HOW TO PARALELIZE SEQUENTIAL PROGRAMS

Felician ALECU, Assistant Lecturer
Economic Informatics Department, A.S.E. Bucharest

Abstract: *The easiest way to parallelize a sequential program is to use a compiler that detects, automatically or based on the compiling directives specified by the user, the parallelism of the program and generates the parallel version with the aid of the interdependencies found in the source code. If the user decides to manually parallelize his program, he can freely decide which parts have to be parallelized and which not. There are several methods used to parallelize source code containing loops and input/output operations. Usually loops spend the most CPU time even if the code contained is very small.*

Keywords: parallel processing, parallel program, loops and I/O operations parallelization

1. INTRODUCTION

The first criterion to be considered when evaluating the performance of a parallel program is the speedup used to express how many times the parallel program works faster than the sequential one, where both programs are solving the same problem.

If a parallel program is executed on a computer with *p* processors, the highest value that can be obtained for the speedup is equal with the number of processors from the system. The maximum speedup value could be achieved in an ideal multiprocessor system where there are no communication costs and the workload of processors is balanced.

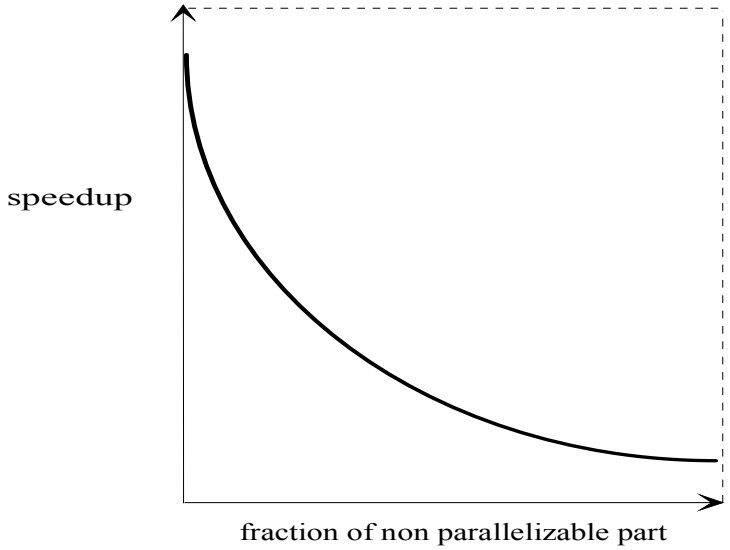
According to the Amdahl law, even in an ideal parallel system it is very difficult to obtain a speedup value equal with the number of processors because each program, in terms of running time, has a fraction α that cannot be parallelized and has to be executed sequentially by a single processor. The rest of (1 - α) will be executed in parallel. This is why the maximum speedup that could be obtained running on a parallel system a program with a fraction α that cannot be parallelized is 1/α no matter of the number of processors from the system.

For example, if a program fraction of 20% cannot be parallelized on a four processors system, the parallel execution time will be 40% of the serial execution time

and the parallel program will be only 2.5 times faster than the sequential one because 20% of the program cannot be parallelized. The maximum speedup that we can obtain is 1 / 0.2 = 5 and this means that the parallel execution time will never be shorter than 20% of the sequential execution time even in a system with infinite number of processors.

Amdahl low states that it is very important to identify the fraction of a program than cannot be parallelized and to minimize it. The next figure shows the upper bound of parallel speedup.

Figure 1. **Upper Bound of Parallel Speedup**



When running a parallel program on a real parallel system there is an overhead coming from processors load imbalance and from communication times needed for changing data between processors and for synchronization. This is why the execution time of the program will be greater than the theoretical value.

So, in order to obtain faster programs we should:

- decrease the fraction of the program that cannot be parallelized – sometimes we have to change the algorithm used by the program in order to achieve a lower value of fraction α;
- minimize the processors load imbalance;
- minimize the communications time by decreasing the amount of data transmitted or by decreasing the number of times that data is transmitted. Let's presume that a process has to send a large matrix to another process. If it will send it element by element then the communication overhead will have a very high value. Instead, if the matrix is copied in a contiguous memory area the process can send the entire memory area content at once

and the communication overhead will be acceptable because usually copying the data needs less time than communication latency.

2. PARALLELIZATION OF SEQUENTIAL PROGRAMS

The easiest way to parallelize a sequential program is to use a compiler that detects, automatically or based on the compiling directives specified by the user, the parallelism of the program and generates the parallel version with the aid of the interdependencies found in the source code. The automatically generated parallel version of the program could be executed on a parallel system. The executables generated by such compilers run in parallel using multiple threads that can communicate with each other by using the shared address space or by using the explicit message passing statements. The user does not have to be concerned about which part of the program is parallelized because the compiler will take such a decision when the automatic parallelization facility is used. On the other hand the user has a very limited control over parallelization. The capabilities of such a compiler are restricted because it is very difficult to find and to analyze the dependencies from complex programs using nested loops, procedure calls and so on.

If the user decides to manually parallelize his program, he can freely decide which parts have to be parallelized and which not. Also, the user has to explicitly define in the parallel program the desired communication mechanisms (message passing or shared memory) and synchronization methods (locks, barriers, semaphores and so on).

3. PARALLELIZING LOOPS

Parallelizing loops is one of the most important challenges because loops usually spend the most CPU time even if the code contained is very small. Usually the loop iterations are distributed over multiple processes and processors. Every process will execute just a subset of the loop iterations range and the results will be synchronized to all processors.

Using the parallelization of a loop we can minimize the load imbalance but the communication overhead will be higher as result of synchronization. The program has to perform more data transmissions in order to synchronize the data affected by the loop iterations.

The code contained by a loop involves arrays whose indices are associated with the loop variable. This is why distributing iterations means dividing arrays and assigning chunks to processes.

There are several methods used to distribute iterations among processes:

- *block distribution* - if there are p processes to be executed in parallel, the iterations will be divided into p parts. For example, if we have a loop with 100 iterations to be distributed over 5 processes, then each process will execute 20 consecutive iterations. If the number of iterations, n, is not divisible by the number of processes, p, we have to distribute among processes the reminder, r ($n = p \cdot q + r$). In such a case, the first r processes

will execute $q + 1$ iterations each and the last $p - r$ processes will execute just q iterations each. Using block distribution method it is possible to have processes assigned to no iterations at all and the workload could be unbalanced among processes. The following table shows how 14 iterations will be block-distributed to four processes.

Table 1. **Block Distribution**

Iteration	1	2	3	4	5	6	7	8	9	10	11	12	13	14
Rank	0	0	0	0	1	1	1	1	2	2	2	2	3	3

- *cyclic distribution* - the loop iterations will be assigned to processes in a round-robin fashion. The cyclic distribution method provides a more balanced workload among processes then block distribution. The following table shows an example of cyclic distribution of 11 iterations to three processes.

Table 2. **Cyclic Distribution**

Iteration	1	2	3	4	5	6	7	8	9	10	11
Rank	0	1	2	0	1	2	0	1	2	0	1

- *block-cyclic distribution* – this distribution combines block and cyclic distributions in a single method. The iterations are partitioned into equally sized blocks and these blocks are assigned to processes in a round-robin manner. The next table shows how 10 iterations are assigned to 3 processes using block-cyclic distribution method with a block size of two.

Table 3. **Block-Cyclic Distribution**

Iteration	1	2	3	4	5	6	7	8	9	10
Rank	0	0	1	1	2	2	0	0	1	1

When parallelizing loops it is necessary to consider that it is more efficient to access arrays in the same order they are stored in the memory than to access them in any other order.

4. INPUT/OUTPUT BLOCKS PARALLELIZATION

The following methods are used to parallelize source code containing I/O operations. When a program uses an input file, all processes have to read the file content concurrently. This could be done in several ways:

- *locate the input file on a shared file system* – each process reads data from the same file. The file system could be a classical one (like NFS) or could be a dedicated one (like GPFS – General Parallel File System).
- *each process uses a local copy of the input file* – the input file is copied to the each node before running the program. In this way processes can read the input file concurrently using the local copy. Copying the input file to every node needs supplementary time and disk space but the performance achieved is better than reading from a shared file system.
- *a single processor reads the input file and then broadcasts the file content to other processes* – every process will receive only the amount of data needed.

If a program generates an output file, it could be created using one of the following methods:

- *output file located on shared file system* – every process writes its data sequentially into an output file located on a shared file system. While a process is writing its data the file is locked and no other process can write in the file. In this way the file content will not be corrupt at the time when the execution of the program ends.
- *gathering of data by one process* – one of the processes gathers data from all other concurrent processes and than writes the data to the output file. In this case, the output file could be located locally or on a shared file system.

REFERENCES

1. G. Dodescu, *Operating Systems*, ASE, 1997
2. G. Dodescu, B. Oancea, M. Raceanu, *Parallel Processing*, Economic Publishing House, Bucharest, 2002
3. H. F. Jordan, H. E. Jordan, *Fundamentals of Parallel Computing*, Prentice Hall, 2002
4. J. Joseph, C. Fellenstein, *Grid Computing*, Prentice Hall, 2003
5. A. S. Tanenbaum, *Computer Networks*, Prentice Hall, 1996