How to parallelize a sequential program
Alecu Felician
Assistant Lecturer, IE Department, ASE 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.

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

The speedup formula is

\[ S = \frac{T_s}{T_p} \]

where

- \( T_s \) is the execution time of the fastest sequential program for our problem;
- \( T_p \) is the execution time of the parallel program used to solve 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. In such a system, every processor needs \( T_s/p \) time units in order to complete its job and the speedup will be:

\[ S = \frac{T_s}{T_s} = p \]

We cannot have a speedup value higher than \( p \) because in such a case we can emulate all the system processors by a single sequential one obtaining a serial execution time lower than \( T_s \) but this situation is not possible because \( T_s \) is the execution time of the fastest sequential program used to solve our problem.

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 $\alpha$ that cannot be parallelized and has to be executed sequentially by a single processor. The rest of $(1 - \alpha)$ will be executed in parallel.

The parallel execution time and the speedup will be:

$$T_p = T_s \cdot \alpha + T_s \cdot (1 - \alpha) / p$$

$$S = \frac{T_s}{T_p} = \frac{T_s}{T_s \cdot \alpha + T_s \cdot (1 - \alpha) / p} = \frac{1}{\alpha + (1 - \alpha) / p} = \frac{1}{\alpha \cdot (p - 1) + 1}$$

When $p \to \infty$ we have

$$\lim_{p \to \infty} S = \frac{1}{\alpha}$$

This is why the maximum speedup that could be obtained running on a parallel system a program with a fraction $\alpha$ that cannot be parallelized is $1/\alpha$ 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 and the speedup will be:

$$T_p = T_s \cdot 0.2 + T_s \cdot 0.8 / 4 = 0.4 \cdot T_s$$

$$S = \frac{T_s}{0.4 \cdot T_s} = \frac{1}{0.4} = 2.5$$

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 says that it is very important to identify the fraction of a program than cannot be parallelized and to minimize it. The next figure (figure 1) shows the upper bound of parallel speedup.

![Figure 1 – Upper Bound of Parallel Speedup](image-url)
In figure 2 we can see how the speedup depends on the number of processor for a few given values of fraction $\alpha$.

![Graph showing speedup depending on the number of processors for different values of $\alpha$.](image)

**Figure 2 – Speedup depending on the number of processor for a few given values of fraction $\alpha$.**

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 $\alpha$;
- 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.
**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).

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 3 we have an example of nested loops.

![Figure 3 – Sequential Program - Nested Loops Example](image)

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 4 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.
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 5).

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

1. A. Inselberg, Parallel Coordinates, Springer, 2004
2. R. Wyrzykowski, Parallel Processing And Applied Mathematics, Springer, 2004