## Contents

### 1 File Index

1.1 File List .......................................................... 1

### 2 File Documentation

2.1 factorial.c File Reference ........................................... 2

2.1.1 Detailed Description .............................................. 2

2.1.2 Macro Definition Documentation ............................... 3

2.1.3 Function Documentation ......................................... 3

2.2 fireplace.c File Reference .......................................... 4

2.2.1 Detailed Description ............................................. 4

2.2.2 Macro Definition Documentation ............................... 5

2.2.3 Function Documentation ......................................... 5

2.2.4 Variable Documentation ........................................ 8

2.3 hello.c File Reference ............................................... 8

2.3.1 Detailed Description ............................................. 8

2.3.2 Function Documentation ......................................... 9

2.3.3 Variable Documentation ........................................ 9

2.4 helloHybrid.c File Reference ..................................... 9

2.4.1 Detailed Description ............................................. 9

2.4.2 Function Documentation ......................................... 10

2.4.3 Variable Documentation ........................................ 10

2.5 integ.c File Reference .............................................. 11

2.5.1 Detailed Description ............................................. 11

2.5.2 Function Documentation ......................................... 12

2.5.3 Variable Documentation ........................................ 14

2.6 matrixadd.c File Reference ....................................... 14

2.6.1 Detailed Description ............................................. 14

2.6.2 Macro Definition Documentation ............................... 16

2.6.3 Function Documentation ......................................... 16

2.7 matrixmult.c File Reference ..................................... 18

2.7.1 Macro Definition Documentation ............................... 18

2.7.2 Function Documentation ......................................... 18

2.8 matrixmult.hybrid.c File Reference ............................ 20

2.8.1 Detailed Description ............................................. 21

2.8.2 Macro Definition Documentation ............................... 21

2.8.3 Function Documentation ......................................... 21

2.9 max.c File Reference ............................................... 23

2.9.1 Detailed Description ............................................. 24
1 File Index

1.1 File List

Here is a list of all files with brief descriptions:

- **factorial.c**
  This program computes N factorial
  2

- **fireplace.c**
  This program utilizes Jacobi iterations to simulate a basic 2-D heat distribution from a given heat source
  4

- **hello.c**
  This program is a simple parallel implementation of hello world
  8

- **helloHybrid.c**
  This program is a hybrid parallel implementation of hello world that includes distributed memory (MPI) and shared memory (OpenMP) constructs
  9

- **integ.c**
  This is an example of integration by approximating the area under a function in parallel
  11
matrixadd.c
This is a parallel implementation of matrix addition 14

matrixmult.c
This is a parallel implementation of matrix multiplication 18

matrixmult.hybrid.c
This is a parallel implementation of matrix multiplication that includes distributed memory and
shared memory parallelism 20

max.c
This program finds the maximum value of a randomly generated array of user provided length 23

montyparallel.c
This example approximates the expected 2/3 odds of the Monty Hall Problem using an embar-
راسingly parallel implementation 25

pl.c
This program finds approximates the value of pi through Monte Carlo approximation 28

sieve_parallel.c
This is a parallel implementation of sieve of eratosthenes for computing all prime values up to
some given value 31

sobel.c
This is a parallel implementation of sobel edge detection which applies a sobel mask to a
greyscale pgm image 34

tsp.c
This is a parallel implementation of a solution to the travelling salesman problem 38

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.
2.1 factorial.c File Reference

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 *This barrier is here so that we can take a time stamp
43 *once we know all processes are ready to go.
44 */
45 #pragma paraguin barrier
46 #pragma paraguin end_parallel
47
48 gettimeofday(&tv1, NULL); 
49 ;
50 #pragma paraguin begin_parallel
51 #pragma paraguin bcast n
52 /*
53 * Fact must be initialized within the parallel region so it is available
54 * to all processes as 1.
55 */
56 fact = 1;
57
58 ;
59 #pragma paraguin forall
60 /*
61 * This implementation of factorial is iterative rather than recursive. This
62 * allows the for loop to be easily broken into segments equal to the number
63 * of available processes through the use of a paraguin forall pragma.
64 */
65 for (i = 1; i <= n; i++)
66 fact = fact * i;
67
68 /*
69 * Reduce the partial products (fact) at each process to a single total
70 * sum (answer).
71 */
72 #pragma paraguin reduce prod fact answer
73 #pragma paraguin end_parallel
74 /*
75 * Take a time stamp. This won’t happen until after the master
This program utilizes Jacobi iterations to simulate a basic 2-D heat distribution from a given heat source.

```c
#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 distributed 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
2.2 File Reference

Date

January 2014

![Graph showing speedup factor vs. number of processors]

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 

```c
#define M 900
```

Definition at line 20 of file fireplace.c.

2.2.2.2 

```c
#define N 900
```

Definition at line 19 of file fireplace.c.

2.2.2.3 

```c
#define TOTAL_TIME 30000
```

Definition at line 18 of file fireplace.c.

2.2.3 Function Documentation

2.2.3.1 

```c
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

<table>
<thead>
<tr>
<th>A</th>
<th>the current array for which the distribution is being computed</th>
</tr>
</thead>
<tbody>
<tr>
<td>i</td>
<td>the i index of the array</td>
</tr>
<tr>
<td>j</td>
<td>the j index of the array</td>
</tr>
</tbody>
</table>

### 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 {  
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    gettimeofday(&tv1, NULL);
81    /*
82     * Compute the Jacobi iterations necessary for the heat distribution problem. Room is the array, n and m are
83     * the array dimensions, max_iterations is the maximum number of iterations
84     * given by the user, and computeValue is the function to compute heat
85     * distribution at each index of the given array. A function must always be
86     * passed into the Paraguin stencil pattern.
87     */
88    #pragma paraguin stencil room n m max_iterations computeValue
89    gettimeofday(&tv2, NULL);
90    elapsed_time = (tv2.tv_sec - tv1.tv_sec) +
91        ((tv2.tv_usec - tv1.tv_usec) / 1000000.0);
92    /*
93     * Only the master should print this because the other processors only
94     * have partial products. The last accessed array 2D array of the 3D array
95     */
96    #pragma paraguin end_parallel
97    /*
98     * Take a time stamp. This won’t happen until after the master
99     * process has gathered all the input from the other processes.
100     */
101    gettimeofday(&tv2, NULL);
102    elapsed_time = (tv2.tv_sec - tv1.tv_sec) +
103        ((tv2.tv_usec - tv1.tv_usec) / 1000000.0);
104    /*
105     * Then print out all the results.
106     */
107    for (i = 0; i < N; i++) {
108        for (j = 0; j < M; j++) {
109            printf("%10.2f", room[0][i][j]);
110        }
111        printf("\n");
112    }
113    for (i = 0; i < N; i++) {
114        for (j = 0; j < M; j++) {
115            printf("%10.2f", room[1][i][j]);
116        }
117        printf("\n");
118    }
119    }
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 (with 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 implements 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.

```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 (without 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.

```c
#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 (without 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 implements 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.

```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 #pragma paraguin end_parallel
64 /*
65 * Prints a statement indicating that the master process is ending.
66 */
67 printf("Goodbye world from process %d.\n", __guin_rank);
68 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 (with 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 $4\sin(1.5x)+5$ 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 $4\sin(1.5x)+5$. 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
Figure 2: This figure shows the speedup factor for an algorithm that approximates the integral of $4\sin(1.5x)+5$ 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 $4\sin(1.5x)+5$ 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 $4\sin(1.5x)+5$ function

Definition at line 31 of file integ.c.

```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.
char *usage = "Usage: %s a b N\n";

int i, error = 0, N;

double a, b, x, y, size, area, overal_area, elapsed_time;

struct timeval tv1, tv2;

/*
 * Command line values need to be provided for the range over which the integral
 * needs to be computed and the number of iterations (N). If there are not enough
 * arguments return an error. If a, the range start, is more than b, the range
 * end, then return an error. Otherwise read arguments 1, 2, and 3 into a,
 * b, and N respectively.
 */

if (argc < 4) {
    fprintf(stderr, usage, argv[0]);
    error = -1;
} else {
    a = atof(argv[1]);
    b = atof(argv[2]);
    N = atoi(argv[3]);

    if (b <= a) {
        fprintf(stderr, "a should be smaller than b\n");
        error = -1;
    }
}

/*
 * The error is broadcast in a parallel region so that all processes receive
 * the error. This prevents a deadlock if an error arises because all processes
 * will receive the error and then exit.
 */
#pragma paraguin begin_parallel
#pragma paraguin bcast error
#else
#pragma paraguin barrier
#endif
if (error) return error;
#pragma paraguin end_parallel

#pragma paraguin begin_parallel

/*
 * This barrier is here so that a time stamp can be taken. This ensures all
 * processes have reached the barrier before exiting the parallel region
 * and then taking a time stamp.
 */
#pragma paraguin barrier
#pragma paraguin end_parallel

gettimeofday(&tv1, NULL);

#pragma paraguin begin_parallel

/*
 * Broadcast the range start (a), range end (b), and number of iterations (N)
 * to all processes.
 */
#pragma paraguin bcast a b N

size = (b - a) / N;
area = 0.0;

/*
 * This parallelizes the following loop nest assigning iterations
 * of the outermost loop (i) to different partitions. The area under the
 * function is estimated by each process for its respective loop partition.
 */
#pragma paraguin forall
for (i = 0; i < N-1; i++) {
    x = a + i * size;
    y = f(x);
    area += y * size;
}
}

#pragma paraguin reduce sum area overal_area
#pragma paraguin end_parallel

ifndef PARAGUIN

overal_area = area;
#endif

#include "paraguin.h"

#ifndef PARAGUIN

timeofday(&tv2, NULL);
#endif

/*
 * The area computed by each process is reduced by a sum operator into the
 * total area under the function from a to b, and this area represents the
 * estimated integral.
 */
#ifndef PARAGUIN

gettimeofday(&tv2, NULL);
#endif

/*

Generated on Wed May 7 2014 00:44:05 for Paraguin Compiler Examples by Doxygen

*/
Only the master should print this because the other processors only have partial products. The quadrant 1 area is multiplied by 4 to get the total area of the circle of radius \(N\). This area will approximate \(\pi\). The estimated area, estimated \(\pi\) value, true \(\pi\) value, and error are all printed.

```
elapsed_time = (tv2.tv_sec - tv1.tv_sec) + ((tv2.tv_usec - tv1.tv_usec) / 1000000.0);
printf("elapsed_time=%lf (seconds)\n", elapsed_time);
printf("area = %lf\n", overal_area);
```
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 \(O(n^2)\) 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.
if (argc < 2) {
    fprintf (stderr, usage, argv[0]);
    error = -1;
}

/*
 * If there is an error in reading the input file containing the matrices to
 * be added an error is given.
 */
if (!error && (fd = fopen (argv[1], "r")) == NULL) {
    fprintf (stderr, "%s: Cannot open file %s for reading.
", argv[0], argv[1]);
    fprintf (stderr, usage, argv[0]);
    error = -1;
}

#pragma paraguin begin_parallel
#pragma paraguin bcast error
if (error) return -1;
#pragma paraguin end_parallel

/*
 * Read input from file for matrices a and b. The I/O is not timed because
 * this I/O needs to be done regardless of whether this program is run
 * sequentially on one processor or in parallel on many processors.
 * Therefore, it is irrelevant when considering speedup.
 */
for (i = 0; i < N; i++)
for (j = 0; j < N; j++)
    fscanf (fd, "%lf", &a[i][j]);

for (i = 0; i < N; i++)
for (j = 0; j < N; j++)
    fscanf (fd, "%lf", &b[i][j]);

fclose(fd);

#pragma paraguin begin_parallel
/*
 * This barrier is here so that a time stamp can be taken. This ensures all
 * processes have reached the barrier before exiting the parallel region
 * and then taking a time stamp.
 */
#pragma paraguin barrier
#pragma paraguin end_parallel

gettimeofday(&tv1, NULL);

#pragma paraguin begin_parallel
/*
 * Scatter a and b matrices to all processes. This divides both matrices
equally between all available processes.
 */
#pragma paraguin scatter a b
#pragma paraguin forall

for (i = 0; i < N; i++)
for (j = 0; j < N; j++)
    c[i][j] = a[i][j] + b[i][j];

#pragma paraguin gather c
#pragma paraguin end_parallel

gettimeofday(&tv2, NULL);

elapsed_time = (tv2.tv_sec - tv1.tv_sec) +
((tv2.tv_usec - tv1.tv_usec) / 1000000.0);

/*
 * Only the master should print this because the other processors only
 * have partial products.
 */
2.6.3.2 print_results (char *prompt, double a[N][N])

Prints a given 2D array.

Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>prompt</td>
<td>a string to represent the 2D array</td>
</tr>
<tr>
<td>a</td>
<td>the 2D array to be printed</td>
</tr>
</tbody>
</table>

Definition at line 141 of file matrixadd.c.

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
2.7 matrixmult.c File Reference

Date
February 2014

![Speedup Factor vs. Number of Processors](image)

**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 

```c
#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.

```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     */
```
/* If there is an error in reading the input file containing the matrices to
 * be added an error is given.
 */
if ((fd = fopen (argv[1], "r")) == NULL) {
    fprintf (stderr, "Cannot open file %s for reading.
", argv[0], argv[1]);
    fprintf (stderr, usage, argv[0]);
    return -1;
}

#pragma paraguin begin_parallel
#pragma paraguin bcast error
if (error) return error;
#pragma paraguin end_parallel

/*
 * Read input from file for matrices a and b. The I/O is not timed because
 * this I/O needs to be done regardless of whether this program is run
 * sequentially on one processor or in parallel on many processors.
 * Therefore, it is irrelevant when considering speedup.
 */
for (i = 0; i < N; i++)
    for (j = 0; j < N; j++)
        fscanf (fd, "%lf", &a[i][j]);

for (i = 0; i < N; i++)
    for (j = 0; j < N; j++)
        fscanf (fd, "%lf", &b[i][j]);

#pragma paraguin begin_parallel
/*
 * This barrier is here so that a time stamp can be taken. This ensures all
 * processes have reached the barrier before exiting the parallel region
 * and then taking a time stamp.
 */
#pragma paraguin barrier
#pragma paraguin end_parallel

gettimeofday(&tv1, NULL);

#pragma paraguin begin_parallel
/*
 * Broadcast the input to all processors. This could be
 * faster if we used scatter, but bcast is easy and scatter
 * is not implemented in Paraguin
 */
#pragma paraguin scatter a
#pragma paraguin bcast b

/*
 * This parallelizes the following loop nest assigning iterations
 * of the outermost loop (i) to different partitions. This allows for
 * each process to multiply its section of a with the corresponding part of b
 * and place the result in the array c.
 */
#pragma paraguin forall
for (i = 0; i < N; i++)
    for (j = 0; j < N; j++)
        c[i][j] = 0.0;
    for (k = 0; k < N; k++)
        c[i][j] = c[i][j] + a[i][k] * b[k][j];

#pragma paraguin gather c
#pragma paraguin end_parallel

/*
 * This gathers the partial values from c into a final result.
 */
2.8 matrixmult.hybrid.c File Reference

/* Take a time stamp. This won’t happen until after the master process has gathered all the input from the other processes. */
gettimeofday(&tv2, NULL);

time_t elapsed_time = (tv2.tv_sec - tv1.tv_sec) + ((tv2.tv_usec - tv1.tv_usec) / 1000000.0);
printf("elapsed_time=\t%f (seconds)\n", elapsed_time);

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

Prints a given 2D array.

<table>
<thead>
<tr>
<th>prompt</th>
<th>a string to represent the 2D array</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>the 2D array to be printed</td>
</tr>
</tbody>
</table>

Definition at line 151 of file matrixmult.c.

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

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 $O(n^3)$ 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 $512 \times 512$ 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`.

```c
27 {
28    int i, j, k, error = 0, tid;
29    double a[N][N], b[N][N], c[N][N];
```
char *usage = "Usage: %s file\n";
FILE *fd;
double elapsed_time, start_time, end_time;
struct timeval tv1, tv2;

if (argc < 2) {
    fprintf (stderr, usage, argv[0]);
    error = -1;
}

if ((fd = fopen (argv[1], "r")) == NULL) {
    fprintf (stderr, "%s: Cannot open file %s for reading.\n", argv[0], argv[1]);
    fprintf (stderr, usage, argv[0]);
    error = -1;
}

#pragma paraguin begin_parallel
#pragma paraguin bcast error
if (error) return error;
#pragma paraguin end_parallel

for (i = 0; i < N; i++)
    for (j = 0; j < N; j++)
        fscanf (fd, "%lf", &a[i][j]);

for (i = 0; i < N; i++)
    for (j = 0; j < N; j++)
        fscanf (fd, "%lf", &b[i][j]);

#pragma paraguin begin_parallel
/*
 * This barrier is here so that a time stamp can be taken. This ensures all
 * processes have reached the barrier before exiting the parallel region
 * and then taking a time stamp.
 */
#pragma paraguin barrier
#pragma paraguin end_parallel

gettimeofday(&tv1, NULL);

#pragma paraguin begin_parallel
/*
 * Broadcast the input to all processors. This could be
 * faster if we used scatter, but bcast is easy and scatter
 * is not implemented in Paraguin
 */
#pragma paraguin scatter a
#pragma paraguin bcast b

/*
 * This parallelizes the following loop nest assigning iterations
 * of the outermost loop (i) to different partitions. An OpenMP parallel
 * pragma is then used to parallelize the inner loop (j) so that the outer
 * loop is parallelized across distributed memory processes while simultaneously
 * parallelizing using shared memory threads. This allows for each process to
 * multiply its section of a with the corresponding part of b and place the
 * result in the array c.
 */
for (i = 0; i < N; i++) {
```c
#pragma omp parallel for private(tID, j, k) num_threads(4)
for (j = 0; j < N; j++) {
    tID = omp_get_thread_num();
    c[i][j] = 0.0;
    for (k = 0; k < N; k++) {
        c[i][j] = c[i][j] + a[i][k] * b[k][j];
    }
}

#pragma omp gather c
#pragma omp end_parallel

/*
 * Take a time stamp. This won't happen until after the master
 * process has gathered all the input from the other processes.
 */
gmtimeofday(&tv2, NULL);

elapsed_time = tv2.tv_sec - tv1.tv_sec +
((tv2.tv_usec - tv1.tv_usec) / 1000000.0);

printf("elapsed_time=%lf (seconds)\n", elapsed_time);
print_results("C = ", c);
}

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

Prints a given 2D array.

Parameters

<table>
<thead>
<tr>
<th>prompt</th>
<th>a string to represent the 2D array</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>the 2D array to be printed</td>
</tr>
</tbody>
</table>

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

```
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.
62  #pragma paraguin bcast N
63  a = (double *) malloc (N * sizeof(double));
64  #pragma paraguin end_parallel
65  gettimeofday(&tv1, NULL);
66  srand(tv1.tv_usec);
67  /*
68   * Populate an array of N length with random values.
69   */
70  for (i = 0; i < N; i++) {
71    a[i] = ((double) random()) / RAND_MAX * 100.0;
72  }
73
74  #pragma paraguin begin_parallel
75  /*
76   * Scatters the array a between all processes. This divides the contents of a
77   * evenly across the total number of processes. For example an array of length
78   * 60 with 6 processes would result in chunks of size 10 being sent to each process.
79   */
80  #pragma paraguin scatter a( N )
81  /*
82   * Sets the local_max to the largest possible negative value. This is a placeholder
83   * for the largest value in each local array.
84   */
85  local_max = -999999;
86
87  #pragma paraguin forall
88  for (i = 0; i < N; i++) {
89    local_max = max(local_max, a[i]);
90  }
91
92  /*
93   * Reduce aggregates the locally computed maximum from each process. The max
94   * operator indicates that the largest value of the aggregate should be stored
95   * in the variable over_max.
96   */
97  #pragma paraguin reduce max local_max over_max
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   * Only the master should print this because the other processors only
107   * have partial products.
108   */
109  printf("Overall maximum value = %d\n", over_max);
110  printf("elapsed_time=\t%lf (seconds)\n", elapsed_time);
111  printf("\n");
112  }
113

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).


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.
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"
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
********* Let’s Make a Deal!!!!!! ********* \n\n");
66 printf(" __________ __________ __________ \n\n");
67 printf("| __ __ | | __ __ | | __ __ | \n\n");
68 printf("| | || | | | | || | | | | || | | \n\n");
69 printf("| | || | | | | || | | | | || | | \n\n");
70 printf("| |__||__| | | |__||__| | | |__||__| | \n\n");
71 printf("| __ __()| | __ __()| | __ __()| \n\n");
72 printf("| | || | | | | || | | | | || | | \n\n");
73 printf("| | || | | | | || | | | | || | | \n\n");
74 printf("| | || | | | | || | | | | || | | \n\n");
75 printf("| | || | | | | || | | | | || | | \n\n");
76 printf("| |__||__| | | |__||__| | | |__||__| | \n\n");
77 printf("|__________| |__________| |__________| \n\n");
78 }
79 #pragma paraguin begin_parallel
80 /*
81 *This barrier is here so that we can take a time stamp.
82 *once we know all processes are ready to go.
83 */
84 #pragma paraguin barrier
85
86 gettimeofday(&tv1, NULL);
87 /*
88 * Broadcast the number of iterations to all processes.
89 */
90 #pragma paraguin bcast iterations
91 /*
92 * Initialize the wins to zero as well as seeding a random using system time
93 * in micro-seconds. This must be done within the parallel region so that the
94 * variables are available to all processes.
95 */
96 srandom(tv1.tv_usec);
97 win=0;
98 }
99 #pragma paraguin forall
100 for (i = 0; i < iterations; i++) {
101 /*
102 * Assign a random value from 0 to 2 to prizedoor (the door hiding the
103 * prize) and pickadoor (the door randomly selected by the contestant).
104 */
105 prizedoor = random() %3;
106 pickadoor = random() %3;
107 /*
108 * The underlying logic of the game proceeds as such:
109 * The game show host will always reveal a door that is not the prize door
110 * and is not the door selected by the contestant. The following series of
This program finds approximates the value of pi through Monte Carlo approximation.

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

2.11 pi.c File Reference

This program finds approximates the value of pi through Monte Carlo approximation.
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 embarrassingly parallel and can be performed by multiple processes independently. Each process estimates the area under a section of the function, 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 100,000,000 iterations as the number of processes increases. The speedup factor is computed by dividing the sequential execution time by the parallel execution time.](image)

Definition in file \texttt{pi.c}. 

Generated on Wed May 7 2014 00:44:05 for Paraguin Compiler Examples by Doxygen
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

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>the radius of the circle</td>
</tr>
</tbody>
</table>

Returns

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

Definition at line 32 of file pi.c.

```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.

```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, overall_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    gettimeofday(&tv1, NULL);
68    /*
69    * Broadcast N (user provided radius) to all parallel processes.
70    */
71    #pragma paraguin bcast N
72    /*
73    * For all; each process will execute a partition of the for loop. This for
74    * loop computes an estimate of an array under the function representing the
75    * quadrant 1 section of circle.
76    */
77    for (i = 0; i < N-1; i++) {
78        x = a + i * size;
79    }
```
\begin{verbatim}
30 CONTENTS
90     y = f(x);
91     area += y * size;
92 }
93
94 /*
95  * A reduction is performed to aggregate the area estimates of all processes
96  * for each subsection of the area under the function. These are then added
97  * using the sum operator to get the total estimated area under the function
98  * representing quadrant 1.
99 */
100 #pragma paraguin reduce sum area overall_area
101
102 #pragma paraguin end_parallel
103
104 #ifndef PARAGUIN
105 overall_area = area;
106 #endif
107 /*
108 * Take a time stamp. This won’t happen until after the master
109 * process has gathered all the input from the other processes. 
110 */
111 gettimeofday(&tv2, NULL);
112 /
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=\tt%lf (seconds)\n", elapsed_time);
121 printf ("area = %lf\n", overall_area);
122 est_pi = overall_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.
\end{verbatim}
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);
35 printf("Number of processors = %i
",__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
gmtimeofday(&tv1, NULL);
46 #pragma paraguin begin_parallel
47 n=N;
48
49 */
50 procSize = (N-1)/__guin_NP;
51 if (2+procSize < (int) sqrt((double) N)) {
52 if (__guin_rank == 0){
53 printf ("Too many processors\n");
54 return 1;
55 }
56 }
57
58 /*
59 * The array is partitioned with each processor given a high value and a low value.
60 */
61 low_value = (__guin_rank*N-1)/__guin_NP;
62 high_value = ((__guin_rank+1)*N-1)/__guin_NP;
63
64 /*
65 * An array of N characters is allocated to serve as the means of indicating
66 * whether values in a given list are prime or not.
67 */
68 marked = malloc(N * sizeof(char));
69 temp = malloc(N * sizeof(char));
70 if(marked==NULL){
71 printf("Error allocating marked! \n");
72 return 1;
73 }
74
gmtimeofday(&tv2, NULL);
75 elapsed_time = (double) (tv2.tv_sec) - (double) (tv1.tv_sec);
76
gmtimeofday(&tv1, NULL);
77 elapsed_time = (double) (tv2.tv_sec) - (double) (tv1.tv_sec);
78 //
79}
for (i = 0; i < N; i++) {
  marked[i] = 1;
}

prime = 2;
/*
 * All primes are computed up to the square root of N.
 */
while(prime*prime<N) {
  /*
   * Compute the first multiples of prime greater than or equal to the low value.
   */
  if (low_value % prime == 0) {
    first = low_value;
  } else {
    first = low_value+prime - (low_value%prime);
  }

  /*
   * Mark the multiples of prime in each processors given range.
   */
  for (i=first+prime; i<high_value; i+=prime) {
    marked[i] = 0;
  }

  /*
   * The master process identifies the next prime value
   */
  if (__guin_rank == 0) {
    while (marked[prime]!=1) {
      prime++;
    }
  }

  /*
   * The newly computed prime is broadcast to all processes
   */
  #pragma paraguin bcast prime
}

/*
 * All partial values of primes are gathered into the array marked.
 */
#pragma paraguin gather marked ( n )
#pragma paraguin end_parallel

/*
 * Take a time stamp. This won’t happen until after the master
 * process has gathered all the input from the other processes.
 */
gettimeofday(&tv2, NULL);
elapsed_time = (tv2.tv_sec - tv1.tv_sec) +
(1000000*(tv2.tv_usec - tv1.tv_usec) / 1000000.0);

/*
 * Only the master should print this because the other processors only
 * have partial products. The list of all primes is printed as well as runtime.
 */
if(print) {
  printf("elapsed_time=%f (seconds)n", elapsed_time);
  if(print) {
    for (i=1;i<N;i++)
      if (marked[i])
        printf("prime = %d
",i);}
  #pragma paraguin begin_parallel
  free(marked);
  free(temp);
  #pragma paraguin end_parallel
  return 0;
}

2.12.4 Variable Documentation

2.12.4.1 int __guin_NP = 0

Definition at line 25 of file sieve_parallel.c.
2.13.4.2 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
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.
FILE *inFile, *oFile;
int grayImage[N][N], edgeImage[N][N];
char type[2];
tw, h, max;
tw, y, i, j, sum, sumx, sumy;
tw, GX[3][3], GY[3][3];
tw, elapsed_time;
struct timeval tv1, tv2;
tw, error = 0;
char buffer[BUFSIZ];

if (argc < 3) {
    fprintf(stderr, "Usage: %s target\n", argv[1]);
    error = -1;
}

if (!error) {
    inFile = fopen(argv[1], "r");
    if (inFile == 0) {
        fprintf(stderr, "Open failed\n");
        fprintf(stderr, "Usage: %s target\n", argv[1]);
        error = -1;
    } else {
        /*
        * Read the file type. Some graphics programs put comments in the
        * file and if we find a #, indicating a comment, we throw away
        * that specific line.
        */
        do
            fgets (buffer, BUFSIZ - 1, inFile);
        while (buffer[0] == '#');
        if (buffer[0] != '#')
            sscanf(buffer, "%s", type);
        else type[0] = '\0';

        /*
        * Read the height and width of the image file. If a comment is
        * found throw away that specific line.
        */
        do
            fgets (buffer, BUFSIZ - 1, inFile);
        while (buffer[0] == '#');
        if (buffer[0] != '#')
            sscanf(buffer, "%d%d", &w, &h);
        else w = h = max = 0;

        /*
        * Read in the pixels
        */
        for (i = 0; i < N; ++i) {
            for (j = 0; j < N; ++j) {
                fscanf(inFile, "%d", &grayImage[i][j]);
            }
        }
        fclose(inFile);
    }
}

#pragma paraguin begin_parallel

/*
 * The error is broadcast in a parallel region so that all processes receive
 * the error. This prevents a deadlock if an error arises because all processes
 */
#pragma paraguin end_parallel

* will receive the error and then exit.

*/
#pragma paraguin bcast error
if (error) return error;
*/
* This barrier is here so that a time stamp can be taken. This ensures all
* processes have reached the barrier before exiting the parallel region
* and then taking a time stamp.
*/
#pragma paraguin barrier
#pragma paraguin end_parallel

gettimeofday(&tv1, NULL);
#pragma paraguin begin_parallel

/*
* These are the 3x3 sobel masks for the image.
*/
GX[0][0] = -1; GX[0][1] = 0; GX[0][2] = 1;
GX[1][0] = -2; GX[1][1] = 0; GX[1][2] = 2;
GX[2][0] = -1; GX[2][1] = 0; GX[2][2] = 1;

GY[0][0] = 1; GY[0][1] = 2; GY[0][2] = 1;
GY[1][0] = 0; GY[1][1] = 0; GY[1][2] = 0;
GY[2][0] = -1; GY[2][1] = -2; GY[2][2] = -1;

*/
* The image that was read in as an array of pixels is broadcast to all
* processes as well as the dimensions of the image as width (w) and height (h)
*/
#pragma paraguin bcast grayImage w h

#pragma paraguin forall
for(x=0; x < N; ++x){
for(y=0; y < N; ++y){
  sumx = 0;
  sumy = 0;
  /*
  * This handles image boundaries
  */
  if(x==0 || x==(h-1) || y==0 || y==(w-1))
    sum = 0;
  else{
    /*
    * This is the x gradient approximation
    */
    for(i=-1; i<=1; i++)
      for(j=-1; j<=1; j++)
        sumx += (grayImage[x+i][y+j] * GX[i+1][j+1]);
    /*
    * This is the y gradient approximation
    */
    for(i=-1; i<=1; i++)
      for(j=-1; j<=1; j++)
        sumy += (grayImage[x+i][y+j] * GY[i+1][j+1]);
    /*
    * This is the gradient magnitude approximation
    */
    sum = (abs(sumx) + abs(sumy));
    edgeImage[x][y] = clamp(sum);
  }
}
#pragma paraguin gather edgeImage
#pragma paraguin end_parallel

gettimeofday(&tv2, NULL);
elapsed_time = (tv2.tv_sec - tv1.tv_sec) +
               ((tv2.tv_usec - tv1.tv_usec) / 1000000.0);

/*
* Only the master should print this because the other processors only
* have partial products. The array consisting of the masked image is printed
* to a new file specified by the user as an argument at execution.
*/
printf("elapsed_time=%f (seconds)\n", elapsed_time);
oFile = fopen(argv[2], "w");
if(oFile != 0){
This is a parallel implementation of a solution to the travelling salesman problem.

```c
#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.
2.14.3.2 void find_unique ( int perm[], int j )

Definition at line 290 of file tsp.c.

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

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

Definition at line 237 of file tsp.c.

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

```c
initialize (perm, n, i+1);
return 1;
#pragma paraguin end_parallel
```

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

Definition at line 277 of file tsp.c.

```c
{ int j;
#pragma paraguin begin_parallel
for (j = i; j < n; j++)
{ perm[j] = 0;
 find_unique (perm, j);
}
#pragma paraguin end_parallel
}
```

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

Definition at line 58 of file tsp.c.

```c
{ int i, j, k, N, p;
 int perm[MAX_NUM_CITIES], minPerm[MAX_NUM_CITIES+1];
 float D[MAX_NUM_CITIES][MAX_NUM_CITIES];
 float dist, minDist, finalMinDist;
 double elapsed_time;
 struct timeval tvi, tv2;
 int abort;

 abort = processArgs(argc, argv);
 if (!abort) {
 for (i = 0; i < n; i++) {
 D[i][i] = 0.0f;
 for (j = 0; j < i; j++) {
 fscanf (fd, "%f", &D[i][j]);
 D[j][i] = D[i][j];
 }
 }
 } else {
 if (n <= 1)
 printf ("0 0 0\n");
 else
 n = 0;
 }

#pragma paraguin begin_parallel
#pragma paraguin barrier
#pragma paraguin end_parallel
```
2.14 tsp.c File Reference

```c
* Take a time stamp.
*/
timeofday(&tv1, NULL);

#pragma paraguin begin_parallel
#pragma paraguin bcast debug n D

perm[0] = 0;
minDist = 9.0e10; // Near the largest value we can represent with a float

if (n == 2) {
    perm[1] = 1; // If n = 2, the N = 0, and we are done.
    minPerm[0] = perm[0]; minPerm[1] = perm[1];
    minDist = computeDist(D, n, perm);
}

/*
 * This computes N = (n-1)(n-2)
*/
N = n*n - 3*n + 2;

/*
 * This parallelizes the following loop by assigning iterations
 * of the loop to different partitions.
 */
#pragma paraguin forall
for (p = 0; p < N; p++) {
    if (debug && __guin_rank == 0 && p == 0) {
        printf (__guin_NP = %d
```

```c
if (debug) {
    printf ("pid %d: p = %d
```

```c
if (debug) {
    printf ("%4d", perm[i]);
}

```c
dist = computeDist(D, n, perm);

```c
if (debug) printf ("tDist = %f\n", dist);
```

```c
if (minDist < 0 || minDist > dist) {
    minDist = dist;
    for (i = 0; i < n; i++)
        minPerm[i] = perm[i];
}
```

```c
} while (increment(perm,n));
```

```c
myAnswer.minDist = minDist;
myAnswer.rank = __guin_rank;
```

```c
#pragma paraguin reduce minloc myAnswer resultAnswer
```

```c
/*
 * Send a message with the minimum Permutation from the rank that computed
 * it to the master process.
 */
#pragma paraguin message minPerm resultAnswer.rank 0
```

```c
if (__guin_rank == resultAnswer.rank) {
    if (debug) {
        printf ("Minimum dist = %f\n", minDist);
        for (i = 0; i < n; i++)
            printf ("%4d", minPerm[i]);
    } else {
        printf (" %f \n", minDist);
        for (i = 0; i < n; i++)
            printf ("\n", minPerm[i]);
    }
```

Generated on Wed May 7 2014 00:44:05 for Paraguin Compiler Examples by Doxygen
printf ("%d ", minPerm[i]);
    
    printf ("%d\n", minPerm[0]);
}
} #pragma paraguin end_parallel

gtimeofday(&tv2, NULL);

elapsed_time = ((tv2.tv_sec - tv1.tv_sec) * 1000.0) +
    ((tv2.tv_usec - tv1.tv_usec) / 1000.0);

if (debug) {
    printf ("Total time to compute =\%lf milliseconds\n", elapsed_time);
} else {
    int etime = (int) roundf (elapsed_time);
    printf ("%f milliseconds \n", elapsed_time);
}
}

2.14.3.6 void printUsage ( char * argv0 )

Definition at line 319 of file tsp.c.

fopen (stderr, "nUsage: %s \[options\] n \[DistanceMatrix\]n\*, argv0);
fprintf (stderr, "noptions are: \n\n-h, --help \n\nprint this help and quit \n\n\n-n \n\n-d, --debug \n\nprint debugging information (do not use with a large n) \n\n\n\nThe DistanceMatrix should be a lower triangular matrix of \n\ndistances between cities. For example, if n = 5, the DistanceMatrix \n\nmight look like: \n\n62.635856 \n25.893111 54.673987 \n38.751828 75.894124 22.376224 \n71.873977 50.518504 47.698433 55.902881 \n\nIf the DistanceMatrix is not provided, then it is read from stdin. \n\n\n\nThe output consists of n+3 integers printed to stdout. \n\ntime to derive the solution in milliseconds \n\nan approximate total distance of the circuit \n\n(rounded to the nearest integer) \n\na permutation of the first n integers starting a zero \n\n(\nzero indicates the originating city) \n\na zero (indicating a return to the originating city) \n\n\n\n\n2.14.3.7 int processArgs ( int argc, char * argv[]) 

Definition at line 352 of file tsp.c.

char *argv0;

argv0 = argv[0];

argc--; argv++;

while (argc > 0 && argv[0][0] == '-') {
    if (strcmp(argv[0], "-d") == 0 ||
        strcmp(argv[0], "--debug") == 0) {
        debug = 1;
    } else {
        printUsage(argv0);
        return 1;
    }
}

argc--; argv++;

Generated on Wed May 7 2014 00:44:05 for Paraguin Compiler Examples by Doxygen
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