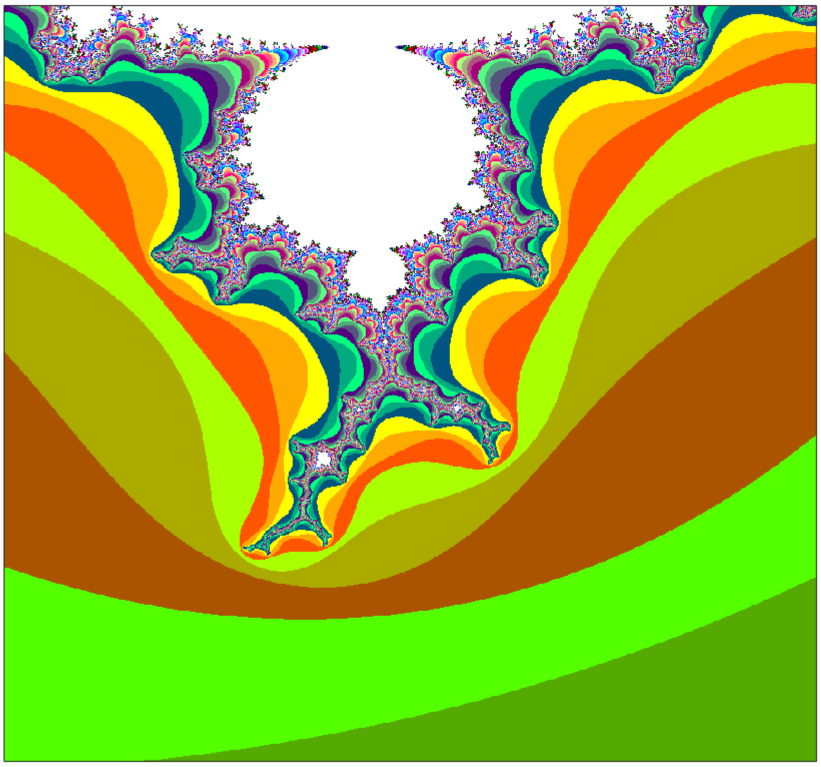


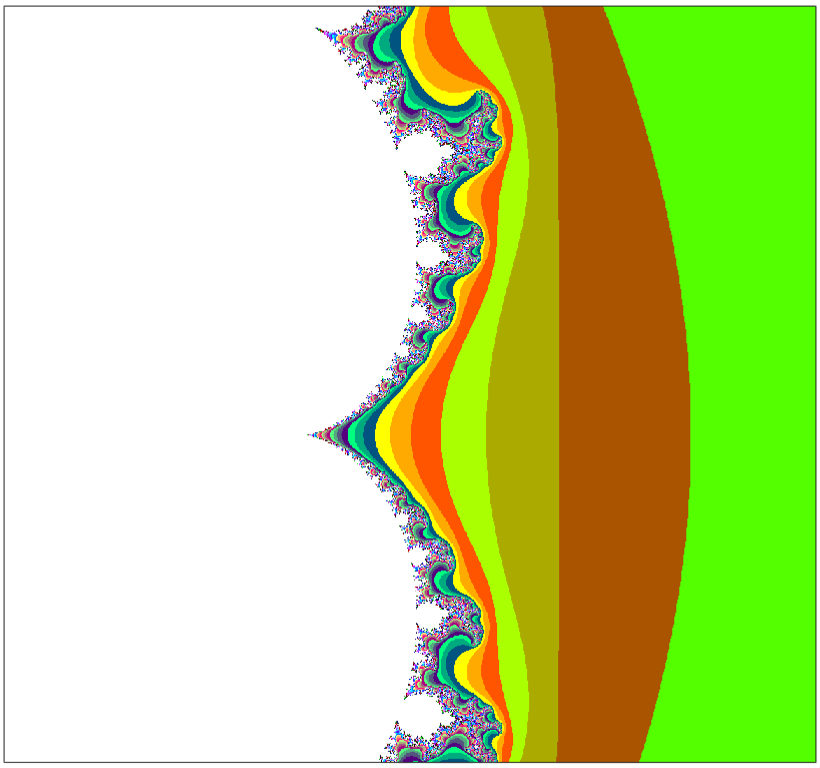
(c) Some of the interesting parts of the figure were magnified by selectively computing certain regions and using high resolution (10240 X 10240 with 256 iterations).











References

[1] Barry Wilkinson and Michael Allen. 2004. Parallel Programming: Techniques and Applications Using Networked Workstations and Parallel Computers (2nd Edition). Prentice-Hall, Inc., USA.

Appendix

```c
#include <stdio.h>
#include <math.h>
#include <mpi.h>
#include <stdlib.h>

int mandelBrot(double Cx, double Cy) {
  double Zx, Zy;
  double Zx2, Zy2;
  int it;
  int MaxIter=100;
  Zx=0.0;
  Zy=0.0;
  Zx2=Zx*Zx;
  Zy2=Zy*Zy;

  for (it = 0;it<MaxIter && ((Zx2+Zy2)<4);it++) {
    Zy=2*Zx*Zy+Cy;
    Zx=Zx2-Zy2+Cx;
    Zx2=Zx*Zx;
    Zy2=Zy*Zy;
  };
  return it;
}

int main(int argc, char ** argv) {
  if (argc!= 6) {
    printf("Usage:   %s <xmin> <xmax> <ymin> <ymax> <Filename>\n", argv[0]);
    printf("Example: %s -2 2 -2 2 fig1.txt\n", argv[0]);
    exit(0);
  }
    int P,rank;
  int xRes=2048;
  int yRes=2048;
  double Cx, Cy,rc,tag;
  double xMin=atof(argv[1]);
  double xMax=atof(argv[2]);
  double yMin=atof(argv[3]);
  double yMax=atof(argv[4]);
  const char* filename = argv[5];
  double PixelWidth=(xMax-xMin)/xRes;
  double PixelHeight=(yMax-yMin)/yRes;
  FILE *fp;
  int *data,*data_start;
  tag=100;
  MPI_Status status;
  rc=MPI_Init(&argc,&argv);
  rc=MPI_Comm_size(MPI_COMM_WORLD,&P);
  rc=MPI_Comm_rank(MPI_COMM_WORLD,&rank);

  int wp=xRes/P;
  if(xRes%P!=0){
      printf("Resolution can not be evenly divided.");
      return 0;
    }
  int start = rank*wp;
  int end = start + wp;

  data=(int*)malloc(wp*yRes*sizeof(int));
  data_start = data;

  for (int iX=start;iX<end;iX++) {

    Cx = xMin + iX*PixelWidth;
    for (int iY = 0; iY < yRes; iY++) {
      Cy = yMin + iY*PixelHeight;
      if (fabs(Cy) < PixelHeight / 2)
        Cy = 0.0;

      int it = mandelBrot(Cx, Cy);
      *data++ = it;
    }
  }
    data=data_start;
  if (rank == 0) {
    fp = fopen(filename, "w");
    printf("Process %d completed.\n", rank);
    for (int i = 0; i < wp; i++) {
      for (int j = 0; j < yRes; j++) {
        fprintf(fp, "%hhu ", (unsigned char) data[j + i * yRes]);
      }
      fprintf(fp, "\n");
    }

    fclose(fp);

    for (int k = 1; k < P; k++) {
      MPI_Recv(data, wp*yRes, MPI_INTEGER, k, tag, MPI_COMM_WORLD, &status);
      printf("Process %d completed.\n", k);
      fp = fopen(filename, "a");
      for (int i = 0; i < wp; i++) {
        for (int j = 0; j < yRes; j++) {
          fprintf(fp, "%hhu ", (unsigned char) data[j + i * yRes]);
        }
        fprintf(fp, "\n");
      }

      fclose(fp);
    }
  } else {
    rc=MPI_Send(data, wp*yRes, MPI_INTEGER, 0, tag, MPI_COMM_WORLD);
  }

 rc=MPI_Finalize();
  return 0;
}
```