Parallelization of Sequential Programs

Alecu Felician, Pre-Assistant Lecturer,
Economic Informatics Department, A.S.E. Bucharest

Abstract

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 automatically detects the parallelism of the program and generates the parallel version. If the user decides to manually parallelize the sequential program he has to explicitly define the desired communication and synchronization mechanisms. There are several ways 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 and the code contained involves arrays whose indices are associated with the loop variable. Typically loops are parallelized by dividing arrays into blocks and distributing iterations among processes.

1. Introduction

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. In figure 1 we can see how the speedup depends on the number of processor for a few given values of fraction α.

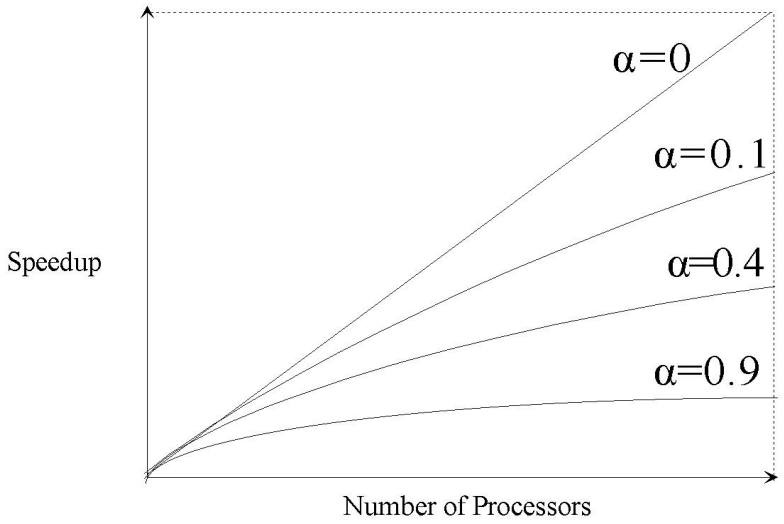

Figure 1 – Speedup depending on the number of processor for a few given values of fraction α

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 have a faster program 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 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. How to parallelize a sequential program

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).

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. In figure 2 we have an example of nested loops.

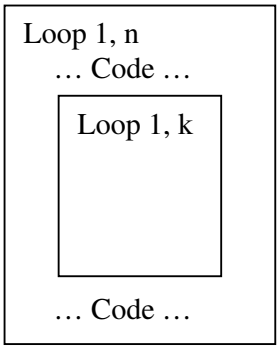

Figure 2 – Sequential Program - Nested Loops Example

Let's presume that the first loop is not time consumer but the second one is using most of the CPU time. This is why it is reasonable to parallelize the second loop by

distributing iterations among processes while the main loop remains unchanged. This is called *partial parallelization of a loop*.

If data affected by the inner loop are then referenced in the main loop, we need to synchronize data just after the end of the inner loop in order to be sure that the values accessed by the main loop are the updated one.

Figure 3 shows how the sequential program code was parallelized. The data synchronization was added just after the second loop end and the inner loop was distributed over multiple processes and processors. Every process will execute just a subset of the inner loop iterations range.

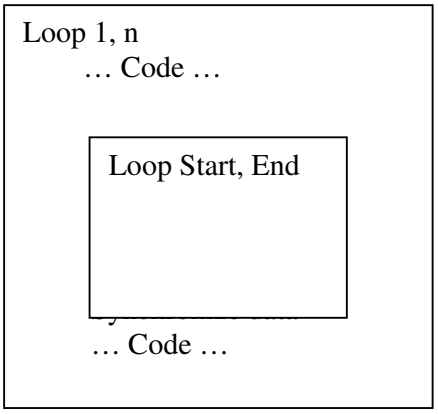

Figure 3 – Parallel Program - Nested Loops Example

Using the partial 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 inner loop.

If the inner loop is located in a function/procedure called by the main loop, we can parallelize the main program or the subroutine depending on the desired level of granularity (figure 4).

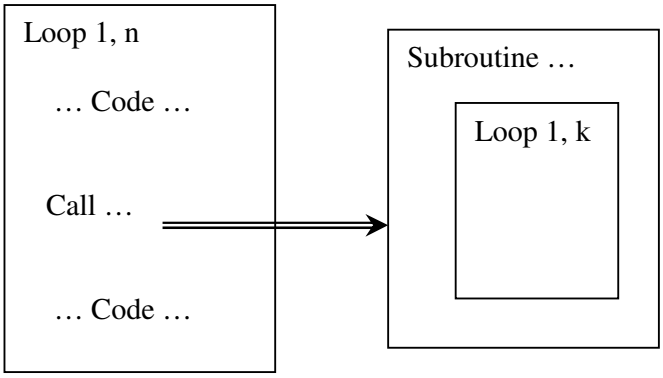

Figure 4 – Sequential Program Using Loops

If the subroutine is parallelized, the workload will be balanced due to the fine-grained parallelization.

3. 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 the following ways:

1. 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).

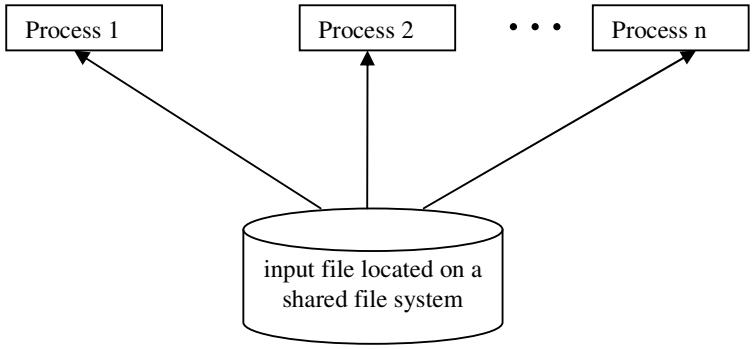

Figure 5 – Input file located on shared file system

2. 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.

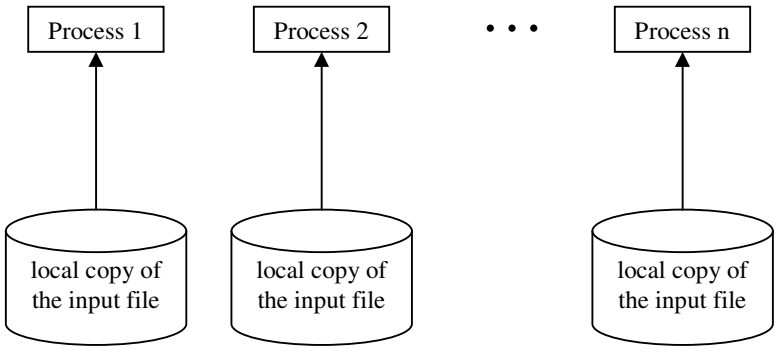

Figure 6 – Local Copy of the Input File

3. 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.

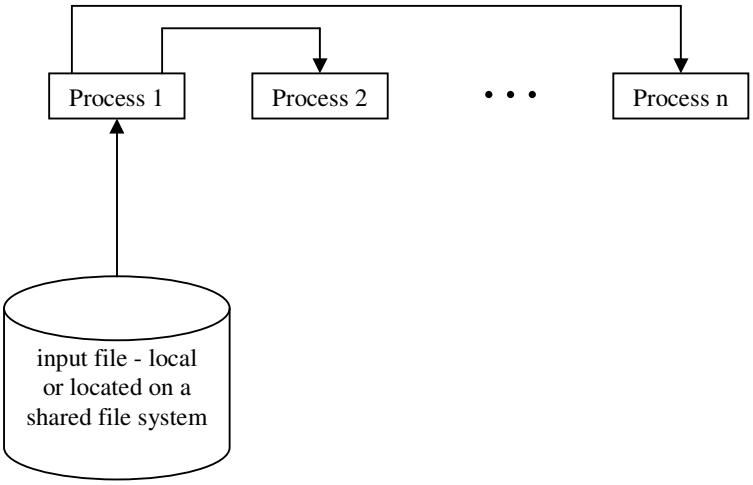

Figure 7 – One Process Reads and Distributes the Input File

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

1. 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.

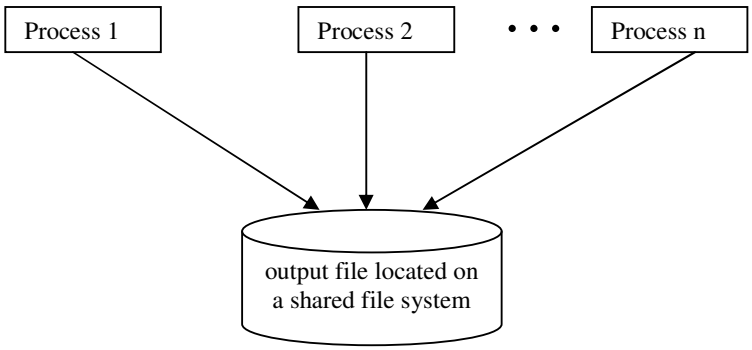

Figure 8 – Output File Located on a Shared File System

2. 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.

Locks, barriers, semaphores and other methods are used to synchronize the processes.

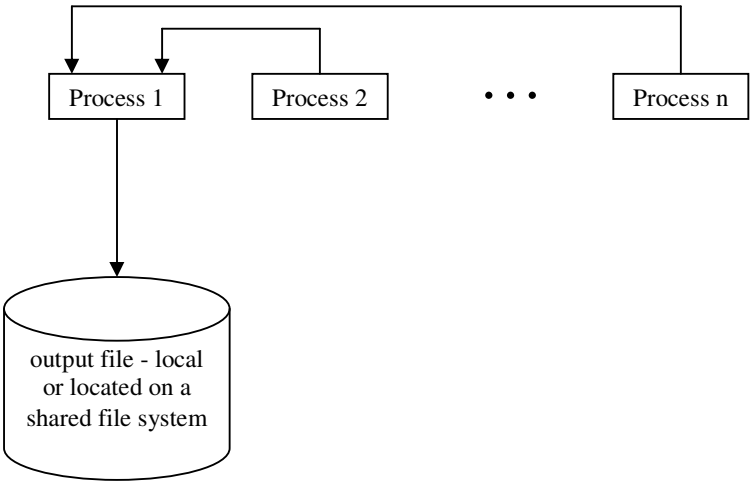

Figure 9 – Gathering of Data by One Process

4. Loops Parallelization

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.

Usually 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.

4.1. Block distribution

If there are p processes to be executed in parallel, the iterations have to be divided into p parts. If we have a loop with 100 iterations executed by 5 processes, 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. This distribution corresponds to expressing n as:

$$n = r \cdot (q+1) + (p-r) \cdot q$$

Having the loop range of iterations and the number of processes it is possible to compute the iterations range for a given rank:

```
sub range_ver1(n1, n2, p, rank, Start, End)
     q = (n2 - n1 + 1) / p
     r = MOD(n2 - n1 + 1, p)
     Start = rank * q + n1 + MIN(rank, r)
     End = Start + q - 1
     If r > rank then End = End - 1
end
```

or

```
sub range_ver2(n1, n2, p, rank, Start, End)
     q = (n2 - n1) / p + 1
     Start = MIN(rank * q + n1, n2 + 1)
     End = MIN(Start + q - 1, n2)
end
```

where

n1, n2 – lowest value, highest value of the iteration variable (IN)
p – number of processes (IN)
rank – the rank for which we want to know the range of iterations (IN)
Start, End – lowest value, highest value of the iteration variable that process rank executes (OUT)

Table 1 shows how these subroutines block-distribute 14 iterations to four processes using the above method.

Iteration	Rank	
	range_ver1	range_ver2
1	0	0
2	0	0
3	0	0
4	0	0
5	1	1
6	1	1
7	1	1
8	1	1
9	2	2
10	2	2
11	2	2
12	3	2
13	3	3
14	3	3

Table 1 – Block Distribution

Using block distribution method it is possible to have processes assigned to no iterations at all and the workload could be unbalanced among processes.

4.2. Cyclic Distribution

Cyclic distribution means that the iterations are assigned to processes in a round-robin fashion. Let's presume that we have the following loop:

```
for i = n1, n2
     computations
end
```

In such a case, the rank *rk* process will perform the following iterations:

```
for i = n1 + rk, n2, p
     computations
end
```

Table 2 we have an example of cyclic distribution of 11 iterations to three processes.

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

Table 2 – Cyclic Distribution

The cyclic distribution method provides a more balanced workload among processes then block distribution.

4.3. Block-Cyclic Distribution

The block-cyclic 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.

Dividing the following loop

```
for i = n1, n2
     computations
end
```

the rank *rk* process will perform the following iterations:

```
for ii = n1 + rk * block, n2, p * block
     for i = ii, MIN(ii + block – 1, n2)
          computations
     end
end
```

where *block* means the block size.

Table 3 shows how 10 iterations are assigned to 3 processes using block-cyclic distribution method with a block size of .two.

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

Table 3 – Block-Cyclic Distribution

4.4. Nested Loops

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

1. S. A. Wiliams, *Programming Models for Parallel Systems*, John Wiley & Sons, 1990.
2. A. Gibbons, P. Spirakis, *Lectures on Parallel Computation*, Cambridge University Press, 1993.
3. M. Zargham, *Computer Architecture – Single and Parallel Systems*, Prentice Hall, 1996.
4. G. Dodescu, *Operating Systems*, ASE Bucharest, 1997.
5. G. Dodescu, *Parallel Processing*, Ed. Economica, 2002