6th Marathon of Parallel Programming

SBAC-PAD'11

October 27th, 2011.

Rules

For all problems, read carefully the input and output session. For all problems, a sequential implementation is given, and it is against the output of those implementations that the output of your programs will be compared to decide if your implementation is correct. You can modify the program in any way you see fit, except when the problem description states otherwise. You must upload a compressed file with your source code, the *Makefile* and an execution script. The program to execute should have the name of the problem. You can submit as many solutions to a problem as you want. Only the last submission will be considered. The *Makefile* must have the rule *all*, which will be used to compile your source code before submit. The execution script must follow the *Cluster Enterprise3* submitting rules – it will be inspected not to corrupt the target machine.

All teams have access to the target machine during the marathon. Your execution is queued by the Sun Grid Engine (SGE) and does not have concurrent process. During the marathon, SGE will be examined to find any suspect execution.

The execution time of your program will be measured running it with *time* program and taking the real CPU time given. Each program will be executed at least twice with the same input and only the smaller time will be taken into account. The sequential program given will be measured the same way. You will earn points in each problem, corresponding to the division of the sequential time by the time of your program (*speedup*). The team with the most points at the end of the marathon will be declared the winner.

This problem set contains 5 problems; pages are numbered from 1 to 12.

Problem A

Shellsort

Shellsort is an algorithm devised by Donald Shell in 1959. It is a generalization of the insertion sort. It does an *h-sort* across the whole array. It means that every *h*th element belongs to a new array that must be sorted. The last step of the algorithm is an ordinary insertion sort of the entire array ($h = 1$). Figure A.1 show part of this sort algorithm.

```
Original array:
[ 13 14 94 33 82 25 59 94 65 23 45 27 73 25 39 10 ]

      5-sort
13 14 94 33 82
25 59 94 65 23
45 27 73 25 39
10
      after
10 14 73 25 23
13 27 94 33 39
25 59 94 65 82
45

Partial ordered array:
[ 10 14 73 25 23 13 27 94 33 39 25 59 94 65 82 45 ]

      3-sort
10 14 73
25 23 13
27 94 33
39 25 59
94 65 82
45
      after
10 14 13
25 23 33
27 25 59
39 65 73
45 94 82
94

Partial ordered array:
[10 14 13 25 23 33 27 25 59 39 65 73 45 94 82 94 ]
```

Figure A.1 – Partial Shellsort with 5-sort and 3 sort.

The real problem of the Shellsort is choosing the values of *h*. Besides, it is difficult to evaluate and to prove their complexity while *h* decreases during the algorithm's iteration. All that is known is its basis complexity: $O(n^2)$.

All over the years, several authors suggested their own sequence based on experimental studies:

Author	Sequence
Donald Shell (1959)	$\lfloor N/2^k \rfloor$
Donald Knuth (1969)	$(3^k-1)/2$
Robert Sedgewick (1986)	$4^k+1 + 3\cdot 2^{k-1} + 1$
Marcin Ciura (2001)	$1, 4, 10, 23, 57, 132, 301, 701, \lfloor h_{k-1}\cdot 5/2 \rfloor+1$

Write a parallel program that uses a Shellsort algorithm to sort keys.

Input

The input file contains only one test case. The first line contains the total number of keys (N) to be sorted ($1 \le N \le 10^{10}$). The following lines contain N keys, each key in a separate line. A key is a seven-character string made up of printable characters (0x21 to 0x7E – ASCII) not including the space character (0x20 ASCII).

The input must be read from a file named *shellsort.in*

Output

The output file contains the sorted keys. Each key must be in a separate line.

The output must be written to a file named *shellsort.out*

Example

Input	Output for the input

```
11
SINAPAD
SbacPad
Wscad11
Sinapad
1234567
LADGRID
WEAC-11
CTDeWIC
sinaPAD
MPP2011
SINApad
```
	```
	1234567
	CTDeWIC
	LADGRID
	MPP2011
	SINAPAD
	SINApad
	SbacPad
	Sinapad
	WEAC-11
	Wscad11
	sinaPAD
	```

```c
void shell_sort_pass(char *a, int length, long int size, int interval)
{
        int i;
        for (i = 0; i < size; i++) {
                /* Insert a[i] into the sorted sublist */
                int j;

                char v[length];
                strcpy(v, a + i * length);

                for (j = i - interval; j >= 0; j -= interval) {
                        if (strcmp(a + j * length, v) <= 0)
                                break;
                        strcpy(a + (j + interval) * length, a + j *
length);
                }
                strcpy(a + (j + interval) * length, v);
        }
}

void shell_sort(char *a, int length, long int size) {
        int ciura_intervals[] = { 701, 301, 132, 57, 23, 10, 4, 1 };
        double extend_ciura_multiplier = 2.3;

        int interval_idx = 0;
        int interval = ciura_intervals[0];
        if (size > interval) {
                while (size > interval) {
                        interval_idx--;
                        interval = (int) (interval *
extend_ciura_multiplier);
                }
        } else {
                while (size < interval) {
                        interval_idx++;
                        interval = ciura_intervals[interval_idx];
                }
        }

        while (interval > 1) {
                interval_idx++;
                if (interval_idx >= 0) {
                        interval = ciura_intervals[interval_idx];
                } else {
                                interval = (int) (interval /
extend_ciura_multiplier);
                }

                shell_sort_pass(a, length, size, interval);
        }

}
```

Problem B

Leibniz's π

There is many ways to calculate the π. In 1682, Gregory–Leibniz proposed a simple formula to calculate it:

$$\pi = 4 \cdot \sum_{n=0}^{\infty} \frac{(-1)^n}{2n+1}$$

This formula is based on a Taylor series, considering that arccot(1)= π/4:

$$arc\cot(1) = \sum_{n=0}^{\infty} \frac{(-1)^n}{(2n+1)\cdot 1^{(2n+1)}} = \frac{\pi}{4}$$

Since π number has infinite decimal places, computational implementation reduces it to only some "trillion" places.

Write a parallel program that computes the π number.

Input

The input contains only one test case. The first line contains two integers: the number of terms in the series ($1 \leq N \leq 1000$) and the amount of decimal places ($1 \leq D \leq 2^{16}-2^4$).

The input must be read from a file named *pi.in*

Output

The output contains only one line printing the π number with exact D decimal places.

The output must be written to a file named *pi.out*

Example

Input	Output for the input
`80 100`	`3.1539378622726156250500586797491430543377733502638185261231945380388686496965829818612584875322139720`

```c
void pi(char* output, const long int n, const long int d) {
        long int digits[d + 11];;
        long int digit, i;
        int signal;
        long unsigned int remainder, div, mod;

        for(digit=0;digit<d+11;++digit) {
                digits[digit]=0;
        }

        signal = 1;
        for(i=0;i<=n;++i) {
                remainder = 4;
                for(digit=0;digit<d+11&&remainder;++digit) {
                        div=remainder/(2*i+1);
                        mod=remainder%(2*i+1);
                        digits[digit]+=(signal*div);
                        remainder=mod*10;
                }
                signal *= -1;
        }

        for(i=d+11-1;i>0;--i) {
                digits[i-1]+=digits[i]/10;
                digits[i]%=10;
                if(digits[i]<0) {
                        digits[i-1]--;
                        digits[i]+=10;
                }
        }

        if(digits[d+1]>=5) {
                ++digits[d];
        }

        for(i=d;i>0;--i) {
                digits[i-1]+=digits[i]/10;
                digits[i]%=10;
        }

        output[0] = digits[0]+'0';
        output[1]= '.';
        for(i=1;i<=d;i++)
        output[i+1] = digits[i]+'0';
```

```c
        output[d+2]=0;
}
```

Problem C

Mandelbrot

There is an interesting way to picture a set of points from a complex equation. Usually, its name is fractal. Professor Benoit Mandelbrot was the first person who used a computer to plot images and saw a visualization of the set in 1979, as shown on Figure C.1.

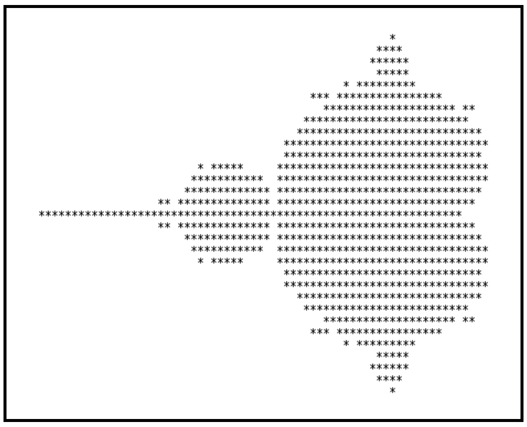

Figure C.1 – The first picture of the Mandelbrot set.

Write a parallel program that plot a Mandelbrot set.

Input

The input contains only one test case. The first line contains two integers: the width and the height of the image ($1 \leq W, H \leq 2^{13}-1$). The next line contains the radius from the center of the fractal ($0 < R \leq 2$). The last line contains one complex number representing the center of the fractal image – a complex number consists of a real part and an imaginary part ($-3 \leq A,B \leq 3$).

The input must be read from a file named mandelbrot.in

Output

The output contains a WxH fractal image in text format.

The output must be written to a file named mandelbrot.out

```
void mandelbrot::create() {
        if (!update)
                return;

        img = new char[width * height];;

        float minWH = width < height ? width : height;
        float scale = 2.0 * radius / minWH;
        float low_x = center.real() - width * scale / 2.0;
        float low_y = center.imag() - height * scale / 2.0;

        int d = 1;
        while (d < width || d < height)
                d <<= 1;
        do {
                for (int i = 0; i < width; i += d) {
                        for (int j = 0; j < height; j += d) {
                                if (i % (d << 1) == 0 && j % (d << 1) ==
0
                                                && (i != 0 || j != 0))
                                        continue;
                                char newColor = valueAt(low_x + i *
scale, low_y + j * scale);
                                if (*(img + (i * height) + j) == 0
                                                || newColor != *(img + (i *
height) + j)) {
                                        fill(i, height - j - d, d, d,
newColor);
                                        for (int s = 0; s < d; ++s)
                                                if (i + s < width)
                                                        for (int t = 0; t <
d; ++t)
                                                                if (j + t <
height)

        *(img + (height * (i + s)) + j + t) =

                newColor; //c[i + s][j + t] = newColor;
                                }
                        }
                }
                d /= 2;
        } while (d > 0);
```

```
        update = false;
}
```

Problem D

Permutation Flowshop Scheduling

Permutation Flowshop Scheduling (PFS) have as goal the deployment of an optimal schedule for *N* jobs on *M* machines. It is a NP-hard problem: *N!* possibilities.

Each n_i job ($1 \leq i \leq N$) must be schedule, in any time, on a m_j machine ($1 \leq j \leq M$). That is because a job has *M* operations and its j^{th} operation must be processed in m_j machine. So, one job can start on m_j machine if its m_{j-1} operation is completed and m_j machine is free. Each operation has its own time (t_j).

For PFS the operating sequence of the jobs are the same on every machine. That is the input job queue must be the same for all machines.

Solving the PFS problem means determining the permutation which gives the smallest makespan value to schedule *N* jobs on *M* machine.

For example, Table D.1 shows three jobs and their operation time for each machine.

Table D.1 – Operation time for each job.

Operation (t_j)	job 1	job 2	job 3
1	*1*	*2*	*1*
2	*1*	*1*	*1*
3	*1*	*1*	*2*
4	*1*	*1*	*1*

Figure D.1 shows a way to schedule these three jobs in four machines.

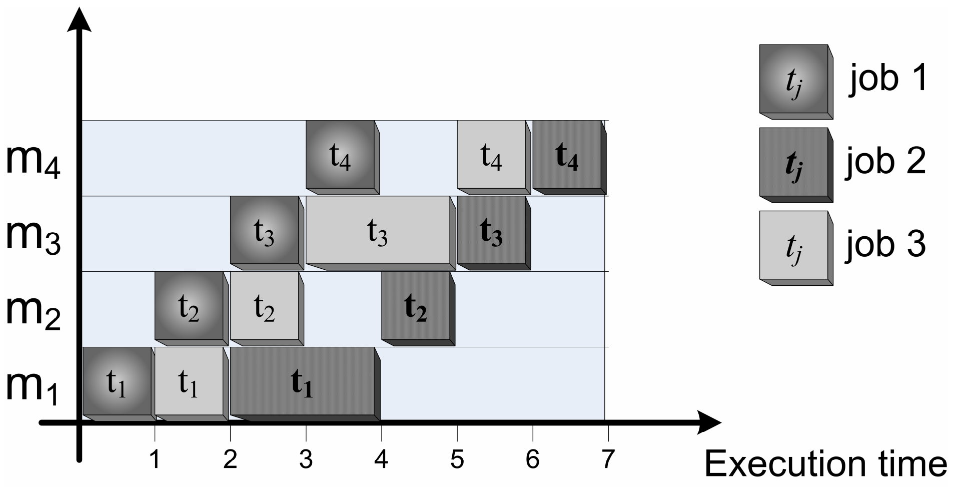

Figure D.1 – PFS problem for 3 jobs in 4 machines.

Write a parallel program that finds the smallest makespan for the PFS problem.

Input

The input contains several test cases. Each test case begins with two integers: the number of jobs ($0 \leq N \leq 100$) and the number of machines ($0 \leq M \leq 20$). The next N lines describe the n_i jobs. Each line has M integers separated by blank spaces. These numbers represent the t_j time that the m_j machine needs to process the j^{th} operation of the n_i job ($0 < t_j \leq 1000$). The test cases end with $N = 0$ and $M = 0$.

The input must be read from a file named pfs.in

Output

For each test case, the program must output an integer representing the smallest makespan for the PFS problem.

The output must be written to a file named pfs.out

Example

Input	Output for the input

1 4
1 1 1 1
2 3
1 1 1
1 1 1
3 4
1 1 1 1
2 1 1 1
1 1 2 1
0 0
	4
	4
	7

```
int main(int argc, char* argv[]) {
  ...
  while (1) {
    memset(tasks, 0, sizeof(tasks));

    fscanf(in, "%d%d", &n, &m);
    if (n == 0 || m == 0)
      break;
    for (i = 0; i < n; i++) {
      for (j = 0; j < m; j++)
        fscanf(in, "%d", &tasks[i].i[j]);
    }
    min_makespan = 0;

    // generate first sequence
    for (i = 0; i < n; i++)
      seq[i] = i;

    do {
      for (i = 0; i < m; i++)
        machines[i] = -1;
      makespan = 0;
      for (i = 0; i < n; i++) {
        memset(tasks[i].exec, 0, sizeof(tasks[i].exec));
        tasks[i].maq = 0;
      }

      cont_n = 0;

      // simulate permutation flow job schedule
      while (cont_n < n) {

        // schedule each task on a machine
        for (i = cont_n; i < n; i++) {
          if (machines[tasks[seq[i]].maq] < 0) {
            machines[tasks[seq[i]].maq] = seq[i];
          }
        }

        //run task
        for (i = 0; i < m; i++) {
          if (machines[i] >= 0) {
            tasks[machines[i]].exec[i]++;
            if (tasks[machines[i]].exec[i]
                >= tasks[machines[i]].i[i]) {
              // free a machine
              tasks[machines[i]].maq++;
              if (tasks[machines[i]].maq >= m)
                cont_n++;
              machines[i] = -1;
            }
          }
        }
        makespan++;
      }

      // calculate makespan
      if (!min_makespan || makespan < min_makespan)
        min_makespan = makespan;

      // generate another sequence
      cont_n = 1;
      while (cont_n) {
        i = n - 1;
        seq[i]++;
        while (seq[i] >= n) {
          for (j = i; j < n; j++)
            seq[j] = 0;
          if ((i - 1) >= 0)
            seq[--i]++;
        }
        cont_n = 0;
        for (i = 0; i < n && !cont_n; i++)
          for (j = i + 1; j < n && !cont_n; j++)
            if (seq[i] == seq[j])
              cont_n = 1;
      }
      cont_n = 1;
      for (i = 0; i < n - 1 && cont_n; i++)
        if (seq[i] > seq[i + 1])
          cont_n = 0;
      if (cont_n)
        break;
    } while (1);
    fprintf(out, "%d\n", min_makespan);
    fflush(out);
  }
  ...
```

Problem E

Minimum Spanning Tree[1]

A tree is a connect graph that contains no circle. The spanning tree of a connect graph is a subgraph that contains all the nodes of the original graph and a subset of just enough edges to constitute a tree.

The Minimum Spanning Tree (MST) is a spanning tree that has the minimal sum of all edge weight. The Prim's algorithm finds the MST in N^2 time, where N is the number of vertices in a graph.

Write a parallel program that finds the MST of an undirected graph.

Input

The input contains only one test case. The first line contains one integer that represents the number of vertices in the graph ($1 \leq N \leq 2^{13}$). The next N lines represent each i vertices. Each line contains N integers representing the weight between i and j ($0 \leq w_{i,j} \leq 2^{16}-1$) – 0 means that there is no connection.

The input must be read from a file named mst.in

Output

The output show an adjacent list of the MST, sorted in ascending order.

The output must be written to a file named mst.out

Example

Input	Output for the input

`5`
`0 0 0 3 5`
`0 0 0 7 0`
`0 0 0 2 0`
`3 7 2 0 1`
`5 0 0 1 0`
	`0 -> 3`
	`3 -> 1,2,4`

[1] *The Art of Concurrency*. Clay Breshears.O'Reilly, 2009.

```c
void prim(float **W, int **T, int N) {
  int i, j, k = 0;
  int *nearNode = (int*) calloc(N, sizeof(int));
  float *minDist = (float*) calloc(N, sizeof(float));
  float min;

  for (i = 1; i < N; i++) {
    nearNode[i] = 0;
    minDist[i] = W[i][0];
  }
  for (i = 0; i < N - 1; i++) {
    min = FLT_MAX;
    for (j = 1; j < N; j++) {
      if (0 <= minDist[j] && minDist[j] < min) {
        min = minDist[j];
        k = j;
      }
    }
    T[i][0] = nearNode[k];
    T[i][1] = k;
    minDist[k] = -1;
    for (j = 1; j < N; j++) {
      if (W[j][k] < minDist[j]) {
        minDist[j] = W[j][k];
        nearNode[j] = k;
      }
    }
  }
  free(nearNode);
  free(minDist);
}

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

  FILE *in, *out;
  int N;
  float **M;
  int **T;
  int i, j, k, f;

  in = fopen("mst.in", "r");
  out = fopen("mst.out", "w");

  fscanf(in, "%d", &N);
```

```c
  M = calloc(N, sizeof(float*));
  for (i = 0; i < N; i++) {
    M[i] = calloc(N, sizeof(float));
    for (j = 0; j < N; j++) {
      fscanf(in, "%f", &M[i][j]);
      if (M[i][j] == 0)
        M[i][j] = FLT_MAX;
    }
  }
  T = calloc(N, sizeof(int*));
  for (i = 0; i < N; i++) {
    T[i] = calloc(2, sizeof(int));
  }

  prim(M, T, N);

  for (i = 0; i < N; i++) {
    f = 0;
    for (j = 0; j < N; j++) {
      for (k = 0; k < N - 1; k++) {
        if (T[k][0] == i && T[k][1] == j) {
          if (f == 0) {
            fprintf(out, "%d -> %d", T[k][0], T[k][1]);
            f++;
          } else
            fprintf(out, ",%d", T[k][1]);
        }
      }
    }
    if (f)
      fprintf(out, "\n");
  }
  fflush(out);

  for (i = 0; i < N; i++) {
    free(M[i]);
    free(T[i]);
  }
  free(M); free(T);
  fclose(in); fclose(out);

  return EXIT_SUCCESS;
}
```