Software Programs, from Sequential to Parallel

Felician ALECU, PhD, University Lecturer

Department of Economic Informatics

Academy of Economic Studies, Bucharest, Romania

E-mail: alecu[at]ase[dot]ro; Web Page: http://alecu.ase.ro

Abstract: *Several methods can be applied to parallelize a program containing input/output operations but one of the most important challenges is to parallelize loops because they usually spend the most CPU time even if the code contained is very small. A loop could by parallelized by distributing iterations among processes. Every process will execute just a subset of the loop iterations range.*

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

1. Introduction

The main reason of parallelization a sequential program is to run the program faster. 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.

2. Input/Output Blocks Parallelization

When a program uses an input file, all processes have to read the file content concurrently, so the following methods can be successfully used to parallelize the source code.

2.1. Locate the input file on a shared file system

Each process reads data from the same file (Figure 1). The file system could be a classical one (like NFS) or could be a dedicated one (like GPFS – General Parallel File System).

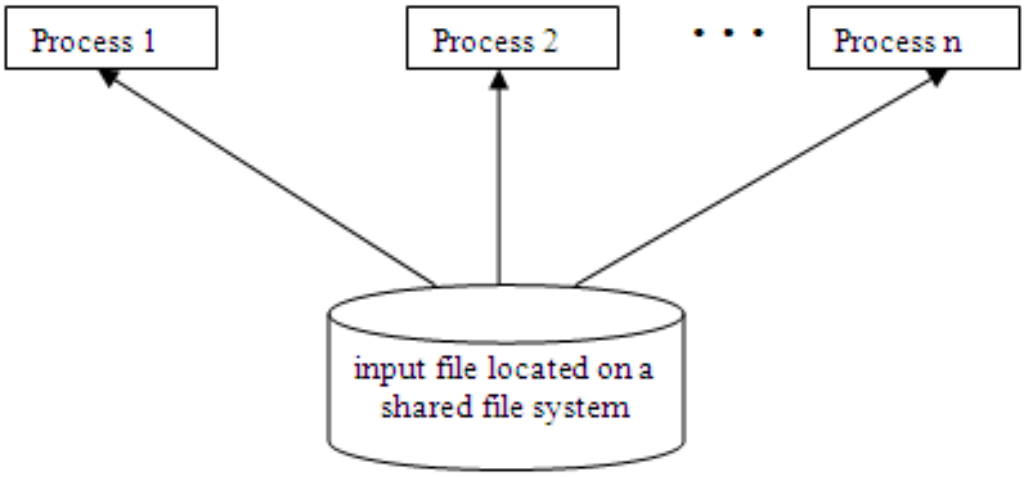

Figure 1 – Input file located on shared file system

2.2. Each process uses a local copy of the input file

The input file is copied to the each node before running the program (Figure 2). 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.

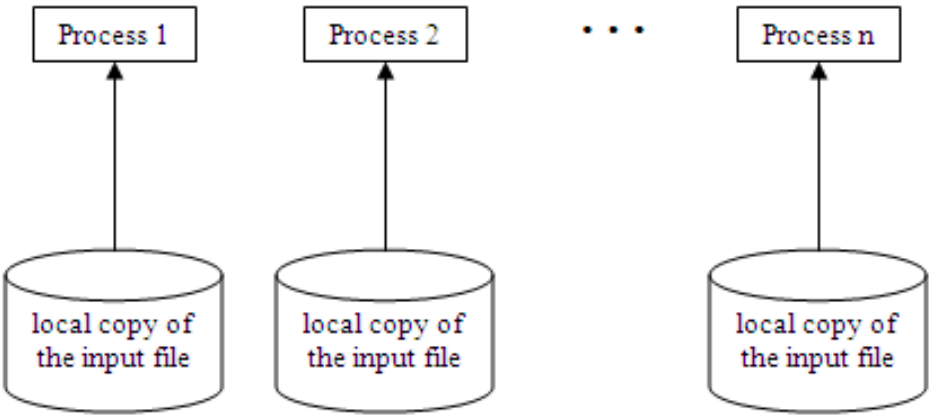

Figure 2 – Local Copy of the Input File

2.3. A single processor reads the input file and then broadcasts the file content to other processes

In this case, each process will receive only the amount of data that it needs (Figure 3).

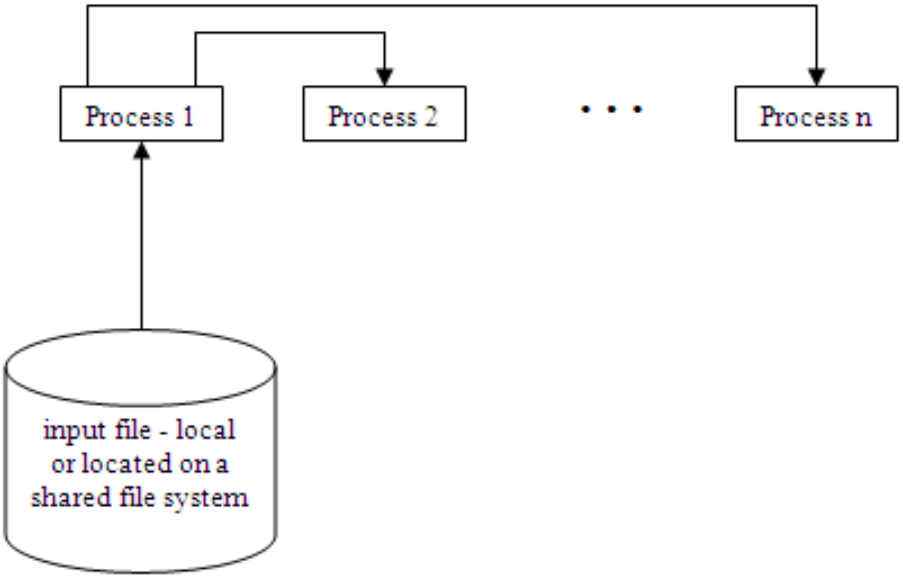

Figure 3 – One Process Reads and Distributes the Input File

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

2.4. Output file located on shared file system

Each process writes its data sequentially into an output file located on a shared file system (Figure 4). 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.

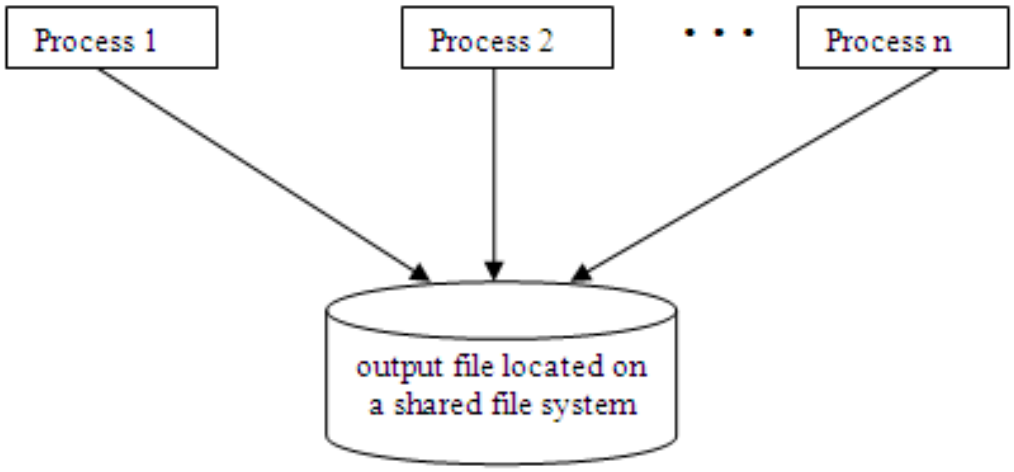

Figure 4 – Output File Located on a Shared File System

2.5. Gathering of data by one single process

A single one process gathers data from all other concurrent processes and than writes the data to the output file (Figure 5). In this case, the output file could be located locally or on a shared file system.

Locks, barriers, semaphores and other similar object can be used to synchronize the processes.

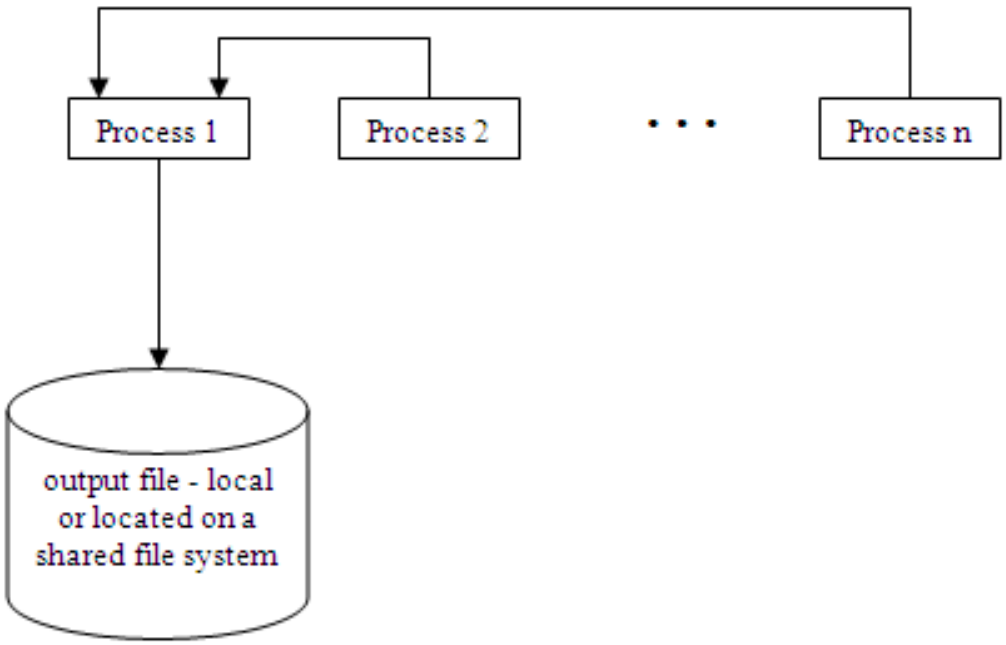

Figure 10 – Gathering of Data by One Process

3. Parallelization of 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. A loop could by parallelized by distributing iterations among processes. Every process will execute just a subset of the loop iterations range.

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. For example, if we have an array stored in a column-major order, the LOOP1 will be slower than LOOP2:

```
LOOP1
    for i = 1, n
        for j = 1, n
            a(i, j) = …
```

```
            end
        end

LOOP2
        for j = 1, n
            for i = 1, n
                a(i, j) = …
            end
        end
```

The LOOP2 could be parallelized in the following ways:

```
LOOP21
        for j = jsta, jend
            for i = 1, n
                a(i, j) = …
            end
        end

LOOP22
        for j = 1, n
            for i = ista, iend
                a(i, j) = …
            end
        end
```

The same rule has to be applied so it is preferable to use the LOOP21 because it is faster than LOOP22.

References

C. Breshears	*The Art of Concurrency: A Thread Monkey's Guide to Writing Parallel Applications*, O'Reilly Media INC., 2009
D. GROSS C.M. HARRIS	*Fundamentals of Queuing Theory*, Wiley, New York, 2003
A. SILBERSCHATZ P.B. GALVIN G. GAGNE	*Operating System Concepts*, Wiley, 8 edition, 2009
F. Alecu	*Security Benefits of Cloud Computing*, Internationa Conference on Security for Information Technoloy and Communication, November 2008, Bucharest, ASE Publishing House, Romania
G. DODESCU	*Parallel Processing*, Economic Publishing House, Bucharest, 2002