American University of Armenia

College of Engineering

Computer and Information Science Program

Master's Project

Parallel Implementation of Master Mind and Duty Scheduling

Student:	Sergey Jilavyan
Supervisor:	Dr. S. Khachatryan
Referee:	Dr. Rajendra Akerkar

Yerevan
2005

Table of contents

Abstract

We program two problems that produce human-like “decision making” outcomes. The first one is an implementation of a well-known game Master Mind. The program must deduce a number secretly kept by a user. The result is achieved through interactive interface and corresponding “thinking”. Expectedly, the computer always beats the user in the sense that, on average, it finds the number much faster than the human being when the roles are reversed.

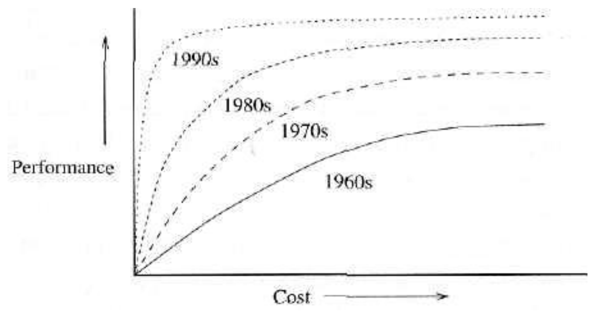
The second problem is daily scheduling for taking some duties. There are given number **n** persons, and every day **m** of them must be on the duties. It is required to collect groups of all possible combinations, and let all the people to have at least **k** day offs.

We offer an approach that solves these two different problems. In both cases, however, lack of resources of a single computer becomes apparent. It becomes impossible to play Master Mind with 7-digit numbers, or takes days to generate a schedule for 20 people to be divided into groups of 10.

The algorithm has been parallelized and the programs have been executed in the AUA educational computer labs. It is shown that the University has sufficient resources to conduct large-scale computations and use the computers in the LAN-operating labs as a sort of a heterogeneous cluster.

1. Introduction

There are not that many countries running supercomputers – highly powerful but hugely expensive computing facilities. For particular problems, however, it is enough to assemble usual PCs into a network and, thus, create a cluster of capabilities close to those of supercomputers, but essentially cheaper. Dependence of computer performance on the cost is shown on the picture below for several decades [1]:



Among cheap computers sharp growing behavior is observed. After some point, however, the curves reach their respective saturation levels. It means that only slight improvement in the performance is possible, but at considerable costs. It is also evident that now very fast and pretty cheap processors can be produced.

Any reasonable efficiency may be achieved only on careful program parallelization. The latter includes such aspects as parallel architectures, algorithms and programming, run-time environment, etc. It is not surprising that the parallel computing is considered among the most advanced fields in Computer and Information Science. Needs in it mainly come from simulations of complex systems and so called "Grand Challenge" problems. Its application areas include:

- Fundamental Physics and Astrophysics
- Chemical and Nuclear Reactions
- Weather and Climate
- Artificial Intelligence
- Mechanical Devices
- Electronic Circuits
- Manufacturing Processes

- Geological Activity and Seismology
- Biological, Human Genome

Undoubtely, acceleration in problem solutions is the most obvious advantage of parallel computing. For example, weather forcast for tomorrow may require several days, if running on a single processor. There is no other choice but to do that in parallel, in order to produce up-to-date results.

But fast solutions are not the only advantages of parallel computers. Dealing with large and complex problems that require too huge data for a single computer memory is also an important issue, to be solved by utilization of data parallelizm.

The main objective of the current work, together with two more concurrently taken CIS 294 projects, is to study and implement the parallel computing in a common LAN-operating environment, particularly, AUA. In order to turn the set of existing personal computers into a computing unit, we should fulfill the following:

1. Setup parallel programming software for the existing hardware;
2. Construct parallel algorithms, develop the corresponding data structures for our problems and work out communications between the sub problems;
3. Distribute the sub problems among the computers.

The AUA software and hardware resources allow direct implementation of parallel computing through RMI, first introduced in JDK 1.1, and / or MPI. There are solid grounds for using Java for high-performance parallel applications. Its clean and type-safe object-oriented programming model and its support for concurrency make it an attractive environment for writing large-scale parallel programs. For shared memory machines, Java offers a common multithreading platform. For distributed memory machines, such as clusters of computers, Java provides Remote Method Invocation (RMI), which is an object-oriented version of Remote

Procedure Call (RPC). Being easy to use, RMI is a powerful technology in the area of distributed object computing. It offers many advantages for distributed programming, including integration with Java object modeland platform-independence [11].

The main objective of the project is to initiate parallel computing in the AUA computer labs. Interconnected by LAN, the computers there are operating under MS Windows 2000 / XP operating systems and communicating through TCP / IP protocol. Such setup makes it possible to explore Message Passing Interface (MPI), particularly MPICH under MS Windows. The programs are coded in C / C++.

The report consists of the following chapters: Chapter 2 The Concept of Parallel Computing introduces the notions of parallel architecture and message passing; Chapter 3 Model Problems describes in details the model problems and the algorithms of their both serial and parallel implementations; Chapter 4 discusses the performed tests and obtained results; Chapter 5 enlists the conclusions and directions for the future work.

2. The Concept of Parallel Computing

Parallel computing can only be used efficiently only on clear definition and full understanding of all its technical issues. They include design of parallel computers and clusters, parallel algorithms and languages, parallel programming tools and environment, performance evaluation, parallel applications. Parallel computers should be designed so that they can be scaled up to a large number of processors and are capable of supporting fast communication and data sharing among processors. The resource of a parallel computer will not be used efficiently unless there are available parallel algorithms, languages, and methods to evaluate the performance. As shown below, convenient parallel programming tools and environment are very important in development of parallel programs [1].

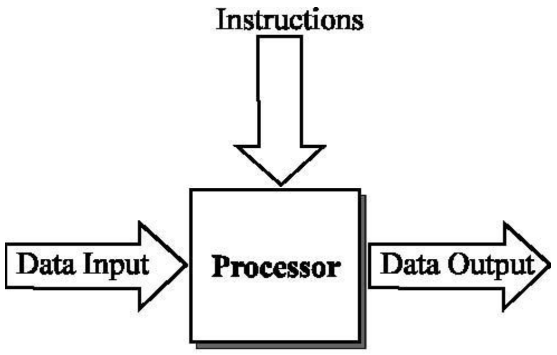
The core elements of parallel processing are CPUs. Based on a number of instruction and data streams that can be processed simultaneously, computer systems are classified into the following four categories:

1. Single Instruction Single Data (SISD)
2. Single Instruction Multiple Data (SIMD)
3. Multiple Instruction Single Data (MISD)
4. Multiple Instruction Multiple Data (MIMD)

In the explanations below we use diagrams from [1].

Single Instruction Single Data (SISD)

A SISD computing system is a uniprocessor machine capable of executing a single instruction which operates on a single data stream.



In SISD machine instructions are processed sequentially and, therefore, computers adopting this model are usually called sequential computers. The execution order of simple arithmetic operations is shown below:

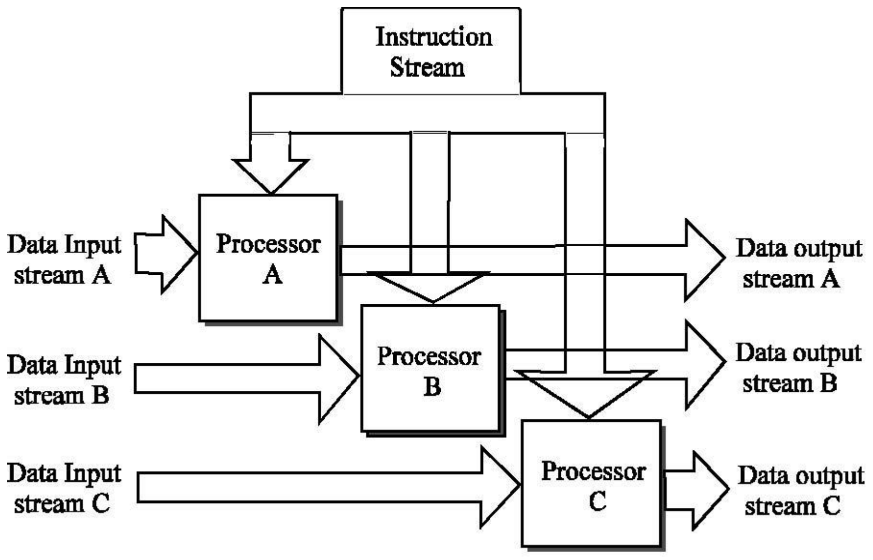
time →

LOAD A → LOAD B → C = A + B → STORE C → A = B * 2 → STORE A

Most of conventional computers are built using SISD model. All the instructions and data to be processed have to be stored in the primary memory. The speed of processing element in SISD model is limited by the rate at which computer can transfer information internally. Dominant representative SISD systems are IBMPC, Macintosh, Workstations, etc.

Single Instruction Multiple Data (SIMD)

A SIMD computing system is a multiprocessor machine capable of executing the same instruction on all the CPUs, but operating on different data streams.



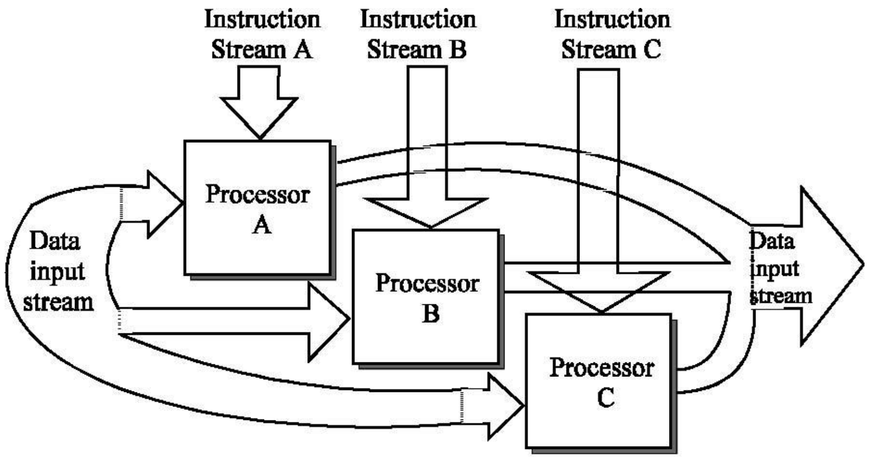
Machines based on SIMD model are well matched for scientific computing since they involve lots on vector and matrix operations. This is a type of parallel computers. In the SIMD architecture all processing units execute the same instruction at any given time and each processing unit operates on its own data segment. This type of machine typically has an instruction dispatcher, a very high-bandwidth internal network, and a very large array of very small-capacity instruction units. Our project is realized by this type of architecture of parallel computing.

time →

Process 1:	**LOAD A(1) → LOAD B(1) → C(1)=A(1)+B(1) → STORE C(1)**
Process 2:	**LOAD A(2) → LOAD B(2) → C(2)=A(2)+B(2) → STORE C(2)**
Process 3:	**LOAD A(3) → LOAD B(3) → C(3)=A(3)+B(3) → STORE C(3)**

Multiple Instruction Single Data (MISD)

A MISD computing system is a multiprocessor machine capable of executing different instructions on different processing elements, but all of them operating on the same data-set . Statements perform different operations on the same data set. Machines built using MISD model, are not useful in most of the applications; a few machines are built, but none of them are available commercially. They become more of an intellectual than practical interest.



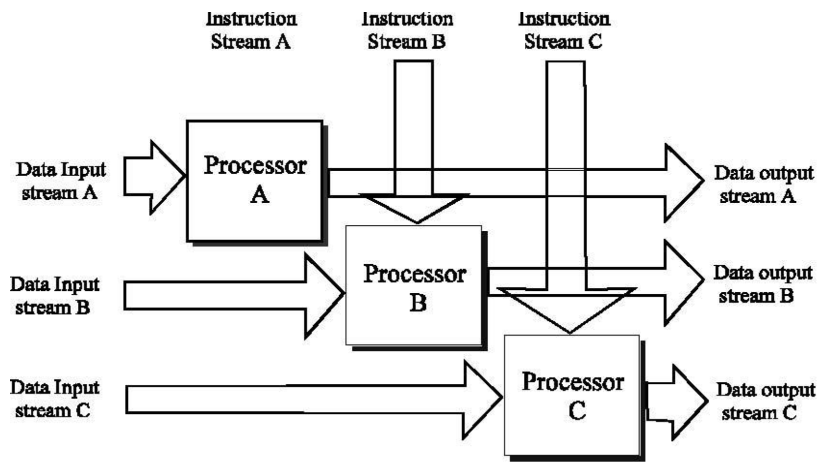
Multiple Instruction Multiple Data (MIMD)

A MIMD computing system is a multiprocessor machine capable of executing multiple instructions on multiple data sets. Each processing elements in MIMD model have separate instruction and data stream and hence machines built using this model are well suited for any kind of applications. This is the most general type of parallel computer. Every processor in MIMD may execute a different instruction stream. Every processor may work with a different data stream.

time →

Process 1: **LOAD A(1) → LOAD B(1) → C(1)=A(1)+B(1) → STORE C(1)**
Process 2: **CALL FUNCA( ) → X = X + 1 → SUM = X + Y → CALL FUNCB( )**
… **…**
Process n: **DO 10 i=1,N → Z = Z*K → C(n)=Z → CONTINUE**

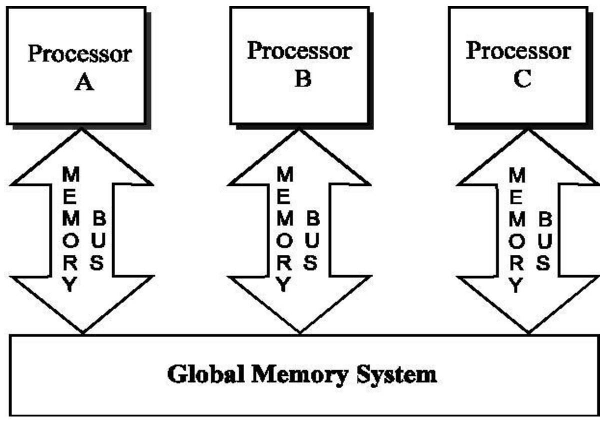
Unlike SIMD and MISD machines, processing elements in MIMD machines work asynchronously.



MIMD machines are broadly categorized into shared-memory MIMD and distributed-memory MIMD machines based on how all the processing elements are coupled to the main memory.

Shared Memory MIMD Machine

In shared memory MIMD model, all the processing elements are connected to a single global memory; all the processing elements have access to this global memory. The communication between processing elements in this model, is done through the shared memory – modification of the data stored in global memory by one PE is visible to all other processing elements. Dominant representative shared-memory MIMD systems are Silicon Graphics machines, Sun.s SMP.s (Symmetric Multi-Processing), BARC.s Anupam, etc.



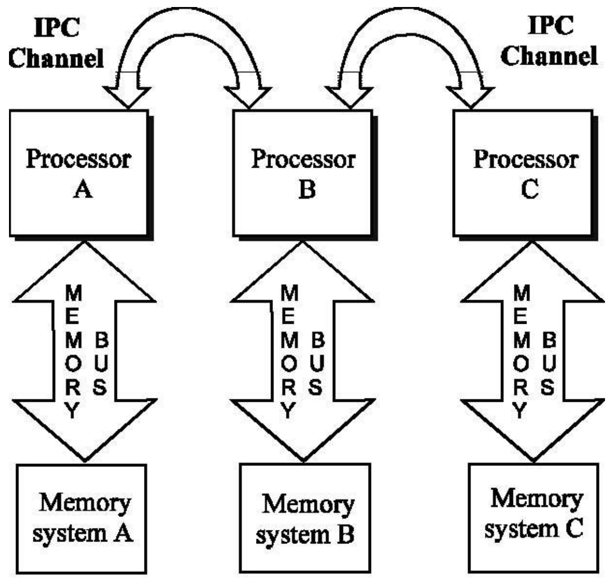
The following are the characteristics of shared-memory MIMD machines:

- **Manufacturability:** Easy to build – well established conventional operating systems can be easily adapted.
- **Programmability:** Easy to program and it does not involve much communication traffic during the communication between the programs executing simultaneously.
- **Reliability:** Failure of a memory component or any processing element affects the whole system.
- **Extensibility and Scalability:** Adding more processing elements to the existing system is very difficult, because it leads to memory contention.

Distributed Memory MIMD machine

In distributed memory MIMD model, all the processing elements have their own local memory. The communication between processing elements in this model, takes place through the interconnection network. The network connecting processing elements can be configured to tree, mesh, cube, etc.

The processors operate asynchronously and if communication/synchronization among tasks is necessary, they can do so by communicating messages between them. Dominant representative distributedmemory MIMD systems are C-DAC.s PARAM, IBM.s SP/2, Intel.s Paragaon, etc.



The following are the characteristics of distributed memory MIMD machines:

- **Manufacturability:** Easy to build, but it requires operating systems consuming little system resources.
- **Programmability:** Slightly difficult when compared to shared-memory, but it is well suited for real-time applications.
- **Reliability:** Failure of any component will not affect the entire system; since any PE can be easily isolated.
- **Extensibility and Scalability:** Adding more processing elements to the existing system is much easier.

The shared-memory MIMD model, is easy to program but suffers from extensibility and scalability and the distributed memory MIMD model difficult to program but it can easily be scaled and hence, such systems are popularly called massively parallel processing (MPP) systems [2, 7].

MPI-MPICH

The Message Passing Interface (MPI) specification is widely used for solving significant scientific and engineering problems on parallel computers. There are many implementations for different computer platforms ranging from IBM SP-2 supercomputers to clusters of PCs running Windows NT or Linux ("Beowulf" machines). The new version of the initial MPI Standard document – MPI-2, contains both significant enhancements to the existing MPI core and new features. The new extensions to include parallel I/O, remote memory access operations, and dynamic process management [3].

MPICH is a freely available, portable implementation of MPI library to be used, among different options, with MS Visual C++ 6 [6]. A more detailed presentation of MPI / MPICH and the commands is given in [9].

3. Model Problems

Problem 1 – The Game of Master Mind

Master Mind is a popular game not difficult to program. In the original version it deals with colors. It is played by two players: one – the keeper, secretly constructs a sequence of 4 – 6 colors, while the counterpart – the discoverer, tries to guess that sequence. At every move the discoverer suggests his /her version, and the keeper grades it using black and white stones. The number of the black stones denotes how many correct colors at their correct places are, while the number of the white stones denote how many colors from the trial are included in the original sequence, but are at wrong places. For example, if the keeper's sequence is RED, BLUE, ORANGE and WHITE, then the following couple of moves are possible:

The keeper's original sequence		RED	BLUE	ORANGE	WHITE		
	Move #					"Black" grade	"White" grade
	1	RED	GREEN	WHITE	BLUE	1	2
	2	BLUE	GREEN	RED	BROWN	0	2

There is only one correctly guessed color in the first trial – RED. Therefore, the "Black" grade is 1. Meantime, there are two more colors in the original sequence that appear in wrong positions in the trial – WHITE and BLUE. So, the "White" grade is 2. Based on these grades, the discoverer may assume, for example, that GREEN is included and is at its correct place, while the places of RED and BLUE must be changed. By incorporating a new color – BROWN, (s)he suggests a new trial, direct examination of which leads to 0 and 2 for the "Black" and "White" grades respectively.

Understandably, the rules require to distinctly specify at the beginning the set of all possible colors involved in the game. The difficulty level directly depends on the length of a color sequence to be found and the number of possible colors to be included in the sequence. Implementing this game, we substitute the colors with decimal digits and assume that all digits from 0 to 9 may be included. Being trivial to program, the keeper's side is left beyond our interest. Instead, we implement the following modification:

1. The program asks and the user decides the number of digits N in the sequences. Realistically, it ranges from 4 to 6.
2. The user writes down a sequence of digits and lets the program to make the first guess.
3. Every move is graded by the user, and, using these grades, the program suggests the next trial sequence.
4. The game finishes when the grade consists of N black "stones".

The Serial Algorithm

It is not hard to construct a "decision making" approach for the program. We outline here and implement in the code the principal algorithm, leaving its optimization for later stages.

1. Once the amount of digits N is entered, numbers from the entire range from 0 to 10^N-1 are placed in an array. The sequence to be found is definitely among them.
2. At random an element from the existing array is chosen and offered as a trial.
3. The grade of the trial is recorded. Then, all the array elements are graded, while being compared with the trial, and those of them are kept that lead to the same recorded grade. Thus, the array is reduced.

4. The algorithm returns to Step 2 if there are elements to choose from and the grade is still less than 10N. Otherwise, either the sequence is found, or the user provided somewhere above a wrong grade.

The corresponding flow chart is given in the Appendix 1.

In order to demonstrate how the algorithm works, we trace the solution for the case of N = 2 assuming that the original number to be found is 12:

Move #	**Array Elements**	**Array Length**	**Trial**	**Grade**
1	Successive numbers from 0 to 99 are included.	100	34	00
2	The numbers containing 3 or 4 are excluded.	64	25	01
3	The numbers staring with 2 or ending with 5 are excluded.	49	61	01
4	The numbers starting with 6 or ending with 1 are excluded.	36	58	00
5	The numbers containing 5 or 8 are excluded.	20	16	10
6	The numbers not starting with 1 or not ending with 6, together with 16, are excluded.	7	19	10
7	The numbers not starting with 1 and 19 are excluded.	3	12	20

The algorithm for computing a relative grade between two numbers of N digits each consists of the following three steps:

1. The digits in the numbers are compared pair wise. The number of coinciding digits makes the “black” grade. Such digits are removed from the respective numbers.
2. The last digit of the second number is brought to its beginning and the Step 1 is repeated. The number of the coinciding digits is added to the “white” grade.
3. In all, the Step 2 is repeated N-1 times. The value of the “white” grade accumulates all N-1 iterations.

It is obvious that the longest analysis is required after getting the first grade. And the main issue is not a lengthy delay with the replay, but rather the volume of required memory resources. A common RAM ranges from 128 MB to 1 GB. By looking at the table below that shows how much of memory is required for different game modes, we can conclude that games with 7 – 8 digits will hardly start.

Sequence length	**Length of the initial array**	**Required memory**
4	10 000	40 KB
5	100 000	400 KB
6	1 000 000	4 MB
7	10 000 000	40 MB
8	100 000 000	400 MB
9	1 000 000 000	4 GB

The Parallel Implementation

Parallelization of the algorithm will provide a natural solution. We just need to incorporate several processors or, equivalently, organize as many processes. Let's denote the number of the processes by p. Every process will keep and handle just p-th part of that general array. Assuming p = 10, we can ensure full functionality of a game of 7 and, probably, 8 digits. For 9 digits, p must be of order of 100. Practically, however, the game loses interest, if the sequences become too long. Therefore, we can fully limit ourselves to the resources of the AUA computer labs. For the case of 7 digits each computer running a process will deal with numbers ranging from (rank * 1 000 000) to ((rank + 1) * 1 000 000 – 1) to be stored in some 4 MB of its memory. Here 0 ≤ rank < p is the index of the process.

Understandably, the algorithm for parallel implementation, shown below, is more complex than its serial counterpart.

1. User provides the process 0 with the sequence length – **numbsize** from user. Then, the process 0 creates a dynamic array **candArray** for temporary storage of candidate trials recived from all "not empty" processes.
2. The process 0 sends to all other processes the value of **numbsize**.

3. Each process creates its own dynamic integer array **localArray** of $10^{numbsize}$/p elements. The elements are initialized by their respective indexes added to ($10^{numbsize}$/p * **current_process_number**), where **current_process_number** is the process' rank. Everything is ready for the main loop to start.
4. Every process checks if its **localArray** is empty.
   a. If Yes: the process sends to the process 0 a negative number (**candcont = -1**) as an indicator;
   b. If No: the process sends to the process 0 a randomly chosen element from its **localArray** array**.**
5. The process 0 receives elements from all other processes and, if non-negative, places them in the array **candArray**. Then, it checks if **candArray** is left empty?
   a. If Yes: the games is terminated, as the user gave a false grade as some stage;
   b. If No: The process 0 randomly chooses the next **trial** from **candArray**.
6. The process 0 sends the newly chosen **trial** to all other processes and waites for the user's grade. If the grade equals to 10 * **numbsize**, then the sequence is found and the game is over
7. The process 0 sends the **grade** to all other processes.
8. Each process shortens its **localArray** by removing those elements that do not match the recent trial modulo the last grade.
9. The Step 4 is repeated.

The corresponding chart is shown in the Appendix 2.

Problem 2 – Duties Scheduling

Duties scheduling is another problem the “decision making” approach can be applied to. The problem deals with **n** people, to be divided into groups of **m** persons each. Some duties must be conducted by one group per day. It is necessary to schedule these groups as to ensure everyone has at least **k** day offs between two successive working days. Well before the current work was proposed, this problem was solved in MS Excel. Obviously, the latter is not the most convenient environment – the problem set for **n = 16**, **m = 2** and **k = 1** generated a spreadsheet of 7.5 MB size. Here, however, we copy the main table, in order to explain the algorithm. The parameters are chosen as **n = 6**, **m = 2** and **k = 1**.

GROUP	**CREW**		**DAYS**													
			2	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**	**15**
1	1	2	0	0	0	0	0	0	0	0	0	0	0	0	0	0
2	1	3	0	0	1	0	0	0	0	0	0	0	0	0	0	0
3	1	4	0	0	1	0	0	0	1	0	0	0	0	0	0	0
4	1	5	0	1	0	0	1	0	0	0	0	0	0	0	0	0
5	1	6	0	1	0	0	1	0	0	0	1	0	0	0	0	0
6	2	3	0	0	1	0	0	1	0	1	0	1	0	0	0	0
7	2	4	0	0	1	2	0	0	0	0	0	0	0	0	0	0
8	2	5	0	1	0	1	0	0	1	2	0	0	0	0	0	0
9	2	6	0	1	0	1	0	1	0	1	0	0	0	1	0	0
10	3	4	1	0	0	0	0	0	0	0	0	0	0	0	0	0
11	3	5	1	0	0	0	1	0	0	1	0	1	0	0	1	0
12	3	6	1	0	0	0	1	2	0	0	0	0	0	0	0	0
13	4	5	1	0	0	1	0	0	1	0	0	1	2	0	0	0
14	4	6	1	0	0	1	0	1	0	0	1	0	1	0	0	1
15	5	6	1	2	0	0	0	0	0	0	0	0	0	0	0	0

First of all, the people are referred to by numbers from 1 to 6. Then, all possible 2-person groups are constructed – there are 15 such groups. Each row in the table corresponds to a distinct group. From the first three columns we see that, for example, the second and the forth persons constitute the group 7.

Without any loss of generality we can assume that the first day duties are taken by the first group. The table is filled column-by-column according to the following rules:

1. Initially all the groups are assigned a rate 0.
2. After a group is scheduled for some day, in all remaining cells in its row zeros are placed. It indicates that this group will never be on duties till the end of the cycle.
3. For a given day the group is assigned a rate 0, if one of its members was on duties the previous day (do not forget that in this particular example **k = 1**). Otherwise, if none of its members were on duties the previous day, the rate of the group is increased by 1.
4. That group is scheduled for the day whose rate is maximal in the corresponding column. In the presented table always the group of the first maximal value is chosen. In columns the cell of the selected group's rate is shaded. Note that, rightfully, there is one and only one shaded cell per each column and each row.

The generalization of this example is pretty transparent:

1. Create an array of all possible groups and keep their rates there. The length of the array is $C_n^m = \dfrac{n!}{m!(n-m)!}$.
2. Randomly choose such a GROUP from the array whose rate is maximal.
3. Write the GROUP in the Duty list and remove it from the array.
4. Assign 0 day off to the members of the GROUP and increase the existing values of the day off of all other persons by 1.
5. In the groups array increase by 1 the rate of those groups where all members have at least k day offs. Set to 0 the rate of all other groups.
6. Go to Step 2 until the array becomes empty.

The corresponding flow chart is shown in the Appendix 3.

The Parallel Implementation

In principle, this problem is computationally expensive – the number of operations is O(square of the number of groups) ≈ $O(2^{n/2} / n^2)$. The last estimation is obtained based on the Stirling approximation. Under the current interpretation, though, the realistic number of persons **n** is of order of 10, and the longest possible schedule consists of 252 days and, therefore, requires only $O(10^5)$ operations. However, we can imagine a different situation. Let's consider an electronic device containing 30 chips. At given time only 15 of them concurrently operate. The number of possible combinations in this case is more than 10^8. A single processor will require 1 – 3 weeks to complete the schedule. Again, parallelization is the only feasible way to get the result. Like above, it requires much more efforts, than the serial version. The detailed algorithm follows:

1. The process 0 starts the game:
    a. Asks a user the values of **n**, **m**, **k;**
    b. Creates a list of process readiness – an array **procs**. It shows the current state of every process. The states are distinguished between **ready**, **not ready** and **empty**. At the beginning all processes are at the **ready** state. They will switch between **ready** and **not ready** states until becoming **empty**.
2. The process 0 broadcasts the values of **n**, **m** and **k**.
3. Every process calculates the number of all possible groups: **nog** $\equiv C_n^m = \frac{n!}{m!(n-m)!}$
4. Every process creates and initializes the following two local arrays:
    a. **candArray** – the array of the groups that belong to the process. All the groups are given the same initial rate.
    b. **persons** – the array of day off of the persons involved in the process's group. The elements are initialized with **k**.

5. The process 0 randomly chooses the very first group by its absolute index **absindexR** ranging from **0** to **nog-1** and broadcasts **absindexR**.
6. The index **cproc** of a process that owns the chosen group **absindexR** is calculated.
7. IF **absindexR** is positive at this step, then the main recording steps are fullfilled:
   a. The process 0 changes all **not ready** states to **ready** in the array **procs**;
   b. The same process records the group in the schedule file;
   c. The process **cproc** removes the group **absindexR** from its **candArray**;
   d. All processes set the day off to 0 in their respective **persons** array for all the members of the newly recorded group;
   e. Also, they increase the rate of thouse groups where all the members already have at least **k** day offs, and assign zero to the rates of all other groups.
8. ELSE, the non-positive value of **absindexR** indicates that **cproc** is either **not ready** or **empty**. The following adjustments are necessary then:
   a. The process 0 assigns to **procs**[**cproc**] **not ready** status, if **absindexR = -1**, or **empty** – if **absindexR = -100**;
   b. If there are no processes with **ready** status in procs, then the SCHEDULE IS OVER!
9. It is time for choosing a group for the next day. It is done in two stages. First, the process 0 randomly chooses from the array **procs** and broadcasts another candidate process index **cproc**. The latter must be in **ready** state.
10. The process **cproc** considers in its **candArray** all the groups with the maximal rate and randomly chooses **absindexR** from them and broadcasts this value. If there are no groups to choose from, then the process is empty and, therefore, **absindexR** is becomes **-100**. It may also happen that the process is not empty, but all its groups have 0 rates. It means that none

of them can be scheduled at the moment. Therefore, the process has **not ready** status and **absindexR = -100**.

11. Go to the Step 8.

The corresponding flow chart is in the Appendix 4 and the code – in Appendix 8.

4. Tests and Results

There are several aspects we would like to check. First of all, it is the performance of the computers and the Local Are Network. In both problems we tried to minimize the data volume to be communicated between the processes. So, this aspect was reduced to the interaction between the installed MPICH and the existing hardware. Most notably, we wanted to clarify the correct configuration and running mode of the MPICH environment.

Secondly, we are interested in the performance of the parallelized algorithms. The main questions are how faster the parallel implementations were producing the results compared with the serial code (Duty Schedule), and by how much the parallelization eased up the issue of the memory resources (Master Mind).

The problems were running on the computers from the 9B and 9D labs. We included computers of different characteristics in the "cluster", in order to emulate the conditions arising when as many AUA computers would be involved as possible. The leading process (process 0) was running on comp# 22 in the 9B Lab. In order to let all the processes work consistently under MS Windows 2000 / XP, namely have equal rights to write in their respective shared folders, the programs were run under an account from either Administrators or Power Users gropus. The same account and password were created on all the cluster computers. Totally, we included 12 computers. The table of the computer characteristics in the AUA labs can be found in [9].

Master Mind

Below is a fragment from the testing results to be found in the Appendix 9. The sequence length in this run is 6.

Move	Serial code (on 9B_22)	Parallel code (12 PC, 12 processes)
1	less then 5.1 sec	less then 0.8 sec
2	less then 1.8 sec	less then 0.2 sec

3	less then 0.3 sec	less then 0.1 sec
4-6	less then 0.1 sec	less then 0.1 sec

The acceleration is apparent, especially for the first move, when the program deals with the longest array of candidates. After that, however, both algorithms generate a replay almost instantaneously. Therefore, at these stages the speed increase is not important at all. The most important gain is the increase in memory resources. If a 7 digits-long game is played, then the serial code stops working in 44.9 seconds and produces “Your systemi is low on virtual memory” error message. Meantime, the paralle code with 12 computers completes the move in some 10 seconds.

Duties Scheduling

During coding we confronted with certain difficulties. The main disadvantage was absence of debugging tools. In serial programming we have full power of debugging tools provided by MS Visual Studio. In the MPICH under Window environment, however, we did not discover anything similar. So, we came to some sort of nonsense – under the increased complexity we were deprived of debugging tools.

The serial Duty Scheduling program, as well as its parallel but one-processor version, worked well from the beginning. But the same program stopped working with several processes – the network connection was lost and the following error was reported:

```
Error 64, process 0, host 9B_22:
   GetQueuedCompletionStatus failed for socket 1 connected to
host '10.2.0.182
```

It required quite substantial time and efforts to find out that reason is in lack of synchronization. We reconstructed the serial code and artificially divided it into processes, in order to find the bug (see the Appendix 7). MPI provides a special function MPI_Barrier(), at which all the processes stop until the last one reaches this point. Surprisingly, it did not behave as expected. Trying to achieve a consistent performance and stability, we ensured equal load on all the processes, avoiding long idle time while other processes intensively worked.

Below is a fragment from the testing results to be found in the Appendix 9.

Serial code:
n=30, m=4, k=1, nog=27405
24 groups were calculated in 1min=60 sec (was checked 3 times).
We estimate all 27405 groups to be calculated in ≈ 68513 sec ≈ 1142 min ≈ 19 hours

Parallel code – 12 processes, 12 computers:
n=30, m=4, k=1, nog=27405
The parallel program worked 1347 secs ≈ 22.5 min

Parallel code – 24 processes, 12 computers:
n=30, m=4, k=1, nog=27405
The parallel program worked 931 secs ≈ 15.5 min

Parallel code – 36 processes, 12 computers:
n=30, m=4, k=1, nog=27405
The first 4000 groups were calculated in 173 sec.
We estimate all 27405 groups to be calculated in ≈1186sec ≈ 20 min

The testing results speak for themselves. The parallel implementation is faster than the serial one by 10 – 100 times. Another interesting effect is the additional acceleration, if each computer runs 2 processes. 3 processes per computer bring the speed to the initial level, and further ioncrease of the processes per computer start slow down the performance.

5. Conclusions and Future Work

1. The completed project fully confirms the idea of promotion of the parallel computing in American University of Armenia. The existing hardware and software resources, as well as the academic preparation of the CIS students make large-scale problems possible to address.
2. The essential part of the project was devoted to learning and adoption of the parallel programming language and methodology.
3. We considered two model problems and observed fundamental improvement in the programs' performances. Such promising results are based not only on involvement of the resources of many computers, but also on carefull parallelization of the algorithms. The main cornerstones are the minimial possible data volume to be communicated and maximal consistency in the nature and load of the tasks between the processes.
4. Parallel implementation appears a convenient opportunity to pay more attention to capabilities of an algorithm and discover better solutions that would be normally neglected in the serial implementation. Meantime, the coding and debugging are more complex.
5. The future work can and, most probably, will include switching to MPICH2, which is a more recent implementation of MPI. In addition to the features in MPICH, MPICH2 includes support for one-side communication, dynamic processes, intercommunicator collective operations, and expanded MPI-IO functionality. MPICH2 is a unified source distribution, supporting most Unix versions and recent versions of MS Windows. For the latter it requires NET thechnology [6].

References.

1. V. Kumar, A. Grama, A. Gupta, G. Kerypis, Design and Analysis of Algorithms, Benjamin / Cummings Publishing Company, Inc., 1994.

2. Parallel Computing at a Glance, www.buyya.com/microkernel/chap1.pdf

3. W. Gropp, E. Lusk and A. Skjellum, Using MPI: Portable Parallel Programming with the Message-Passing Interface, MIT Press, Cambridge, MA, 1994.

4. Introduction to Parallel Computing, http://www.llnl.gov/computing/tutorials/workshops/workshop/parallel_comp/MAIN.html

5. Message Passing Interface (MPI), http://www.mhpcc.edu/training/workshop/mpi/MAIN.html

6. MPICH-A Portable Implementation of MPI, http://www-unix.mcs.anl.gov/mpi/mpich/

7. A. R. Rossi, Parallel Computing Overview, Institute for Advanced Study, Advanced Computing Technology Center, 2004.

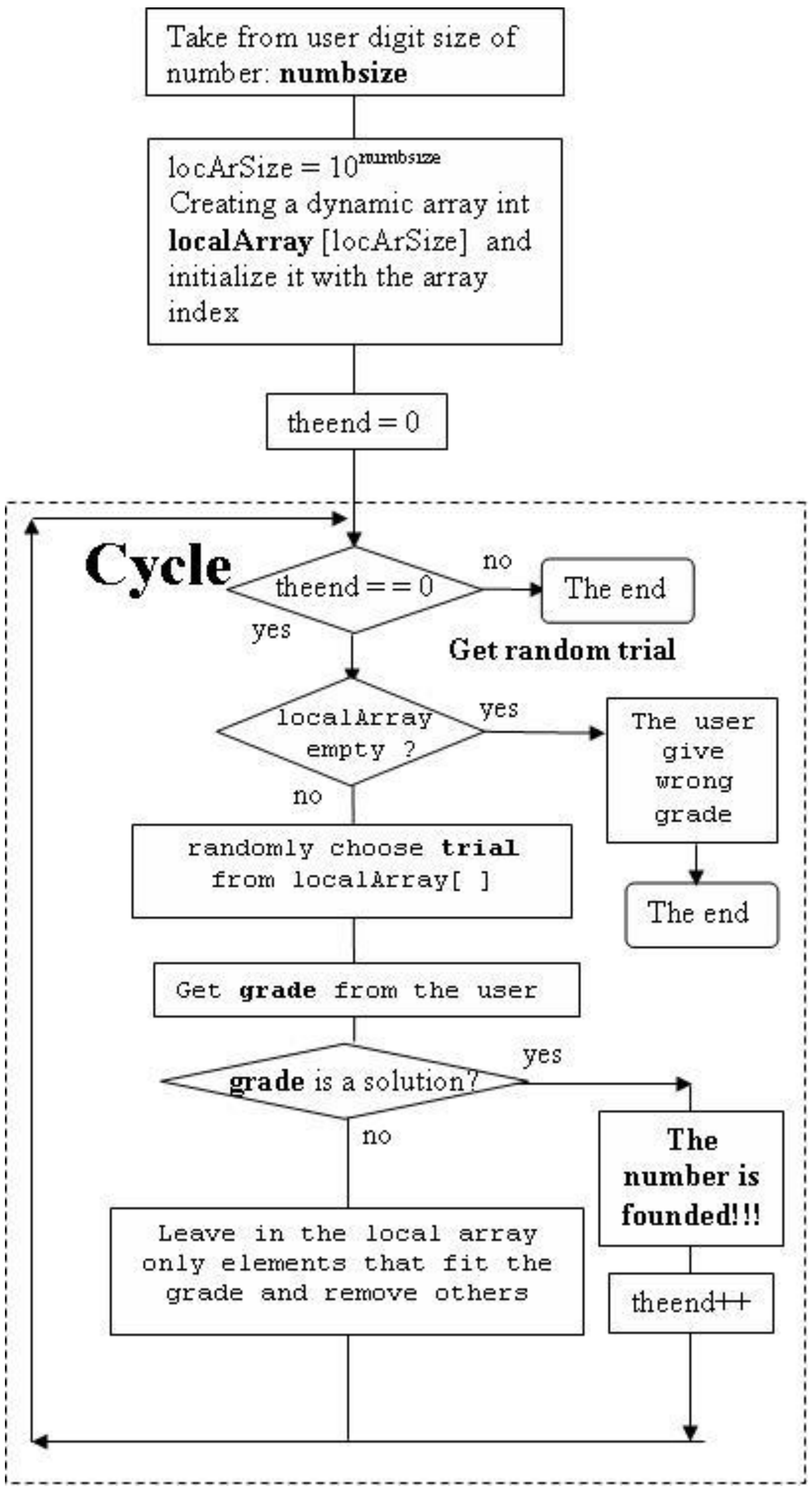
8. What is Parallel Computing? http://www.eecs.umich.edu/~qstout/parallel.html

9. V. Khachatryan , Parallel Computation of Derivatives on One- and Two-Dimensional Domains, Master's project, CIS, AUA, Yerevan, 2005.

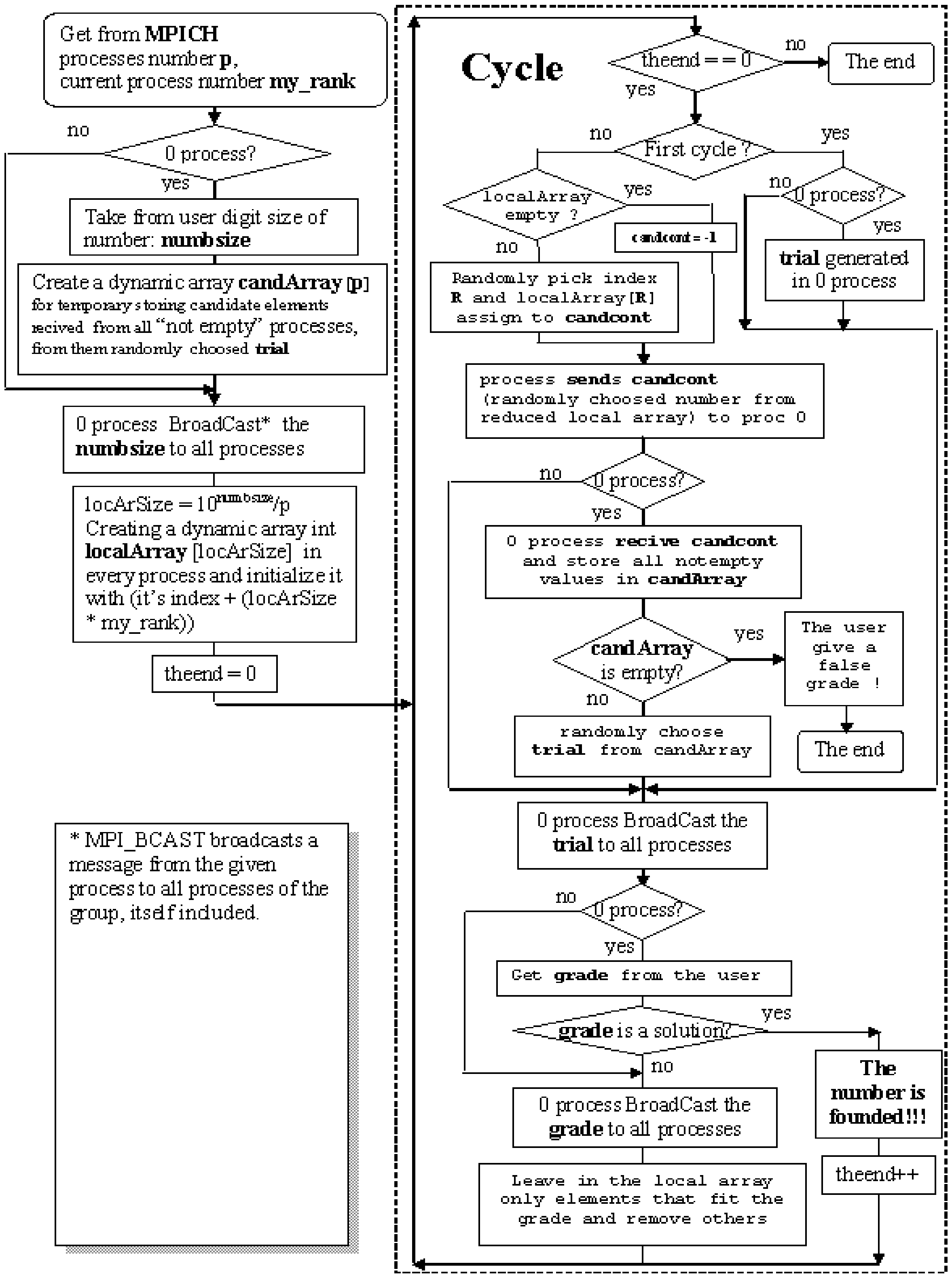
10. A. Darbinyan, Parallel Implementation of Quantum Algorithms, Master's project, CIS, AUA, Yerevan, 2005.

11. J. Waldo, Remote procedure calls and Java Remote Method Invocation. *IEEE Concurrency 6*, 3 (July–Sept.), 1998.

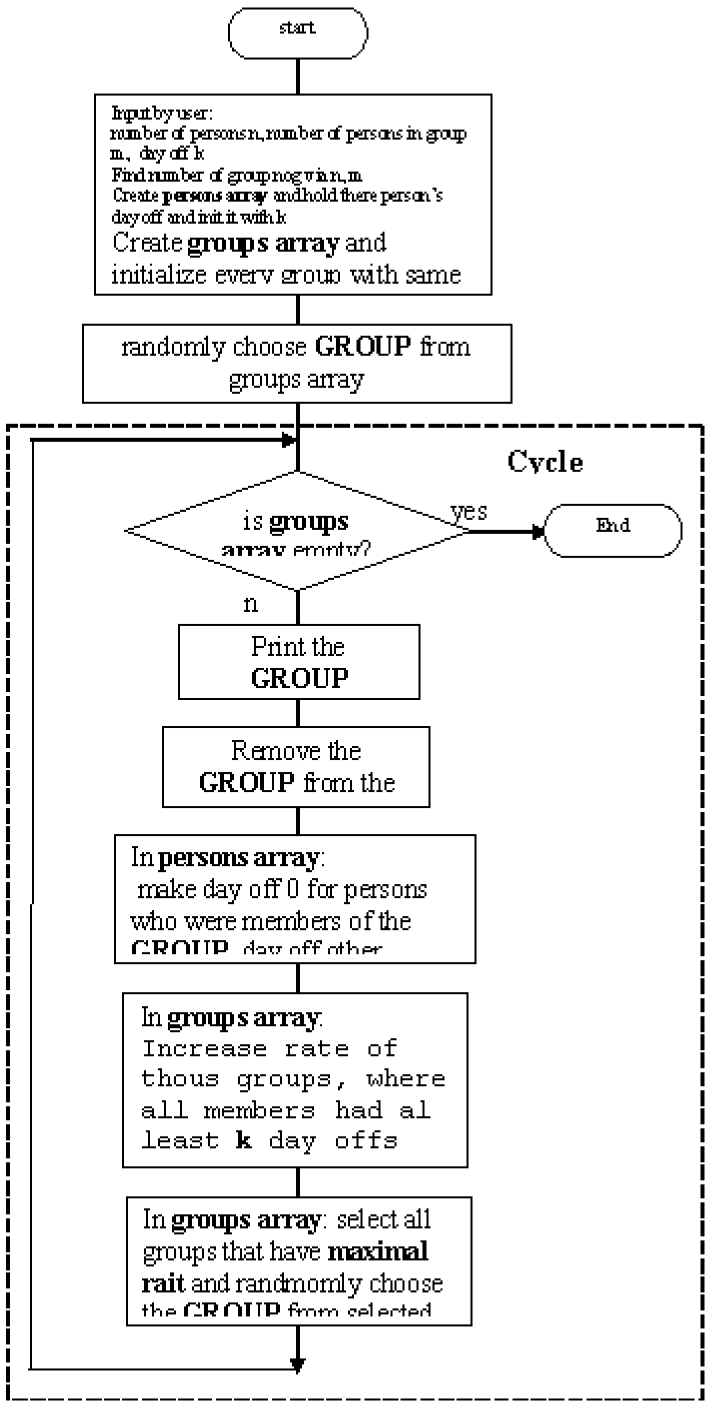
Appendix 1. Flow chart of Serial MasterMind


Take from user digit size of number: numbsize
locArSize = 10^numbsize
Creating a dynamic array int localArray [locArSize] and initialize it with the array index
theend = 0
Cycle
theend = = 0
no
The end
yes
Get random trial
localArray empty ?
yes
The user give wrong grade
The end
no
randomly choose trial from localArray[ ]
Get grade from the user
grade is a solution?
yes
The number is founded!!!
theend++
no
Leave in the local array only elements that fit the grade and remove others

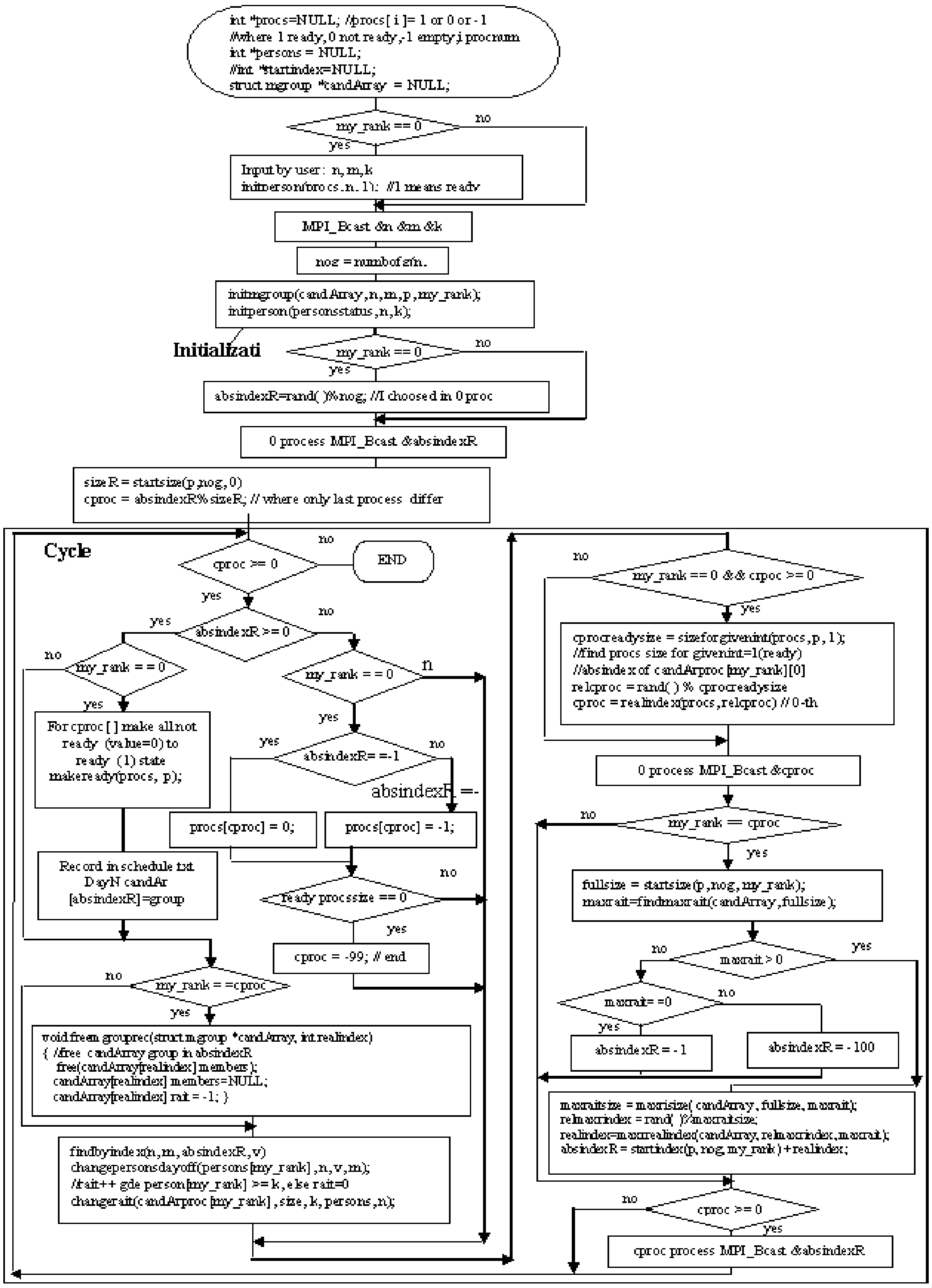
Appendix 2. Flow chart of Parallel Master Mind


Get from MPICH processes number p, current process number my_rank
no
0 process?
yes
Take from user digit size of number: numbsize
Create a dynamic array candArray [p] for temporary storing candidate elements recived from all "not empty" processes, from them randomly choosed trial
0 process BroadCast* the numbsize to all processes
locArSize = 10^numbsize/p Creating a dynamic array int localArray [locArSize] in every process and initialize it with (it's index + (locArSize * my_rank))
theend = 0
* MPI_BCAST broadcasts a message from the given process to all processes of the group, itself included.
Cycle
theend == 0
no
The end
yes
no
First cycle ?
yes
localArray empty ?
yes
candcont = -1
no
Randomly pick index R and localArray[R] assign to candcont
no
0 process?
yes
trial generated in 0 process
process sends candcont (randomly choosed number from reduced local array) to proc 0
no
0 process?
yes
0 process recive candcont and store all notempty values in candArray
candArray is empty?
yes
The user give a false grade !
The end
no
randomly choose trial from candArray
0 process BroadCast the trial to all processes
no
0 process?
yes
Get grade from the user
grade is a solution?
yes
The number is founded!!!
theend++
no
0 process BroadCast the grade to all processes
Leave in the local array only elements that fit the grade and remove others

Appendix 3. Flow chart of Serial Duties Scheduling


start
Input by user:
number of persons n, number of persons in group m, day off k
Find number of group nog via n, m
Create persons array and hold there person's day off and init it with k
Create groups array and initialize every group with same
randomly choose GROUP from groups array
Cycle
is groups array empty?
yes
End
n
Print the GROUP
Remove the GROUP from the
In persons array: make day off 0 for persons who were members of the GROUP, day off other
In groups array: Increase rate of thous groups, where all members had al least k day offs
In groups array: select all groups that have maximal rait and randmomly choose the GROUP from selected

Appendix 4. Flow block of Parallel Duties Scheduling


int *procs=NULL; //procs[i]= 1 or 0 or -1
//where 1 ready, 0 not ready, -1 empty, i procrunn
int *persons = NULL;
//int *startindex=NULL;
struct mgroup *candArray = NULL;
my_rank == 0
no
yes
Input by user: n, m, k
initperson(procs, n, 1); //1 means ready
MPI_Bcast &n &m &k
nog = numbofs/n.
initmgroup(candArray, n, m, p, my_rank);
initperson(personsstatus, n, k);
Initializati
my_rank == 0
no
yes
absindexR=rand( )%nog; //I choosed in 0 proc
0 process MPI_Bcast &absindexR
sizeR = startsize(p, nog, 0)
cproc = absindexR%sizeR; // where only last process differ
Cycle
cproc >= 0
no
END
yes
absindexR >= 0
yes
no
my_rank = = 0
no
yes
For cproc [ ] make all not ready (value=0) to ready (1) state makeready(procs, p);
Record in schedule txt DayN candAr [absindexR]=group
my_rank = = 0
n
yes
absindexR= =-1
yes
no
absindexR =-
procs[cproc] = 0;
procs[cproc] = -1;
ready procssize == 0
no
yes
cproc = -99; // end
my_rank = =cproc
no
yes
void freem grouprec(struct mgroup *candArray, int realindex)
{ //free candArray group in absindexR
free(candArray[realindex] members);
candArray[realindex] members=NULL;
candArray[realindex] rait = -1; }
findbyindex(n, m, absindexR, v)
changepersonsdayoff(persons[my_rank], n, v, m);
//rait++ gde person[my_rank] >= k, else rait=0
changerait(candArproc[my_rank], size, k, persons, n);
my_rank == 0 && cproc >= 0
no
yes
cprocreadysize = sizeforgivenint(procs, p, 1);
//find procs size for givenint=1(ready)
//absindex of candArproc[my_rank][0]
relcproc = rand( ) % cprocreadysize
cproc = realindex(procs, relcproc) // 0-th
0 process MPI_Bcast &cproc
my_rank == cproc
no
yes
fullsize = startsize(p, nog, my_rank);
maxrait=findmaxrait(candArray, fullsize);
maxrait > 0
no
yes
maxrait= =0
no
yes
absindexR = -1
absindexR = -100
maxraitsize = maxrisize(candArray, fullsize, maxrait);
relmaxrindex = rand( )%maxraitsize;
realindex=maxrealindex(candArray, relmaxrindex, maxrait);
absindexR = startindex(p, nog, my_rank) + realindex;
cproc >= 0
no
yes
cproc process MPI_Bcast &absindexR

Appendix 5. Parallel Master Mind listing

```
/* Parallel Number finding game
 */
#include <iostream.h>
#include <stdio.h>
#include <math.h>
#include <stdlib.h>
#include <malloc.h>
#include <time.h>
#include "mpi.h"

const int empty = -1;

void ArrayCopy(char *arrayto, int number,int size);
void takeFromUser(int *number, char *text);
void extractArr(int Arr[], int *arsize, int trial,int grade, int digitsize);
int FitDigit(char *chtrial, char *chlocalElement, int fulldigitsize);
int isGrade(int trial, int localElement, int grade, int digitsize);

void main(int argc, char** argv)
{
      int candstep;
      int theend = 0;
      int my_rank;       // my process rank
      int numbsize;      // Number of digits, size of digits, qani nish e tiv@
      int locArSize;
      int step=0;
      int temprint;
      //char *temprint=NULL;
      /*char *trial=NULL;
      char *tempCand = NULL;
      char *candcont = NULL;
      char **candArray = NULL;
      char **localArray = NULL;*/
      int trial;        // sizeof(int and long)=4byte max=2^32/2-
1=4294967296/2-1=2147483647
      int tempCand;
      int candcont;
      int *candArray = NULL;
      int *localArray = NULL;

      time_t t;
    srand((unsigned) time(&t));

      int grade;
      int p;                   // The number of processes
      MPI_Status status;

      MPI_Init(&argc, &argv); // let the slystem do what int needs to a start
up MPI
      MPI_Comm_rank(MPI_COMM_WORLD, &my_rank); //Get my process rank
      MPI_Comm_size(MPI_COMM_WORLD, &p); //Find out how many processes are
being used
      /**********************
       1. Ask and gets size - proc0*/
      if (my_rank == 0)
      {
            takeFromUser(&numbsize, "Type digit size of your number:\n");
      // take number digit size from user
            if (candArray == NULL)
            {
                  //candArray = new char* [p];
                  candArray = new int[p];
            }
            //if(temprint == NULL)
            //    temprint = new char[numbsize+1];
      }
      /***********************
      2. Send the size to all processes from proc0 - proc0 broadcast*/
      //MPI_Bcast(&locArSize, 1, MPI_INT, 0, MPI_COMM_WORLD);
      MPI_Bcast(&numbsize, 1, MPI_INT, 0, MPI_COMM_WORLD);
      locArSize = (int)pow(10.0,numbsize)/p; //calculate for process's
localarray size
      /***********************
      3. Create and initial local array dinamically - all processes */
      if (localArray == NULL)
      {
            //int temp;
            localArray = new int[locArSize];
            //all local arrays hold the same digits from 0 to locArSize-1
            //abs number = my_rank*locArSize + localArray[i];
            for(int i = 0;i<locArSize;i++)
            {
                  localArray[i] = i + (locArSize * my_rank);
            }
      }
      do {
            /*********************
            4. Get random trial
            4a. if step = 0. Generate random trial */
            if(step == 0)
            {
                  /*if (trial == NULL)
                  {
                        trial = new char [numbsize + 1];
                  } */

                  if(my_rank == 0)
                  {
                        int temp, i;
                        trial = 0;
                        for (i = 0; i<numbsize;i++)
                        {
                              temp = rand()%10;
                              //trial[i] = char(int('0') + temp);
                              trial += temp * (int)pow(10.0, i);
                        }
                        //trial[numbsize] = '\0';
```

```
			}
		}
		/****************
		4.b all processes send randomly choosed from reduced local array
to proc 0*/
		else
		{
			if(locArSize>0)
			{
				candcont = localArray[rand()%locArSize];
			}
			else
				candcont = empty; // sm define empty -1;
//2100000000; //if nothing to send then value > 1000000000-1
				//candcont = "\0";

			//MPI_Send(candcont, numbsize + 1, MPI_CHAR, 0, my_rank,
MPI_COMM_WORLD);
			MPI_Send(&candcont, 1, MPI_INT, 0, my_rank,
MPI_COMM_WORLD);
			if(my_rank == 0)
			{
				candstep = 0;
				for (int i = 0;i< p;i++)
				{
					MPI_Recv(&tempCand, 1, MPI_INT, i, i,
MPI_COMM_WORLD, &status);
					if(tempCand > empty)
					{
						candArray[candstep] = tempCand;
						candstep++;

					}
				}
				if (candstep > 0)
					trial = candArray[rand()%candstep];
				else
				{
				//add here ostatoksize = (int)pow(10.0,numbsize)%p
that was not include in arrays
				// for example digits%process = 10000%3=1 10000%6=4
numbers not included in game
					printf("You have given me false grade!\n");
					fflush(stdout);
					theend++;
					//new
					trial = -1;
				}
			}
		}
		step++;
		/**********************************
		5. Send the trial and its grade to all proc - from 0 proc*/
		MPI_Bcast(&trial, 1, MPI_INT, 0, MPI_COMM_WORLD);
		//new changes
		if(trial< 0)
		{
```

```
                  theend++;
                  break;
            }
            /*******************************
            6. Get grade from the user */
            if(my_rank == 0)
            {
                  //printf("Give grade for %d",trial);
                  printf("Give grade for ");
                  fflush(stdout);
                  temprint = trial;
                  for(int m=numbsize-1;m>=0;m--)
                  {
                        printf("%d",temprint/(int)pow(10.0,m));
                        fflush(stdout);
                        temprint%=(int)pow(10.0,m);
                  }
                  takeFromUser(&grade, ". Please type grade:\n");
            }
            MPI_Bcast(&grade, 1, MPI_INT, 0, MPI_COMM_WORLD);
            if(grade/10 == numbsize)  // if (40/10 == 4) then theendo = 1
            {
                  if(my_rank==0)
                  {
                        printf("I hardly worked and found the number in %d
step!!!\n", step);
                        fflush(stdout);
                  }
                  theend++;   //exit the while loop;
            }


            //7.Extract from local array all elements that fit the grade and
reduce locArSize
            //extractArr(localArray, &locArSize, trial, grade, numbsize+1);
            if(theend == 0)
                  extractArr(localArray, &locArSize, trial, grade, numbsize);

      } while (theend==0);  // poka theend == 0


      //Shut down MPI
      MPI_Finalize();
      return;

} // end main

// int -> char *  01234 -> char [0,1,2,3,4];
void ArrayCopy(char *arrayto, int number,int size)
{
      int temp = number;
      for(int d=size-1;d>=0;d--)
      {
            arrayto[d] = (char)((int)'0' + temp%10);
            temp /= 10;
      }
```

```
}

void takeFromUser(int *numberptr, char *text)
{
      printf("%s",text);
      fflush(stdout);
      scanf("%d", numberptr);
}


//void extractArr(char **Arr, int *arsize, char *trial,int grade, int
digitsize)
void extractArr(int Arr[], int *arsize, int trial,int grade, int digitsize)
{
      int flag = 0;
      for(int i=0;i<*arsize;i++)
      {
            if (isGrade(trial, Arr[i], grade, digitsize))
            { //if Arr[i] don't fit
                  Arr[flag]  = Arr[i];
                  flag++;
            }
      }
      *arsize = flag;
}

int isGrade(int trial, int localElement, int grade, int digitsize)
{
      // if localElement's digits fit grades of trial than true (return 1),
esle 0
      //grade 2 digit 1s digit the digit's are the same , second grad's digit
the digit present but not in it place
      int fit1, fit2=0;
      char tempch;
      char *temp        = new char[digitsize]; //NULL; //= localElement; //,
digit;
      char *tmptrial    = new char[digitsize];
      ArrayCopy(temp, localElement,digitsize);
      ArrayCopy(tmptrial, trial,digitsize);
      fit1 = FitDigit(tmptrial, temp, digitsize);
      if (!(grade/10 - fit1))
      {
            for(int i=1;i < digitsize;i++)

            {
                  tempch = temp[0];
                  for(int j=1;j<digitsize;j++)
                  {
                        temp[j-1] = temp[j];
                  }
                  temp[digitsize-1] = tempch;
                  fit2 += FitDigit(tmptrial, temp, digitsize);
            }

            if(!(grade%10 - fit2))
            {
                  //printf("'lE=%d->%d%d' return1 ", localElement,fit1,fit2);
                  return 1;
            }
      }
      //printf("'lE=%d->%d%d' return0 ", localElement,fit1,fit2);
      return 0;
}
//int FitDigit(long trial, long localElement, int digitsize)
int FitDigit(char *chtrial, char *chlocalElement, int fulldigitsize)
{
      //count how many digits int trial and localElement the same (fits each
other)
      //if last char of trial is '\0' then
      int digitsize = fulldigitsize;
      int   count=0;
      for ( int i = 0; i < digitsize; i++)
      {
            //trialchr[0] = tria
            if (chtrial[i] == chlocalElement[i])
            {
                  count++;
                  chtrial[i] = 'A';
                  chlocalElement[i] = 'B';
            }
      }
      return count;
}
```

Appendix 6. The listing of my_funct.c, the library of functions are used in the Duties Scheduling

```
#include <stdio.h>
#include <malloc.h>
#include <math.h>
#include <stdlib.h>

#include <time.h>

struct mgroup
{
	int *members;
	int rait;
};

void initperson(int *persons, int n, int k); /*initialize array of n integers with k value*/
int numbofg(int n, int m);
int startsize(int p, int nog, int my_rank); /*calculate CandArray size of current process, last proc
p-1 otlichaetsya*/
int startindex(int p, int nog, int my_rank); /*output: return the first index of current process*/
int findbyindex(int n, int m, int index, int *v); /* index must be  0 <= index < nog, abnormal
index return -999*/
void initmgroup(struct mgroup *candArray, int n, int m, int p, int my_rank);
void filemgroup(FILE *outFile, struct mgroup *candArray, int m, int nog);
void changepersonsdayoff(int *persons, int n, int *v, int m);
/* NULL        0 1      NULL  3 4 NULL       5 6 NULL       7   8  9    relindex 9 ->
// 0      1 2       3                   4 5 6               7 8 9               10 11 12         realindex = 12 */
int findrealindex(struct mgroup *candArray, int relindex);
int maxrisize(struct mgroup *candArray, int realsize, int maxrait); /*show how many candArray
groups has maxrait realsize = startsize(p, nog, my_rank)*/
int maxrrealindex(struct mgroup *candArray, int relmaxrindex, int maxrait);
/*0<=relmaxrindex<size of CandArray; 0<=realindex<startsize() index of CandArray real size*/
void freemgrouprec(struct mgroup *candArray, int realindex); //free group in absindexR
void changerait(struct mgroup *candArray, int size, int k, int *persons, int n, int m); /*size is
quontity of not NULL members;*/
int notnullsize(struct mgroup *candArray, int realsize); /*show how many candArray has not
NULL members*/
int sizeforgivenint(int *procs, int fullsize, int givenint);
int realindex(int *procs, int relcproc, int givenint); /* realindex of relindex*/
void makeready(int *procs, int p); /* all not ready (0) make ready (1)*/
int findmaxrait(struct mgroup *candArray, int size);
//void printinfo(FILE *outFile, struct mgroup **candArproc, int n, int m, int p, int nog, int
my_rank, int *procs, int **persons);
```

```
void printfreegrp(FILE *outFile, struct mgroup candArray, int count, int proc,int realindexR,int
absindexR, int m);
void printit(struct mgroup *candArray, int n, int m, int p, int my_rank, char *start, char *end,
char *delim); /*start = "[", end = "]=", delim = "\t" */
void writeinfile(FILE* outFile, struct mgroup *candArray, int n, int m, int p, int my_rank, char
*start, char *end, char *delim); /*start = "[", end = "]=", delim = "\t"*/

/*initialize array of n integers with k value*/
void initperson(int *persons, int n, int k)
{
	int i;
	for(i=0;i<n;i++)
	{
		persons[i]=k;
	}
}

int numbofg(int n, int m)
{
	/* if overflow then return -1;*/
	double niz=1.0, verx=1.0;
	double maxint = pow(2, sizeof(int) * 8)/2 - 1;
	double maxdouble = pow(2, sizeof(double) * 8)/2 - 1;
	double maxnumb = maxdouble/100;
	double result;
	/*   nog = (n-(m-1))*...*(n-1)*n)/m!  verx = (n-(m-1))*...*(n-1)*n   niz = m! */
	int i;
	double forniz, forverx;
	double prev=-1.0;
	for (i=0;i<m;i++)
	{
		forniz  = i+1;
		forverx = n-i;
		if (verx > maxnumb)
		{
			verx /= forniz;
			forniz = 1;
		}
		prev = verx;
		verx *= forverx;

		if ( niz > maxnumb )
		{
			verx /= niz;
			niz = 1;
		}
		niz *= forniz;
		if (verx < prev)  /* overflow*/
			return -1;
	}
	/*printf("verx/miz=%f",verx/niz);*/
	result = verx/niz;
	if(result < maxint)
		return (int)result;
	else
		return -1 ;
}

/*calculate CandArray size of current process, last proc p-1 otlichaetsya*/
int startsize(int p, int nog, int my_rank)
{
	int size, drob = nog%p;
	size = nog/p;
	if (drob > 0.6)
		 size++;
	if(my_rank == p-1)
		size = nog - (p-1)*size;
	return size;
}

/*output: return the first index of current process*/
int startindex(int p, int nog, int my_rank)
{
	int result;
	result = startsize(p,nog, 0); /* size otlichaetsya u poslednego proc */
	result *= my_rank;
	return result; /*output: return the first index of current process*/
}

/* index must be  0 <= index < nog, abnormal index return -999*/
int findbyindex(int n, int m, int index, int *v)
{
	/*having index find represents in group and place it in v[m] array*/
	int csum = -1; /*= 0;   /*    v  [3] [2] [1] [0]   with vind = 3, 2, 1, 0*/
	int vind = m-1; /*  vind= 3 | 2 | 1 | 0 |*/
	int p = 0;      /*    p = 3   5   6   8*/
	int cp;
	/*v = new int[m];
	//          vind        (n-p-1)!
	//cp=C      = ---------------------
	//    n-p-1   vind!*((n-p-1)-vind)!
	//test printf("%d\n",index);*/
	if(index >= numbofg(n,m) || index < 0)
		return -999; /*abnormal index, index not in range*/

	while (vind >= 0)
	{
		cp = numbofg(n-p-1,vind);
		/*if (p==0) cp--;  /* if p = 0 the counting begins with 0*/
		if ( (csum < index) && (index <= (cp + csum)) )
		{
			v[vind] = p;
			vind--;
		}
		else
		{
			csum += cp;
		}
		p++;
		/*test printf("%d\r",csum);*/

	}
	return 1;

}



void initmgroup(struct mgroup *candArray, int n, int m, int p, int my_rank)
{

	/*candArray[i].members[m-1],[m-2]......[0]     [4],[3] [2] [1] [0]    for m=4
	/* first group i = 0 ->  0    1   234...(m-1)   0   1   2   3   4*/
	int count, shift,temp, min, max, i, j, nog, startind, size;
	nog = numbofg(n, m);
	startind = startindex(p, nog, my_rank);
	size = startsize(p, nog, my_rank);

	candArray[0].members = (int *)malloc(m * sizeof(int));
	candArray[0].rait = 1;

	findbyindex(n, m, startind, candArray[0].members);
	/*zdes vnachale realindex = relindex*/
	for(i=1;i<size;i++)
	{
		candArray[i].members = (int *)malloc(m * sizeof(int));
		shift = 1;
		for(j=0;j<m;j++)
```

```
        {
            temp = candArray[i-1].members[j] + shift;
            max = n - j - 1;
            if (temp <= max)
            {
                candArray[i].members[j]=temp;
                shift = 0;
            }
            else
            {
                for(count=1;count<n-j;count++)
                {
                    min = candArray[i-1].members[j+count] + count + 1;
                    if (min <= max)
                        break;
                }


                candArray[i].members[j] = min>max?max:min;
                shift = 1;

            }

        }
        candArray[i].rait=1;
    }

}

void filemgroup(FILE *outFile, struct mgroup *candArray, int m, int nog)
{
    //write whole candArray[nog] in file  in absindexes
    int i, j;
    for(i=0;i < nog; i++)
    {
        if(candArray[i].members != NULL)
        {

            fprintf(outFile, "\tcandArray[%d]\t=\t", i);
            for(j=m-1;j>=0;j--)
            {
                fprintf(outFile, "%d ", candArray[i].members[j]);
            }
            fprintf(outFile, "\n");
        }
    }
```

```
}

void changepersonsdayoff(int *persons, int n, int *v, int m)
{
	int i, j, temp;
	for(i = 0; i<n; i++)
	{
		temp = 0;
		/* viyasnayem etot person est v gruppe? esli est to day off = 0, else ++*/
		for(j=m-1;j>=0;j--)
		{
			if(i == v[j])
			{
				temp++;
				break;
			}
		}
		if(temp>0)
			persons[i]=0;
		else
			persons[i]++; /* esli ego net v gruppe to eshe odin den etot person ne
rabotal*/
	}
}

/* NULL		0 1	NULL	3 4 NULL	5 6 NULL	7  8  9    relindex 9 ->
/* 0	1 2	3		4 5 6		7 8 9		10 11 12	realindex = 12 */
int findrealindex(struct mgroup *candArray, int relindex)
{
	int realindex=-1, i;
	/* i is relative index*/
	for(i=0;i<= relindex;i++)
	{
		realindex++;
		while(candArray[realindex].members == NULL)
		{
			realindex++;
		}
	}
	return realindex;
}

/*show how many candArray groups has maxrait realsize = startsize(p, nog, my_rank)*/
int maxrisize(struct mgroup *candArray, int realsize, int maxrait)
{
	int i, result = 0;

	if (realsize == 0 || maxrait == 0)
		return result;
	/*i is relative index*/
	for(i=0;i<realsize;i++)
	{
		if(candArray[i].rait==maxrait && candArray[i].members != NULL)
		{
			result++;
		}

	}
	return result;
}

/*0<=relmaxrindex<size of CandArray; 0<=realindex<startsize() index of CandArray real size*/
int maxrrealindex(struct mgroup *candArray, int relmaxrindex, int maxrait)
{
	int realindex=-1, i;
	/*i is relative index*/
	for(i=0;i<= relmaxrindex;i++)
	{
		realindex++;
		while(candArray[realindex].rait != maxrait)
		{
			realindex++;
		}
	}
	return realindex;
}

void freemgrouprec(struct mgroup *candArray, int realindex) /*free group in absindexR */
{
	free(candArray[realindex].members);
	candArray[realindex].members=NULL;
	candArray[realindex].rait = -1;
}

/*size is quontity of not NULL members;*/
void changerait(struct mgroup *candArray, int size, int k, int *persons, int n, int m)
{
	/*6. Increase value of thous groups, where all members had al least k day offs
	// (t.e. te personi kotorie uzhe otdoxnuli polozhennie denoff i teper prostaivaut zhdut
dezhurstva)
	//   and rait of other grpoups became zero
	// fi { 0 , 1, X, 3, X, X, 6, 7, X, X}  fi=factilndx ca=candArray mb=members size= 5
	//  i   0   1     2       3  4 < size=5
	//################Podozrenie chto oshibka zdes
	//absindex = startindex;*/
	int relind, i, temp, j;
	for(relind= 0; relind<size; relind++)
	{
		i = findrealindex(candArray, relind); /*realindex*/
		if( candArray[i].members != NULL)
		{
			/* esli xot odin person iz group imeet dni prostoya < k(dayoff), to on eshe
ne naotdixalsya
			// to poetomu budet temp=0, t.e. eta grupa eshe ne budet poka
dezhurit(rabotat)*/
			temp = 1;
			for(j=0;j<m;j++)
			{
				if(persons[candArray[i].members[j]] < k)
				{
					temp = 0;
					break;
				}
			}
			if(temp)
				candArray[i].rait++;
			else
				candArray[i].rait=0;
		}

	}

}



/*show how many candArray has not NULL members*/
int notnullsize(struct mgroup *candArray, int realsize)
{
	int i, result = 0;

	/*i is relative index*/
	for(i=0;i<realsize;i++)
	{
		if(candArray[i].members != NULL)
		{
			result++;
		}
```

```
          }
          return result;
}

int sizeforgivenint(int *procs, int fullsize, int givenint)
{
          int i, result = 0;

          if (givenint == 0)
                    return result;
          /*i is real index*/
          for(i=0;i<fullsize;i++)
          {
                    if(procs[i] == givenint)
                    {
                              result++;
                    }

          }
          return result;
}

/* !!!realindex of relindex */
int realindex(int *procs, int relcproc, int givenint)
{
   int i, result=-1;
   for(i=0;i<=relcproc;i++)
   {
                    result++;
      while ( procs[result] != givenint)
          result++;
   }
          return result;
}


void makeready(int *procs, int p)
{
          // all not ready (0) make ready (1)
          int i;
          for(i=0;i<p;i++)
          {
                    if(procs[i] == 0)
                              procs[i] = 1;
          }
```

```
}

int findmaxrait(struct mgroup *candArray, int size)
{
        int i, temp, maxr = -1; /*, realindex;*/
        /*i is relative index*/
        for(i=0; i < size; i++)
        {
                /*realindex = findrealindex(candArray, i);*/
                temp = candArray[i].rait;
                if(maxr < temp)
                        maxr = temp;
        }
        return maxr;
}

void printfreegrp(FILE *outFile, struct mgroup candArray, int count, int proc,int realindexR,int
absindexR, int m)
{
        int i;
        fprintf(outFile, "Day %d candArproc[%d][%d %d]=", count, proc, realindexR,
absindexR);
        if(candArray.members ==      NULL)
        {
                fprintf(outFile, "The candArray.members = NULL");
                fflush(outFile);
                exit(1);
                return;
        }


        for(i=m-1;i>=0;i--)
        {
                fprintf(outFile,"%d ", candArray.members[i]);
        }
        fprintf(outFile,"->%d\n", candArray.rait);
}

/*start = "[", end = "]=", delim = "\t"*/
void printit(struct mgroup *candArray, int n, int m, int p, int my_rank, char *start, char *end,
char *delim)
{
        int i, j, nog, size;
        int startind;
        nog = numbofg(n, m);
        startind = startindex(p, nog, my_rank);
        size = startsize(p, nog, my_rank);
        for (i= 0;i<size;i++)
        {
                if( candArray[i].members != NULL)
                {
                        printf(start);
                        printf("%d %d",i, i + startind);
                        printf(end);
                        for(j=m-1;j>=0;j--)
                                printf("%d ",candArray[i].members[j]);
                        printf("->%d",candArray[i].rait);
                        printf(delim);
                }
                else
                {
                        printf(start);
                        printf("%d %d",i, i + startind);
                        printf(end);
                        printf("NULL");
                        printf("->%d",candArray[i].rait);
                        printf(delim);
                }
        }
        printf("\n");
}

/*start = "[", end = "]=", delim = "\t"*/
void writeinfile(FILE* outFile, struct mgroup *candArray, int n, int m, int p, int my_rank, char
*start, char *end, char *delim)
{
        int i, j, size, nog, startind;
        nog = numbofg(n, m);
        startind= startindex(p, nog, my_rank);
        size = startsize(p, nog, my_rank);
        /*i is realindex*/
        for (i=0;i<size;i++)
        {
                if( candArray[i].members != NULL)
                {
                        /*outFile << start << i << end;*/
                        fprintf(outFile, "%s%d %d%s", start, i, i + startind,end);
                        for(j=m-1;j>=0;j--)
                                fprintf(outFile, "%d ", candArray[i].members[j]);
                        fprintf(outFile,"->%d", candArray[i].rait);
                        fprintf(outFile,"%s", delim);
                }
```

```
				else
				{
					fprintf(outFile, "%s%d %d%s", start, i, i + startind,end);
					fprintf(outFile, "NULL");
					fprintf(outFile,"->%d", candArray[i].rait);
					fprintf(outFile,"%s", delim);
				}
				if((i+1)%5==0)
					fprintf(outFile, "\n");
			}
		}
```

Appendix 7. Serial Duties Scheduling listing

The code below is the serial implementation of the Duties Scheduling artificially divided into "processes". It was done for debugging purposes.

```
#include "my_funct.c"  // see Appendix 6

int main()
{
        /*Declaration*/
        time_t t;
        int j, p, my_rank, n, m, k, nog, cproc, fullsize, maxrait, maxraitsize, relmaxrindex, i;
        int absindexR, sizeR, realindexR, fin=1, count = 0, relcproc, cprocreadysize, procsize;
        int *procs = NULL, **persons = NULL, *v = NULL;
        /*struct mgroup *candArray  = NULL;*/
        struct mgroup **candArproc = NULL;
        FILE *outFile;
        FILE *output;
        outFile = fopen("output.txt","w+");
        output  = fopen("schedule.txt","w+");

        srand((unsigned) time(&t));
        /*Input by user*/
        printf("Insert p n m k: ");
        scanf("%d %d %d %d", &p, &n, &m, &k);

        /*Initializaton procs, v, persons[], candArproc[]*/
        procs = (int *)malloc(p*sizeof(int)); /* index iz process, el-t 1 ready, 0 not ready, -1
empty*/
        persons = (int **)malloc(p*sizeof(int *));
        candArproc = (struct mgroup **)malloc(p * sizeof(struct mgroup *));
        nog = numbofg(n, m);
        printf("nog=%d", nog);
        fprintf(outFile,"p=%d, n=%d, m=%d, k=%d, nog=%d\n", p, n, m, k, nog);
        fprintf(output,"p=%d, n=%d, m=%d, k=%d, nog=%d\nDay N: cAr[absindexR] =
groupmembers\n", p, n, m, k, nog);


        printf("\n");

        initperson(procs, p, 1); /*initialize all persons[] with 1*/

        for(my_rank = 0; my_rank < p; my_rank++)
        {
```

```
			persons[my_rank] = (int *)malloc(n*sizeof(int));
			candArproc[my_rank] = (struct mgroup *)malloc(startsize(p, nog, my_rank) *
sizeof(struct mgroup));
			initperson(persons[my_rank], n, k);
			initmgroup(candArproc[my_rank], n, m, p, my_rank);
		}

		printinfo(outFile, candArproc, n, m, p, nog, my_rank, procs, persons);

		/*I time we choose absindex from 0 process*/
		absindexR=rand( )%nog; /*I choosed in 0 proc*/
		//printf("n=%d, m=%d, ", n, m);
		v = (int *)malloc(m * sizeof(int));
		sizeR = startsize(p,nog, 0); /* where last process  differ*/
		cproc = absindexR/sizeR;
		realindexR = absindexR%sizeR;
		//printf("\n");
		my_rank=0;
		count = 0;
		while(count <= nog)
		{
			if(absindexR >= 0)
			{

				findbyindex(n, m, absindexR, v);
				fprintf(outFile, "####################Cycle %d###########\n",
count);
				for(my_rank = 0; my_rank < p; my_rank++)
				{
					if(my_rank == 0)
					{
						count++;
						//fprintf(output, "Day %d: [absindexR=%d] = ", count,
absindexR );
						fprintf(output, "Day %d: candArray[%d] = ", count,
absindexR );
						for(j=m-1;j>=0;j--)
							fprintf(output, "%d ", v[j]);
						fprintf(output, "\n");
					}
					if(my_rank == cproc)
					{
						printfreegrp(outFile, candArproc[cproc][realindexR],
count, cproc, realindexR, absindexR, m);
						freemgrouprec(candArproc[my_rank], realindexR); /*free
group in absindexR */
```

```
				}
				/*in persons  kto est v grupe v[ ], ostalnie ++*/
				changepersonsdayoff(persons[my_rank], n, v, m);
				changerait(candArproc[my_rank],
notnullsize(candArproc[my_rank], startsize(p, nog, my_rank)), k, persons[my_rank], n, m);
			}
			printinfo(outFile, candArproc, n, m, p, nog, my_rank, procs, persons);
		}
		my_rank = 0;

		cprocreadysize = sizeforgivenint(procs, p, 1); /*find procs size for
givenint=1(ready)*/

		if(cprocreadysize > 0)
			relcproc = rand( ) % cprocreadysize;
		else
		{
			if(count == nog)
				fprintf(outFile, "All right. All possible combinations are
written.\n");

			fprintf(outFile, "There are no ready processes cprocreadysize = 0\n");
			printf("\n");
			exit(0);
		}
		cproc = realindex(procs, relcproc, 1); /* 1 means ready*/
		my_rank=cproc;
		fullsize = startsize(p, nog, my_rank);
		maxrait=findmaxrait(candArproc[my_rank], fullsize);
		fprintf(outFile, "cprocreadysize=%d; cproc=%d; maxrait=%d", cprocreadysize,
cproc, maxrait);
		if(maxrait > 0)
		{
			maxraitsize = maxrisize( candArproc [my_rank] , fullsize, maxrait);
			relmaxrindex = rand( )%maxraitsize;
			realindexR = maxrrealindex(candArproc[my_rank], relmaxrindex,
maxrait);
			absindexR = startindex(p, nog, my_rank) + realindexR;
			fprintf(outFile, "\nmaxraitsize=%d; relmaxrindex=%d;
absindexR=%d\n",maxraitsize, relmaxrindex, absindexR);

		}
		else
		{
			if(maxrait == 0)
				absindexR = -1;
```

```
                else
                        absindexR = -100;
                fprintf(outFile, "DON'T CHOOSE, because absindexR=%d\n",
absindexR);
        }
        my_rank = 0;
        if(absindexR >=0)
        {
                makeready(procs, p);
        }
        else if(absindexR == -1)
        {
                procs[cproc] = 0;
        }
        else
        {
                /* absindexR = -100 */
                procs[cproc] = -1;
        }
        procsize = sizeforgivenint(procs, fullsize, 1); /* size of ready processes */
        if(procsize == 0)
        {
                /*Only for serial program, we controlling how many not NULL
                  candArrproc[my_rank 0,1,2,..p-1][sizeR] are there
                fprintf(output, "candArrproc "); */
                /* Find how many groups not included */
                count=0;
                for(i=0; i<p; i++)
                {
                        for(j=0;j<startsize(p,nog,i);j++)
                        {
                                if(candArproc[i][j].members != NULL)
                                {
                                        count++;
                                }
                        }
                }
                fprintf(output, "*********p=%d, n=%d, m=%d, k=%d,
nog=%d************\n", p, n, m, k, nog);
                if(count != 0)
                {
                        fprintf(output, "%d groups are not included:\n", count);
                        for(i=0; i<p; i++)
                        {
                                for(j=0;j<startsize(p,nog,i);j++)
                                {
```

```
                                                        if(candArproc[i][j].members != NULL)
                                                        {
                                                                fprintf(output, "[%d]", i);
                                                                fprintf(output, "[%d]=", j);
                                                                for(k=0;k<m;k++)
                                                                        fprintf(output,"%d ",
candArproc[i][j].members[k]);
                                                                fprintf(output,"->%d\n",
candArproc[i][j].rait);
                                                        }
                                                }
                                                /*fprintf(output,"\n");*/
                                        }
                                        fprintf(output,"\n");
                                }
                                /*end for serial program part*/
                                count=nog;
                        }
                        printf("\r                ");
                        printf("\r%d", count);
                }
                printf("\n");
        return 0;
}
```

Appendix 8. Parallel Duties Scheduling listing

```
#include "my_funct.c"  // see Appendix 6
#include "mpi.h"

int main(int argc, char** argv)
{
        /*Declaration*/
        time_t t,t1, t2;
        int kukucount;
        int i, j, p, my_rank, nog, cproc, fullsize, maxrait, maxraitsize, relmaxrindex, n, m, k;
        int nmk[3];
        int absindexR, sizeR, realindexR, count, relcproc, cprocreadysize, procsize;
        int *procs = NULL, *personstatus = NULL, *v = NULL;
        struct mgroup *candArray = NULL;
        FILE *outFile = NULL;
        FILE *output = NULL;

        srand((unsigned) time(&t));
        //MPI_Status status;
        MPI_Init(&argc, &argv); // let the system do what int needs to a start up MPI
        MPI_Comm_rank(MPI_COMM_WORLD, &my_rank); //Get my process rank
        MPI_Comm_size(MPI_COMM_WORLD, &p); //Find out how many processes are
being used

        cproc = 0;
        count = 0;
        /*Input by user*/
        if(my_rank==0)
        {
                fflush(stdout);
                output  = fopen("schedule.txt","w+");
                outFile = fopen("candArray.txt","w+");
                printf("Insert n m k: ");
                fflush(stdout);
                scanf("%d %d %d", &nmk[0], &nmk[1], &nmk[2]);
        }

        MPI_Bcast(nmk, 3, MPI_INT, 0, MPI_COMM_WORLD);
        n=nmk[0];
        m=nmk[1];
        k=nmk[2];
        fflush(stdout);
        nog = numbofg(n, m);
        MPI_Barrier( MPI_COMM_WORLD);
```

```
	if(my_rank == 0)
	{
		printf("n=%d, m=%d, k=%d, nog=%d\n", n, m, k, nog);
		fflush(stdout);
		fprintf(output, "n=%d, m=%d, k=%d, nog=%d\n", n, m, k, nog);
		fflush(output);
	}
	t1 = time(NULL);
	candArray = (struct mgroup *)malloc(startsize(p, nog, my_rank) * sizeof(struct
mgroup));
	initmgroup(candArray, n, m, p, my_rank);
	//if(my_rank == 0)
	{
		procs = (int *)malloc(p*sizeof(int)); /* index iz process, el-t 1 ready, 0 not ready,
-1 empty*/
		personstatus = (int *)malloc(n*sizeof(int));
		initperson(procs, p, 1); /*initialize all procs[] with ready state (1) */
		initperson(personstatus, n, k);
		/* I time we choose absindex from 0 process */
		absindexR = rand( )%nog; /* I choosed in 0 proc */
	}
	MPI_Barrier( MPI_COMM_WORLD);
	MPI_Bcast(&absindexR, 1, MPI_INT, 0, MPI_COMM_WORLD);
	MPI_Barrier( MPI_COMM_WORLD);
	v = (int *)malloc(m * sizeof(int));
	sizeR = startsize(p,nog, 0); /* where last process  differ*/
	cproc = absindexR/sizeR;
	realindexR = absindexR%sizeR;
	count = 0;
	while (cproc >= 0)
	{
		findbyindex(n, m, absindexR, v);
		if(absindexR >=0)
		{
			makeready(procs, p);
			if(my_rank == 0)
			{
				count++;
				printf("\r          \r%d <= %d", count, nog);
				fflush(stdout);
				fprintf(output, "Day %d: candArray[%d] = ", count, absindexR );
				fflush(output);
				for(j=m-1;j>=0;j--)
				{
					fprintf(output, "%d ", v[j]);
					fflush(output);
```

```
                    }
                    fprintf(output, "\n");
                    fflush(output);
                }
                if(my_rank == cproc)
                {
                    freemgrouprec(candArray, realindexR); //free group in absindexR
                }
                //0 for persons  who is in group v[ ], the rest ++
                changepersonsdayoff(personstatus, n, v, m);
                changerait(candArray, notnullsize(candArray, startsize(p, nog, my_rank)),
k, personstatus, n, m);
        }
        else
        {
                if(my_rank == 0)
                {
                    if(absindexR == -1)
                    {
                        procs[cproc] = 0;
                    }
                    else
                    {
                        procs[cproc] = -1;
                    }
                    procsize = sizeforgivenint(procs, p, 1);
                    if(procsize == 0)
                    {
                        cproc = -99;
                    }
                }
        }
        MPI_Barrier( MPI_COMM_WORLD);
        if(my_rank == 0 && cproc >= 0)
        {
                cprocreadysize = sizeforgivenint(procs, p, 1); //find procs size for
givenint=1(ready)
                relcproc = rand( ) % cprocreadysize;
                cproc = realindex(procs, relcproc, 1); // 1 means ready
        }

        MPI_Barrier( MPI_COMM_WORLD);
        MPI_Bcast(&cproc, 1, MPI_INT, 0, MPI_COMM_WORLD);
        MPI_Barrier( MPI_COMM_WORLD);

        if(my_rank==cproc)
```

```
			{
				fullsize = startsize(p, nog, my_rank); // fullsize of candArray of the current
process
				maxrait=findmaxrait(candArray, fullsize);
				if(maxrait > 0)
				{
					maxraitsize = maxrisize( candArray, fullsize, maxrait);
					relmaxrindex = rand( )%maxraitsize;
					realindexR = maxrrealindex(candArray, relmaxrindex, maxrait);
					absindexR = startindex(p, nog, my_rank) + realindexR;
				}
				else
				{
					if(maxrait == 0)
						absindexR = -1;
					else
						absindexR = -100;
				}

			}

			if(cproc >= 0)
			{
				MPI_Bcast(&absindexR, 1, MPI_INT, cproc, MPI_COMM_WORLD);
			}
		}
		if(my_rank==0)
		{
			if(nog - count > 0)
			{
				//printf("\nnog=%d, count=%d!!!!\n", nog, count);
				fprintf(output, "***************\n%d group(s) was(were) not
included\n", nog - count);
				fflush(output);
				//filemgroup(output, candArray, m, nog);
				printf("\n**********\n%d group(s) not included", nog - count);
				fflush(stdout);
			}
			t2 = time(NULL);
			printf("\nThe parallel program worked %ld secs\n",t2-t1);
			fprintf(output, "The parallel program wint p processes worked %ld secs\n", p, t2-t1);
		}
   MPI_Finalize();
		fflush(stdout);
   return 0;
}
```

Appendix 9. Test results

```
Duties Scheduling test
18.06.2005 2comp: Wg_lab9b( 9B_22, 9B_14)
***************
serial on 9B_22
n=20, m=3, k=1, nog=1140
2 group(s) not included
The parallel program worked 6 secs
************************************
2 process 1 comp
n=20, m=3, k=1, nog=1140
all members are included
The parallel program worked 8 secs
*************************
2 proces 2 comp
n=20, m=3, k=1, nog=1140
2 group(s) not included
The parallel program worked 6 secs
***************************
2 process 2 comp
n=30, m=3, k=1, nog=4060
4 group(s) not included
The parallel program worked 59 secs
*************************************
serial on 9B_22
n=30, m=3, k=1, nog=4060
all members are included
The parallel program worked 196 secs
*******************************
serial on 9B_22
n=30, m=3, k=1, nog=4060
1 group(s) not included
The parallel program worked 107 secs
****************************
4 process 1 comp
n=30, m=3, k=1, nog=4060
1 group(s) not included
The parallel program worked 91 secs
****************************
4 process 2 comp
n=30, m=3, k=1, nog=4060
2 group(s) not included
The parallel program worked 55 secs
*****************
```

```
*****************
cis294 as user
25.06.2005 12comp:Wg_lab9b( 9B_22), Pub9d_group(LAB9D_01-LAB9D_05,LAB9D_08,
LAB9D_11, LAB9D_14, LAB9D_16, LAB9D_18,LAB9D_21)
**************
12 process, 12 computers
n=30, m=3, k=1, nog=4060
19 group(s) not included
The parallel program worked 34 secs
**************
12 process, 12 computers
n=30, m=3, k=1, nog=4060
4 group(s) not included
The parallel program worked 34 secs
***********************************
24 process, 12 computers
n=30, m=3, k=1, nog=4060
5 group(s) not included
The parallel program worked 63 secs
***********************************
6 processes, 12 computers
n=30, m=3, k=1, nog=4060
8 group(s) not included
The parallel program worked 33 secs
**********************************
3 processes, 12 computers
n=30, m=3, k=1, nog=4060
2 group(s) not included
The parallel program worked 65 secs
***********************************
1 process, 12 computers
n=30, m=3, k=1, nog=4060
1 group(s) not included
The parallel program worked 190 secs
***********************************
1 process, 12 computers
n=30, m=4, k=1, nog=27405
24 group was calculated in 1min=60 sec (was checked 3 times),
so 27405 groups would be calculated in 68512.5 sec = 1141.875 min = 19.03125 hours
************************************
12 process, 12 computers
n=30, m=4, k=1, nog=27405
1000 was calculated in 68 sec
so 27405 groups would be calculated in 1863.5sec~31.1 min
62 group(s) not included
The parallel program worked 1347 secs
```

```
**********************************************
24 process, 12 computers
n=30, m=4, k=1, nog=27405
3000 groups was calculated in 122 sec,
so 27405 groups would be calculated in 1114.47sec=18.6min
30 group(s) not included
The parallel program worked 931 secs
************************************
36 process, 12 computers
n=30, m=4, k=1, nog=27405
4000 groups was calculated in 173 sec,
so 27405 groups would be calculated in 1185.47sec=19.8min

*******************************************************
Master Mind result

serial on 9B_22
Type digit size of your number:
7
Give grade for 4587332. Please type grade:
05
After thinking 44.9 sec the computer stop working give next message box
"Your systemi is low on virtual memory."

So programm stop working! There are no enough resources
so on one computer there are no enough resources to execute task

//Master Mind 7 digit number can't work even on 2 comp with 4 processes (low virtual memory)

12 processes 12 computers
Type digit size of your number:
7
Give grade for 8772250. Please type grade:
02
Give grade for 9055936. Please type grade:
04
Give grade for 2993115. Please type grade:
05
Give grade for 1539001. Please type grade:
22
Give grade for 3139562. Please type grade:
33
Give grade for 1369542. Please type grade:
25
Give grade for 3649521. Please type grade:
16
```

```
Give grade for 1234569. Please type grade:
70
I hardly worked and found the number in 8 step!!!

*****************************************
Chronometr
12 processes 12 computers

Type digit size of your number:
6
Give grade for 065224. Please type grade:
04                                    (Interval between Enter after 04(grad) and displaying
nuber 904156 is less then 0.8 sec)
Give grade for 904156. Please type grade:
22                                    (less then 0.2 sec)
Give grade for 694452. Please type grade:
22                                    (less then 0.1 sec)
Give grade for 912652. Please type grade:
13                                    (less then 0.1 sec)
Give grade for 127456. Please type grade:
50                                    (less then 0.1 sec)
Give grade for 126456. Please type grade:
50                                    (less then 0.1 sec)
Give grade for 123456. Please type grade:
60
I hardly worked and found the number in 7 step!!!
*******************************************************
serial on 9B_22
Type digit size of your number:
6
Give grade for 794012. Please type grade:
03                                    (less then 5.1 sec)
Give grade for 000871. Please type grade:
01                                    (less then 1.8 sec)
Give grade for 442660. Please type grade:
03                                    (less then 0.3 sec)
Give grade for 511244. Please type grade:
04                                    (less then 0.1 sec)
Give grade for 125426. Please type grade:
41                                    (less then 0.1 sec)
Give grade for 165423. Please type grade:
24                                    (less then 0.1 sec)
Give grade for 123456. Please type grade:
60
I hardly worked and found the number in 7 step!!
```