Chapter 1:

Parallel Algorithms Fundamentals

Section 4:

1.4 Performance and Scalability of Parallel Systems

The most important evaluation parameter for sequential algorithms is their execution time expressed as a function of the input size n. For parallel algorithms the execution time depends not only on the input size but on the number of processors p and the architecture, as well. That is why a parallel system has to be evaluated as a combination of the parallel algorithms and the architecture. We need appropriate metrics for their evaluation. Scalability of the algorithm on a given architecture is a measure how much the algorithm is able to achieve performance proportional to the number of processors.

1.4.1 Performance metrics

This section is devoted to the metrics that may be used for evaluation of parallel systems. In the introductory chapter we gave a very brief discussion on metrics, here we discuss it in more details.

Run Time

Parallel run time is the time that elapses from the beginning of the execution of a parallel algorithm to the moment when all processing is finished and all the processors have finished their operation. Let us denote the time for the serial algorithms by T_s and for the parallel one by T_p.

Speedup denoted by S_p, is the measure that expresses the relative benefit of using parallel systems. It may be defined as the ratio of the time to solve the problem on a single processor to the time to solve it on a parallel computer with p processors.

$$S_p = T_s / T_p$$

Several sequential algorithms may solve a problem. Not all (or none) of them may serve for parallelization. So, the question arises, for the developed parallel algorithm evaluation, which serial one has to be used. According to the commonly accepted view the *best sequential algorithm* run time is used, even if the parallel algorithm has been developed on the base of another serial origin. If we do not know which is the best one, then we have to select the best known one.

Sometimes we hear about a speedup value exceeding p. It would be possible only if each of the processors would take less time for calculations than the single (identical one) for the best algorithm. It is impossible, that each processor spends less time than T_s /p. If someone speaks on **superlinear speedup** then either he does not compare identical circumstances (e.g. faster or differently organized memory is used in the parallel processor), or he uses another definition, not the strict one stated above.

Efficiency denoted by E**:** In most cases we do not achieve speedup equal to the number of processors. When we execute parallel algorithms, the processors take not 100% of their time to the calculations strictly for the algorithm itself, but they spent some time to the so called overhead (like communication, etc.) actions. **Efficiency** is the measure expressing how much part of their running time the processors spend on 'useful' calculations. In other words it is the ratio of speedup to the number of processors.

$$E = S_p /p.$$

Cost denoted by W_p: Sometimes it is called **work.** This is the total time that p processors spend on the algorithm execution; i.e. it is the product of p and T_p. A parallel system is *cost-optimal* if the cost of solving the task is proportional to the execution time of the best sequential algorithm.

1.4.2 Granularity and Data Mapping

Best results may be achieved if the algorithm contains as many parallel partitions as possible and we assign an individual processor (of the total number of processors p_o) to each of the paralleled partitions of data. In some examples this would mean that a separate processor treats each input data item. In practice this is not feasible neither in terms of the size and complexity, nor the cost of the parallel computer system. Naturally, we have to apply fewer processors, which means that we assign larger pieces of the task to any of the processors. We **scale down** the parallel system regarding the number of the processors p, where $p < p_o$, i.e. we increase the granularity of the computer. Usually that is not a good solution if in this case we leave the parallel algorithm intact, i.e. we leave there as many parallel parts as it would be supposing that the system was not scaled down, and make the processors simulate a large number of originally assumed processors. In this way, as an average, a physical processor would simulate p_o/p virtual processors. The computation at each processor now increases by this factor.

If the number of processors does not match the granularity of the algorithm, the solution is regarded not cost-optimal. If originally the system was not cost-optimal, then by simulation of p_o / p processors by each of the physical processors we do not convert the system into a cost-optimal one. On the contrary, the scaled down system may behave in an inefficient way. We may achieve a better solution redesigning the algorithm. Usually, there is no trivial way for strategies of scaling down the algorithm itself.

If the original algorithm requires W basic steps, it could be mapped onto W processors. However, if we use less processors, then each processor takes a bigger part of the task, running some sequential program, that can be implemented in many different ways, and naturally resulting in different performance values. So, the optimal parallel algorithm can not be designed in a trivial way. Besides the above mentioned, the communication affects a lot of the performance of the system. For example, if neighboring processors communicate only, Store-and-Forward and Cut-Through routing methods give the same results. But this is not the case if processors at a longer distance have to communicate to each other.

1.4.3 Scalability

In optimal cases we achieve speedup that is equal or close to the number of processors. However, this is a very rare case. In Figure 1.4.1 we illustrate an example, using a parallel computer with hypercube topology for adding numbers. One can see that the speedup does not increase linearly with the number of processors. This situation has been formulated by **Amdahl's law.** According to it after a certain value the speedup starts to saturate and does not increase further. Amdahl distinguished among two parts of any algorithmic actions: there are parts that are sequential in their nature, and never can be parallelized, the second part of the actions may be executed concurrently.

Let us represent the total number of basic operation 100% = 1. We denote the sequential part W_s of the algorithm by *a*, and so *1-a* is the part that may be paralleled and assigned to *p* processors. Then during the execution of the sequential part only one processor will be busy, the others are idle. Thus we get the following execution and speedup value (T_1 is the time for the processor that works alone):

$$T_p = a\,T_1 + (1 - a)\,T_1/p,$$

$$S_p = T_s / T_p = T_1 / (a\,T_1 + (1 - a)\,T_1/p),$$

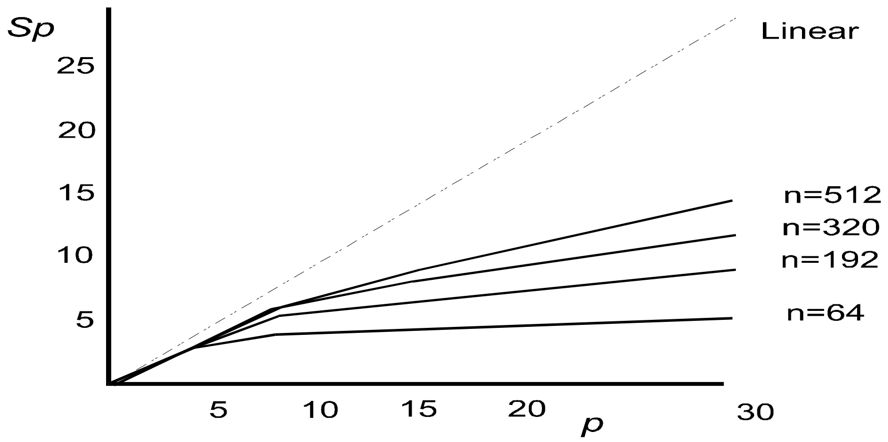
$$S_p = T_s / T_p = p / (a\,p + 1 - a) = p / (a\,(p - 1) + 1),$$

Suppose that the parallelizable part can be parallelized to the finest possible solution resulting in very big number of parts $(p \rightarrow \infty)$. The speedup in this case becomes:

$$S_p = T_s / T_p = 1/a = W / W_s.$$

The efficiency:

$$E_p = S_p / p = p / a\,.$$



Speedup (Sp) versus the number of processors (p)
for adding n numbers on a hypercube

Figure 4.1.1

Opposite to increasing the number of processors, if we expand the size of the problem (i.e. increasing the input size) then based on more data, we may increase the number of parallelizable parts; thus the saturation of the speedup curves is reached later, at a larger number of *p*.

Thus we get that growing the problem size increases the speedup value and thus the efficiency, while increasing the number of processors decreases efficiency. Hence, we may try to keep the efficiency at a certain value increasing both the problem size *n* and the number of processors *p*. The ability to maintain efficiency at a given value (close to 1) by simultaneously increasing the processors number and the problem size is a feature of a lot of parallel systems. These systems are called **scalable** systems, and **scalability** is the measure how much the speedup may be thus maintained. It characterizes the ability of the parallel system to utilize the increased number of resources. Scalable systems have linear (or close to it) speedup.

1.4.4. The Problem Size and the Overhead Function

We have already discussed the input size. The system scalability may be maintained with the processor number increasing provided, the problem size is also increasing. But the input size *n* and the problem size *W* are not the same. For parallel systems the problem size (not the input size) must be kept growing with the number of processors *p* increasing if we want to achieve constant efficiency.

The **problem size** is a measure closely related to the amount of computations. The input size is not necessarily in linear connection with the amount of computations, just regard matrix-matrix multiplication operation, where the problem size increases by the cube of the matrix size increase, while in the case of adding two matrices, the problem size squares with the input size (in traditional straightforward algorithms). So, **problem size** *(W)* may be defined as the number of basic operations in the best sequential algorithm. It is a function of the input size.

In real systems efficiency does not reach the value of 1, or speedup does not reach the value of *p*, the number of processors. Besides the communication time loss, there may be other reasons for efficiency decrease. All causes leading to efficiency losses are called **overhead** actions that are due to parallel processing.

The total **overhead**, or the **overhead function** (T_o) of a parallel system is part of the total cost (the processor-time product). This is not met in the best-known sequential algorithm. Hence this is the cost increase of algorithms in parallel systems compared to the cost of sequential algorithms. In other words it is the total time spent by a parallel system (all processors of it) in addition to the time required by the best sequential algorithm. Naturally it is a function of the problem size and number of processors: $T_o(W,p)$. The cost of problem solving in parallel systems is: $p\,T_p$, the total time spent by *p* processors. Hence:

$$T_o = p\,T_p - W$$

The basic causes for overhead are *communication cost, data rearrangement, synchronization cost*, etc.

1.4.5 The Isoefficiency function

Based on the previously discussed material, parallel execution time is a function of the problem size, overheads, and the number of processors. In other words:

$$T_p = (W + T_o(W,p))/p$$

For the speedup and efficiency, we get the following expression:

$$S_p = W/T_p = p\, W / (W + T_o(W,p))$$

$$E = S_p / p = W / (W + T_o(W,p) = 1 / (1 + T_o(W,p) / W)$$

If the ratio of the overhead function and the problem size $T_o(W,p) / W$ is maintained constant, then efficiency is kept fixed.

For different parallel systems W must be increased differently with the number of processors p in order to maintain efficiency. This is due to the fact that the overhead is a function of the problem size and the processors' number. Reordering the above equations we may get:

$$(1 - E) / E = T_o(W,p) / W$$

and denoting $E/(1-E) = K$, provided that efficiency is constantly maintained, we get:

$$W = K\, T_o(W,p).$$

Based on this equation we usually may express for a given system W as a function of p. It expresses how the problem size should be increased for a fixed efficiency. This function is called the **isoefficiency function.** A small isoefficiency function means, that small increases are needed in the problem size to keep efficiency fixed and the system scalable. On the contrary high isoefficiency function means poor scalability.

1.4.6. Cost optimality

It is commonly agreed that a parallel system is cost-optimal if the product of parallel execution time and the number of processors is proportional to the execution cost of the best sequential algorithm W. That means, the condition of cost optimality is:

$$p\, T_p = Q(W)$$

or modifying the equation:

$$W + T_o(W,p) = Q(W)$$

or replacing *Q(W) - W = O(W):*

$$T_o(W,p) = O(W) \text{ and so:}$$

$$W = W(T_o(W,p)).$$

This means that a real system is cost optimal if its overhead function does not exceed the problem size.

1.4.7. Degree of concurrency

The maximum number of parallel tasks (at any time) in a parallel algorithm is called the **degree of concurrency** *C(W)***.** It is independent of the architecture; it expresses the number of operations that may concurrently run. This means that no more processors can be employed usefully in a parallel algorithm with a given problem size.

1.4.8 Sources of Parallel Overhead

Previously we defined the overhead function. Here we will discuss the various sources for the system overhead in parallel systems. The most important sources of overhead are:

- ⁕ interprocess communication
- ⁕ load imbalance
- ⁕ extra computations.

Interprocess communications

Any parallel algorithm requires communication between the partitions executed by different processors. The time spent on data transfer among the processors is usually the major source of the overhead. If we denote t_{comm} the time a processor spends for interprocess communication, then the component that communication contributes to overhead is $p\ t_{comm}$.

Load imbalance

In many cases we can not predict the size of each subtask in algorithms. Thus it is also impossible to know how much time a given processor devotes to such a subtask. We can not equally partition the task among the processors. During the work, some processors will be kept busy while others become idle.

On the other hand, at some points the processors have to communicate to each other. If one processor reaches this point earlier, but the other is not ready for communication yet, then the first has to wait, i.e. it becomes idle until the other is ready for communication.

A parallel algorithm may have some sequential part (see Amdahl's law). In the case when one of the processors takes this part, the others are idle.

All these factors resulting in idle processors contribute to the overhead functions, since the time of the idle processors is wasted.

Extra computations

As we stated earlier, the performance of the parallel algorithm is evaluated in comparison with the best known sequential algorithm. Usually the best sequential algorithms do not allow good parallelization. In these cases we select a poorer algorithm with higher degree of concurrency, and with more **cost** (in terms of sequential execution). If *W* is the time for the fastest sequential algorithm, and *W'* is a poorer but parallelizable algorithm size, then *W*-*W'* is regarded as overhead, it expresses extra amount of work.

In sequential algorithms some intermittent results may be reused by other parts of the algorithm. In parallel case, these intermittent results are calculated and used first at a given processor, and later they may be needed for another one. In this case it may be more expensive to communicate and synchronize with the other processor, in order to pass the data for reuse, instead it is less expensive to recalculate them. This is again extra computation as to the sequential algorithm, and so it is regarded as an overhead component for the parallel algorithm.