Paraguin Compiler Examples

2.1

Generated by Doxygen 1.8.6

Wed May 7 2014 00:44:05

Contents

1 File Index

1.1 File List

Here is a list of all files with brief descriptions:

2 File Documentation

2.1 factorial.c File Reference

This program computes N factorial.

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

Macros

- #define N 20

Functions

- int main (int argc, char ∗argv[])

2.1.1 Detailed Description

This program computes N factorial.

This is an example of an iterative implementation of factorial. The for loop is easily subdivided into easily parallelized subsets using a forall. Since the iterations are divided among available processors, each processors will have a partial product. Therefore, a reduction must be applied to arrive at the final solution.

Author

Roger Johnson and Clayton Ferner

Date

January 2014

Definition in file factorial.c.

2.1.2 Macro Definition Documentation

2.1.2.1 #define N 20

Definition at line 23 of file factorial.c.

2.1.3 Function Documentation

2.1.3.1 int main (int *argc*, char * *argv[]*)

Definition at line 25 of file factorial.c.

```
26 {
27     int i, n;
28     long answer, fact;
29     double elapsed_time;
30     struct timeval tv1, tv2;
31     /*
32     * If a command line argument is provided for the computation of factorial use
33     * that, else use the N defined as 20.
34     */
35     if (argc > 1)
36         n = atoi(argv[1]);
37     else n = N;
38 
39     ;
40     #pragma paraguin begin_parallel
41 
42     /*
43     *This barrier is here so that we can take a time stamp
44     *once we know all processes are ready to go.
45     */
46     #pragma paraguin barrier
47     #pragma paraguin end_parallel
48 
49     gettimeofday(&tv1, NULL);
50 
51     ;
52     #pragma paraguin begin_parallel
53     #pragma paraguin bcast n
54     /*
55     * Fact must be initialized within the parallel region so it is available
56     * to all processes as 1.
57     */
58     fact = 1;
59 
60     ;
61     #pragma paraguin forall
62      /*
63     * This implementation of factorial is iterative rather than recursive. This
64     * allows the for loop to be easily broken into segments equal to the number
65     * of available processes through the use of a paraguin forall pragma.
66     */
67     for (i = 1; i <= n; i++)
68         fact = fact * i;
69 
70     /*
71     * Reduce the partial products (fact) at each process to a single total
72     * sum (answer).
73     */
74     ;
75     #pragma paraguin reduce prod fact answer
76     #pragma paraguin end_parallel
77     /*
78     * Take a time stamp.  This won't happen until after the master
```

```
79     * process has gathered all the input from the other processes.
80     */
81     gettimeofday(&tv2, NULL);
82     /*
83     * Only the master should print this because the other processors only
84     * have partial products.
85     */
86     printf ("%ld! = %ld\n", n, answer);
87     elapsed_time = (tv2.tv_sec - tv1.tv_sec) + ((tv2.tv_usec - tv1.tv_usec) / 1000000.0);
88     printf ("elapsed_time=\t%lf (seconds)\n", elapsed_time);
89 }
```

2.2 fireplace.c File Reference

This program utilizes Jacobi iterations to simulate a basic 2-D heat distribution from a given heat source.

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

Macros

- #define TOTAL_TIME 30000
- #define N 900
- #define M 900

Functions

- double computeValue (double A[ ][M], int i, int j)

  Computes the heat distribution for a given index by multiplying its value by that of its horizontal and vertical neighbors.
- int main (int argc, char ∗argv[ ])

Variables

- int __guin_rank = 0
- int __guin_current = 0
- int __guin_next = 1

2.2.1 Detailed Description

This program utilizes Jacobi iterations to simulate a basic 2-D heat distribution from a given heat source.

This program implements the Paraguin compiler's stencil pattern to perform a heat distribution simulation using Jacobi iterations. Heat is distibuted over a room of size NxM (provided by user) from a simulated "fireplace". The simulation works by generating a 3D array consisting of 2 2D arrays. Two arrays are used as a means of optimizing the Jacobi iterations by allowing the toggling between the two arrays to prevent the copying of values back into the original array. The array is set to size NxM populated by zeros except for a "fireplace" of size (2/5 of M) x 1. This is populated by 100. Then each index in the array is multiplied by its adjacent neighbors for a given number of iterations (TOTAL_TIME). A single 2D array is then printed out that represents the distribution of heat from the fireplace into the simulated room over a given interval. The Paraguin pragma stencil is used to divide the room into subsections given to each process to compute the heat distribution in parallel

Author

Clayton Ferner

Date

January 2014

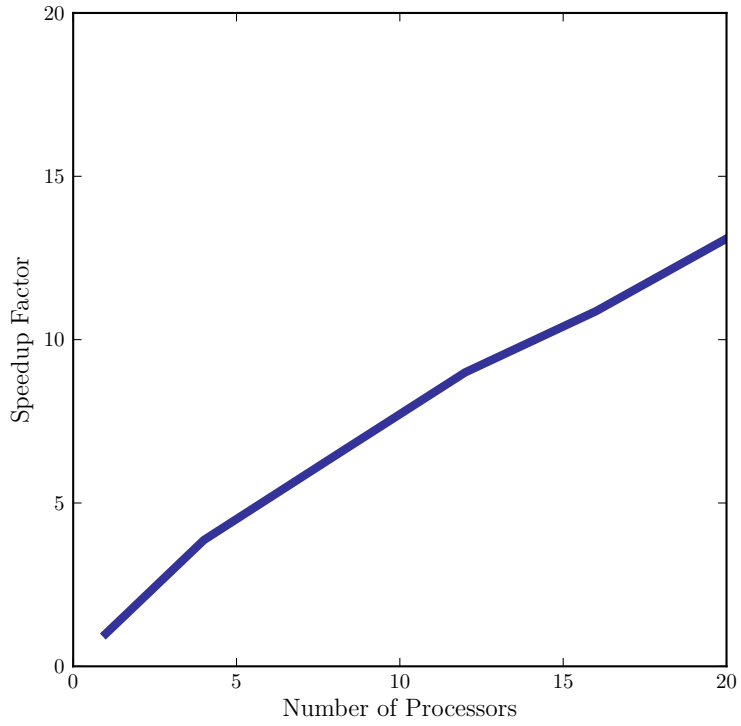

Figure 1: This figure shows the speedup factor of the fireplace heat distribution simulation for a room of size 900x900 over 30000 iterations. The speedup factor is computed by dividing the sequential execution time by the parallel execution time.

Definition in file fireplace.c.

2.2.2 Macro Definition Documentation

2.2.2.1 #define M 900

Definition at line 20 of file fireplace.c.

2.2.2.2 #define N 900

Definition at line 19 of file fireplace.c.

2.2.2.3 #define TOTAL_TIME 30000

Definition at line 18 of file fireplace.c.

2.2.3 Function Documentation

2.2.3.1 double computeValue (double *A[][M],* int *i,* int *j*)

Computes the heat distribution for a given index by multiplying its value by that of its horizontal and vertical neighbors.

Parameters

A	the current array for which the distribution is being computed
i	the i index of the array
j	the j index of the array

Returns

the new value of a given index based on the effects of the heat distribution

Definition at line 45 of file fireplace.c.

```
46 {
47     return (A[i-1][j] + A[i+1][j] + A[i][j-1] + A[i][j+1]) * 0.25;
48 }
```

2.2.3.2 int main (int *argc*, char * *argv[]*)

Definition at line 52 of file fireplace.c.

```
53 {
54     int i, j,n, m, max_iterations;
55     int left, right;
56     double room[2][N][M];
57     double elapsed_time, start_time, end_time;
58     struct timeval tv1, tv2;
59     left = (int) round(M * 0.3);
60     right = M - left;
61     /*
62     * For the "wall" of the array, if the j is between the left and right cutoff
63     * then set all values to 100 to simulate a hot fireplace. Otherwise set all
64     * values to 0.
65     */
66     for (j = 0; j < N; j++) {
67         if (j >= left && j <= right)
68             room[0][0][j] = room[1][0][j] = 100.0;
69         else
70             room[0][0][j] = room[1][0][j] = 0.0;
71     }
72     /*
73     * Set the remainder of the simulated room values to 0.
74     */
75     for (i = 1; i < N; i++) {
76         for (j = 0; j < N; j++) {
77             room[0][i][j] = room[1][i][j] = 0.0;
78         }
79     }
80
81     gettimeofday(&tv1, NULL);
82
83     ;
84     #pragma paraguin begin_parallel
85
86     n = N;
87     m = M;
88     max_iterations = TOTAL_TIME;
89     /*
90     * Utilize the stencil pattern in Paraguin to perform the Jacobi iterations
91     * necessary for the heat distribution problem. Room is the array, n and m are
92     * the array dimensions, max_iterations is the maximum number of iterations
93     * given by the user, and computeValue is the function to compute heat
94     * distribution at each index of the given array. A function must always be
95     * passed into the Paraguin stencil pattern.
96     */
97     #pragma paraguin stencil room n m max_iterations computeValue
98
99     #pragma paraguin end_parallel
100     /*
101     * Take a time stamp.  This won't happen until after the master
102     * process has gathered all the input from the other processes.
103     */
104     gettimeofday(&tv2, NULL);
105
106     elapsed_time = (tv2.tv_sec - tv1.tv_sec) +
107     ((tv2.tv_usec - tv1.tv_usec) / 1000000.0);
108     /*
109     * Only the master should print this because the other processors only
110     * have partial products. The last accessed array 2D array of the 3D array
```

```
111     * pair is printed out, and this represents the simulated heat distribution
112     * given the user defined room dimensions and number of iterations.
113     */
114     printf ("elapsed_time=\t%lf (seconds)\n", elapsed_time);
115
116     for (i = 0; i < N; i++) {
117         for (j = 0; j < N; j++) {
118             printf (" %lf", room[__guin_current][i][j]);
119         }
120         printf ("\n");
121     }
122     printf ("\n\n");
123
124
125 }
```

2.2.4 Variable Documentation

2.2.4.1 int __guin_current = 0

Definition at line 34 of file fireplace.c.

2.2.4.2 int __guin_next = 1

Definition at line 35 of file fireplace.c.

2.2.4.3 int __guin_rank = 0

Definition at line 33 of file fireplace.c.

2.3 hello.c File Reference

This program is a simple parallel implementation of hello world.

Functions

- int main (int argc, char ∗argv[])

Variables

- int __guin_rank = 0

  This is a predefined Paraguin identifier. It explicitly defines the master process as rank 0. This is done so that the program can be compiled with gcc (wiith no modification to the source code) to create a sequential version of the program.

2.3.1 Detailed Description

This program is a simple parallel implementation of hello world.

This program implments hello world in parallel. The master process will print a unique hello world statement. All other processes will print identical hello world statements that differ only in process number and machine number.

Author

Clayton Ferner

Date

January 2014

Definition in file hello.c.

2.3.2 Function Documentation

2.3.2.1 int main (int *argc,* char * *argv[]*)

Definition at line 26 of file hello.c.

```
27 {
28     char hostname[256];
29
30     #pragma paraguin begin_parallel
31     /*
32     * Print hello world from each process that is not the master process.
33     */
34     gethostname(hostname, 255);
35     printf("Hello world from process %d on machine %s\n", __guin_rank, hostname);
36
37     #pragma paraguin end_parallel
38     /*
39     * Print hello world from the master process. This must be done outside a
40     * parallel region so that only the master process prints the hello world.
41     */
42     printf ("Hello world from master process %d running on %s\n", __guin_rank, hostname);
43 }
```

2.3.3 Variable Documentation

2.3.3.1 int __guin_rank = 0

This is a predefined Paraguin identifier. It explicitly defines the master process as rank 0. This is done so that the program can be compiled with gcc (wiith no modification to the source code) to create a sequential version of the program.

Definition at line 24 of file hello.c.

2.4 helloHybrid.c File Reference

This program is a hybrid parallel implementation of hello world that includes distributed memory(MPI) and shared memory (OpenMP) constructs.

```
#include <stdio.h>
```

Functions

- int main (int argc, char *argv[])

Variables

- int __guin_rank = 0

  This is a predefined Paraguin identifier. It explicitly defines the master process as rank 0. This is done so that the program can be compiled with gcc (wiith no modification to the source code) to create a sequential version of the program.

2.4.1 Detailed Description

This program is a hybrid parallel implementation of hello world that includes distributed memory(MPI) and shared memory (OpenMP) constructs.

This program implments hello world in parallel. The master process will print a unique hello world statement. All other processes will print identical hello world statements that differ only in process number and machine number. Each parallel process generated using a pragma parallel region will also have four OpenMP threads generated using an OpenMP parallel pragma. Each thread will print out the thread ID as well as the process it belongs to. This

is a purely demonstrative example to show how OpenMP shared memory parallel pragmas can be integrated with Paraguin distributed memory parallel code.

Author

Clayton Ferner

Date

January 2014

Definition in file helloHybrid.c.

2.4.2 Function Documentation

2.4.2.1 int main (int *argc*, char * *argv[]*)

Definition at line 31 of file helloHybrid.c.

```
32 {
33     char hostname[256];
34     int tID, x;
35     /*
36     * Prints a statement indicating the master process is starting.
37     */
38     printf("Master process %d starting.\n", __guin_rank);
39 
40     #pragma paraguin begin_parallel
41 
42     gethostname(hostname, 255);
43     /*
44     * Print hello world from each process that is not the master process.
45     */
46     printf("Hello world from process %3d on machine %s.\n", __guin_rank, hostname);
47 
48     x = 0;
49     /*
50     * OpenMP parallel pragma construct, defines a parallel region with four
51     * threads. As this is already within a paraguin parallel region than four
52     * threads will be created for each process.
53     */
54     #pragma omp parallel private(tID) num_threads(4)
55     if (x == 0) {
56         tID = omp_get_thread_num();
57         /*
58         * Prints the process ID for the current distributed memory process (MPI),
59         * and the thread ID from the current shared memory thread (OpenMP).
60         */
61         printf ("<pid %d>: tid = %d\n", __guin_rank, tID);
62     }
63 
64     #pragma paraguin end_parallel
65     /*
66     * Prints a statement indicating that the master process is ending.
67     */
68     printf("Goodbye world from process %d.\n", __guin_rank);
69 
70     return 0;
71 }
```

2.4.3 Variable Documentation

2.4.3.1 int __guin_rank = 0

This is a predefined Paraguin identifier. It explicitly defines the master process as rank 0. This is done so that the program can be compiled with gcc (wiith no modification to the source code) to create a sequential version of the program.

Definition at line 29 of file helloHybrid.c.

2.5 integ.c File Reference

This is an example of integration by approximating the area under a function in parallel.

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <sys/time.h>
```

Functions

- double f (double x)

  Returns the integral that represents 4sin(1.5x)+5 over a given interval.

- int main (int argc, char ∗argv[])

Variables

- int __guin_rank = 0

2.5.1 Detailed Description

This is an example of integration by approximating the area under a function in parallel.

This program works by approximating an integral over a specific range given by the user. The function for which this is computed is 4sin(1.5x)+5. The only user definable variables are range values, from a to b, and the number of segments the area under the curve is to be divided into, indicated by N. The greater the value of N the greater the accuracy with which the integral can be computed. This is a variation of Monte Carlo approximation like that implemented in the pi approximation example.

Author

Clayton Ferner

Date

February 2014

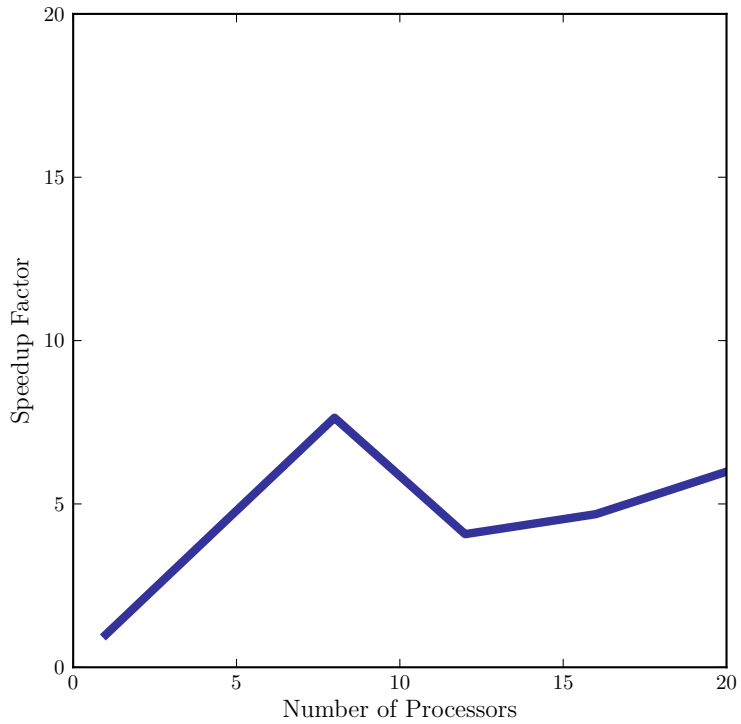

Figure 2: This figure shows the speedup factor for an algorithm that approximates the integral of 4sin(1.5x)+5 from 0 to 1 by finding the area divided into 100000000 subsections. The speedup factor is computed by dividing the sequential execution time by the parallel execution time.

Definition in file integ.c.

2.5.2 Function Documentation

2.5.2.1 double f (double *x*)

Returns the integral that represents 4sin(1.5x)+5 over a given interval.

Parameters

x	the width of the subunit used to quantify area under a function

Returns

the area of a given subsection based on the 4sin(1.5x)+5 function

Definition at line 31 of file integ.c.

```
32 {
33     #pragma paraguin begin_parallel
34      return 4.0 * sin(1.5*x) + 5;
35     #pragma paraguin end_parallel
36 }
```

2.5.2.2 int main (int *argc*, char * *argv[]*)

Definition at line 39 of file integ.c.

```
40 {
41     char *usage = "Usage: %s a b N\n";
42     int i, error = 0, N;
43     double a, b, x, y, size, area, overal_area, elapsed_time;
44     struct timeval tv1, tv2;
45     /*
46     * Command line values need to be provided for the range over which the integral
47     * needs to be computed and the number of iterations (N). If there are not enough
48     * arguments return an error. If a, the range start, is more than b, the range
49     * end, then return an error. Otherwise read arguments 1, 2, and 3 into a,
50     * b, and N respectively.
51     */
52     if (argc < 4) {
53         fprintf (stderr, usage, argv[0]);
54         error = -1;
55     } else {
56 
57         a = atof(argv[1]);
58         b = atof(argv[2]);
59         N = atoi(argv[3]);
60         if (b <= a) {
61             fprintf (stderr, "a should be smaller than b\n");
62             error = -1;
63         }
64     }
65     /*
66     * The error is broadcast in a parallel region so that all processes receive
67     * the error. This prevents a deadlock if an error arises because all processes
68     * will receive the error and then exit.
69     */
70     #pragma paraguin begin_parallel
71     #pragma paraguin bcast error
72     if (error) return error;
73     #pragma paraguin end_parallel
74 
75     #pragma paraguin begin_parallel
76     /*
77     * This barrier is here so that a time stamp can be taken. This ensures all
78     * processes have reached the barrier before exiting the parallel region
79     * and then taking a time stamp.
80     */
81     #pragma paraguin barrier
82     #pragma paraguin end_parallel
83 
84     gettimeofday(&tv1, NULL);
85 
86     #pragma paraguin begin_parallel
87     ;
88     /*
89     * Broadcast the range start (a), range end (b), and number of iterations (N)
90     * to all processes.
91     */
92     #pragma paraguin bcast a b N
93 
94     size = (b - a) / N;
95     area = 0.0;
96     /*
97     * This parallelizes the following loop nest assigning iterations
98     * of the outermost loop (i) to different partitions. The area under the
99     * function is estimated by each process for its respective loop partition.
100     */
101     #pragma paraguin forall
102     for (i = 0; i < N-1; i++) {
103         x = a + i * size;
104         y = f(x);
105         area += y * size;
106     }
107 
108     ;
109     /*
110     * The area computed by each process is reduced by a sum operator into the
111     * total area under the function from a to b, and this area represents the
112     * estimated integral.
113     */
114     #pragma paraguin reduce sum area overal_area
115 
116     #pragma paraguin end_parallel
117 
118 #ifndef PARAGUIN
119     overal_area = area;
120 #endif
121     /*
122     * Take a time stamp.  This won't happen until after the master
123     * process has gathered all the input from the other processes.
124     */
125     gettimeofday(&tv2, NULL);
126    /*
```

```
127     * Only the master should print this because the other processors only
128     * have partial products. The quadrant 1 area is multiplied by 4 to get the
129     * total area of the circle of radius N. This area will approximate pi. The
130     * estimated area, estimated pi value, true pi value, and error are all printed.
131     */
132     elapsed_time = (tv2.tv_sec - tv1.tv_sec) + ((tv2.tv_usec - tv1.tv_usec) / 1000000.0);
133     printf ("elapsed_time=\t%lf (seconds)\n", elapsed_time);
134     printf ("area = %lf\n", overal_area);
135 }
```

2.5.3 Variable Documentation

2.5.3.1 int __guin_rank = 0

Definition at line 19 of file integ.c.

2.6 matrixadd.c File Reference

This is a parallel implementation of matrix addition.

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

Macros

- #define N 512

Functions

- void print_results (char *prompt, double a[N][N])

  Prints a given 2D array.

- int main (int argc, char *argv[])

2.6.1 Detailed Description

This is a parallel implementation of matrix addition.

This divides the process of matrix addition between multiple parallel processes and gathers these partial results back together to get a final result.

Author

Peter Lawson

Date

February 2014

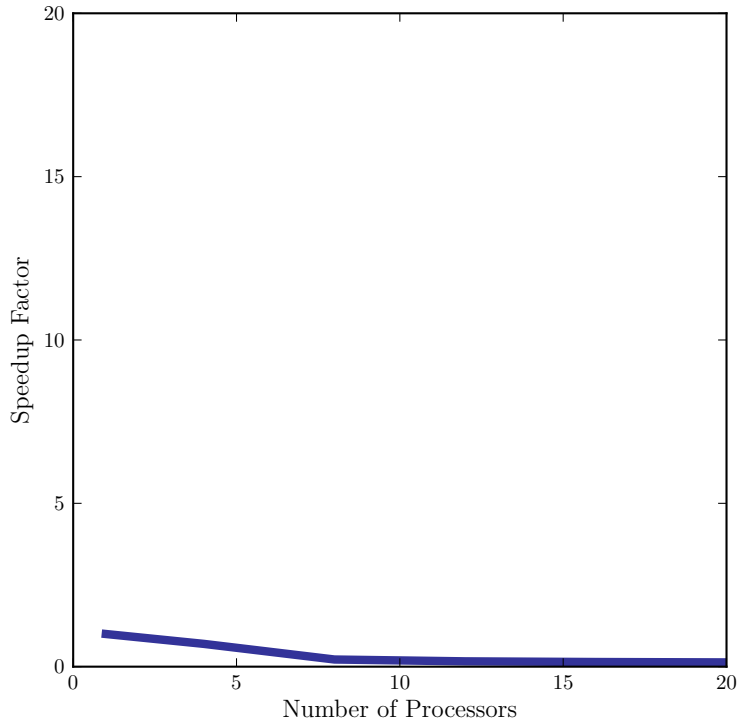

Figure 3: This figure shows the speedup factor for matrix addition for an increasing number of parallel processes for 2 512x512 matrices. Matrix multiplication is O(n2) and as such does not experience a significant performance advantage for except in exceedingly large matrices as indicated by the minimal speedup. The speedup factor is computed by dividing the sequential execution time by the parallel execution time.

Definition in file matrixadd.c.

2.6.2 Macro Definition Documentation

2.6.2.1 #define N 512

Definition at line 13 of file matrixadd.c.

2.6.3 Function Documentation

2.6.3.1 int main (int *argc*, char * *argv[]*)

Definition at line 28 of file matrixadd.c.

```
29 {
30     int i, j, error = 0;
31     double a[N][N], b[N][N], c[N][N];
32     char *usage = "Usage: %s file\n";
33     FILE *fd;
34     double elapsed_time;
35     struct timeval tv1, tv2;
36     char message[80];
37
38     /*
39     * If no command line argument is provided with a text file containing the
40     * matrices to added, than an error is given.
41     */
```

```
42      if (argc < 2) {
43          fprintf (stderr, usage, argv[0]);
44          error = -1;
45      }
46
47      /*
48      * If there is an error in reading the input file containing the matrices to
49      * be added an error is given.
50      */
51      if (!error && (fd = fopen (argv[1], "r")) == NULL) {
52          fprintf (stderr, "%s: Cannot open file %s for reading.\n",
53                               argv[0], argv[1]);
54          fprintf (stderr, usage, argv[0]);
55          error = -1;
56      }
57
58      #pragma paraguin begin_parallel
59      #pragma paraguin bcast error
60      if (error) return -1;
61      #pragma paraguin end_parallel
62
63      /*
64      * Read input from file for matrices a and b. The I/O is not timed because
65      * this I/O needs to be done regardless of whether this program is run
66      * sequentially on one processor or in parallel on many processors.
67      * Therefore, it is irrelevant when considering speedup.
68      */
69      for (i = 0; i < N; i++)
70          for (j = 0; j < N; j++)
71              fscanf (fd, "%lf", &a[i][j]);
72
73      for (i = 0; i < N; i++)
74          for (j = 0; j < N; j++)
75              fscanf (fd, "%lf", &b[i][j]);
76
77      fclose(fd);
78
79      #pragma paraguin begin_parallel
80      /*
81      * This barrier is here so that a time stamp can be taken. This ensures all
82      * processes have reached the barrier before exiting the parallel region
83      * and then taking a time stamp.
84      */
85      #pragma paraguin barrier
86      #pragma paraguin end_parallel
87
88      gettimeofday(&tv1, NULL);
89
90      #pragma paraguin begin_parallel
91
92      /*
93      * Scatter a and b matrices to all processes. This divides both matrices
94      * equally between all available processes.
95      */
96      #pragma paraguin scatter a b
97
98      /*
99      * This parallelizes the following loop nest assigning iterations
100      * of the outermost loop (i) to different partitions. This allows for
101      * each process to add its section of a and b into the array c.
102      */
103     #pragma paraguin forall
104
105     for (i = 0; i < N; i++) {
106         for (j = 0; j < N; j++) {
107             c[i][j] = a[i][j] + b[i][j];
108         }
109     }
110
111     ;
112     /*
113     * This gathers the partial values from c into a final result.
114     */
115     #pragma paraguin gather c
116     #pragma paraguin end_parallel
117     /*
118     * Take a time stamp.  This won't happen until after the master
119     * process has gathered all the input from the other processes.
120     */
121     gettimeofday(&tv2, NULL);
122
123     elapsed_time = (tv2.tv_sec - tv1.tv_sec) +
124                         ((tv2.tv_usec - tv1.tv_usec) / 1000000.0);
125     /*
126     * Only the master should print this because the other processors only
127     * have partial products.
128     */
```

```
129     printf ("elapsed_time=\t%lf (seconds)\n", elapsed_time);
130
131     print_results("C = ", c);
132
133 }
```

2.6.3.2 print_results (char * *prompt,* double *a[N][N]*)

Prints a given 2D array.

Parameters

prompt	a string to represent the 2D array
a	the 2D array to be printed

Definition at line 141 of file matrixadd.c.

```
142 {
143     int i, j;
144
145     printf ("\n\n%s\n", prompt);
146     for (i = 0; i < N; i++) {
147         for (j = 0; j < N; j++) {
148             printf(" %.2f", a[i][j]);
149         }
150         printf ("\n");
151     }
152     printf ("\n\n");
153 }
```

2.7 matrixmult.c File Reference

This is a parallel implementation of matrix multiplication.

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

Macros

- #define N 512

Functions

- void print_results (char *prompt, double a[N][N])

  Prints a given 2D array.
- int main (int argc, char *argv[])

2.7.1 Detailed Description

This is a parallel implementation of matrix multiplication.

This divides the process of matrix multiplication between multiple parallel processes and gathers these partial results back together to get a final result. Matrix multiplication is O(n3) and benefits greatly from parallelism.

Author

Clayton Ferner

Date

February 2014

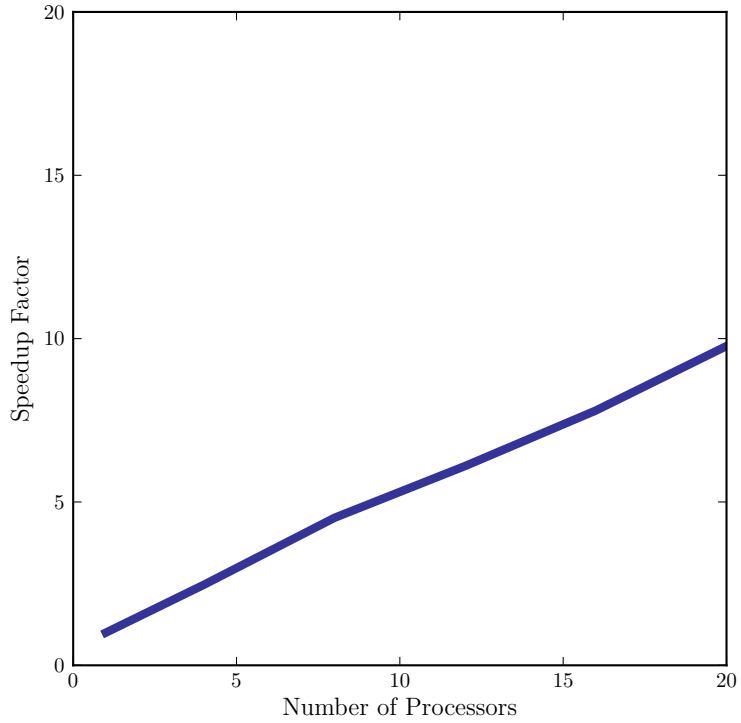

Figure 4: This figure shows the speedup factor for matrix multiplication for an increasing number of parallel processes for 2 512x512 matrices. The speedup factor is computed by dividing the sequential execution time by the parallel execution time.

Definition in file matrixmult.c.

2.7.2 Macro Definition Documentation

2.7.2.1 #define N 512

Definition at line 13 of file matrixmult.c.

2.7.3 Function Documentation

2.7.3.1 int main (int *argc*, char * *argv[]*)

Definition at line 27 of file matrixmult.c.

```
28 {
29     int i, j, k;
30     double a[N][N], b[N][N], c[N][N];
31     char *usage = "Usage: %s file\n";
32     FILE *fd;
33     double elapsed_time, start_time, end_time;
34     struct timeval tv1, tv2;
35
36     /*
37     * If no command line argument is provided with a text file containing the
38     * matrices to added, than an error is given.
39     */
40     if (argc < 2) {
41         fprintf (stderr, usage, argv[0]);
42         return -1;
```

```
43      }
44
45      /*
46      * If there is an error in reading the input file containing the matrices to
47      * be added an error is given.
48      */
49      if ((fd = fopen (argv[1], "r")) == NULL) {
50          fprintf (stderr, "%s: Cannot open file %s for reading.\n",
51              argv[0], argv[1]);
52          fprintf (stderr, usage, argv[0]);
53          return -1;
54      }
55
56      ;
57      #pragma paraguin begin_parallel
58      #pragma paraguin bcast error
59      if (error) return error;
60      #pragma paraguin end_parallel
61
62      /*
63      * Read input from file for matrices a and b. The I/O is not timed because
64      * this I/O needs to be done regardless of whether this program is run
65      * sequentially on one processor or in parallel on many processors.
66      * Therefore, it is irrelevant when considering speedup.
67      */
68      for (i = 0; i < N; i++)
69          for (j = 0; j < N; j++)
70              fscanf (fd, "%lf", &a[i][j]);
71
72      for (i = 0; i < N; i++)
73          for (j = 0; j < N; j++)
74              fscanf (fd, "%lf", &b[i][j]);
75
76      ;
77      #pragma paraguin begin_parallel
78      /*
79      * This barrier is here so that a time stamp can be taken. This ensures all
80      * processes have reached the barrier before exiting the parallel region
81      * and then taking a time stamp.
82      */
83      #pragma paraguin barrier
84      #pragma paraguin end_parallel
85
86      gettimeofday(&tv1, NULL);
87
88      ;
89      #pragma paraguin begin_parallel
90      /*
91      * Broadcast the input to all processors.  This could be
92      * faster if we used scatter, but Bcast is easy and scatter
93      * is not implemented in Paraguin
94      */
95       /*
96      * Scatter a to all processes, and broadcast b to all processes. Scatter breaks
97      * the a matrix up based on the number of available processes. Broadcast sends
98      * the entire array to each process. Scatter is more efficient but in order to
99      * facilitate matrix multiplication each section of the scattered b matrix would
100      * need to be transposed which is not currently in the capacity of scatter.
101      * Instead the entire b array is broadcast to each process.
102      *
103      */
104      #pragma paraguin scatter a
105      #pragma paraguin bcast b
106
107      /*
108      * This parallelizes the following loop nest assigning iterations
109      * of the outermost loop (i) to different partitions. This allows for
110      * each process to multiply its section of a with the corresponding part of b
111      * and place the result in the array c.
112      */
113      #pragma paraguin forall
114
115      for (i = 0; i < N; i++) {
116          for (j = 0; j < N; j++) {
117              c[i][j] = 0.0;
118              for (k = 0; k < N; k++) {
119                  c[i][j] = c[i][j] + a[i][k] * b[k][j];
120              }
121          }
122      }
123
124      ;
125      /*
126      * This gathers the partial values from c into a final result.
127      */
128      #pragma paraguin gather c
129      #pragma paraguin end_parallel
```

```
130
131
132     /*
133     * Take a time stamp.  This won't happen until after the master
134     * process has gathered all the input from the other processes.
135     */
136     gettimeofday(&tv2, NULL);
137
138     elapsed_time = (tv2.tv_sec - tv1.tv_sec) +
139     ((tv2.tv_usec - tv1.tv_usec) / 1000000.0);
140
141     printf ("elapsed_time=\t%lf (seconds)\n", elapsed_time);
142     print_results("C = ", c);
143
144 }
```

2.7.3.2 void print_results (char * *prompt,* double *a[N][N]*)

Prints a given 2D array.

Parameters

prompt	a string to represent the 2D array
a	the 2D array to be printed

Definition at line 151 of file matrixmult.c.

```
152 {
153     int i, j;
154
155     printf ("\n\n%s\n", prompt);
156     for (i = 0; i < N; i++) {
157         for (j = 0; j < N; j++) {
158             printf(" %.2f", a[i][j]);
159         }
160         printf ("\n");
161     }
162     printf ("\n\n");
163 }
```

2.8 matrixmult.hybrid.c File Reference

This is a parallel implementation of matrix multiplication that includes distributed memory and shared memory parallelism.

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

Macros

- #define N 512

Functions

- print_results (char *prompt, double a[N][N])

  Prints a given 2D array.
- int main (int argc, char *argv[])

2.8.1 Detailed Description

This is a parallel implementation of matrix multiplication that includes distributed memory and shared memory parallelism.

This divides the process of matrix addition between multiple parallel processes and gathers these partial results back together to get a final result. Matrix multiplication is O(n3) and benefits greatly from parallelism.

Author

Clayton Ferner

Date

February 2014

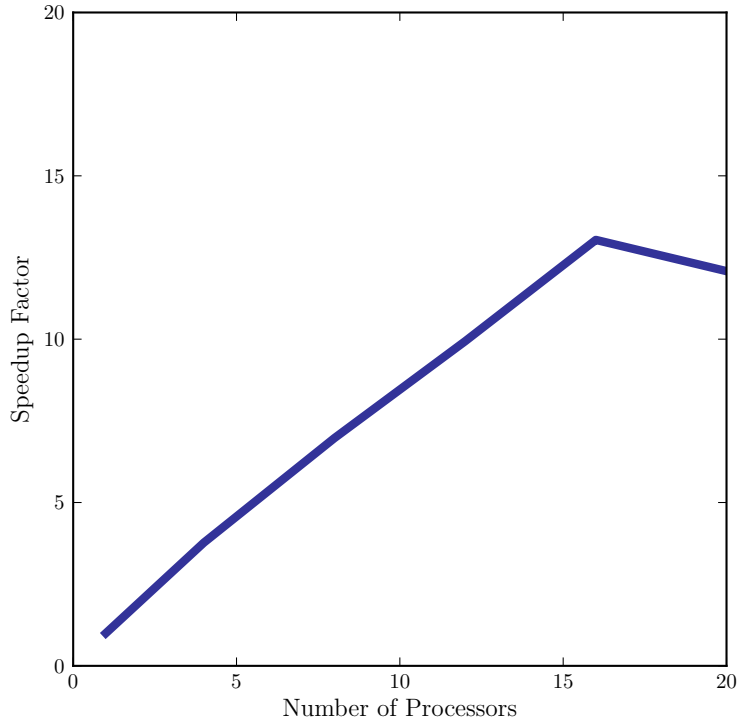

Figure 5: This figure shows the speedup factor for a hybrid parallel matrix multiplication algorithm with for an increasing number of parallel processes for 2 512x512 matrices. The speedup factor is computed by dividing the sequential execution time by the parallel execution time.

Definition in file matrixmult.hybrid.c.

2.8.2 Macro Definition Documentation

2.8.2.1 #define N 512

Definition at line 13 of file matrixmult.hybrid.c.

2.8.3 Function Documentation

2.8.3.1 int main (int *argc*, char * *argv[]*)

Definition at line 26 of file matrixmult.hybrid.c.

```
27 {
28     int i, j, k, error = 0, tID;
29     double a[N][N], b[N][N], c[N][N];
```

```
30      char *usage = "Usage: %s file\n";
31      FILE *fd;
32      double elapsed_time, start_time, end_time;
33      struct timeval tv1, tv2;
34
35
36      /*
37      * If no command line argument is provided with a text file containing the
38      * matrices to added, than an error is given.
39      */
40      if (argc < 2) {
41          fprintf (stderr, usage, argv[0]);
42          error = -1;
43      }
44
45      /*
46      * If there is an error in reading the input file containing the matrices to
47      * be added an error is given.
48      */
49      if ((fd = fopen (argv[1], "r")) == NULL) {
50          fprintf (stderr, "%s: Cannot open file %s for reading.\n",
51                           argv[0], argv[1]);
52          fprintf (stderr, usage, argv[0]);
53          error = -1;
54      }
55
56      ;
57      #pragma paraguin begin_parallel
58      #pragma paraguin bcast error
59      if (error) return error;
60      #pragma paraguin end_parallel
61
62      /*
63      * Read input from file for matrices a and b. The I/O is not timed because
64      * this I/O needs to be done regardless of whether this program is run
65      * sequentially on one processor or in parallel on many processors.
66      * Therefore, it is irrelevant when considering speedup.
67      */
68      for (i = 0; i < N; i++)
69          for (j = 0; j < N; j++)
70              fscanf (fd, "%lf", &a[i][j]);
71
72      for (i = 0; i < N; i++)
73          for (j = 0; j < N; j++)
74              fscanf (fd, "%lf", &b[i][j]);
75
76      ;
77      #pragma paraguin begin_parallel
78      /*
79      * This barrier is here so that a time stamp can be taken. This ensures all
80      * processes have reached the barrier before exiting the parallel region
81      * and then taking a time stamp.
82      */
83      #pragma paraguin barrier
84      #pragma paraguin end_parallel
85
86      gettimeofday(&tv1, NULL);
87
88      ;
89      #pragma paraguin begin_parallel
90      /*
91      * Broadcast the input to all processors.  This could be
92      * faster if we used scatter, but Bcast is easy and scatter
93      * is not implemented in Paraguin
94      */
95       /*
96      * Scatter a to all processes, and broadcast b to all processes. Scatter breaks
97      * the a matrix up based on the number of available processes. Broadcast sends
98      * the entire array to each process. Scatter is more efficient but in order to
99      * facilitate matrix multiplication each section of the scattered b matrix would
100      * need to be transposed which is not currently in the capacity of scatter.
101      * Instead the entire b array is broadcast to each process.
102      *
103      */
104      #pragma paraguin scatter a
105      #pragma paraguin bcast b
106
107      /*
108      * This parallelizes the following loop nest assigning iterations
109      * of the outermost loop (i) to different partitions. An OpenMP parallel
110      * pragma is then uses to parallelize the inner loop (j) so that the outer
111      * loop is parallelized across distributed memory processes while simultaneously
112      * parallelizing using shared memory threads. This allows for each process to
113      * multiply its section of a with the corresponding part of b and place the
114      * result in the array c.
115      */
116      for (i = 0; i < N; i++) {
```

```
117         #pragma omp parallel for private(tID, j,k) num_threads(4)
118         for (j = 0; j < N; j++) {
119             tID = omp_get_thread_num();
120             c[i][j] = 0.0;
121             for (k = 0; k < N; k++) {
122                 c[i][j] = c[i][j] + a[i][k] * b[k][j];
123             }
124         }
125     }
126
127     ;
128     /*
129     * This gathers the partial values from c into a final result.
130     */
131     #pragma paraguin gather c
132     #pragma paraguin end_parallel
133
134
135     /*
136     * Take a time stamp.  This won't happen until after the master
137     * process has gathered all the input from the other processes.
138     */
139     gettimeofday(&tv2, NULL);
140
141     elapsed_time = (tv2.tv_sec - tv1.tv_sec) +
142     ((tv2.tv_usec - tv1.tv_usec) / 1000000.0);
143
144     printf ("elapsed_time=\t%lf (seconds)\n", elapsed_time);
145     print_results("C = ", c);
146
147 }
```

2.8.3.2 print_results (char * *prompt*, double *a[N][N]*)

Prints a given 2D array.

Parameters

prompt	a string to represent the 2D array
a	the 2D array to be printed

Definition at line 154 of file matrixmult.hybrid.c.

```
155 {
156     int i, j;
157
158     printf ("\n\n%s\n", prompt);
159     for (i = 0; i < N; i++) {
160         for (j = 0; j < N; j++) {
161             printf(" %.2f", a[i][j]);
162         }
163         printf ("\n");
164     }
165     printf ("\n\n");
166 }
```

2.9 max.c File Reference

This program finds the maximum value of a randomly generated array of user provided length.

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <sys/time.h>
```

Macros

- #define max(x, y) (x > y ? x : y)

Functions

- int main (int argc, char ∗argv[])

2.9.1 Detailed Description

This program finds the maximum value of a randomly generated array of user provided length.

This program generates an array populated with random values with an array length specified by an int argument passed as a command line argument at runtime. The array is then split based on the number of processes. Each process finds the local maximum for that process, and then each local maximum is reduced to a final maximum representing the global maximum for the original array. This maximum is then printed to screen.

Author

Clayton Ferner

Date

January 2014

Definition in file max.c.

2.9.2 Macro Definition Documentation

2.9.2.1 #define max(*x*, *y*) (x > y ? x : y)

Definition at line 22 of file max.c.

2.9.3 Function Documentation

2.9.3.1 int main (int *argc*, char ∗ *argv[]*)

Definition at line 29 of file max.c.

```
30 {
31     char *usage = "Usage: %s N\n";
32     int i, error = 0, N, local_max, over_max;
33     double *a, elapsed_time;
34     struct timeval tv1, tv2;
35     /*
36     * If no command line value is provided for the length of the array to be
37     * created than an error is generated.
38     */
39     if (argc < 2) {
40         fprintf (stderr, usage, argv[0]);
41         error = -1;
42     }
43
44     #pragma paraguin begin_parallel
45     /*
46     * The error is broadcast in a parallel region so that all processes receive
47     * the error. This prevents a deadlock if an error arises because all processes
48     * will receive the error and then exit.
49     */
50     #pragma paraguin bcast error
51     if (error) return error;
52     #pragma paraguin end_parallel
53
54     N = atoi(argv[1]);
55
56     #pragma paraguin begin_parallel
57     /*
58     * Broadcast N (user provided array length) to all parallel processes. Then
59     * allocate memory to an array a of length N. This needs to be done within a
60     * parallel region so a can be available to all processes.
61     */
```

```
62      #pragma paraguin bcast N
63      a = (double *) malloc (N * sizeof(double));
64      #pragma paraguin end_parallel
65
66      gettimeofday(&tv1, NULL);
67      srandom(tv1.tv_usec);
68      /*
69      * Populate an array of N length with random values.
70      */
71      for (i = 0; i < N; i++) {
72          a[i] = ((double) random()) / RAND_MAX * 100.0;
73      }
74
75      #pragma paraguin begin_parallel
76      /*
77      * Scatters the array a between all processes. This divides the contents of a
78      * evenly across the total number of processes. For example an array of length
79      * 60 with 6 processes would result in chunks of size 10 being sent to each process.
80      */
81      #pragma paraguin scatter a( N )
82      /*
83      * Sets the local_max to the largest possible negative value. This is a placeholder
84      * for the largest value in each local array.
85      */
86      local_max = -999999;
87
88      #pragma paraguin forall
89      for (i = 0; i < N; i++) {
90          local_max = max(local_max, a[i]);
91      }
92
93      ;
94      /*
95      * Reduce aggregates the locally computed maximum from each process. The max
96      * operator indicates that the largest value of the aggregate should be stored
97      * in the variabe over_max.
98      */
99      #pragma paraguin reduce max local_max over_max
100
101      #pragma paraguin end_parallel
102      /*
103      * Take a time stamp.  This won't happen until after the master
104      * process has gathered all the input from the other processes.
105      */
106      gettimeofday(&tv2, NULL);
107      /*
108      * Only the master should print this because the other processors only
109      * have partial products.
110      */
111      printf ("Overall maximum value = %d\n", over_max);
112      elapsed_time = (tv2.tv_sec - tv1.tv_sec) + ((tv2.tv_usec - tv1.tv_usec) / 1000000.0);
113      printf ("elapsed_time=\t%lf (seconds)\n", elapsed_time);
114
115 }
```

2.10 montyparallel.c File Reference

This example approximates the expected 2/3 odds of the Monty Hall Problem using an embarrassingly parallel implementation.

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <sys/time.h>
```

Functions

- int main (int argc, char *argv[])

2.10.1 Detailed Description

This example approximates the expected 2/3 odds of the Monty Hall Problem using an embarrassingly parallel implementation.

The Monty Hall problem is named for its similarity to the Let's Make a Deal television game show hosted by Monty Hall. The problem is stated as follows. Assume that a room is equipped with three doors. Behind two are goats, and behind the third is a shiny new car. You are asked to pick a door, and will win whatever is behind it. Let's say you pick door 1. Before the door is opened, however, someone who knows what's behind the doors (Monty Hall) opens one of the other two doors, revealing a goat, and asks you if you wish to change your selection to the third door (i.e., the door which neither you picked nor he opened). The Monty Hall problem is deciding whether you do. The correct answer is that you do want to switch. If you do not switch, you have the expected 1/3 chance of winning the car, since no matter whether you initially picked the correct door, Monty will show you a door with a goat. But after Monty has eliminated one of the doors for you, you obviously do not improve your chances of winning to better than 1/3 by sticking with your original choice. If you now switch doors, however, there is a 2/3 chance you will win the car (counterintuitive though it seems).

Weisstein, Eric W. "Monty Hall Problem." From MathWorld–A Wolfram Web Resource. http://mathworld.↩wolfram.com/MontyHallProblem.html

This program implements a simulation of the Monty Hall Problem in which the contestant always switches, thus resulting in an approximate win ratio of 0.66 repeating. The user inputs a command line argument representing the number of games. The greater the number of games the closer the win ration converges to 0.66 repeating, thus demonstrating that the intuition about selecting between the last two remaining doors being 50/50 odds is false.

Author

Peter Lawson

Date

February 2014

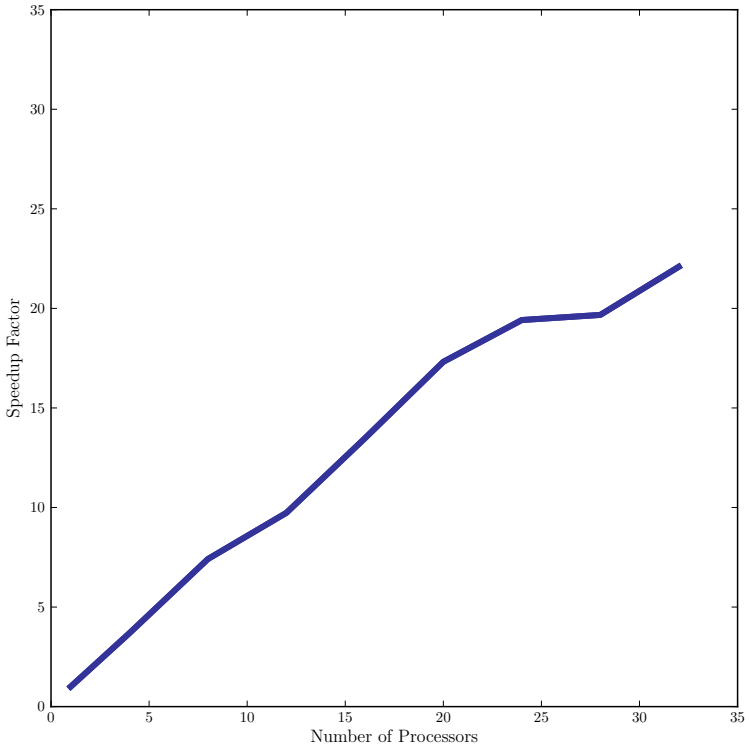

Figure 6: This figure shows the speedup factor of the Monty Hall simulation over an increasing number of parallel processes for 10 billion iterations. The speedup factor is computed by dividing the sequential execution time by the parallel execution time.

Definition in file montyparallel.c.

2.10.2 Function Documentation

2.10.2.1 int main (int *argc*, char * *argv[]*)

Definition at line 38 of file montyparallel.c.

```
39 {
40     char *usage = "Enter the number of games to simulate at console: %s N\n";
41     int i, iterations, prizedoor, pickadoor, showadoor, switchdoor, win, temp, loss, error;
42     double elapsed_time;
43     struct timeval tv1, tv2;
44     /*
45     * If no command line value is provided for the number of games to simulate,
46     * return an error.
47     */
48     if (argc < 2) {
49         fprintf (stderr, usage, argv[0]);
50         error = -1;
51     }
52     ;
53     #pragma paraguin begin_parallel
54     /*
55     * The error is broadcast in a parallel region so that all processes receive
56     * the error. This prevents a deadlock if an error arises because all processes
57     * will receive the error and then exit.
58     */
59     #pragma paraguin bcast error
60     if (error) return error;
61     #pragma paraguin end_parallel
62
63     iterations = atoi(argv[1]);
64
65     printf("\n\n ********* Let's Make a Deal!!!!!  ******** \n");
66     printf(" ___________      ___________      ___________ \n");
67     printf("|  __  __  |     |  __  __  |     |  __  __  |\n");
68     printf("| |  ||  | |     | |  ||  | |     | |  ||  | |\n");
69     printf("| |  ||  | |     | |  ||  | |     | |  ||  | |\n");
70     printf("| |__||__| |     | |__||__| |     | |__||__| |\n");
71     printf("|  __  __()|     |  __  __()|     |  __  __()|\n");
72     printf("| |  ||  | |     | |  ||  | |     | |  ||  | |\n");
73     printf("| |  ||  | |     | |  ||  | |     | |  ||  | |\n");
74     printf("| |  ||  | |     | |  ||  | |     | |  ||  | |\n");
75     printf("| |  ||  | |     | |  ||  | |     | |  ||  | |\n");
76     printf("| |__||__| |     | |__||__| |     | |__||__| |\n");
77     printf("|___________|     |___________|     |___________|\n\n\n");
78     ;
79     #pragma paraguin begin_parallel
80
81     /*
82     *This barrier is here so that we can take a time stamp
83     *once we know all processes are ready to go.
84     */
85
86     #pragma paraguin barrier
87
88     gettimeofday(&tv1, NULL);
89
90     /*
91     * Broadcast the number of iterations to all processes.
92     */
93     #pragma paraguin bcast iterations
94
95     /*
96     * Initialize the  wins to zero as well as seeding a random using system time
97     * in micro-seconds. This must be done within the parallel region so that the
98     * variables are available to all processes.
99     */
100     srandom(tv1.tv_usec);
101     win=0;
102
103     ;
104     #pragma paraguin forall
105     for (i = 0; i < iterations; i++) {
106
107         /*
108         * Assign a random value from 0 to 2 to prizedoor (the door hiding the
109         * prize) and pickadoor (the door randomly selected by the contestant).
110         */
111         prizedoor = random()%3;
112         pickadoor = random()%3;
113
114         /*
115         * The underlying logic of the game proceeds as such:
116         * The game show host will always reveal a door that is not the prize door
117         * and is not the door selected by the contestant. The following series of
```

```
118         * if statements implements this logic. In this simulation the contestant
119         * will always switch to the remaining closed door, which over a large number
120         * of iterations will approximate a win ratio of 0.66 repeating.
121         *
122         */
123         if (prizedoor==pickadoor)
124         {
125             if (prizedoor==2)
126             {
127                showadoor=random()%2;
128            }
129            if (prizedoor==0)
130            {
131             showadoor=(random()%2)+1;
132         }
133         if (prizedoor==1)
134         {
135             if ((random()%2)==0)
136             {
137                 showadoor=0;
138             }
139             else
140             {
141                 showadoor=2;
142             }
143         }
144     }
145     if (prizedoor!=pickadoor)
146     {
147         showadoor = 3-(prizedoor+pickadoor);
148     }
149
150     switchdoor = 3-(showadoor+pickadoor);
151
152     /*
153     * The final reveal! If the door that the contestant switched to is the same
154     * as the one revealed as the prize door by the host than win is incremented once.
155     */
156     if (switchdoor==prizedoor)
157     {
158         win++;
159     }
160 }
161
162 /*
163 * The partial win sums on each process are gathered and added to a single value
164 * (temp) representing the total number of wins.
165 */
166 ;
167 #pragma paraguin reduce sum win temp
168
169 #pragma paraguin end_parallel
170 /*
171 * Take a time stamp.  This won't happen until after the master
172 * process has gathered all the input from the other processes.
173 */
174 gettimeofday(&tv2, NULL);
175 printf ("Number of wins = (%d)\n", temp);
176 printf ("Number of losses = (%d)\n", (iterations-temp));
177 printf ("Percent of wins by switching = (%f)\n", ((double)temp/(double)iterations));
178
179 elapsed_time = (tv2.tv_sec - tv1.tv_sec) +
180 ((tv2.tv_usec - tv1.tv_usec) / 1000000.0);
181
182 printf ("elapsed_time=\t%lf (seconds)\n", elapsed_time);
183
184 }
```

2.11 pi.c File Reference

This program finds approximates the value of pi through Monte Carlo approximation.

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

Functions

- double f (double x)

  Returns the integral that represents the line of the circle within the first quadrant The ratio of the area under the curve with respect to the total quadrant will represent pi/4.
- int main (int argc, char ∗argv[])

2.11.1 Detailed Description

This program finds approximates the value of pi through Monte Carlo approximation.

This program estimates pi through a Monte Carlo method of approximation. This method is embarrassingly parallel as it decomposes into obviously independent tasks that can be done in parallel without any task communications during the computation. Monte Carlo pi approximation works by creating a function that represents a circle inscribed in a square. The circle is split into four quadrants, only one of which, quadrant 1, is needed to compute the area of a circle. An estimate of the area under the function representing the circle in quadrant 1 is performed. The process of estimation is embarassingly parallel and can be performed by multiple processes independently. Each process estimates the area under a section of the fucntion, and the resultant areas are summed and multiplied by 4 to get an estimate of pi.

Author

Clayton Ferner

Date

January 2014

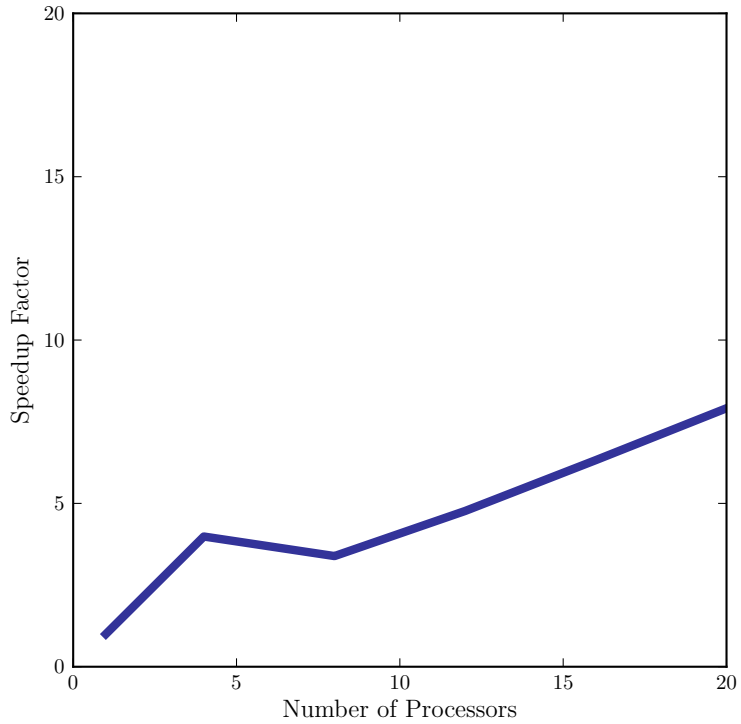

Figure 7: This figure shows the speedup for the approximation of pi for 100000000 iterations as the number of processes increases. The speedup factor is computed by dividing the sequential execution time by the parallel execution time.

Definition in file pi.c.

2.11.2 Function Documentation

2.11.2.1 double f (double *x*)

Returns the integral that represents the line of the circle within the first quadrant The ratio of the area under the curve with respect to the total quadrant will represent pi/4.

Parameters

x	the radius of the circle

Returns

the integral representing the first quadrant of a circle of a given radius

Definition at line 32 of file pi.c.

```
33 {
34     #pragma paraguin begin_parallel
35     return sqrt(1.0 - x*x);
36     #pragma paraguin end_parallel
37 }
```

2.11.2.2 int main (int *argc*, char * *argv[]*)

Definition at line 40 of file pi.c.

```
41 {
42     char *usage = "Usage: %s a b N\n";
43     int i, error = 0, N;
44     double a, b, x, y, size, area, overal_area, elapsed_time;
45     double est_pi, piError, truePI = 3.1415926535;
46     struct timeval tv1, tv2;
47     /*
48     * If no command line value is provided for the radius of the circle to be
49     * created for Monte Carlo approximation than an error is generated.
50     */
51     if (argc < 2) {
52         fprintf (stderr, usage, argv[0]);
53         error = -1;
54     } else {
55         N = atoi(argv[1]);
56     }
57
58     #pragma paraguin begin_parallel
59     /*
60     * The error is broadcast in a parallel region so that all processes receive
61     * the error. This prevents a deadlock if an error arises because all processes
62     * will receive the error and then exit.
63     */
64     #pragma paraguin bcast error
65     if (error) return error;
66     #pragma paraguin end_parallel
67
68     gettimeofday(&tv1, NULL);
69
70     #pragma paraguin begin_parallel
71     ;
72     /*
73     * Broadcast N (user provided radius) to all parallel processes.
74     */
75     #pragma paraguin bcast N
76
77     a = 0.0;
78     b = 1.0;
79
80     size = (b - a) / N;
81     area = 0.0;
82     /*
83     * For all; each process will execute a partition of the for loop. This for
84     * loop computes an estimate of an array under the function representing the
85     * quadrant 1 section of circle.
86     */
87     #pragma paraguin forall
88     for (i = 0; i < N-1; i++) {
89         x = a + i * size;
```

```
90          y = f(x);
91          area += y * size;
92      }
93
94      ;
95      /*
96      * A reduction is performed to aggregate the area estimates of all processes
97      * for each subsection of the area under the function. These are than added
98      * using the sum operator to get the total estimated area under the function
99      * representing quadrant 1.
100      */
101     #pragma paraguin reduce sum area overal_area
102
103     #pragma paraguin end_parallel
104
105 #ifndef PARAGUIN
106     overal_area = area;
107 #endif
108     /*
109     * Take a time stamp.  This won't happen until after the master
110     * process has gathered all the input from the other processes.
111     */
112     gettimeofday(&tv2, NULL);
113     /*
114     * Only the master should print this because the other processors only
115     * have partial products. The quadrant 1 area is multiplied by 4 to get the
116     * total area of the circle of radius N. This area will approximate pi. The
117     * estimated area, estimated pi value, true pi value, and error are all printed.
118     */
119     elapsed_time = (tv2.tv_sec - tv1.tv_sec) + ((tv2.tv_usec - tv1.tv_usec) / 1000000.0);
120     printf ("elapsed_time=\t%lf (seconds)\n", elapsed_time);
121     printf ("area = %lf\n", overal_area);
122     est_pi = overal_area * 4.0;
123     piError = fabs(est_pi - truePI);
124     printf ("Estimate of pi = %lf\n", est_pi);
125     printf ("Error = %lf\n", piError);
126 }
```

2.12 sieve_parallel.c File Reference

This is a parallel implementation of sieve of eratosthenes for computing all prime values up to some given value.

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <sys/time.h>
```

Macros

- #define N 100000000

Functions

- int main (int argc, char ∗argv[])

Variables

- int __guin_rank = 0
- int __guin_NP = 0

2.12.1 Detailed Description

This is a parallel implementation of sieve of eratosthenes for computing all prime values up to some given value.

The algorithm creates list of consecutive integers from 2 to n. The multiples of the list are enumerated and each value that is a factor of one of the integers in the list is marked. In this way all values that are not prime are marked, and those remaining are primes. By dividing the list across multiple processes the algorithm can be made parallel.

Author

Roger Johnson and Clayton Ferner

Date

January 2014

Definition in file sieve_parallel.c.

2.12.2 Macro Definition Documentation

2.12.2.1 #define N 100000000

Definition at line 22 of file sieve_parallel.c.

2.12.3 Function Documentation

2.12.3.1 int main (int *argc*, char * *argv[]*)

Definition at line 27 of file sieve_parallel.c.

```
27                                  {
28     char *marked,*temp;
29     int print=0,numIterations,n,prime,composite,i,j;
30     int procSize,low_value,high_value,first;
31     double elapsed_time;
32     struct timeval tv1, tv2;
33 
34     printf ("N = %d\n", N);
35     printf("Number of processors = %i\n",__guin_NP);
36 
37     #pragma paraguin begin_parallel
38     /*
39     * This barrier is here so that a time stamp can be taken. This ensures all
40     * processes have reached the barrier before exiting the parallel region
41     * and then taking a time stamp.
42     */
43     #pragma paraguin barrier
44     #pragma paraguin end_parallel
45 
46     gettimeofday(&tv1, NULL);
47 
48     #pragma paraguin begin_parallel
49     n=N;
50 
51     /*
52     * The number of processes cannot exceed the square root of N, if it does an
53     * error is returned indicating that the number of processes is too high.
54     */
55     procSize = (N-1)/__guin_NP;
56     if (2+procSize < (int) sqrt((double) N)) {
57         if (__guin_rank == 0){
58             printf ("Too many processors\n");
59             return 1;
60         }
61     }
62 
63     /*
64     * The array is partitioned with each processor given a high value and a low value.
65     */
66     low_value = (__guin_rank*N-1)/__guin_NP;
67     high_value = ((__guin_rank+1)*N-1)/__guin_NP;
68 
69     /*
70     * An array of N characters is allocated to serve as the means of indicating
71     * whether values in a given list are prime or not.
72     */
73     marked = malloc(N * sizeof(char));
74     temp = malloc(N * sizeof(char));
75     if(marked==NULL){
76         printf("Error allocating marked! \n");
77         return 1;
78     }
79 
```

```
80      for (i = 0; i < N; i++){
81          marked[i] = 1;
82      }
83
84      prime = 2;
85      /*
86      * All primes are computed up to the square root of N.
87      */
88      while(prime*prime<N)
89      {
90          /*
91          * Compute the first multiples of prime greater than or equal to the low value.
92          */
93          if (low_value % prime == 0) {
94              first = low_value;
95          }else{
96              first = low_value+prime - (low_value%prime);
97          }
98
99          /*
100          * Mark the multiples of prime in each processors given range.
101          */
102          for (i=first+prime; i<high_value; i+=prime) {
103              marked[i] = 0;
104          }
105
106          /*
107          * The master process identifies the next prime value
108          */
109          if(__guin_rank == 0){
110              while (marked[prime]!=1){
111                  prime++;
112              }
113          }
114          ;
115          /*
116          * The newly computed prime is broadcast to all processes
117          */
118          #pragma paraguin bcast prime
119      }
120      ;
121      /*
122      * All partial values of primes are gathered into the array marked.
123      */
124      #pragma paraguin gather marked ( n )
125
126      #pragma paraguin end_parallel
127
128
129       /*
130      * Take a time stamp.  This won't happen until after the master
131      * process has gathered all the input from the other processes.
132      */
133      gettimeofday(&tv2, NULL);
134
135      elapsed_time = (tv2.tv_sec - tv1.tv_sec) +
136                          ((tv2.tv_usec - tv1.tv_usec) / 1000000.0);
137      /*
138      * Only the master should print this because the other processors only
139      * have partial products. The list of all primes is printed as well as runtime.
140      */
141
142      printf ("elapsed_time=\t%lf (seconds)\n", elapsed_time);
143
144      if(print)
145          for (i=1;i<N;i++)
146              if (marked[i])
147                  printf("prime = %d\n",i);
148
149      #pragma paraguin begin_parallel
150      free(marked);
151      free(temp);
152      #pragma paraguin end_parallel
153
154      return 0;
155  }
```

2.12.4 Variable Documentation

2.12.4.1 int __guin_NP = 0

Definition at line 25 of file sieve_parallel.c.

2.12.4.2 int __guin_rank = 0

Definition at line 24 of file sieve_parallel.c.

2.13 sobel.c File Reference

This is a parallel implementation of sobel edge detection which applies a sobel mask to a greyscale pgm image.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <math.h>
#include <ctype.h>
#include <sys/time.h>
```

Macros

- #define N 1000

Functions

- static int clamp (int val)
- int main (int argc, char **argv)

2.13.1 Detailed Description

This is a parallel implementation of sobel edge detection which applies a sobel mask to a greyscale pgm image.

Author

Clayton Ferner

Date

February 2014

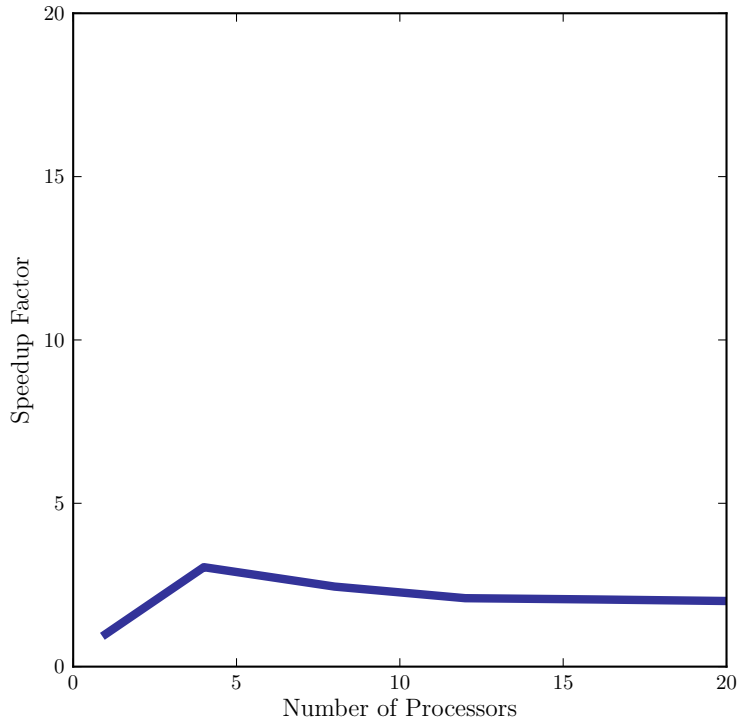

Figure 8: This figure shows the speedup factor for a parallel sobel edge detection algorithm operating on a 1000x1000 pgm image file over an increasing number of parallel processes.. The speedup factor is computed by dividing the sequential execution time by the parallel execution time.

Definition in file sobel.c.

2.13.2 Macro Definition Documentation

2.13.2.1 #define N 1000

Definition at line 23 of file sobel.c.

2.13.3 Function Documentation

2.13.3.1 static int clamp (int *val*) [static]

Definition at line 25 of file sobel.c.

```
25                          {
26      #pragma paraguin begin_parallel
27      if(val < 0)
28           val = 0;
29      else if(val > 255)
30           val = 255;
31      return val;
32      #pragma paraguin end_parallel
33 }
```

2.13.3.2 int main (int *argc*, char ** *argv*)

Definition at line 35 of file sobel.c.

```
35                                {
36
37      FILE *inFile, *oFile;
38      int grayImage[N][N], edgeImage[N][N];
39      char type[2];
40      int w, h, max;
41      int r, g, b, y, x, i, j, sum, sumx, sumy;
42
43      int GX[3][3], GY[3][3];
44      double elapsed_time;
45      struct timeval tv1, tv2;
46      int error = 0;
47      char buffer[BUFSIZ];
48
49      if (argc < 3) {
50           fprintf(stderr, "Usage: %s target\n", argv[1]);
51           error = -1;
52      }
53
54      if (!error) {
55          inFile = fopen(argv[1], "r");
56          if(inFile == 0) {
57               fprintf(stderr, "Open failed\n");
58               fprintf(stderr, "Usage: %s target\n", argv[1]);
59               error = -1;
60          }else{
61
62               /*
63               * Read the file type. Some graphics programs put comments in the
64               * file and if we find a #, indicating a comment, we throw away
65               * that specific line.
66               */
67               do
68                   fgets (buffer, BUFSIZ-1, inFile);
69               while (buffer[0] == '#');
70
71               if (buffer[0] != '#')
72                   sscanf(buffer, "%s", type);
73               else type[0] = '\0';
74
75               /*
76               * Read the height and width of the image file. If a comment is
77               * found throw away that specific line.
78               */
79               do
80                   fgets (buffer, BUFSIZ-1, inFile);
81               while (buffer[0] == '#');
82
83               if (buffer[0] != '#')
84                   sscanf(buffer, "%d%d", &w, &h);
85               else w = h = max = 0;
86
87               /*
88               * Read the height and width of the image file. If a comment is
89               * found throw away that specific line.
90               */
91               do
92                   fgets (buffer, BUFSIZ-1, inFile);
93               while (buffer[0] == '#');
94
95               if (buffer[0] != '#')
96                   sscanf(buffer, "%d", &max);
97               else w = h = max = 0;
98          }
99
100          if(type[1] != '2'){
101               fprintf(stderr, "%s is not a valid pgm file", argv[1]);
102               error = -1;
103          }
104
105
106          /*
107          * Read in the pixels
108          */
109          for(i=0;i < N; ++i){
110               for(j=0;j < N; ++j){
111                    fscanf(inFile, "%d", &grayImage[i][j]);
112               }
113          }
114
115          fclose(inFile);
116      }
117
118     #pragma paraguin begin_parallel
119     /*
120     * The error is broadcast in a parallel region so that all processes receive
121     * the error. This prevents a deadlock if an error arises because all processes
```

```
122     * will receive the error and then exit.
123     */
124     #pragma paraguin bcast error
125     if (error) return error;
126     /*
127     * This barrier is here so that a time stamp can be taken. This ensures all
128     * processes have reached the barrier before exiting the parallel region
129     * and then taking a time stamp.
130     */
131     #pragma paraguin barrier
132     #pragma paraguin end_parallel
133
134      gettimeofday(&tv1, NULL);
135
136      #pragma paraguin begin_parallel
137
138      /*
139      * These are the 3x3 sobel masks for the image.
140      */
141      GX[0][0] = -1; GX[0][1] = 0; GX[0][2] = 1;
142      GX[1][0] = -2; GX[1][1] = 0; GX[1][2] = 2;
143      GX[2][0] = -1; GX[2][1] = 0; GX[2][2] = 1;
144
145      GY[0][0] =  1; GY[0][1] =  2; GY[0][2] =  1;
146      GY[1][0] =  0; GY[1][1] =  0; GY[1][2] =  0;
147      GY[2][0] = -1; GY[2][1] = -2; GY[2][2] = -1;
148     /*
149     * The image that was read in as an array of pixels is broadcast to all
150     * processes as well as the dimensions of the image as width (w) and height (h)
151     */
152      #pragma paraguin bcast grayImage w h
153
154      #pragma paraguin forall
155      for(x=0; x < N; ++x){
156          for(y=0; y < N; ++y){
157              sumx = 0;
158              sumy = 0;
159              /*
160              * This handles image boundaries
161              */
162              if(x==0 || x==(h-1) || y==0 || y==(w-1))
163                  sum = 0;
164              else{
165                  /*
166                  * This is the x gradient approximation
167                  */
168                  for(i=-1; i<=1; i++)  {
169                      for(j=-1; j<=1; j++){
170                          sumx += (grayImage[x+i][y+j] * GX[i+1][j+1]);
171                      }
172                  }
173                  /*
174                  * This is the y gradient approximation
175                  */
176                  for(i=-1; i<=1; i++)  {
177                      for(j=-1; j<=1; j++){
178                          sumy += (grayImage[x+i][y+j] * GY[i+1][j+1]);
179                      }
180                  }
181                  /*
182                  * This is the gradient magnitude approximation
183                  */
184                  sum = (abs(sumx) + abs(sumy));
185              }
186              edgeImage[x][y] = clamp(sum);
187          }
188      }
189
190      ;
191      #pragma paraguin gather edgeImage
192
193      #pragma paraguin end_parallel
194
195      gettimeofday(&tv2, NULL);
196
197      elapsed_time = (tv2.tv_sec - tv1.tv_sec) +
198                         ((tv2.tv_usec - tv1.tv_usec) / 1000000.0);
199
200     /*
201     * Only the master should print this because the other processors only
202     * have partial products. The array consisting of the masked image is printed
203     * to a new file specified by the user as an argument at execution.
204     */
205
206     printf ("elapsed_time=\t%lf (seconds)\n", elapsed_time);
207      oFile = fopen(argv[2], "w");
208      if(oFile != 0){
```

```
209           fprintf(oFile, "P2\n%d %d\n%d\n", w,h,max);
210           for(i=0;i<h;++i){
211                for(j=0;j<w;++j){
212                     fprintf(oFile, "%d ", edgeImage[i][j]);
213                }
214                fprintf(oFile, "\n");
215           }
216           fclose(oFile);
217      }
218      return 0;
219 }
```

2.14 tsp.c File Reference

This is a parallel implementation of a solution to the travelling salesman problem.

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

Macros

- #define MAX_NUM_CITIES 100
- #define ABS(a) (((a) > 0) ? (a) : -(a))

Functions

- float computeDist (float D[MAX_NUM_CITIES][MAX_NUM_CITIES], int n, int perm[])
- int increment (int perm[], int n)
- void initialize (int perm[], int n, int i)
- void find_unique (int perm[], int j)
- void printUsage (char *argv0)
- int processArgs (int argc, char *argv[])
- float roundf (float x)
- int main (int argc, char *argv[])

Variables

- struct {
  float minDist
  int rank
  } myAnswer

- struct {
  float minDist
  int rank
  } resultAnswer

- int debug = 0
- int n = 0
- FILE * fd
- int __guin_rank = 0
- int __guin_NP = 0

2.14.1 Detailed Description

This is a parallel implementation of a solution to the travelling salesman problem.

Author

Clayton Ferner

Date

February 2014

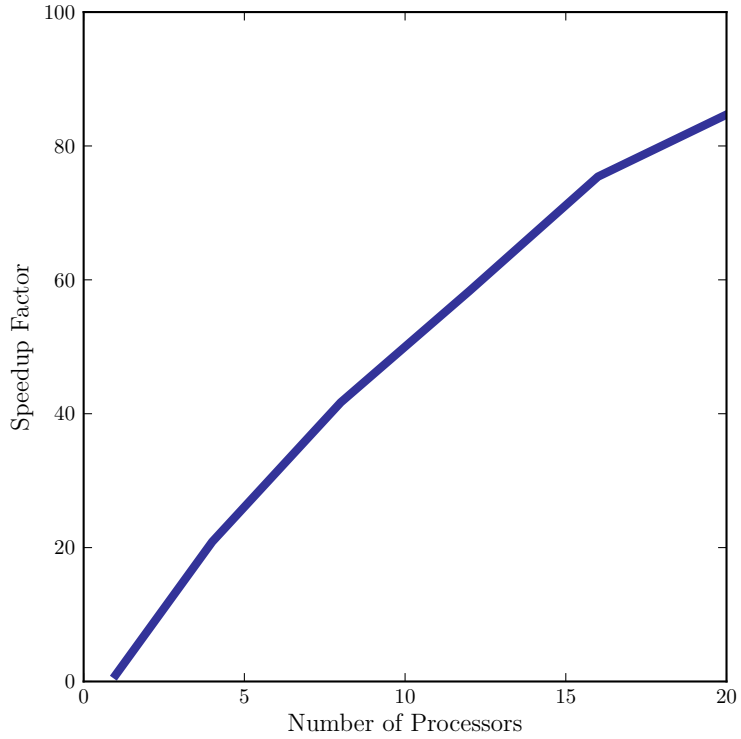

Figure 9: This figure shows the speedup factor for a parallel travelling salesman algorithm for an increasing number of parallel processes given a 12 city distance matrix. The speedup factor is computed by dividing the sequential execution time by the parallel execution time.

Definition in file tsp.c.

2.14.2 Macro Definition Documentation

2.14.2.1 #define ABS(*a*) (((a) > 0) ? (a) : -(a))

Definition at line 22 of file tsp.c.

2.14.2.2 #define MAX_NUM_CITIES 100

Definition at line 18 of file tsp.c.

2.14.3 Function Documentation

2.14.3.1 float computeDist (float *D[MAX_NUM_CITIES][MAX_NUM_CITIES],* int *n,* int *perm[ ]*)

Definition at line 221 of file tsp.c.

```
222 {
223     float dist;
224     int i;
225 
226     #pragma paraguin begin_parallel
227     dist = 0.0f;
228     for (i = 0; i < n-1; i++) {
229         dist += D[perm[i]][perm[i+1]];
230     }
231     dist += D[perm[n-1]][perm[0]];
232 
233     return dist;
234     #pragma paraguin end_parallel
235 }
```

2.14.3.2 void find_unique (int *perm[]*, int *j*)

Definition at line 290 of file tsp.c.

```
291 {
292     int k, unique;
293 
294     #pragma paraguin begin_parallel
295     unique = 0;
296     while (!unique) {
297 
298         /*
299         * Assume that it is unique unless we find out otherwise
300         */
301         unique = 1;
302 
303         for (k = 0; k < j; k++) {
304 
305             /*
306             * If it is not unique, try the next value and restart the check.
307             */
308             if (perm[k] == perm[j]) {
309                 perm[j]++;
310                 unique = 0;
311                 break;  // Start the k for loop over
312             }
313         }
314 
315     }
316     #pragma paraguin end_parallel
317 }
```

2.14.3.3 int increment (int *perm[]*, int *n*)

Definition at line 237 of file tsp.c.

```
238 {
239     int done;
240     int i;
241 
242     #pragma paraguin begin_parallel
243     done = 0;
244     i = n-1;
245     while (!done) {
246 
247 
248         perm[i]++;
249         find_unique (perm, i);
250         /*
251         * If there are no more unique values left (that haven't already
252         * been used) then set it back to zero and increment the previous
253         */
254         location (i.e. carry).
255 
256         if (perm[i] >= n) {
257             perm[i] = 0;
258             i--;
259 
260             /*
261             * If we carry beyond the first position, then we can no longer increment
262             */
263             if (i < 3)
264                 return 0;
265         } else
266         done = 1;
```

```
267     }
268
269     /*
270     * Then initialize (or reset) the rest of the array
271     */
272     initialize (perm, n, i+1);
273     return 1;
274     #pragma paraguin end_parallel
275 }
```

2.14.3.4 void initialize (int *perm[]*, int *n*, int *i*)

Definition at line 277 of file tsp.c.

```
278 {
279     int j;
280
281     #pragma paraguin begin_parallel
282     for (j = i; j < n; j++)
283     {
284         perm[j] = 0;
285         find_unique (perm, j);
286     }
287     #pragma paraguin end_parallel
288 }
```

2.14.3.5 int main (int *argc*, char * *argv[]*)

Definition at line 58 of file tsp.c.

```
59 {
60     int i, j, k, N, p;
61     int perm[MAX_NUM_CITIES], minPerm[MAX_NUM_CITIES+1];
62     float D[MAX_NUM_CITIES][MAX_NUM_CITIES];
63     float dist, minDist, finalMinDist;
64     double elapsed_time;
65     struct timeval tv1, tv2;
66     int abort;
67
68     abort = processArgs(argc, argv);
69
70     if (!abort) {
71         for (i = 0; i < n; i++) {
72             D[i][i] = 0.0f;
73             for (j = 0; j < i; j++) {
74                 fscanf (fd, "%f", &D[i][j]);
75                 D[j][i] = D[i][j];
76             }
77         }
78     } else {
79         if (n <= 1)
80             printf ("0 0 0\n");
81         else
82             n = 0;
83     }
84
85     #pragma paraguin begin_parallel
86     #pragma paraguin bcast abort
87     if (abort) return -1;
88     #pragma paraguin end_parallel
89
90
91     if (debug) {
92         printf ("\n\nDistances:\n");
93         for (i = 0; i < n; i++) {
94             for (j = 0; j < n; j++) {
95                 printf ("%8.1f", D[i][j]);
96             }
97             printf ("\n");
98         }
99         printf ("\n\n");
100     }
101
102
103     ;
104     #pragma paraguin begin_parallel
105     #pragma paraguin barrier
106     #pragma paraguin end_parallel
107
108     /*
```

```
109     * Take a time stamp.
110     */
111     gettimeofday(&tv1, NULL);
112
113     #pragma paraguin begin_parallel
114     #pragma paraguin bcast debug n D
115
116     perm[0] = 0;
117     minDist = 9.0e10; // Near the largest value we can represent with a float
118
119     if (n == 2) {
120         perm[1] = 1; // If n = 2, the N = 0, and we are done.
121         minPerm[0] = perm[0]; minPerm[1] = perm[1];
122         minDist = computeDist(D, n, perm);
123     }
124     /*
125     * This computes N = (n-1)(n-2)
126     */
127     N = n*n - 3*n + 2;
128    /*
129     * This parallelizes the following loop by assigning iterations
130     * of the loop to different partitions.
131     */
132     #pragma paraguin forall
133     for (p = 0; p < N; p++) {
134
135         if (debug && __guin_rank == 0 && p == 0) {
136             printf ("__guin_NP = %d\n", __guin_NP);
137         }
138
139         if (debug) {
140             printf ("pid %d: p = %d\n", __guin_rank, p);
141         }
142
143         perm[1] = p / (n-2) + 1;
144         perm[2] = p % (n-2) + 1;
145
146         if (perm[2] >= perm[1])
147             perm[2]++;
148
149         initialize(perm, n, 3);
150         do {
151
152             if (debug) {
153                 printf ("pid %d:", __guin_rank);
154                 for (i = 0; i < n; i++)
155                     printf ("%4d", perm[i]);
156             }
157
158             dist = computeDist(D, n, perm);
159
160             if (debug) printf ("\tdist = %f\n", dist);
161
162             if (minDist < 0 || minDist > dist) {
163                 minDist = dist;
164                 for (i = 0; i < n; i++)
165                     minPerm[i] = perm[i];
166             }
167
168         } while (increment(perm,n));
169     }
170
171     myAnswer.minDist = minDist;
172     myAnswer.rank = __guin_rank;
173
174     #pragma paraguin reduce minloc myAnswer resultAnswer
175
176     /*
177     * Send a message with the minumum Permutation from the rank that computed
178     * it to the master process.
179     */
180     #pragma paraguin message minPerm resultAnswer.rank 0
181     if (__guin_rank == resultAnswer.rank) {
182
183         if (debug) {
184             printf ("Minimum dist = %f\n\n", minDist);
185
186             for (i = 0; i < n; i++) {
187                printf ("%4d ->", minPerm[i]);
188            }
189
190            printf ("%4d\n", minPerm[0]);
191        } else {
192
193         printf (" %f ", minDist);
194
195         for (i = 0; i < n; i++) {
```

```
196             printf ("%d ", minPerm[i]);
197         }
198
199         printf ("%d\n", minPerm[0]);
200     }
201 }
202     #pragma paraguin end_parallel
203
204 gettimeofday(&tv2, NULL);
205
206 elapsed_time = ((tv2.tv_sec - tv1.tv_sec) * 1000.0) +
207 ((tv2.tv_usec - tv1.tv_usec) / 1000.0);
208
209 if (debug) {
210
211     printf ("Total time to compute =\t%lf milliseconds\n", elapsed_time);
212 } else {
213
214     int etime = (int) roundf (elapsed_time);
215     printf ("%f milliseconds \n", elapsed_time);
216 }
217
218
219 }
```

2.14.3.6 void printUsage (char * *argv0*)

Definition at line 319 of file tsp.c.

```
320 {
321     fprintf (stderr, "\nUsage: %s [options] n [DistanceMatrix]\n", argv0);
322     fprintf (stderr, "\n \
323         options are: \n \
324         -h, --help \n \
325         print this help and quit \n \
326         \n \
327         -d, --debug \n \
328         print debuging information (do not use with a large n) \n \
329         \n \
330         The DistanceMatrix should be a lower triangular matrix of  \n \
331         distances between cities. For example, if n = 5, the DistanceMatrix \n \
332         might look like: \n \
333         62.635856 \n \
334         25.893111 54.673987 \n \
335         38.751828 75.894124 22.376224 \n \
336         71.873977 50.518504 47.698433 55.902881 \n \
337         \n \
338         If the DistanceMatrix is not provided, then it is read from stdin. \n \
339         \n \
340         The output consists of n+3 integers printed to stdout. \n \
341         The output is: \n \
342         time to derive the solution in milliseconds \n \
343         an approximate total distance of the circuit \n \
344         (rounded to the nearest integer) \n \
345         a permuation of the first n integers starting a zero \n \
346         (zero indicates the originating city) \n \
347         a zero (indicating a return to the originating city) \n \
348         \n\n");
349 }
```

2.14.3.7 int processArgs (int *argc*, char * *argv[]*)

Definition at line 352 of file tsp.c.

```
353 {
354     char *argv0;
355
356     argv0 = argv[0];
357     argc--; argv++;
358     while (argc > 0 && argv[0][0] == '-') {
359
360         if (strcmp(argv[0], "-d") == 0 ||
361             strcmp(argv[0], "--debug") == 0) {
362             debug = 1;
363     } else {
364         printUsage(argv0);
365         return 1;
366     }
367     argc--; argv++;
368 }
```

```
369
370 if (argc < 1) {
371     printUsage(argv0);
372     return 1;
373 }
374
375 n = atoi (argv[0]);
376
377 if (n <= 1) {
378     return 1;
379 }
380
381 if (argc > 1 && (fd = fopen(argv[1], "r")) == NULL) {
382     fprintf (stderr, "%s: unable to open file \"%s\" for reading\n",
383        argv0, argv[1]);
384     return 1;
385 } else if (argc <= 1) {
386     fd = stdin;
387 }
388
389 return 0;
390 }
```

2.14.3.8 float roundf (float *x*)

2.14.4 Variable Documentation

2.14.4.1 int __guin_NP = 0

Definition at line 54 of file tsp.c.

2.14.4.2 int __guin_rank = 0

Definition at line 53 of file tsp.c.

2.14.4.3 int debug = 0

Definition at line 51 of file tsp.c.

2.14.4.4 FILE* fd

Definition at line 52 of file tsp.c.

2.14.4.5 float minDist

Definition at line 30 of file tsp.c.

2.14.4.6 struct { ... } myAnswer

2.14.4.7 int n = 0

Definition at line 51 of file tsp.c.

2.14.4.8 int rank

Definition at line 31 of file tsp.c.

2.14.4.9 struct { ... } resultAnswer