PERIODICA POLYTECHNICA SER. EL. ENG. VOL. 46, NO. 3–4, PP. 137–149 (2002)

PERFORMANCE MODELLING OF COOPERATING TASKS IN PC CLUSTERS

Sándor JUHÁSZ and Hassan CHARAF

Department of Automation and Applied Informatics
Budapest University of Technology and Economics
H–1111 Budapest, Goldmann György tér 3, Hungary
juhasz.sandor@aut.bme.hu

Received: July 11, 2003

Abstract

Although parallel processing is a promising way of increasing the performance cost efficiently, it is important to find the balance between the potential speed-up benefits and overheads due to the organization and the increased communication. Finding the optimal distribution of co-operating tasks, minimizing overheads and maximizing execution speed are often completed based on performance prediction.

Complexity of prediction gradually increases with the number of links between the cooperating tasks. In this paper the efforts are focused on building up a performance model for a category of tasks running on clusters of workstations, where the result is expected at the same node the input was fed in, and a strong dependence between the partial solutions must be resolved to obtain the final result. In this domain we investigate the possibilities of prediction and minimization of execution time in the function of the cluster size.

To show the utility of our model, the results are demonstrated on the common, widely used area of integer sorting. Modelling the execution time of different sorting algorithms has a strong mathematical background, which enables to easily build up formulas for the expected execution times helping to determine the optimal cluster size. As a conclusion, we show the execution times of the sorting algorithm measured on a test cluster, and compare the predicted times and the measured results.

Keywords: modelling execution on cluster of workstations, execution time prediction, parallel integer sorting, and cluster performance.

1. Introduction

In the last ten years, clusters of workstations began to become possible candidates for a cheaper environment to accomplish certain computationally intensive tasks. With the advanced network technologies, connecting a number of workstations to each other is not a problem anymore. The significance of the bandwidth is also decreasing, though there is still much to do in the software domain. There is no ultimate solution for the cluster middleware, which should provide the image of a single coherent system even for a continuously changing hardware configuration [3, 9]. The design of parallel algorithms that can be executed efficiently on any general-purpose distributed systems is also a domain subject to major research efforts.

The algorithms should be efficient and scalable in a wide range of problem areas. It is often important to predict execution times and load values depending on the available configuration, or to propose a resource configuration that is optimal for the task to complete [4]. To do this, the calculation algorithm must be aware of the properties of the available resources (processing power, network speed and latency, current load, etc.). Although it seems to be impossible to provide a general prediction algorithm, good mathematical models can be proposed for certain types of applications, which provide solid foundations for optimization and predictive calculations.

This paper seeks to present a methodology, which allows predicting the execution time of a given algorithm with a known cluster configuration, and this prediction will be used for choosing a configuration that minimizes the running time. Our approach is aimed for clusters of physically separated workstations, and models a demand-driven case, where data are fed in and the result is expected at the same node. This usual condition certainly limits scalability, which implies that increasing cluster size after a limit will not increase the performance anymore, or indeed, may even result in a slower execution.

The main idea behind running time prediction is to measure some characteristics of the execution environment taking into consideration the important features of the algorithm to run. These features include effective bandwidth with the current communication pattern, constants characterizing execution speed of the algorithm on a certain kind of processing unit, or I/O and background storage speed on the cluster nodes. To simplify the model only homogeneous clusters are considered. Section 2 presents the computation, network and I/O model, which the further equations are based on.

The measures are supposed to be completed only once for an algorithm. After analyzing the algorithm with the help of the measured values and the parameters (e.g. problem size, input distribution, error bounds) some formulas can be created for the processing speed of different algorithm steps, allowing to find the performance bottleneck that determines the final running time. Section 3 presents a generic model for data processing applications and gives the formulas for execution time prediction in a cluster.

In Section 4 the previous equations are applied to an important practical application: a parallel integer sorting algorithm. Many sources handling the theory of sequential and parallel sorting algorithms are available. The different algorithms designed to be used on high-performing computers (SMPs or clusters of SMPs, for references see [1] and [7]) are difficult to adapt to clusters because this latter has a much slower communication infrastructure. As the aim was to present a running time prediction methodology, a quite simple parallel sorting algorithm had been chosen as an example, which is based on simple merging of data already sorted sequentially by the worker nodes.

In Section 5 some execution times are measured in a test cluster with different problem and cluster sizes and are compared to the predicted values. Finally, a practical example is given of how to calculate the optimal cluster size for the presented case.

2. Modelling Computation, I/O and Network Transactions

To create a simple, but well fitting model for applications running on clusters of workstations, the attention must be focused to a restricted domain of interest. Parallel sorting is a kind of data processing application where limited number of processing steps are done on a high amount of data. This will require continuous usage of local background storage devices and continuous data transfer through the network between the nodes.

In the execution environment of PC clusters a parallel algorithm can be defined as a sequence of local computations interleaved with storage device I/O and network communication steps, where all the three different kinds of operations are allowed to overlap. This approach claims for a model, where computation, network communication, and local I/O times are calculated in an orthogonal way.

Two different families of methods are known to set up *computational models*: the relation between the problem size and computational time can be determined based either on the number of operations to complete or on the memory access pattern. The former is applied in the domain of processing-intensive problems (heavy use of floating point arithmetic), the latter is applied in case of applications that deal with lots of data, on which they usually do limited amount of simple processing steps. Sorting is indeed the member of the second group, where the overall performance is dominated by memory access time rather than computational power. The formulas given for sorting execution times in Section 4 are based on counting the average number of memory accesses in function of the problem size in the algorithms.

In the *communication model* we assume homogeneous fully switched interconnection network with no congestions, where the transfer of a block containing s contiguous data units takes $(l_n + s/b_n)$ time. In the formula, l_n is the network latency, including latencies due to the physical transmission protocol to the switching elements and to the communicating operating systems, and b_n is the network bandwidth. This model is valid for the data distribution phase, where only a single source is in operation. In this case the data distribution bandwidth b_{dist} is:

$$b_{\text{dist}} = \frac{s}{t_{\text{nettransfer}}} = \frac{s}{l_n + s/b_n}. \tag{1}$$

For the data collection phase, where many sources are continuously sending data to a single node, we use an average data gathering bandwidth b_{gather}. In this case the communication bottleneck is the transfer capacity of the line between the sink node and its switch port, which means that b_{gather} can be considered independent of the number of the source nodes. Although b_{gather} is a function of the block size s used for the data transfer, for large enough values of s, b_{gather} can be considered constant, because the implementation of the transmission protocol breaks down the large blocks into smaller pieces.

The *I/O access model* is very simple, because in the domain of our investigation the background storage system is only used for reading and writing sequential files. Our experiments showed that by this usage pattern it is sufficient to model the storage system by its average bandwidth b_i (amount of transferred data in a unit of time). The value b_i characterizes the average bandwidth between the background storage device and a given task, which means that b_i does not only depend on the hardware characteristics, but also on the disk access pattern of the task, on the operational system cache strategy and even on other applications running concurrently on the same workstation.

As the measurements are carried out in a system built up of workstations with multitasking, it must be noted, that the constants in the presented models are more averages of dynamically changing values assigned to an application than a fixed attribute of the system.

3. Predicting and Optimizing Execution Time

When considering a problem of size N, there is a twofold goal to be achieved: firstly a running time prediction should be given in case of solving the problem with a cluster of p nodes, and secondly a cluster size p_{opt} should be calculated, which is optimal in the sense of minimizing execution time. During the calculations we assume that the following constraints are valid:

- The initial problem is fed in at any node (P_1) into the system, and the result is expected at the same place (outer constraint).
- The data of the problem of size N can be fitted in the total capacity of local memories present on the nodes. This avoids undesired use of background storage system by the virtual memory paging mechanisms.
- The solving algorithm is partitioned in a way that the communication between tasks is restricted to a global data distribution and a global result-gathering step, controlled by the input node P_1. The resolution of global dependencies is also done here.
- One task per node is working on the solution of the partitions mapped to that node. The dependencies between the partitions processed at the same node are resolved locally.
- At every node the same set S of resources is available for the task working on the initial problem, where S is composed of a network communication channel (latency l_n and bandwidth b_n), of a computational resource and of a dedicated average local storage I/O bandwidth b_i.
- The nodes of the cluster are connected to each other by a LAN network, which is fully switched.
- The storage I/O, network communication, and computation steps may overlap at each node.

The scalability of an algorithm requires the absence of any central, non-distributed element, which forms a potential bottleneck as the system grows. One way to achieve this is using symmetrical and balanced communication and computation [2], [9]. In our case this principle is immediately violated by requirement 1 assigning a distinguished role to the user input/output node P_1, which certainly will limit the scalability and achievable performance at the same time. We must note that this asymmetry not only limits the speed of our algorithm, but also introduces imbalanced resource consumption in the cluster (effects other tasks in a non-dedicated system). According to the assumption the algorithm solving the problem will work in a system of p nodes as follows:

Step 1 The input data is placed on an arbitrarily chosen node P_1. Other nodes are numbered from 2 to p.

Step 2 On node P_1 the input is continuously read through the storage I/O channel, split into n blocks of size s, where n is an integer multiple of p. The i^{th} $(1 \leq j \leq n)$ block is sent to node $(i \bmod p) + 1$. (P_1 also processes its part like the other nodes, because computation is allowed to overlap with network and storage I/O.)

Step 3 Each node P_j $(1 \leq j \leq p)$ will receive n/p blocks, and process these in the order of their arrival. Only one block can be processed at a time, the others are waiting in a queue, until the processing unit becomes available.

Step 4 After having received and processed all n/p blocks P_j resolves the local inter-block dependencies, and so completes a partial result of size N/p. The partial result is divided into n/p parts of size s for the global partial result gathering and global dependency resolution steps. The first part of the result is sent immediately to P_1, while the rest will be polled by P_1 as it is needed.

Step 5 As P_1 has received the first part of all the p partial results, it begins to resolve global dependencies, and polls the following parts from the other nodes as they are needed. The calculated results are written to the result file.

The aim is to determine the execution time of the above algorithm in function of the number of nodes p. **Step 1** is considered as a preparation step, which is not taken into consideration when calculating the execution time. The execution time t_{read} of **step 2** depends only on problem size N and the effective bandwidth b_{read}. As the I/O reading and the network sending, and the local processing steps overlap, t_{read} is bounded by the slower operation:

$$t_{\text{read}} = N/b_{\text{read}} = N/\min(b_{io}, b_{\text{dist}}, b_{\text{proc}}), \tag{2}$$

where b_{io} is a constant indicating physical data reading speed, and b_{dist} is the network data distribution bandwidth calculated according to formula (1).

If we suppose that the nodes are able to process the received blocks before the next one arrives, all the p nodes having an equal number of blocks to process, the completion time of **step 3** and **step 4** will be determined by the completion time of the node, which receives the last block. The last block will be processed in $t_{\text{proc}}(s)$ time, and the local dependency resolution will be run during t_{local} time for the N/p elements that one node holds.

Step 5 deals with storage of the results, where the storage bandwidth is bounded by the slowest of the three operations running overlapped: collecting data through the network, global dependency resolution, and storage I/O all done by P_1.

$$t_{\text{write}} = N/b_{\text{write}} = N/\min(b_{io}, b_{\text{gather}}, b_{\text{resolution}}). \tag{3}$$

The total execution time of the algorithm is the sum of the time functions detailed above:

$$t_{\text{total}} = t_{\text{read}} + t_{\text{proc}}(s) + t_{\text{local}}(N/p) + t_{\text{write}}. \tag{4}$$

To be able to predict execution times, we have to measure the parameters of our system (l_n, b_n, b_{io}), choose a block size s, and write the times t_{read}, t_{proc}, t_{local} and t_{write} as functions of p, and then the optimal cluster size can be derived from:

$$\frac{\mathrm{d}t_{\text{total}}(p)}{\mathrm{d}p} = 0. \tag{5}$$

4. Application for Parallel Sorting

This section presents a way of using the previous result in a parallel sorting application. Several sorting and parallel sorting algorithms have been proposed for hierarchical memory models in the literature [1], [7]. The different versions of distribution sorting cannot be used in our case, because this approach supposes that all the data are present from the first moment (these algorithms partition the elements into buckets first, and then sort the content of the individual buckets). Here the use of bottom-up algorithms is more beneficial, as the remote nodes can begin working as soon as the first block arrives. The dependencies of sorted blocks are solved by merge sorting. Although merging is very fast, the complexity of merging z already sorted sequences is $O(zN)$, it is inherently sequential, but here it is done in parallel with the likewise sequential I/O writing step.

To demonstrate the running time prediction algorithm, the following methods were chosen to sort the input sequence: while the first node reads the data sequentially and distributes them block by block, the other nodes sort the blocks as received using the quick sort algorithm of average complexity $O(N \ln N)$. The sorting time of one block of size s is:

$$t_{\text{proc}}(s) = c_q s \ln s, \tag{6}$$

where c_q is a constant independent of the problem size, depending only on the attributes of the local processing unit and on the implementation of the algorithm. This results in a processing bandwidth of:

$$b_{\text{proc}} = \frac{\text{size}}{t_{\text{proc}}} = \frac{s}{c_q s \ln s}. \tag{7}$$

After having received and sorted all the n/p blocks, each node P_j $(1 \leq j \leq p)$ executes an n/p way merge sort. To be able to forward the first packet of the result, only a sorting of size s must be done:

$$t_{\mathrm{local}}(s) = c_m s \frac{n}{p} = c_m s \frac{N/s}{p} = c_m s \frac{N}{sp}, \tag{8}$$

while the next blocks of results will be produced overlapped with global dependency resolution, which actually is a p-way merge sort done by P_1 overlapped with the storage I/O. A block of size s of the global dependency resolution is produced according to the function $t_{\mathrm{global}}(s) = c_m sp$. P_1 does its own local dependency resolution and the global dependency resolution in a parallel way, and both steps require the use of the same resources, thus they cannot overlap, so the bandwidth of the dependency resolution will be:

$$b_{\mathrm{resolution}} = \frac{\mathrm{size}}{t_{\mathrm{local}} + t_{\mathrm{global}}} = \frac{s}{c_m s \dfrac{N}{sp} + c_m sp} = \frac{1}{c_m \left(\dfrac{N}{sp} + p \right)}. \tag{9}$$

5. Experimental Results

Let us consider a problem of sorting N piece of 32 bit integers with this method. Our test cluster was built up from uniform PCs with 225 MHz processor, 64 MB RAM and a 700 MB hard disk. The nodes of the cluster were connected with a fully switched 100 Mbit Ethernet-based network. When measuring the sort constants, the values to sort were 32 bit integers between 0 and 100000 of uniform distribution. Under these conditions we measured the following values:

$$c_q = 0.066 \cdot 10^{-6}, \quad c_m = 0.04 \cdot 10^{-6}, \quad b_{io} = 0.54 \text{ M/s},$$
$$l_n = 30 \text{ ms}, \quad b_n = 1.2 \text{ M/s}, \quad b_{\mathrm{gather}} = 0.387 \text{ M/s}.$$

The unit M/s stands for million (32 bit) integer values in a second. During the measurements the same multitasking operating system was running on each computer, and the measured transfer speeds and latencies also include the operating system overhead. We found that by such conditions, b_{gather} is independent of the used block size (s) if this latter is greater than 2^{15}. If we choose the block size s to be 65536 (2^{16} integers $= 2^{18}$ bytes in every transferred block) from (1) and (7) we get:

$$b_{\mathrm{dist}} = \frac{\mathrm{size}}{t_{\mathrm{nettransfer}}} = \frac{2^{16}}{l_n + 2^{16}/b_n} = 0.775 \text{ M/s}, \tag{10}$$

$$b_{\mathrm{proc}} = \frac{\mathrm{size}}{t_{\mathrm{proc}}} = \frac{s}{c_q s \ln s} = \frac{2^{16}}{c_q \cdot 2^{16} \cdot \ln 2^{16}} = 1.366 \text{ M/s}. \tag{11}$$

The above results indicate that the $b_{io} = 0.54$ M/s will be the bottleneck limiting the reading time:

$$t_{\text{read}} = \frac{N}{\min(b_{io}, b_{\text{dist}}, b_{\text{proc}})} = \frac{N}{\min(0.54\ 0.775\ 1.3) \cdot 10^6} = \frac{N}{0.54} \cdot 10^{-6}\ \text{s}. \quad (12)$$

The global dependency resolution and the writing phase can only begin when reading is completely finished. After the last block is processed and the local dependency resolution is ready to provide the first partial result packet of size s:

$$t_{\text{proc}}(s = 65536) = c_q s \ln s = c_q \cdot 2^{16} \cdot \ln 2^{16} = 0.04749\ \text{s}, \quad (13)$$

$$t_{\text{local}}(p) = c_m s \frac{n}{p} = c_m s \frac{N/s}{p} = c_m \frac{N}{p} = 0.038 \cdot 10^{-6} \cdot \frac{N}{p}. \quad (14)$$

To get the time of writing t_{write} the value of $b_{\text{resolution}}$ must also be calculated:

$$b_{\text{resolution}}(p) = \frac{\text{size}}{t_{\text{local}} + t_{\text{global}}} = \frac{1}{c_m\left(\frac{N}{sp} + p\right)} = \frac{sp}{c_m(N + sp^2)}$$
$$= 1.72 \cdot 10^{12} \frac{p}{N + 65535p^2}, \quad (15)$$

$$t_{\text{write}}(p) = \frac{N}{b_{\text{write}}} = \frac{N}{\min[0.54 \cdot 10^{-6} 0.775 \cdot 10^{-6} b_{\text{resolution}}(p)]}. \quad (16)$$

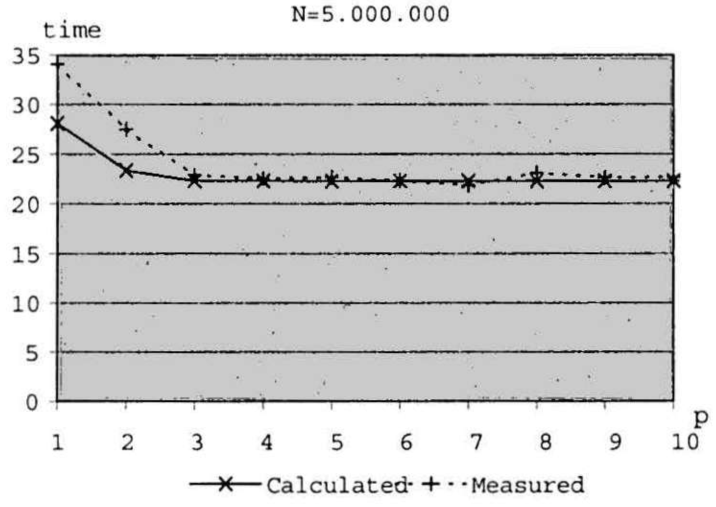
Using these values in (4), we have a total execution time of:

$$t_{\text{total}} = t_{\text{read}} + t_{\text{proc}}(s) + t_{\text{local}}(p) + t_{\text{write}}(p)$$
$$= \frac{N}{0.54} \cdot 10^{-6} + 0.04749 + 0.038 \cdot 10^{-6} \frac{N}{p} + t_{\text{write}}(p). \quad (17)$$

The execution times can now be calculated for every N and p. We represent three series of measured and calculated execution times respectively for $N_1 = 5 \cdot 10^6$ (*Fig. 1*), $N_2 = 10^7$ (*Fig. 2*), and $N_3 = 2 \cdot 10^7$ (*Fig. 3*).

As the figures show, the estimated and the measured execution times are quite close, with a relative error below 6 percent in most of the domain of our interest. The only significant difference is for low p values in *Fig. 2* and *Fig. 3*, where the calculation strongly underestimates the running time. This error is due to the fact that the solution of the problem requires more memory than the amount available on the nodes, and the virtual memory paging highly degrades the overall performance.

The easy determination of the optimal cluster size by simple differentiation is hindered by the minimum formula present in t_{write}, so another minimization method should be applied. It is easy to see, that b_{gather} remains practically a constant, while $b_{\text{resolution}}$ is monotonously increasing in our domain of small cluster sizes. The

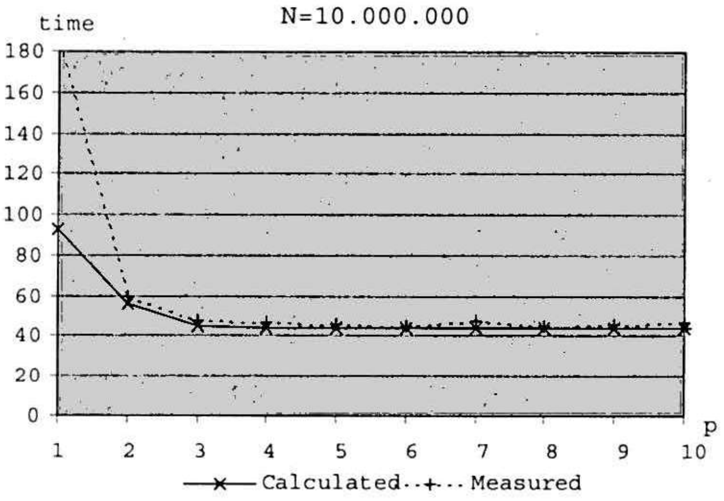


P	Measured time	Calculated time	Relative error [%]
1	34.08	28.1	21.3
2	27.50	23.3	17.8
3	22.88	22.3	2.6
4	22.54	22.3	1.1
5	22.67	22.3	1.8
6	22.32	22.3	0.2
7	21.85	22.3	1.9
8	23.12	22.3	3.8
9	22.63	22.3	1.6
10	22.66	22.3	1.7
11	22.37	22.3	0.4

Fig. 1. Execution times for a problem size $N = 5\,000\,000$ with different cluster sizes

minimum value of t_{write} (and so that of t_{total} likewise) is expected when $b_{\text{resolution}}$ equals to b_{gather}:

$$b_{\text{resolution}}(p) = \frac{sp}{c_m\left(N + sp^2\right)} = b_{\text{gather}},$$
$$\left(b_{\text{gather}} c_m s\right) p^2 - sp + b_{\text{gather}} c_m N = 0. \tag{18}$$



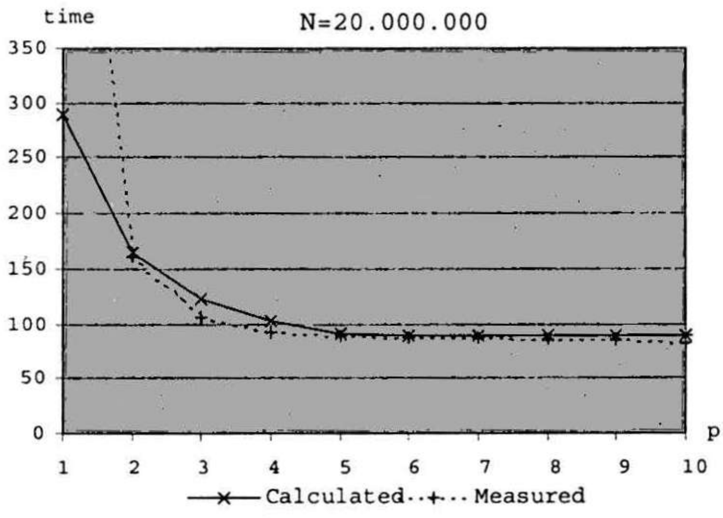
p	Measured time	Calculated time	Relative error [%]
1	183.79	92.7	98.2
2	60.01	56.4	6.5
3	48.11	44.6	8.0
4	47.27	44.5	6.2
5	45.61	44.5	2.5
6	44.81	44.5	0.7
7	47.40	44.5	6.6
8	45.38	44.5	2.0
9	45.48	44.5	2.3
10	47.25	44.5	6.3
11	44.98	44.5	1.2

Fig. 2. Execution times for a problem size $N = 10\,000\,000$ with different cluster sizes.

Solving the quadratic equation for the smaller p:

$$p = \frac{s - \sqrt{s^2 - 4N\left(b_{\text{gather}}c_m s\right) b_{\text{gather}}c_m}}{2b_{\text{gather}}c_m s} = \frac{1 - \sqrt{1 - 4\frac{N\left(b_{\text{gather}}c_m\right)^2}{s}}}{2b_{\text{gather}}c_m}. \quad (19)$$

Considering the third problem, and replacing $N_3 = 2 \cdot 10^7$ in (19) we get for optimal



p	Measured time	Calculated time	Relative error [%]
1	700.6	288.9	143
2	161.4	164.2	1.7
3	106.5	123.2	13.6
4	92.1	103.1	10.6
5	88.5	91.4	3.1
6	87.2	88.9	1.9
7	87.3	88.8	1.8
8	85.9	88.8	3.3
9	85.3	88.8	4.0
10	82.9	88.8	6.7
11	86.2	88.8	3.0

Fig. 3. Execution times for a problem size $N = 20\,000\,000$ with different cluster sizes

cluster size:

$$p = \frac{1 - \sqrt{1 - 4\dfrac{(0.387 \cdot 0.04)^2 \cdot 2 \cdot 10^7}{65536}}}{2 \cdot 0.387 \cdot 0.04} = \frac{1 - 0.8411}{2 \cdot 0.01548} = 5.13. \qquad (20)$$

Naturally, p is an integer, so we may choose the closest value $p = 5$ as cluster size, which is close to optimal, or $p = 6$, which has a processing power actually beyond the limit of network capacity and introduces only slight speed increase. The choice of bigger cluster sizes will not cause any further performance gain, or may even result in performance degradation, as it can also be seen in the graph of *Fig. 3*.

It is worth to mention that the result is quite close to the maximum achievable result with the given b_{io}, because the reading and writing time of this problem without calculations would be:

$$t_{\text{read}}(b_{io}) + t_{\text{write}}(b_{io}) = N/b_{io} + N/b_{io} = \frac{2N}{0.54 \cdot 10^{-6}} s = 73.98 \text{ s.} \tag{21}$$

6. Conclusion

This paper presented a methodology for obtaining good execution time approximations in case of problems where the computation effort of processing and that of dependency solving can be well described with mathematical formulas. An algorithm for parallel sorting was given as a good example of the problem class where relatively little processing effort is done on a large amount of data with global dependencies.

The execution time approximation was based on finding the bandwidth of the bottleneck in the system. After describing the different aspects of the problem with the formulas derived from the performance model, some test measurements were done, and statistical methods are used to compute the few parameters required. As the estimation itself requires only few operations, even runtime usage becomes possible. The obtained results can be used in decision support for resource allocation (how many nodes to assign to a task) or for scheduling problems (how long the resources will be held).

From the point of view of execution speed the mathematical model automatically gives the possibility to obtain an optimal cluster size. As moving a big amount of data generates heavy overhead, the decreasing performance gain with every new node is clearly visible for this class of problem. As it is shown in the example, in the present PC clusters only the first few nodes have significant effect on the performance, because of the slow interconnection network compared to the processing speed. It is clearly visible from the formulas that the efficiency of cluster computing increases with the increase of the processing to communication ratio.

Uniting the resources of the cluster has another beneficial effect. Although not explicitly included in the presented model, it is very important that the data amount not fitting in the memory of a single node could easily fit in the total memory of an extended cluster. In assumption 2 we supposed that the problem data will completely fit in the memory of the cluster during the processing. Should this not be the case, the use of virtual memory paging to disk may cause so serious performance degradation, that even super-linear execution speed increase can be achieved in this domain (e.g. in problem 3 introducing the second node speeds up the computation by a factor of 4).

The sorting example showed that good approximation can be obtained for problems where a well fitting mathematical model exists for the execution time, which is a crucial point of this kind of prediction. Network and background storage operation can be handled more easily; even a single bandwidth value can characterize these operations well. The presented model is not strictly restricted to mathematically well described problems, as for cases, where the processing operations are difficult to treat with simple formulas, a statistical approach may also be applicable.

References

[1] HELMAN, D. R. – BADER, D. A. – JÁJÁ, J., A Randomized Parallel Sorting Algorithm with an Experimental Study, *Journal of Parallel and Distributed Computing*, **52** (1) (July, 1998), pp. 1–23.
http://citeseer.nj.nec.com/21672.html

[2] BAL, H., *Programming Distributed Systems*, Prentice-Hall, London, UK, 1990.

[3] BUYYA, R. (editor), High Performance Cluster Computing, Vol. 1: *Architectures and Systems*, Vol. 2: *Programming and Applications*, Prentice Hall, New Jersey, USA
http://www.dgs.monash.edu.au/~rajkumar/cluster

[4] SMITH, W. – FOSTER, I. – TAYLOR, V., Predicting Application Run Times Using Historical Information, *Proc. IPPS/SPDP '98 Workshop on Job Scheduling Strategies for Parallel Processing*, 1998.

[5] SIMON, E., *Distributed Information Systems*, McGraw-Hill, London, UK, 1997.

[6] AKL, S. G., *The Design and Analysis of Parallel Algorithms*, Prentice Hall, New Jersey, USA, 1989. Chapter 4: Sorting, pp 85–112.

[7] HELMAN, D. R. – JÁJÁ, J., *Sorting on Clusters of SMPs*, 1998,
http://acs.umd.edu/research/EXPAR/papers/smpsort.ps

[8] FOSTER, I., *Designing and Building Parallel Programs*, Vol. 1.3, Addison-Wesley Inc., Argonne National Laboratory. Chapter 2. Designing Parallel Algorithms,
http://www-unix.mcs.anl.gov/dbpp/

[9] *Parallel Computing Projects*,
http://www.hlrs.de/structure/organisation/par/projects

[10] BUYYA, R., *Cluster Computing Info Centre*,
http://www.dgs.monash.edu.au/~rajkumar/cluster