Mikhail Platonov platomik@fit.cvut.cz

Problem MAS: Magic square

Introduction. Task description.

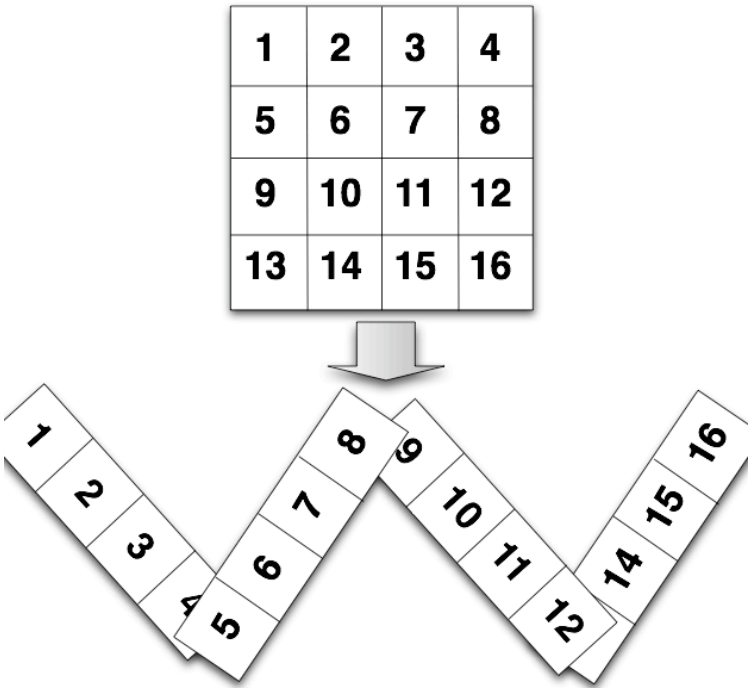
Task.
Find an arrangement of the numbers from set *N* in an *n x n* matrix, with each number occurring exactly once, and such that the sum of the entries of any row, any column, or any main diagonal is *M*. This arrangement is called a **magic square**.

Requirements.

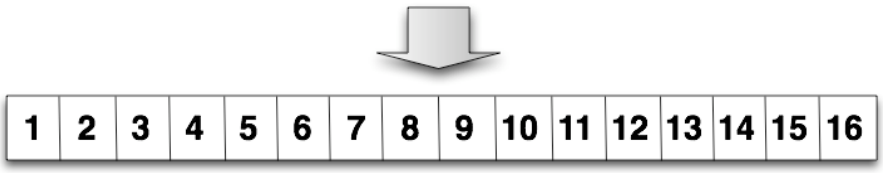
n = integer number, *n>3*
N = set of numbers *{1,..,n2}*
p = number of processors
M = $\frac{n(1+n^2)}{2}$

Sequential algorithm solution and implementation.

Let's assume that the magic square (matrix) is one-dimensional array. And then we will work with this array.



Mikhail Platonov platomik@fit.cvut.cz

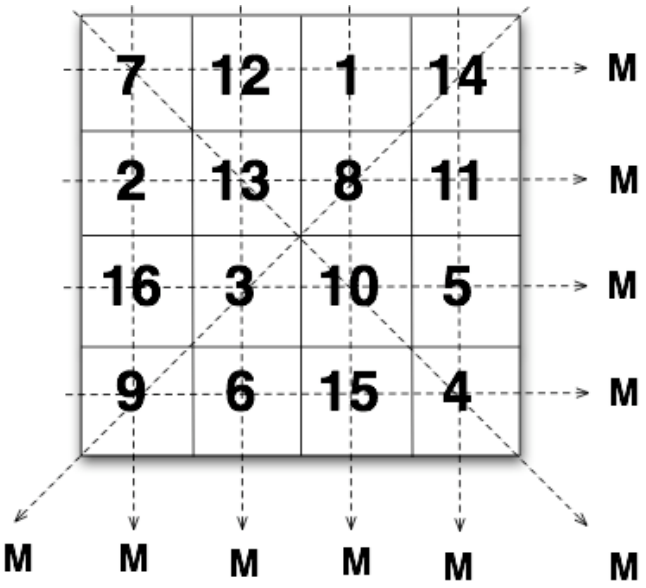


To find a solution we will use exhaustive search algorithm. As a result of this algorithm we will get the set of data which will be checked on magic square conditions.

Magic square conditions:

- The sum of elements in each row is equal to M, so that $M = \frac{n(n^2+1)}{2}$
- The sum of elements in each column is equal to M.
- The sum of elements in each of two main diagonals is equal to M.

Show these conditions on the plot.



$$M(n) = \frac{n(n^2+1)}{2}$$

As the main algorithm for getting a solution we will use anti-lexicographic permutation algorithm. We should generate permutations of a given sequence of values (one-dimensional array). The simple example of anti-lexicographic permutation is shown on plot below:

1 2 3
2 1 3
1 3 2
3 1 2
2 3 1
3 2 1

In sequential algorithm implementation program we generate initial data set - one-dimensional array of N elements (which is equal ***n x n*** matrix, and ***N=n*n;***). This array will be sent to anti-lexicographic function, that generate all permutation (***n!*** permutation totally). In the end each permutation will be checked on magic square condition.

Let's have a look at the parts of the sequential algorithm.

Part 1. Constant definition and library initialization. This part defines initial data for solving the task in 4x4 matrix.

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  #define n 4 //number of elements in row,column
5  #define N 16 //number of elements in square. it must be = rows*columns. N=n*n
6  #define M 34 // sum of elements in any row, col, diag M = n*(n^2+1)/2
7
```

Part 2. Main procedure. Here the one-dimensional array of 16 elements was generated. This array was sent to anti-lexicographic permutation algorithm.

```
83 void main()
84 {
85   int i, a[N], c=0;
86   // fill array [1,2,3..N]
87   for(i=0; i<N; i++)
88      a[i] = i+1;
89   ant(a,N-1,&c);
90 }
```

Part 3. The main part of program that finds all possible solution for a given initial array. Anti-lexicographic algorithm. As we see the algorithm consists of two sub procedures ***swap*** and ***reverse*** and one procedure ***ant*** that checks on the condition of receiving permutation.

```
 8 void swap(int *x, int *y){ //swap x for y and y for x
 9   int t;
10   t=*x;  *x=*y;  *y=t;
11 }
12
13 void reverse(int *a,int m){ //reverse first m elements
14    int i=0, j=m;
15    while(i<j){
16      swap(&a[i], &a[j]);
17      ++i;  --j;
18    }
19 }
```

```
21 void ant(int *a,int m, int *c){ //antilexicographic permutations
22   int i,row,col,sum;
23 
24   if(m==0){//permutation is ready
25   //.............................//
26     // here are conditions for checking array on magic square requirements
27   //.............................//
28     }else{
29     for(i=0; i<=m; i++){
30       ant(a,m-1,c);
31       if(i<m){
32         swap(&a[i], &a[m]);
33         reverse(a,m-1);
34       }
35     }
36   }
37 }
```

Part 4. When permutation is ready we check whether it satisfies the properties of magic square. After that the array is printed if the result is successful or exit if the first array was found.

```
25     *c=*c+1;
26         //check magic in rows
27         for (col=0; col<n; col++){
28                 sum=0;
29                 for (row=col*n; row<col*n+n; row++) {sum+=a[row];}
30                 if (sum != M) break;
31         }
32 
33         if (sum==M) {
34                 //check magic in cols
35                 for (row=0; row<n; row++) {
36                         sum=0;
37                         for (col=row; col<N; col+=n) {sum+=a[col];}
38                         if (sum != M) break;
39                 }
40                 if (sum==M) {
41                         //check magic in 1 diag
42                         sum=0;
43                         for (row=0; row<N; row=row+n+1) {sum+=a[row];}
44                         //check magic in 2 diag
45                         if (sum==M)     {
46                                 sum=0;
47                                 for (i=n-1; i<N-1; i+=n-1) {sum+=a[i];}
48                                 if (sum==M){
49                                         //print square
50                                     for(i=0; i<N; i++){printf("%d ",a[i]);
51                                         if ((i+1)%n==0) printf("\n");
52                                     }
53                                     //EXIT!
54                                     exit(EXIT_SUCCESS);
55                                 }
56                         }
57                 }
58         }
```

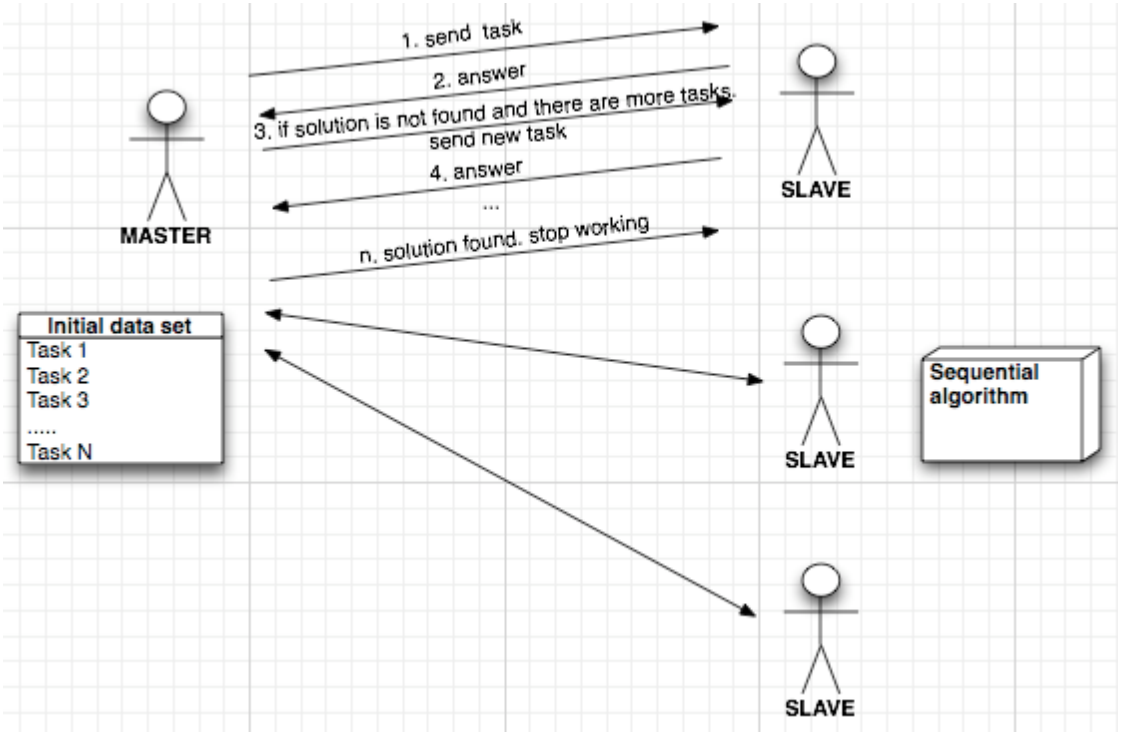
Mikhail Platonov platomik@fit.cvut.cz

Parallel algorithm.

We should divide the task of implementing the parallel algorithm into several steps. On the first step we generate initial data set which is then sent to SLAVE processes on the next step. SLAVE processes realize the same algorithm as in sequential part. When SLAVE process receives the task, It will solve it and send the solution back.

MASTER checks whether the solution is satisfactory or not, then MASTER sends new data to SLAVE process. And all is repeated until the first successful solution is found.

Logical schema of interaction in Parallel algorithm:



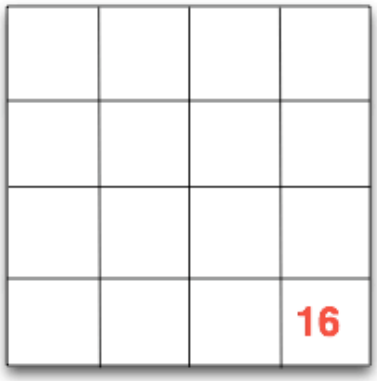
This parallel algorithm is of MASTER-SLAVE type and should be implemented using MPI communication library on the computing cluster star.fit.cvut.cz.

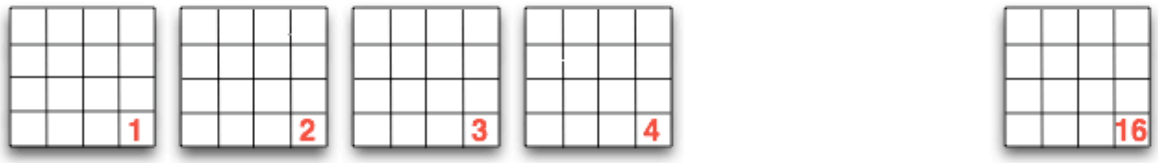
Parallel algorithm implementation on star.fit.cvut.cz

Initial data set

Let's fix the last element in one-dimensional array of N elements. In this case we can generate N independent tasks for future solving on p processes.

For matrix 4x4 of N=16 element:





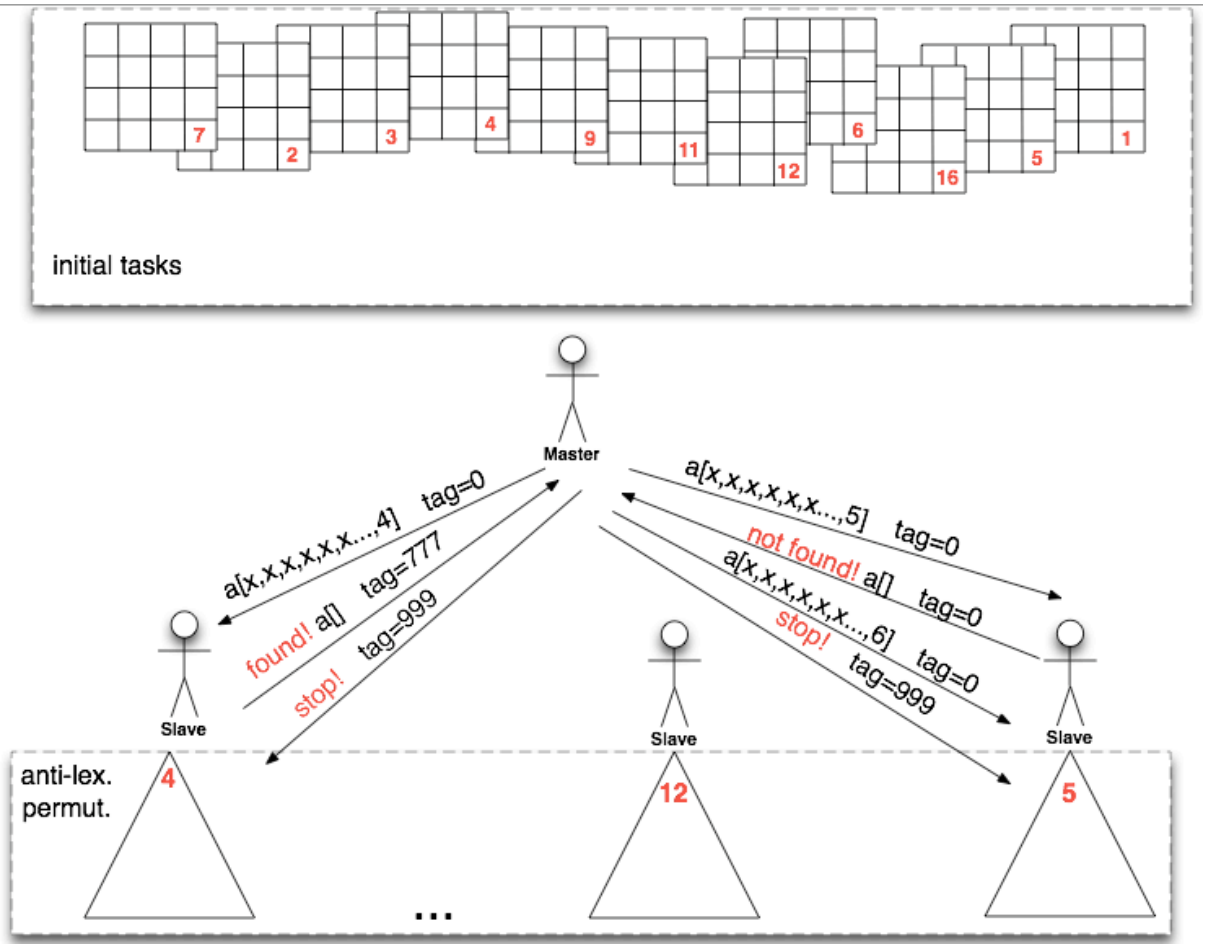
Master and Slave in MPI terms.

When we develop program on MPI communication library we should distinguish the tasks for MASTER from tasks for SLAVE. There can be only one MASTER. It's the main process that sends/receives information to/from other process. MASTER process has always **rank=0.** The number of SLAVE processes depends on the parameters of the calling program. SLAVE processes implement sequential algorithm for searching a solution. SLAVE process receives information from MASTER and sends information to MASTER. All information is sent in messages.

Message consists of several fields: data (address, number of elements, type), destination, tag and communicator. The most important field for interconnection between Master and Slave and managing is *tag* field.

We denote ***tag = 777*** if SLAVE has found a successful solution and sends the result to MASTER. **tag = 999** when MASTER is sending stop signal to all SLAVE processes. In other cases MASTER/SLAVE should examine data fields in the message.

For our task we can produce the following logical schema.



Algorithm for Slave

```
96   if (my_rank != 0) {
97     /* SLAVE */
98         for (i=1;i<=N;i++)   { //estimate max number of tasks.
99              //receive task
100             MPI_Recv (a, N, MPI_INT, MPI_ANY_SOURCE, MPI_ANY_TAG, MPI_COMM_WORLD, &status);
101
102             if (status.MPI_TAG==999){break;} /* Receive STOP tag. Stop working.*/
103             ant(a,N-2);
104             // send to MASTER
105             dest=0;
106             MPI_Send (a, N, MPI_INT, dest, tag, MPI_COMM_WORLD);
107         }
108
109
110  }
```

In this part of program SLAVE process receives a task from MASTER and tries to find a solution using anti-lexicographic algorithm from sequential part of project. But in parallel program we use the anti-lexicographic algorithm for permutation N-1 elements (it was permutation of N elements for sequential program). It means that we have N initial tasks and can send to N SLAVE processes simultaneously.

SLAVE process checks the tag of incoming message. If tag = 999, it means other SLAVE process have found a successful solution and MASTER sends the stop signal to all SLAVE processes. SLAVE stops working.

If the current SLAVE has found a successful solution then SLAVE sends a message to MASTER with tag=777 that contains one-dimensional array.

Algorithm for Master.

In the first part of algorithm for MASTER process the initial tasks have been generated and sent to *p* SLAVE processes.

```
112     /* MASTER */
113     printf("There are %d tasks.\n", N);
114     printf("There are %d slave processes.\n", p-1);
115     t0 = MPI_Wtime(); // start timer
116     
117     /* send initial data for all processes*/
118     for (i=1;i<p;i++) {
119       if (i>N) {break;}     /* if number of processes more than number of tasks*/
120       arr=i; /* how many elements already sent*/
121       printf ("process %d ",i);
122       printf ("get task to make permutation for a[1..N]=[");
123 
124         /*moving array clockwise*/
125           for(j=0; j<N; j++) {
126                a[j] = (i+j)%N+1;
127                if (j==N-1){printf ("%d",a[j]);}else{printf ("%d, ",a[j]);}
128           }
129       printf ("]\n");
130 
131       // send task to slave
132       MPI_Send(a, N, MPI_INT, i, tag, MPI_COMM_WORLD);
133     }
```

In the next part of algorithm the MASTER process is waiting for the answer from SLAVE processes after sending initial tasks. If SLAVE process answer and successful solution was not found. MASTER sends a new task to SLAVE process. If SLAVE process has found a solution (tag=777 in receiving message) - MASTER stops the work of all SLAVE processes by sending message with tag=999.

MASTER process should print out the successful solution - one-dimensional array. Tasks are sent until the first solution has been found or the task list become empty.

```
135     /* receive initial data end send next portion*/
136     for (source=1;source<=N;) {
137       /* checking if message has arrived */
138       MPI_Iprobe(MPI_ANY_SOURCE, MPI_ANY_TAG, MPI_COMM_WORLD, &flag, &status);
139       /* */
140       if (flag) {
141         // slave solved task
142         MPI_Recv(a, N, MPI_INT, MPI_ANY_SOURCE, MPI_ANY_TAG, MPI_COMM_WORLD, &status);
143         t1 = MPI_Wtime();
144
145          /* found */
146         if (status.MPI_TAG==777){
147             printf ("\t Found after %f seconds from process %d answered ", t1-t0, status.MPI_SOURCE);
148
149             /*print result*/
150               for(j=0; j<N; j++) { printf ("%d ",a[j]);}
151               printf ("\n");
152             /* send STOP to all SLAVES*/
153 //              for (i=1;i<p;i++) { MPI_Send(&i, 1, MPI_INT, i, 999, MPI_COMM_WORLD);}
154         }
155
156         /* if not the end of tasks */
157         if (arr<N){
158             arr++; /* increase number tasks already sent*/
159               printf ("process %d ",status.MPI_SOURCE);
160               printf ("get task to make permutation for a[1..N]=[");
161                 |
162               /*moving array clockwise*/
163                 for(j=0; j<N; j++) {
164                      a[j] = (arr+j)%N+1;
165                      if (j==N-1){printf ("%d",a[j]);}else{printf ("%d, ",a[j]);}
166                 }
167               printf ("]\n");
168
169               // send new task to slave
170               MPI_Send(a, N, MPI_INT, status.MPI_SOURCE, tag, MPI_COMM_WORLD);
171         }else{
172             /* end of tasks send STOP to all SLAVES*/
173             for (i=1;i<p;i++) {
174                 MPI_Send(&i, 1, MPI_INT, i, 999, MPI_COMM_WORLD);
175             }
176         }
177         source++;
178       }
179     }
```

Parallel program should be correctly finalized.

```
183   /* shut down MPI */
184   MPI_Finalize();
```

Measurement

For estimate of the time for program execution with different parameters (initial data set, number of slave processes) and then drawing graph plot we should use MPI_Wtime() function.

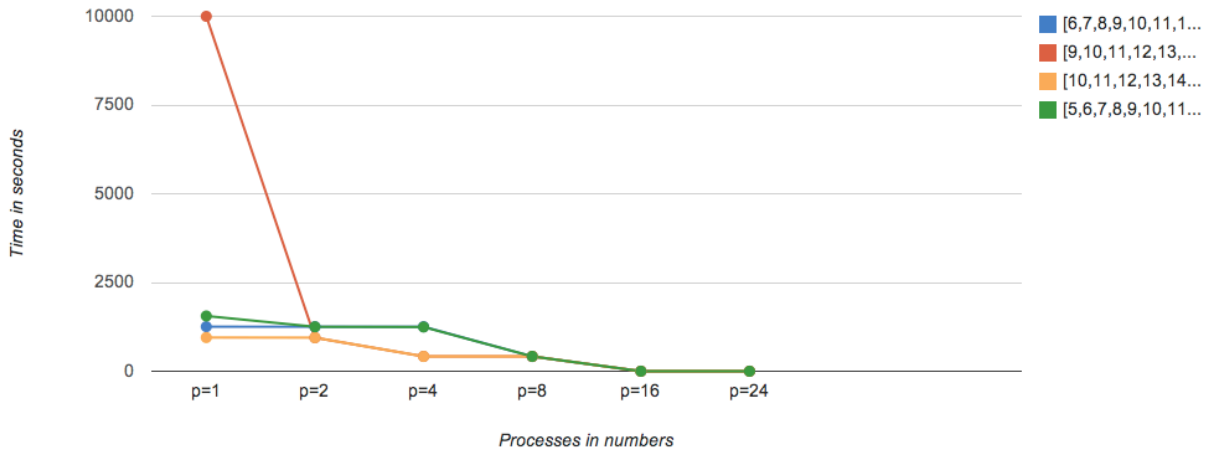
At the beginning of program we store the result of function MPI_Wtime() in variable t0. When solution is found we call MPI_Wtime() function again and save the result in variable t1. The difference between these two variables is equal to the time in seconds for finding a successful solution.

```
110 ....
111
112     t0 = MPI_Wtime(); // start timer
113
114 ....
115
116     t1 = MPI_Wtime();
117     printf ("\t Found after %f seconds from process %d answered ", t1-t0, status.MPI_SOURCE);
118 ....
```

Filling the table of measurements for matrix 4x4 with 16 elements from 1 to 16. For 5 different data we have next results on several processes (p=1,2,4,16,24).

	p=1	p=2	p=4	p=8	p=16	p=24
[6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 1, 2, 3, 4, 5]	1256	1254	1255	417	1	1
[9, 10, 11, 12, 13, 14, 15, 16, 1, 2, 3, 4, 5, 6, 7, 8]		946	419	419	1	1
[10, 11, 12, 13, 14, 15, 16, 1, 2, 3, 4, 5, 6, 7, 8, 9]	950	947	418	418	1	1
[13, 14, 15, 16, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12]	1464	1254	1253	1	1	1
[5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 1, 2, 3, 4]	1557	1253	1247	419	1	1

Drawing a graph according to this table.



It is obvious that for the second data set there is no time for solving the task on p=1 process. It means cluster stopped the program execution before the first solution was

found.

Tests have been made for each data set (16) but only 5 were chosen because other 11 are almost similar and do not contribute any results to the graph.

Performance of parallel solution.

Speedup refers to how much a parallel algorithm is faster than a corresponding sequential algorithm. Speedup is defined by the following formula :

$$S_p(n) = T_1(n) / T_p(n),$$

where *p* - number of processes, *T1* is the execution time of sequential algorithm, *Tp* is the excution time of parallel algorithm with *p* processes, *n* - depends on initial data set.

Speedup for initial data set (array[5,6,7,8,9,10,11,12,13,14,15,16,1,2,3,4]):

$$S_2 = \frac{1557}{1253} = 1{,}24$$

$$S_4 = \frac{1557}{1247} = 1{,}25$$

$$S_8 = \frac{1557}{419} = 3{,}7$$

Efficiency is a performance metric defined as:

$$E_p(n) = T_1(n) / (pT_p(n)) = S_p(n) / p$$

It is a value, typically between zero and one, estimating how well-utilized the processors are in solving the problem, compared to how much effort is wasted in communication and synchronization.

$$E_2 = \frac{1{,}24}{2} = 0{,}62$$

$$E_4 = \frac{1{,}25}{4} = 0{,}31$$

$$E_8 = \frac{3{,}7}{8} = 0{,}46$$

Useful could be estimation of ***cost*** parallel algorithm

$$C_p = pT_p$$

$$C_1 = 1 * 1557 = 1557$$

$$C_2 = 2 * 1253 = 1506$$

$$C_4 = 4 * 1247 = 4988$$

$$C_8 = 8 * 419 = 3352$$

Scalability analysis

Total overhead is the amount of time required to coordinate parallel tasks, as opposed to doing useful work. Total overhead can include factors such as: task start-up time; synchronizations; data communications; software overhead imposed by parallel compilers, libraries, tools, operating system, etc.; task termination time.

$$T_0 = pT_p - T_1 .$$

for 2 processes $T_0 = 2 * 1253 - 1557 = 949$

for 4 processes $T_0 = 4 * 1247 - 1557 = 3431$

for 8 processes $T_0 = 2 * 419 - 1557 = 1795$

Mikhail Platonov platomik@fit.cvut.cz

Review

During solving the Problem Magic square the sequential algorithm was developed. Then sequential algorithm was rewritten above into a parallel one and was implemented on the computing cluster STAR.

A series of time complexity measurements of the parallel algorithm on various numbers of processors was conducted. Tests have been made for matrix 4x4. The results have been filled into the table upon which the graph of measurements was drawn.

Analysis of efficiency and scalability of the parallel program was performed. Speedup, efficiency and cost of parallel algorithm was calculated for one initial set. Moreover scalability analysis was made with estimating parallel overhead parameter.