CS/SE 4F03 Distributed Computer Systems

Final Project:

Moby Dick

Noel Brett - 1137089
Ashley General - 1141035
Michael Liut - 1132938
Matthew Halleran - 1154779

1. Video

You can find our video on the SVN under the subdirectory FINAL.

2. Parallel Approach

We decided to parallelize the code using MPI as the advantages provided by MPI would endure greater benefits than using OpenMP or OpenACC. OpenACC would produce a great deal of overhead and would result in slower times per computation, as well as it would require the systems to have a graphics card and the appropriate compiler. OpenMP could only be run in shared memory computers and would require the appropriate compiler. MPI is already included with the C compiler and can accommodate for distributed or shared memory architectures.

2.1 Description of Implementation

The files where you can find our implementation are: main.cc and renderer.cc. Main.cc starts up the processes and sends them to renderer.cc. Once in renderer.cc, the processes divide the work evenly by splitting the picture by the height, letting each one of them compute a row of the final image. When each has finished computing, MPI_Reduce is then called in main on process 0 so it is able to put the whole image together and save the final picture.

2.2 Explanation of Parallelization

We followed this approach since renderer.cc was the only file that had room for improvement with parallelization as the for loops did not have any dependencies on previous iterations. Moreover, splitting the work according to the height would mean that the image is being rendered by row rather than by column. Computing by row is much faster compared to by columns as the cache stores the rows from the main memory and the loop will be accessing the cache data. This means that the cache will be changed less often, yielding faster times.

2.3 Navigation

We navigate through the object by slowly incrementing the camera coordinate and the look at coordinate. We created a python script (included with source code) that produces params.dat files based on elif statements that update the x, y, and z coordinates of the two coordinates. We came up with the numbers in the elif statements using our knowledge of graphical programming (OpenGL) as well as some trial and error. To find a good path we produced images of low quality (480p) so that we could use trial and error without wasting too much time. Once we found a path through the cube we rendered the images in 4K. In order to make the camera go straight, we calculated the vector between the camera and its look at position. We then divided that vector by an arbitrary number to produce small increments which we can add to the cameras position.

3. Summary of computation

Specs of Machines used

Machine #1 (Matt's Laptop)

- Intel(R) Core(TM) i7-3632QM CPU @ 2.20GHz
- total # of cores : 4
- total computation time : 66 hours
- number of frames : 4800
- seconds per frame : ~50

Machine #2 (Amazon Web Services - c3.8xlarge)

- Intel(R) Xeon(R) CPU E5-2680 v2 @ 2.80GHz
- cache size : 25600 KB
- cpu MHz: 2800.100
- total # of cores : 32
- total computation time : 4.5 Hours
- number of frames : 2400
- seconds per frame : ~9

If our code was to be rendered completely on AWS - c3.8xlarge the files would have been rendered in under 14 hours, versus the serial 120 hours.

4. Source Code

```
/* Main.cc */
#include <stdlib.h>
#include <stdio.h>
#include "camera.h"
#include "renderer.h"
#include "mandelbox.h"
#include "mpi.h"
#include <sys/time.h>
#include <iostream>

void getParameters(char *filename, CameraParams *camera_params, RenderParams
*renderer_params,
                MandelBoxParams *mandelBox_paramsP);
void init3D       (CameraParams *camera_params, const RenderParams *renderer_params);
void renderFractal(const CameraParams &camera_params, const RenderParams
&renderer_params, unsigned char* image, int p, int my_rank);
void saveBMP      (const char* filename, const unsigned char* image, int width, int
height);

MandelBoxParams mandelBox_params;

int main(int argc, char** argv)
{
  struct timeval t0;
```

```
  struct timeval t1;
  CameraParams    camera_params;
  RenderParams    renderer_params;
  int my_rank, p;
  gettimeofday(&t0,0);

   /* Start up MPI */
  MPI_Init(&argc, &argv);

  /* Find out process rank  */
  MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);

  /* Find out number of processes */
  MPI_Comm_size(MPI_COMM_WORLD, &p);


  getParameters(argv[1], &camera_params, &renderer_params, &mandelBox_params);

  int image_size = renderer_params.width * renderer_params.height;
  /* temporary image that has sections created by each processor */
  unsigned char *image = (unsigned char*)malloc(3*image_size*sizeof(unsigned char));
  /* Image that will be output */
  unsigned char *lastimage = (unsigned char*)calloc(3*image_size*sizeof(unsigned
char),sizeof(unsigned char));


  init3D(&camera_params, &renderer_params);


  renderFractal(camera_params, renderer_params, image, p, my_rank);
  /* adds all the images to last image */
  MPI_Reduce(image, lastimage, 3*image_size*sizeof(unsigned char),
MPI_UNSIGNED_CHAR,MPI_SUM,0, MPI_COMM_WORLD);

  if (my_rank == 0){
    saveBMP(renderer_params.file_name, lastimage, renderer_params.width,
renderer_params.height);

    free(image);
    gettimeofday(&t1,0);
    double elapsed = (double)((t1.tv_sec-t0.tv_sec)*1000000LL + t1.tv_usec-
t0.tv_usec)/(double)(1e6);
    FILE *outFile = fopen("stats.txt", "a");
    fprintf(outFile, "%s %f %s\n", renderer_params.file_name, elapsed, "seconds.");
    fclose(outFile);
  }
```

```
  MPI_Finalize();
  return 0;
}
/*****************************************************************************************************/
```

Renderer.cc

```
/* Renderer.cc */

#include <stdio.h>
#include "color.h"
#include "mandelbox.h"
#include "camera.h"
#include "vector3d.h"
#include "3d.h"
#include <time.h>
#include "mpi.h"

extern double getTime();
extern void   printProgress( double perc, double time );

extern void rayMarch (const RenderParams &render_params, const vec3 &from, const vec3
&to, pixelData &pix_data);
extern vec3 getColour(const pixelData &pixData, const RenderParams &render_params,
                      const vec3 &from, const vec3  &direction);

void renderFractal(const CameraParams &camera_params, const RenderParams
&renderer_params, unsigned char* image, int p, int my_rank)
{

  double farPoint[3];
  vec3 to, from;

  from.SetDoublePoint(camera_params.camPos);

  int height = renderer_params.height;
  int width  = renderer_params.width;
  int lowerHeight, chunksHeight;

  /* each processor divides up the work*/
  chunksHeight = height/p;
  /* process 0 starts at height = 0 */
  if (my_rank == 0)
```

```
  lowerHeight = 0;
else
  /* all the other processors start at the computer chunk size * its rank */
  lowerHeight = chunksHeight * my_rank;
/* every processor but the last has the height computer by its lower height + chunk size */
if (my_rank < (p-1))  height = lowerHeight + chunksHeight;


pixelData pix_data;

/*each processor then goes through their respected chunks, lowerheight - height */
for(int j = lowerHeight; j < height; j++)
  {
    //for each column pixel in the row
     for(int i = 0; i < width; i++)
         {

           //send, retrive color
           vec3 color;
           if( renderer_params.super_sampling == 1 )
            {
              vec3 samples[9];
              int idx = 0;
              for(int ssj = -1; ssj < 2; ssj++){
                   for(int ssi = -1; ssi< 2; ssi++){
                     UnProject(i+ssi*0.5, j+ssj*0.5, camera_params, farPoint);

                     // to = farPoint - camera_params.camPos
                     to = SubtractDoubleDouble(farPoint,camera_params.camPos);
                     to.Normalize();

                     //render the pixel
                     rayMarch(renderer_params, from, to, pix_data);

                     //get the colour at this pixel
                     samples[idx] = getColour(pix_data, renderer_params, from, to);
                     idx++;
                   }
              }
              color = (samples[0]*0.05 + samples[1]*0.1 + samples[2]*0.05 +
                         samples[3]*0.1  + samples[4]*0.4 + samples[5]*0.1  +
                         samples[6]*0.05 + samples[7]*0.1 + samples[8]*0.05);

           }
```

```
            else
             {
               // get point on the 'far' plane
               // since we render one frame only, we can use the more specialized method
               UnProject(i, j, camera_params, farPoint);

               // to = farPoint - camera_params.camPos
               to = SubtractDoubleDouble(farPoint,camera_params.camPos);
               to.Normalize();

               //render the pixel
               rayMarch(renderer_params, from, to, pix_data);

               //get the colour at this pixel
               color = getColour(pix_data, renderer_params, from, to);


            //save colour into texture
            int k = (j * width + i)*3;
            image[k+2] = (unsigned char)(color.x * 255);
            image[k+1] = (unsigned char)(color.y * 255);
            image[k]   = (unsigned char)(color.z * 255);

             }
           }
   }
}
```

Params.py

Description: This python script is used to generate a param file for each file. Each param file is named in the following formate: ‘fxxxx.bmp’ where xxxx is the frame number. The parameters for each frame are calculated based on the frame number.

```
import sys

def main():
        camX = 15
        camY = 0
        camZ = 0
        lookX = 0
        lookY = 0
        lookZ = 0
        upY = 1
        upX = 0
```

```
upZ = 0

# For final : double = 2, half = 0.5
# For testing : double = 1, half = 1
double = 2
half = 0.5

for i in xrange(0 * double, 2200 * double):
 if i < (200 * double):
          camX = camX - (0.047 * half)          #camX = 5.6
          camY = camY + (0.019 * half)         #camY = 3.8
          camZ = camZ + (0.0185 * half)        #camZ = 3.7
          lookX = lookX + (0.018 * half)       #lookX = 3.6
          lookY = lookY + (0.019 * half)       #lookY = 3.8
          lookZ = lookZ + (0.0185 * half)      #lookZ = 3.7
 elif i < (212 * double):
          camX = camX - (0.01 * half)                  #camX = 3.48
 elif i < (260 * double):
          camX = camX + (0.01 * half)                  #camX = 6.08
          camY = camY + (0.01 * half)                  #camY = 4.28
          camZ = camZ - (0.01 * half)                  #camZ = 3.22
elif i < (400 * double):
          camX = camX - (0.01 * half)                  #camX = 4.68
elif i < (600 * double):
…
…
…
n = str(i)
if i <= 9:
filenameDAT = 'params' + n.rjust(5-len(n),'0') + '.dat'
filenameBMP = 'f' + n.rjust(5-len(n),'0') + '.bmp'
elif i <= 99:
filenameDAT = 'params' + n.rjust(6-len(n),'0') + '.dat'
filenameBMP = 'f' + n.rjust(6-len(n),'0') + '.bmp'
else:
filenameDAT = 'params' + n.rjust(7-len(n),'0') + '.dat'
filenameBMP = 'f' + n.rjust(7-len(n),'0') + '.bmp'
file = open(filenameDAT, 'w+')
fileStuff = '# CAMERA \n# location  camX,camY,camZ  \n'
fileStuff = fileStuff + str(camX) + ' ' + str(camY) + ' ' + str(camZ) + '\n'
fileStuff = fileStuff + '# look_at (target)  x y z \n'
fileStuff = fileStuff + str(lookX) + ' ' + str(lookY) + ' ' + str(lookZ) + '\n' + \
'# up vector camX,camY,camZ;  (0, 1, 0) \n' + \
str(upX) + ' ' + str(upY) + ' ' + str(upZ) + ' \n' + \
```

```
        '# field of view (1) \n' + \
        '1.1  \n' + \
        '# IMAGE \n' + \
        '# width height \n' + \
        '3840 2160 \n' + \
        '# detail level, the smaller the more detailed (-3) \n' + \
        '-3.5 \n' + \
        '# MANDELBOX \n' + \
        '# scale, rMin, rFixed  (2 0.5 1) \n' + \
        '2.0 0.5 1 \n' + \
        '# max number of iterations, escape time \n' + \
        '18 100 \n' + \
        '# COLORING \n' + \
        '# type 0 or 1 \n' + \
        '0 \n' + \
        '# brightness \n' + \
        '1.2 \n' + \
        '# super sampling anti aliasing 0 = off, 1 = on \n' + \
        '0 \n' + \
        '# IMAGE FILE NAME \n' + filenameBMP

        # 3840 2160
        # 640 480

        file.write(fileStuff)
        file.close()
main()
```

RunMe.py

Description: When this script is ran, it generates a bash command stored in a string. Once the string has been generated, the bash command is sent to the operating system for execution.

```
#!/usr/bin/env python
def runScript(n):
    import subprocess
    import sys
    import os

    # 2200 to 2250
    for i in range (2098 , 2371):

        n = str(i)
            if i <= 9:
            filenameDAT = 'params' + n.rjust(5-len(n),'0') + '.dat'
```

```
            elif i <= 99:
            filenameDAT = 'params' + n.rjust(6-len(n),'0') + '.dat'
            else:
            filenameDAT = 'params' + n.rjust(7-len(n),'0') + '.dat'
        bashCommand = "mpirun -np 8 ./mandelbox " + filenameDAT + ';'
        os.system(bashCommand)
runScript(200)
```

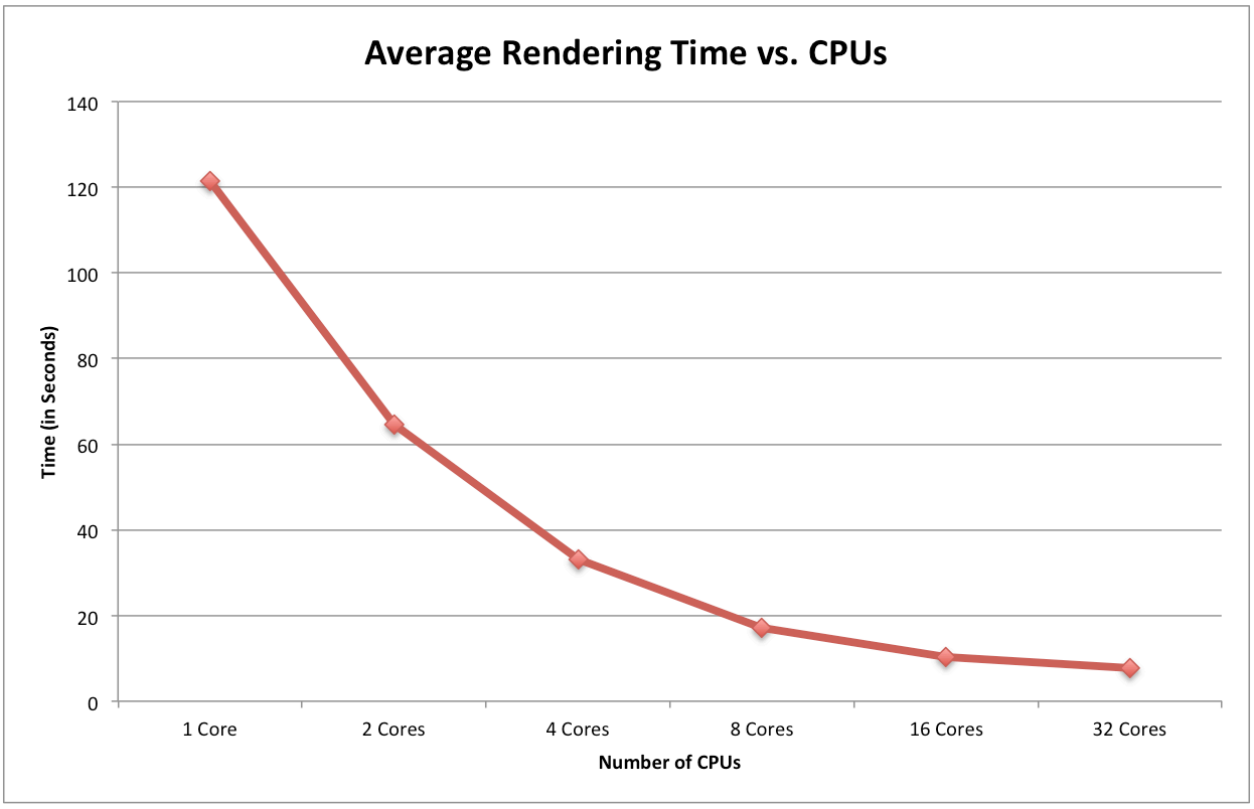
5. Efficiency Claims

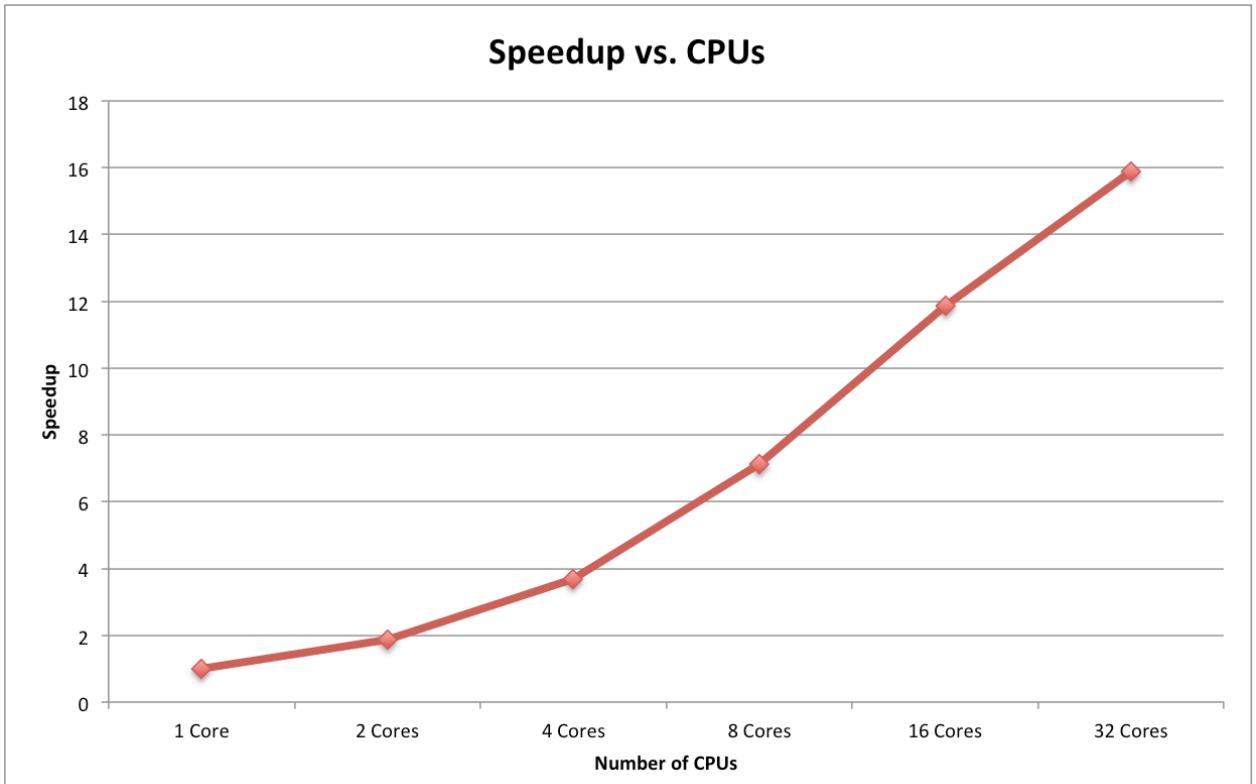
Raw Times:

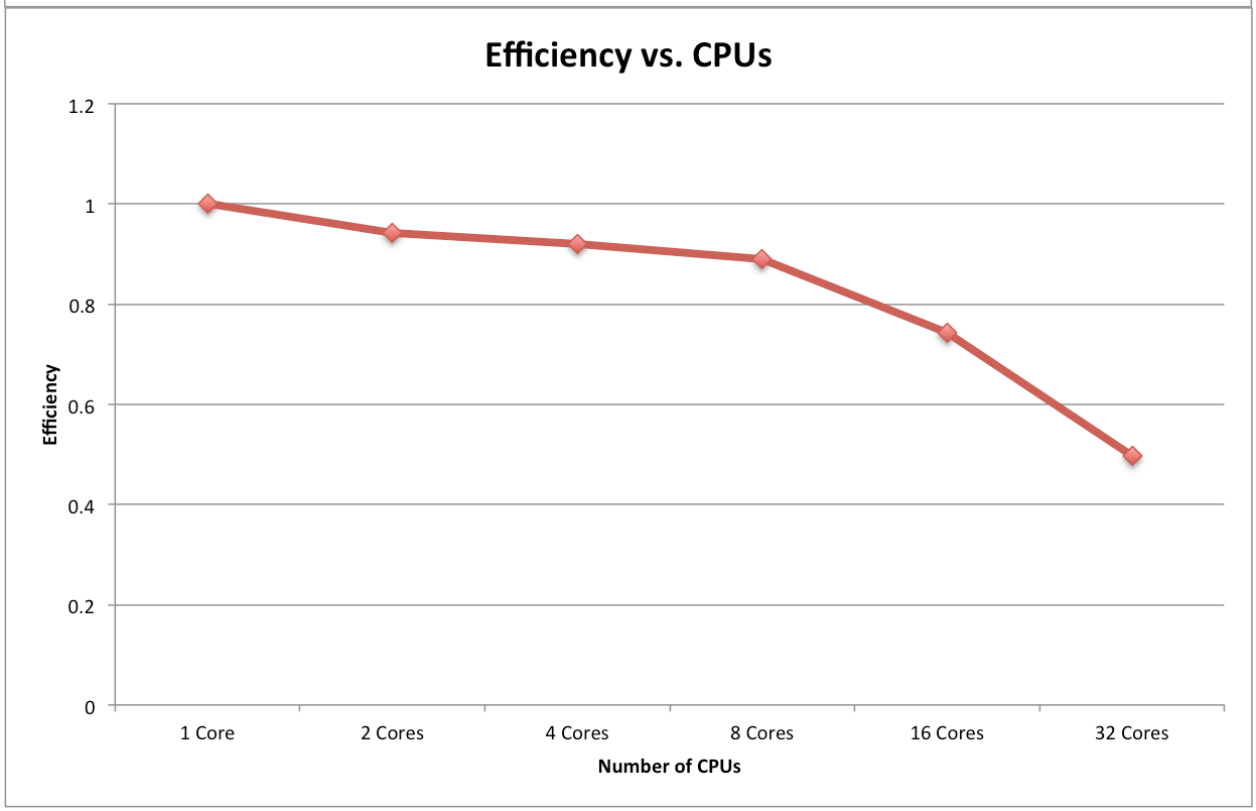
Frame	1 Core	2 Cores	4 Cores	8 Cores	16 Cores	32 Cores
f4400.bmp	121.11632	64.490515	33.161222	16.82935	9.057498	7.484354
f4400.bmp	121.09715	64.478898	33.100678	16.86671	9.096332	7.536114
f4400.bmp	121.060995	64.97826	33.220599	16.869683	9.200881	7.840562
f4400.bmp	121.050796	64.414242	32.919158	16.979917	9.674412	7.722511
f4400.bmp	121.221628	64.439331	32.082984	17.042009	9.191024	7.816108
f4400.bmp	121.153489	64.445829	33.216714	16.953851	10.499452	7.657605
f4400.bmp	121.189598	65.033608	32.877898	17.079782	9.980208	7.76437
f4400.bmp	121.151351	64.424086	32.907117	16.951193	10.629126	7.73306
f4400.bmp	121.245215	64.402925	32.873619	16.967772	9.202032	7.782776
f4400.bmp	121.93088	64.399634	33.015355	17.234088	9.393452	7.528119
f4400.bmp	121.985113	64.695886	33.06944	16.989761	11.501978	7.606963
f4400.bmp	121.305214	64.748789	32.847856	17.025819	9.410023	7.69966
f4400.bmp	121.276757	64.410315	32.872752	17.536003	12.032902	7.545203
f4400.bmp	121.381291	64.380071	32.84221	17.033558	9.970702	7.727643
f4400.bmp	121.437396	64.307034	33.153096	17.017032	9.290414	7.623221
f4400.bmp	121.490441	64.265263	32.833518	17.151726	11.274861	7.56329

f4400.bmp	122.652383	64.316097	34.48236	17.053036	9.805401	7.656346
f4400.bmp	122.457513	64.42842	32.968776	17.819126	9.418057	7.550747
f4400.bmp	121.538958	65.023761	32.804641	17.057164	9.697304	7.63112
f4400.bmp	121.806191	64.218097	33.223744	17.063516	12.368017	7.498079
f4400.bmp	121.716669	64.232176	33.138911	17.082564	14.146197	7.599957

	1 Core	2 Cores	4 Cores	8 Cores	16 Cores	32 Cores
Average	121.4888261	64.50158271	33.02917371	17.07636476	10.23048919	7.646086095
Speedup	1	1.883501474	3.678227834	7.114443138	11.87517271	15.88902147
Efficiency	1	0.941750737	0.919556959	0.889305392	0.742198295	0.496531921




Speedup vs. CPUs
18
16
14
12
10
8
6
4
2
0
Speedup
1 Core
2 Cores
4 Cores
8 Cores
16 Cores
32 Cores
Number of CPUs


Efficiency vs. CPUs
1.2
1
0.8
0.6
0.4
0.2
0
Efficiency
1 Core
2 Cores
4 Cores
8 Cores
16 Cores
32 Cores
Number of CPUs