To performance evaluation of distributed parallel algorithms

Juraj Hanuliak and Ivan Hanuliak
Faculty of Control and Informatics, University of Zilina, Slovakia

Abstract

Purpose – To address the problems of high performance computing by using the networks of workstations (NOW) and to discuss the complex performance evaluation of centralised and distributed parallel algorithms.

Design/methodology/approach – Defines the role of performance and performance evaluation methods using a theoretical approach. Presents concrete parallel algorithms and tabulates the results of their performance.

Findings – Sees that a network of workstations based on powerful personal computers belongs in the future and as very cheap, flexible and perspective asynchronous parallel systems. Argues that this trend will produce dynamic growth in the parallel architectures based on the networks of workstations.

Research limitations/implication – We would like to continue these experiments in order to derive more precise and general formulae for typical used parallel algorithms from linear algebra and other application oriented parallel algorithms.

Practical implications – Describes how the use of NOW can provide a cheaper alternative to traditionally used massively parallel multiprocessors or supercomputers and shows the advantages of unifying the two disciplines that are involved.

Originality/value – Produces a new approach and exploits the parallel processing capability of NOW. Gives the concrete practical examples of the method that has been developed using experimental measuring.

Keywords Cybernetics, Programming and algorithm theory, Computer networks

Paper type Research paper

1. Introduction

There has been an increasing interest in the use of networks (cluster) of workstations connected together by high-speed networks for solving large computation-intensive problems. This trend is mainly driven by the cost effectiveness of such systems as compared to massive multiprocessor systems with tightly coupled processors and memories. Parallel computing on a cluster of workstations connected together by high-speed networks has given rise to a range of hardware and network related issues on any given platform. Load balancing, inter-processor communication, and transport protocol for such machines are being widely studied (Greenberg *et al.*, 1996; Hanuliak, 1999; Hesham and Lewis, 1997; Hwang and Xu, 1998; Kumar *et al.*, 2001; Sveda and Vrba, 2001). With the availability of cheap personal computers, workstations and networking devices, the recent trend is to connect a number of such workstations to solve computation-intensive tasks in parallel on such clusters.

To exploit the parallel processing capability of a network of workstation (NOW), the application program must be paralleled. The effective way how to do it for a concrete application problem (decomposition strategy) belongs to a most important step in developing a effective parallel algorithm (Hanuliak, 2001a; Marinescu and Rice, 1995; Nancy, 1996).

Kybernetes
Vol. 34 No. 9/10, 2005
pp. 1633-1650
© Emerald Group Publishing Limited
0368-492X
DOI 10.1108/03684920510614858

NOW (Hanuliak, 1999; Hesham and Lewis, 1997; Hwang and Xu, 1998; Kumar *et al.*, 2001; Williams, 2001) has become a widely accepted form of high-performance parallel computing. As in conventional multiprocessors, parallel programs running on such a platform are often written in an SPMD form (Single – program – multiple data) to exploit data parallelism or in a improved SPMD form to take into account also the potential of functional parallelism of a given application. Each workstation in a NOW is treated similarly to a processing element in a multiprocessor system. However, workstations are far more powerful and flexible than processing elements in conventional multiprocessors. We can also use the advantages of the new Intel's SIMD (Single instruction Multiple data) or MMX (Multimedia extensions) instructions in the latest personal processors.

2. The role of performance

Quantitative evaluation and modelling of hardware and software components of parallel systems are critical for the delivery of high performance. Performance studies apply to initial design phases as well as to procurement, tuning, and capacity planning analysis. As performance cannot be expressed by quantities independent of the system workload, the quantitative characterisation of resource demands of application and of their behaviour is an important part of any performance evaluation study. Among the goals of parallel systems performance analysis are to asses the performance of a system or a system component or an application, to investigate the match between requirements and system architecture characteristics, to identify the features that have a significant impact on the application execution time, to predict the performance of a particular application on a given parallel system, to evaluate different structures of parallel applications. To the performance evaluation we briefly review the techniques most commonly adopted for the evaluation of parallel systems and its metrics.

2.1 Performance evaluation methods

To the performance evaluation we can use following methods.

1. Analytical methods:
   - application of queueing theory results (Hanuliak, 1999; 2002; and 2001a, b; Harrison and Patel, 1993; Hsu and Pen-Chung, 1997)
   - petri nets (Hanuliak, 1999; Hwang and Xu, 1998).
2. Simulation methods (Banks and Dai, 1997; Fodor *et al.*, 1998).
3. Experimental measurement (Hanuliak, 1999; Hwang and Xu, 1998):
   - benchmarks; and
   - direct measuring of concrete developed parallel application.

In order to extend the applicability of analytical techniques to the parallel processing domain, various enhancements have been introduced to model phenomena such as simultaneous resource possession, fork and join mechanism, blocking and synchronisation. Hybrid modelling techniques allow to model contention both at hardware and software levels by combining approximate solutions and analytical methods. However, the complexity of parallel systems and algorithms limit the applicability of these techniques. Therefore, in spite of its computation and time requirements, simulation is extensively used as it imposes no constraints on modelling.

Evaluating system performance via experimental measurements is a very useful alternative for parallel systems and algorithms. Measurements can be gathered on existing systems by means of benchmark applications that aim at stressing specific aspects of the parallel systems and algorithms. Even though benchmarks can be used in all types of performance studies, their main field of application is competitive procurement and performance assessment of existing systems and algorithms. Parallel benchmarks extend the traditional sequential ones by providing a wider set of suites that exercise each system component targeted workload. The Parkbench suite especially oriented to message passing architectures and the SPLASH suite for shared memory architectures are among the most commonly used benchmarks (Hwang and Xu, 1998).

2.2 Performance evaluation metrics

For evaluating parallel algorithms there have been developed several fundamental concepts. Tradeoffs among these performance factors are often encountered in real-life applications.

2.2.1 Performance concepts. Let $O(s,p)$ be the total number of unit operations performed by p-processor system for size s of the computational problem and $T(s,p)$ be the execution time in unit time steps. In general, $T(s,p) < O(s,p)$ if more than one operation is performed by p processors per unit time, where $p \geq 2$. Assume $T(s,1) = O(s,1)$ in a single-processor system (sequential system). The speedup factor is defined as:

$$S(s,p) = \frac{T(s,1)}{T(s,p)}$$

It is a measure of the speedup factor obtained by given algorithm when p processors are available for the given problem size s. Ideally, since $S(s,p) \leq p$, we would like to design algorithms that achieve $S(s,p) \approx p$.

The system efficiency for an p-processor system is defined by:

$$E(s,p) = \frac{S(s,p)}{p} = \frac{T(s,1)}{pT(s,p)}$$

A value of $E(s,p)$ approximately equal to 1, for some p, indicates that such a parallel algorithm, using p processors, runs approximately p times faster than it does with one processor (sequential algorithm).

2.2.2 The isoefficiency concept. The workload w of an algorithm often grows in the order $O(s)$, where s is the problem size. Thus, we denote the workload $w = w(s)$ as a function of s. In parallel computing is very useful to define an isoefficiency function relating workload to machine size p needed to obtain a fixed efficiency E when implementing a parallel algorithm on a parallel system. Let h be the total communication overhead involved in the algorithm implementation. This overhead is usually a function of both machine size and problem size, thus denoted $h = h(s,p)$.

The efficiency of a parallel algorithm implemented on a given parallel computer is thus defined as

$$E(s,p) = \frac{w(s)}{w(s) + h(s,p)}$$

The workload $w(s)$ corresponds to useful computations while the overhead $h(s, n)$ are useless times attributed to synchronisation and data communication delays. In general, the overhead increases with respect to both increasing values of s and p. Thus, the efficiency is always less than one. The question is hinged on relative growth rates between $w(s)$ and $h(s,p)$.

With a fixed problem size (fixed workload), the efficiency decreases as p increase. The reason is that the overhead $h(s,p)$ increases with p. With a fixed machine size, the overload h grows slower than the workload w. Thus, the efficiency increases with increasing problem size for a fixed-size machine. Therefore, one can expect to maintain a constant efficiency if the workload w is allowed to grow properly with increasing machine size.

For a given algorithm, the workload w might need to grow polynomial or exponentially with respect to p in order to maintain a fixed efficiency. Different algorithms may require different workload growth rates to keep the efficiency from dropping, as p is increased. The isoefficiency functions of common parallel algorithms are polynomial functions of p; i.e. they are $O(p^{k})$ for some $k \geq 1$. The smaller a power of p in the isoefficiency function is, the more scalable the parallel system. Here, the system includes the algorithm and architecture combination.

2.2.3 Isoefficiency concept. We can rewrite equation for efficiency $E(s,p)$ as $E(s,p) = 1/(1 = h(s,p)/w(s))$. In order to maintain a constant E, the workload $w(s)$ should grow in proportion to the overhead $h(s,p)$. This leads to the following relation:

$$w(s) = \frac{E}{1-E} h(s,p)$$

The factor $C = E/1 - E$ is a constant for a fixed efficiency E. Thus, we can define the isoefficiency function as follows: $f_E(p) = Ch(s,p)$. If the workload grows as fast as $f_E(p)$ then a constant efficiency can be maintained for a given algorithm-architecture combination.

3. Complex performance evaluation

To the complex performance evaluation of parallel algorithms we can use in the case of in the world used centralised parallel multiprocessor system (synchronous SIMD – Single instruction Multiple data parallel architectures and SMP – Symmetrical multiprocessors) and asynchronous (centralised or distributed MIMD – Multiple instructions multiple data parallel architectures) analytical approach to get under given constraints some analytical laws (Hesham and Lewis, 1997; Hwang and Xu, 1998; Kumar *et al.*, 2001) (Amdahl's law, Gustafson law) or some other derived analytical relations. Indeed, Amdahl's Law and the extension described by Gustafson and others are only properly applied as limiting cases and have been successfully used to evaluate limitations and potential of parallel processing. The known analytical relations have been derived without considering architecture and communication complexity. That means a performance $P_p = f$ (calculation). Such assumptions could be real in some existed massively multiprocessor systems in the world but not in NOW based on personal computers.

In NOW as a new form of asynchronous parallel systems (Andrews, 2000; Hanuliak, 1999; Hesham and Lewis, 1997; Hwang and Xu, 1998; Kumar *et al.*, 2001; Williams, 2001), we have to take into account all aspects that are important for complex performance evaluation according to the relation $P_p = f$ (architecture, communication, calculation). In such a case we can use the following solution methods to get a complex performance.

- Direct measurement – real experimental measure of P_p and its components for a concrete developed parallel algorithm on the concrete parallel system.
- Analytic modelling – to find P_p on the basis of some closed analytical expressions or statistical distributions for individual overheads.
- Simulation technique – simulation modelling on concrete developed parallel algorithms on the concrete parallel system.

4. The theoretical part

4.1 Method of direct measuring

For suggested direct measuring of complex performance evaluation in a NOW we used the structure according to Figure 1.

The way of measuring can be illustrated by flow diagrams in Figure 2 for complex performance evaluation of common sequential algorithms (for evaluation of some specified speed-up concepts), in Figure 3 for centralised parallel algorithms (SMP – symmetrical multiprocessors, etc.) and in Figure 4 for distributed parallel algorithms (in our case for NOW's implementation). The difference between Figure 3

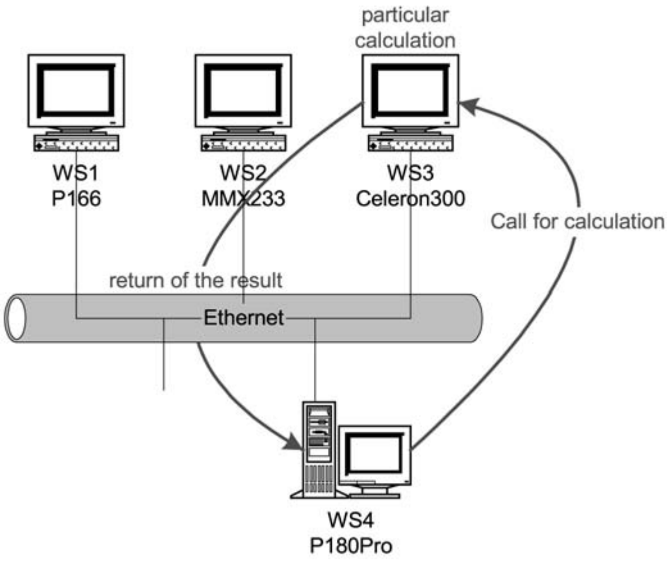


Figure 1.
The measure on the NOW (ethernet network)

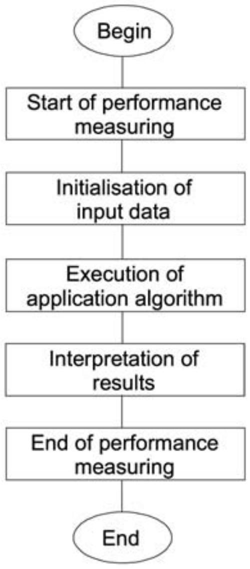

Begin
Start of performance measuring
Initialisation of input data
Execution of application algorithm
Interpretation of results
End of performance measuring
End

Figure 2.
Flow diagram to sequential performance measuring

and Figure 4 is principally in the way of implementing IPC (Inter-process communication) among decomposed parallel processes.

4.2 The concrete parallel algorithms

4.2.1 Numerical integration. A numerical integration is the typical example of the implicitly parallel algorithm, that the parallelism is the integral part of the own algorithm. Concretely for the calculation of the value π the following standard formula is used (Hanuliak, 1999; Hwang and Xu, 1998):

$$\pi = \int_0^1 \frac{4}{1+x^2}\,\mathrm{d}x$$

For the parallel way of calculation we used the potentially possibility of decomposition in the algorithms with numerical integration to the mutual independent parts (processes). In our concrete example we divide the whole calculation to its individual processes according to Figure 5.

The individual independent processes we distribute for the calculation to the computer network in such a way that every process will be concurrently executed on the different node of a NOW network (the mapping of processes to individual workstations). After the parallel computation in the individual nodes of a network of workstations have been made we need only to sum the particular results to achieve the final value. To handle this task we have to choose one of the nodes of a network of workstations. As well at the beginning the chosen node (for example, node 0) must know the value n (the number of the strips in every process) and this node has to let it know to the other active nodes. The example of the parallel algorithm of π computation is then following:

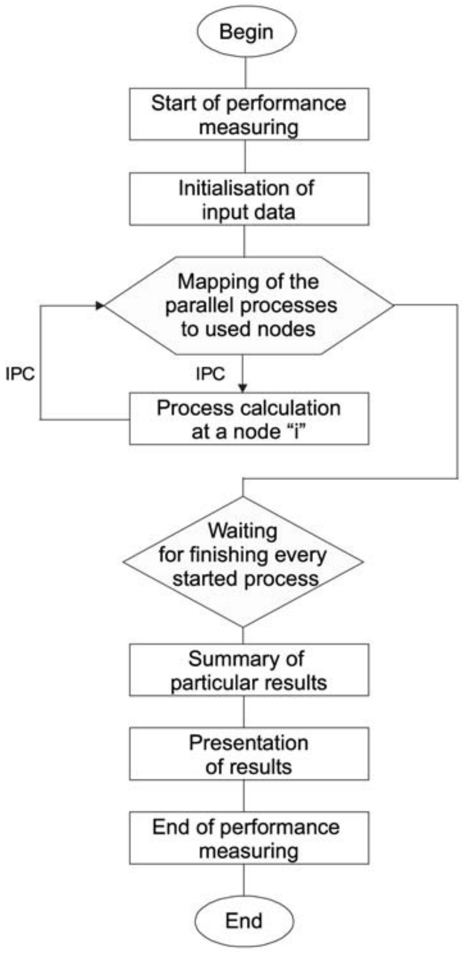

Begin
Start of performance measuring
Initialisation of input data
Mapping of the parallel processes to used nodes
IPC
IPC
Process calculation at a node "i"
Waiting for finishing every started process
Summary of particular results
Presentation of results
End of performance measuring
End

Figure 3.
Flow diagram for centralised architectures (SMP)

```
if my node is 0
  read the number n of strips desired and send it to all
other nodes
else
  receive n from node 0
end if
for each strip assigned to this node
  calculate the height of rectangle (at midpoint) and sum
result
end for
if my node is not 0
  send sum of result to node 0
```

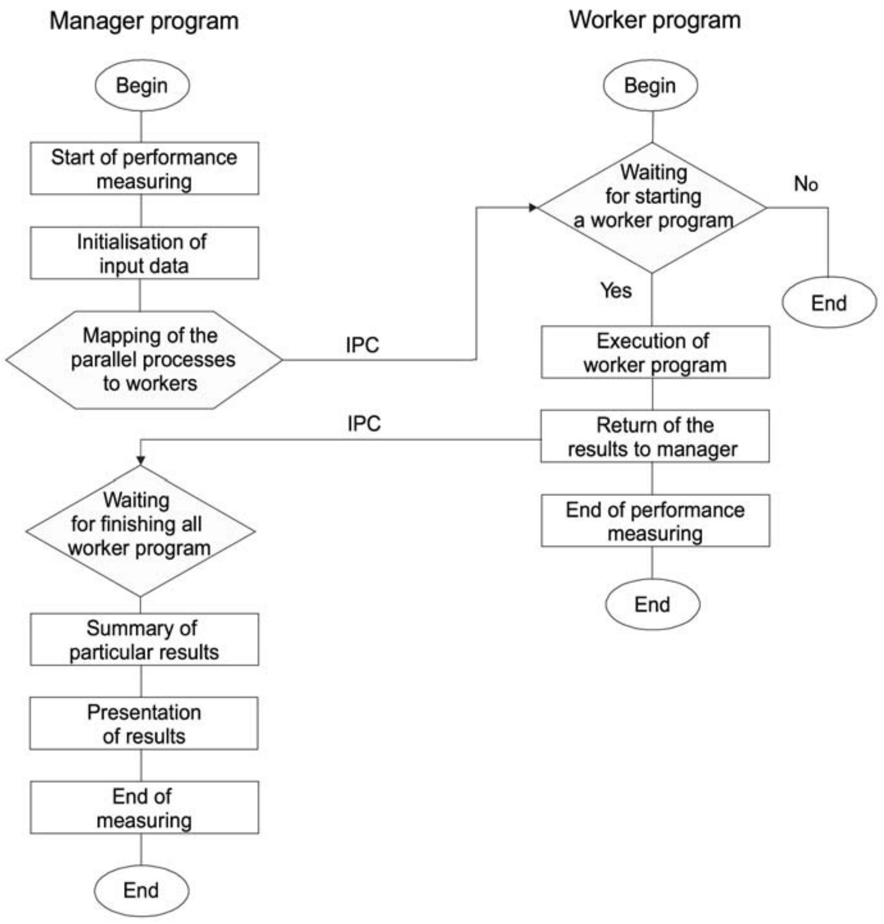

Manager program
Begin
Start of performance measuring
Initialisation of input data
Mapping of the parallel processes to workers
IPC
Waiting for finishing all worker program
Summary of particular results
Presentation of results
End of measuring
End
Worker program
Begin
Waiting for starting a worker program
No
End
Yes
Execution of worker program
IPC
Return of the results to manager
End of performance measuring
End

Figure 4.
The flow diagram for measuring in NOW's

```
else
  receive results from all nodes and sum
multiple the sum by the width of the strips to get $\pi$
return
```

4.2.2 The discrete Fourier transform. The discrete Fourier transform (DFT) has played an important role in the evolution of digital signal processing techniques. It has opened new signal processing techniques in the frequency domain, which are not easily realisable in the analogue domain.

The DFT is defined (Basoglu *et al.*, 1997; Hanuliak, 1999; Hwang and Xu, 1998) as:

$$X_n = \sum_{m=0}^{N-1} x_m w^{mn}, \quad n = 0, 1, \ldots, N-1$$

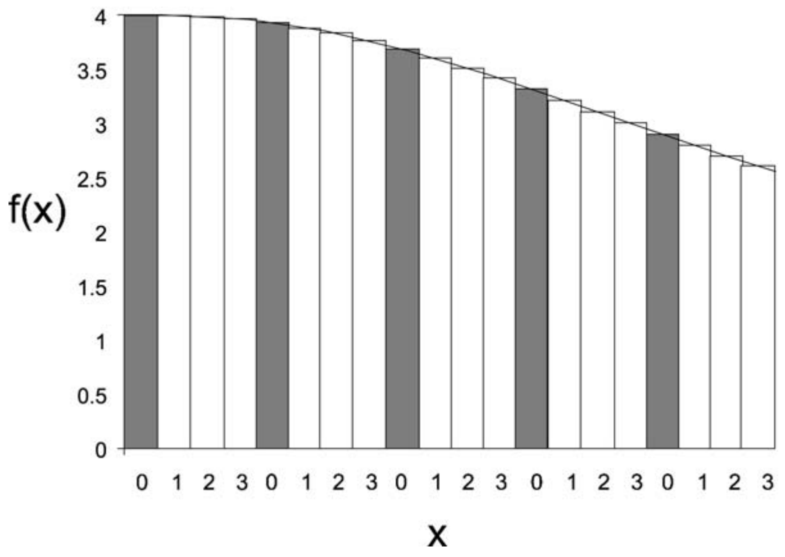


Figure 5.
The decomposition of the numerical integration problems

and the inverse discrete Fourier transform (IDFT) as:

$$x_m = \frac{1}{N}\sum_{R=0}^{N-1} X_n w^{-mn}, \quad m = 0, 1, \ldots, N-1$$

where w is N - root of unity, i.e. $w = \mathrm{e}^{-i(2p/N)}$ for generally complex numbers. In principle the mentioned equations are the linear transforms. Direct computations of the DFT or the IDFT, according to definitions require N^2 complex arithmetic operations. In such a way we could take into account only the calculation times and not also the overheads times caused through a parallel way of an algorithm implementation.

Cooley and Tukey (Basoglu *et al.*, 1997; Hanuliak, 1999) developed a fast DFT algorithm which requires only $O(N\log_2(N))$ operations. The difference in execution time between a direct computation of the DFT and the new DFFT algorithm is very large for large N. Direct computations of the DFT or the IDFT, according to the following program, requires N^2 complex arithmetic operations.

```
Program Direct_DFT;
var
  x, Y: array[0..Nminus1] of complex;
begin
  for k := 0 to N − 1 do
    begin
      Y[k] := x[0];
      for n := 1 to N − 1 do
        Y[k] := Y[k] + W^nk * x[n];
      end;
end.
```

For example, the time required for just the complex multiplication in a 1024-point FFT is $T_{\mathrm{mult}} = 0,5N\log_2(N)4T_{\mathrm{real}} = 0,5.1024\log_2(1024)4T_{\mathrm{real}}$, where the complex multiplication corresponds approximately to four real multiplication. The principle of Cooley and Tukey algorithm, which use a divide-and-conquer strategy, shows Figure 6. Several variations of the Cooley-Tukey algorithm have since been derived. These algorithms are collectively referred to as the discrete fast Fourier transform (DFFT) algorithms. The basic form of parallel DFFT is the one-dimensional (1D), unordered, radix-2 (a use of divide and conquer strategy according the principle in Figure 6). The effective parallel computing of DFFT tends to computing 1D FFT's with radix equals and greater than two and computing multidimensional FFT's by using the polynomial transfer methods. In practical part of this paper we computed 2DFFT (two-dimensional DFFT). In general, a radix-q DFFT is computed by splitting the input sequence of size s into a q sequences of size n/q each, computing faster the q smaller DFFT's, and then combining the result. For example, in a radix-4 FFT's, each step computes four outputs from four inputs, and the total number of iterations is $\log_4 s$ rather than $\log_2 s$. The input length should, of course, be a power of four. Parallel formulations of higher - radix strategies (e.g. radix-3 and 5) 1D or multidimensional DFFT's are similar to the basic form because the underlying ideas behind all sequential DFFT are the same. An ordered DFFT is obtained by performing bit reversal (permutation) on the output sequence of an unordered DFFT. Bit reversal does not affect the overall complexity of a parallel implementation.

5. The results

To measure both a calculation and overhead times the function "Query Performance Counter", which measures calculation times in ms was used. For these purposes, we used common personal computers according to Table I (the principles in using the more powerful personal computers are the same but we would get better results).

The developed parallel algorithms were divided in to two logical parts – manager and worker programs. All programs are written on the WNT (Windows New Technology)

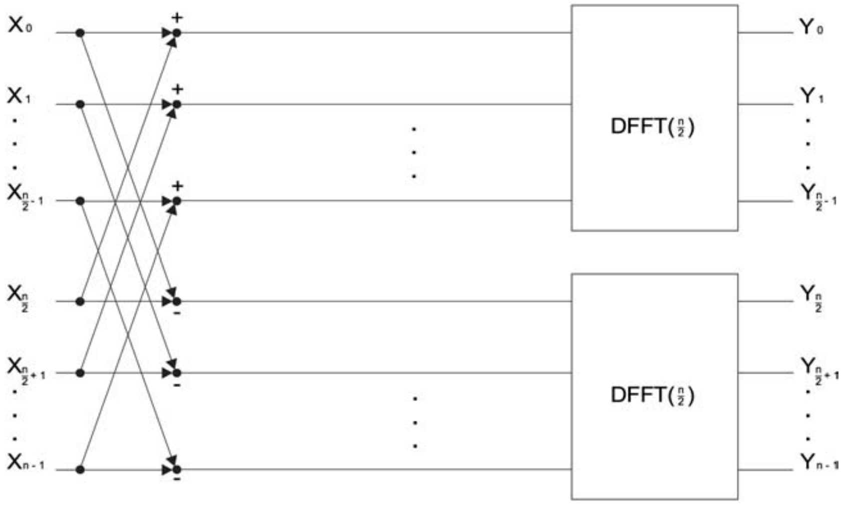


Figure 6.
An illustration of divide-and-conquer strategy for DFFT

platform. Manager control the computer with starting services, makes the connections and starts in parallel way the remote functions. At the end sums the particular results.

Every server waits for calculation starting and then calculates the particular results. At the end of calculation returns to the manager the calculated results and the calculation time. The results of the whole calculation are not only the calculated results but also the calculation and communication times with individual server. To measure the calculation time it used the function “Query Performance Counter”, which measures calculation times in ms. Calibration power of the used workstations based on personal computers at our experiments for π calculation are shown in Figure 7 and for 2DFFT in Figure 8. From Figure 7 (algorithm of numerical integration) we can see that with decreasing epsilon (proportionally increasing number of intervals) the execution time increases in a linear way. It is caused through linear increasing of the calculation sum (input load according to Table II). But communication overheads remain constant. The achieved results for 2DFFT algorithm document is geometrically increasing of both computation and communication parts with the quotient value nearly four for analysed matrix dimensions (increasing matrix dimension means to do twice more computation on columns and twice more on rows). Therefore, for better illustration we have used dependencies on relative input load defined according to Table III.

The results in ethernet NOW (numerical integration of π calculation) are graphically illustrated in Figure 9 and for 2DFFT in Figure 10. In both cases we limited for better graphical illustration the measured values for WS1 network node.

The influence of matrix dimension to the network load is shown in Figure 11.

The percentile amount of the individual parts (computation, overheads – network load, initialisation) are illustrated for π execution time with epsilon $= 10^{-5}$ in Figure 12

Label	Processor	RAM [MB]	Operation systém
WS1	Pentium I 233 MHz	64	Windows 2000
WS2	Pentium II 450 MHz	128	Windows 2000
WS3	Pentium III 933 MHz	256	Windows 2000
WS4	Pentium IV 1 Ghz	512	Windows 2000
WS5	Pentium IV 1.4 Ghz	512	Windows 2000
WS6	Pentium IV 2.26 Ghz	1000	Windows 2000
WS7	Pentium IV Xeon – 2 proc., 2.2 GHz	1000	Windows 2000 Server

Table I. Parameters of the used personal computers

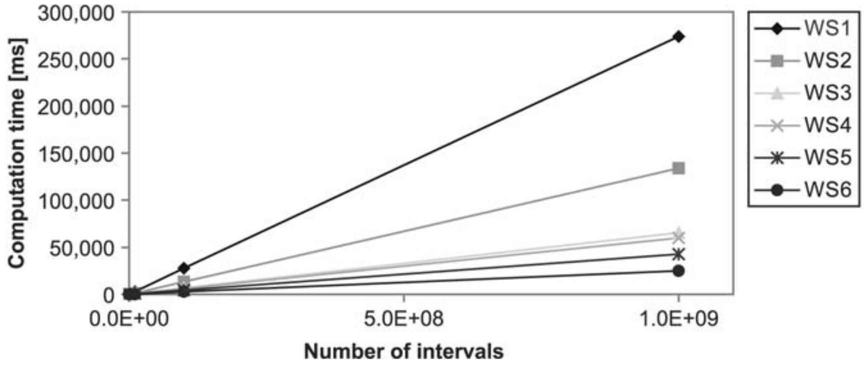


Figure 7. Calibration power results for π calculation

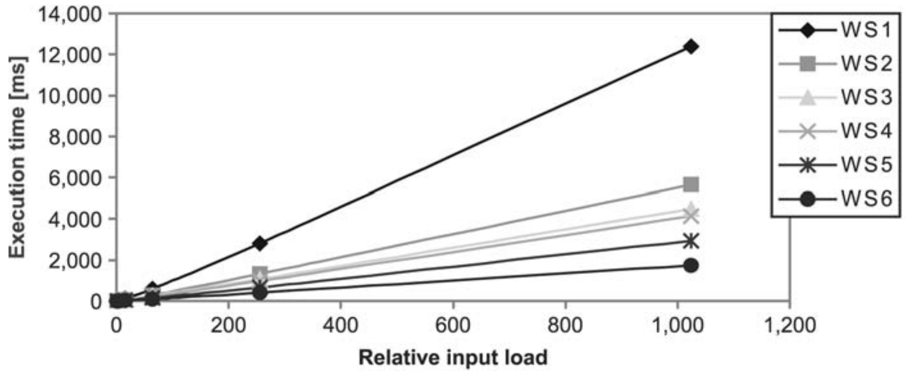


Figure 8.
Calibration power results for 2DFFT

Table II.
Measured results for π calculation

Number of intervals	10^5	10^6	10^7	10^8	10^9
Epsilon	10^{-5}	10^{-6}	10^{-7}	10^{-8}	10^{-9}
WS1	27	279	2 772	27 487	274,193
WS2	14	134	1 355	13 400	133,976
WS3	7	70	668	6 584	65,459
WS4	6	58	598	6 116	59,844
WS5	4	42	425	4 279	42,542
WS6	2	25	250	2 510	25,022

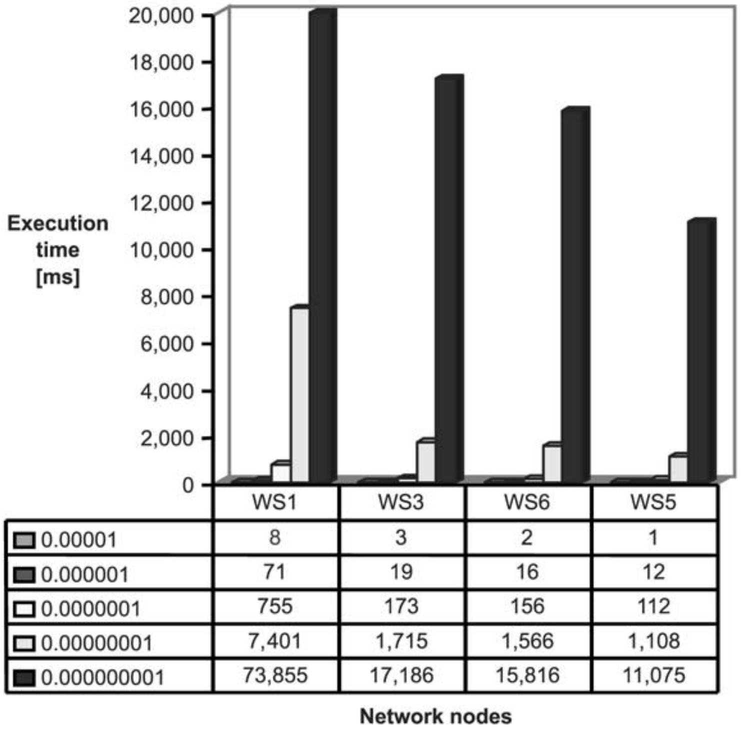
Table III.
Measured results for 2DFFT computation

Relative input load	1	4	16	64	256	1024
Matrix dimension	32 × 32	64 × 64	128 × 128	256 × 256	512 × 512	1024 × 1024
WS1	7	31	138	606	2,819	12,390
WS2	4	15	65	293	1,316	5,657
WS3	3	13	54	246	1,066	4,498
WS4	3	12	49	220	992	4,125
WS5	2	8	36	159	671	2,926
WS6	1	5	22	94	402	1,728

and for epsilon = 10^{-9} in Figure 13. We can see that increasing epsilon (higher input load) results in dominating influence of computation time (network loads remain constant).

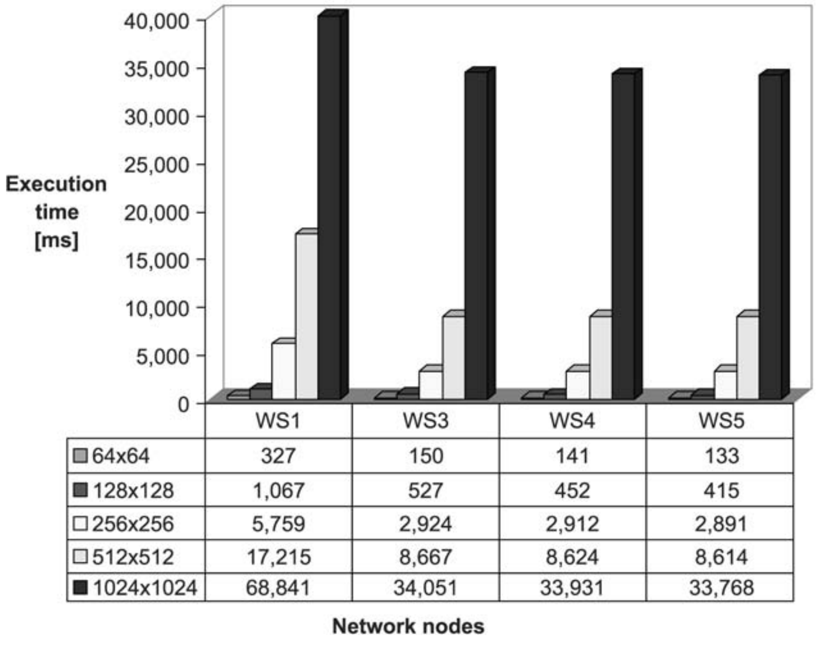
The percentile amount of the individual parts (computation, overheads – network load, initialisation) at 2DFFT execution time for the matrix 512 × 512 is shown in Figure 14 and for matrix 1024 × 1024 in Figure 15. In both cases the percentile amount of individual parts are nearly the same. The high network loads are involved through the needed matrix transpositions during 2DFFT computation.

The comparison of the sequential and parallel way of execution on SMP parallel system (WS7 according Table I) are shown in Figure 16 for π execution and for 2DFFT algorithm in Figure 17.



	WS1	WS3	WS6	WS5
■ 0.00001	8	3	2	1
■ 0.000001	71	19	16	12
□ 0.0000001	755	173	156	112
□ 0.00000001	7,401	1,715	1,566	1,108
■ 0.000000001	73,855	17,186	15,816	11,075

Figure 9.
The results in NOW for π calculation (ethernet network)



	WS1	WS3	WS4	WS5
■ 64x64	327	150	141	133
■ 128x128	1,067	527	452	415
□ 256x256	5,759	2,924	2,912	2,891
□ 512x512	17,215	8,667	8,624	8,614
■ 1024x1024	68,841	34,051	33,931	33,768

Figure 10.
The results in NOW for 2DFFT calculation (ethernet network)

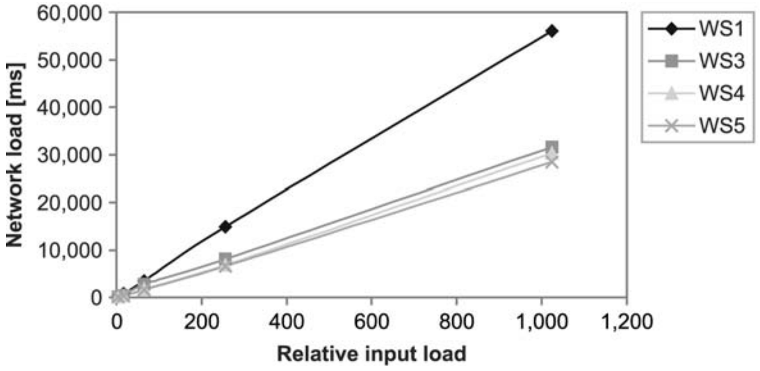
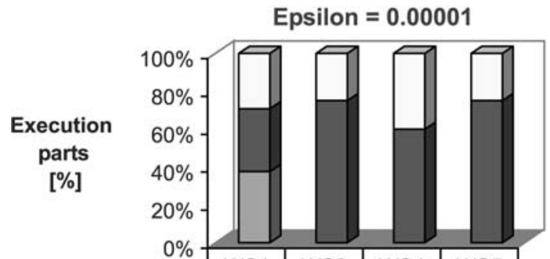


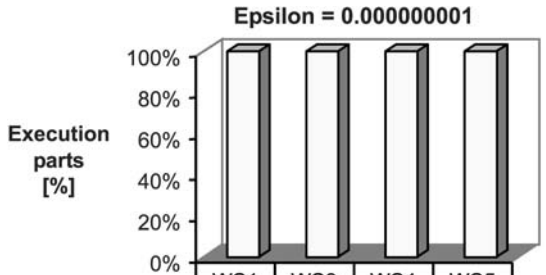
Figure 11.
The influence of matrix dimension to network load



	WS1	WS3	WS4	WS5
□ Computation	7	2	2	1
■ Network load	8	6	3	3
▨ Initialisation	9	0	0	0

Network nodes

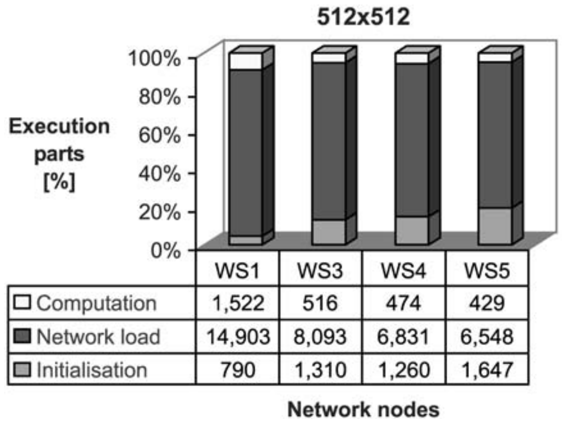
Figure 12.
The individual parts for π execution time (epsilon $= 10^{-5}$)



	WS1	WS3	WS4	WS5
□ Computation	73836	17850	15810	11066
■ Network load	10	5	4	3
▨ Initialisation	9	0	0	0

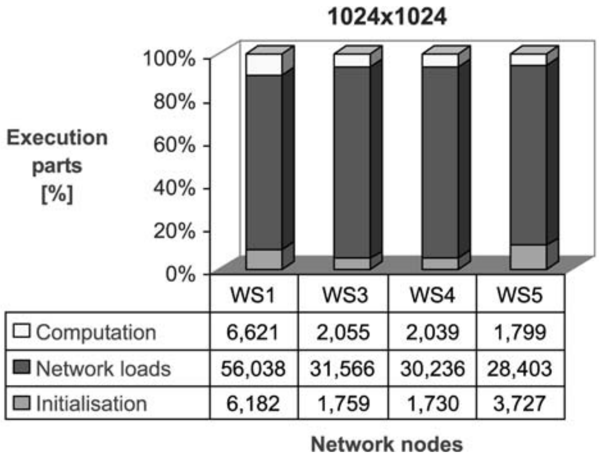
Network nodes

Figure 13.
The individual parts for π execution time (epsilon $= 10^{-9}$)



	WS1	WS3	WS4	WS5
□ Computation	1,522	516	474	429
■ Network load	14,903	8,093	6,831	6,548
▨ Initialisation	790	1,310	1,260	1,647

Figure 14. The individual parts of the 2DFFT execution time (512×512)



	WS1	WS3	WS4	WS5
□ Computation	6,621	2,055	2,039	1,799
■ Network loads	56,038	31,566	30,236	28,403
▨ Initialisation	6,182	1,759	1,730	3,727

Figure 15. The individual parts of the 2DFFT execution time (1024×1024)

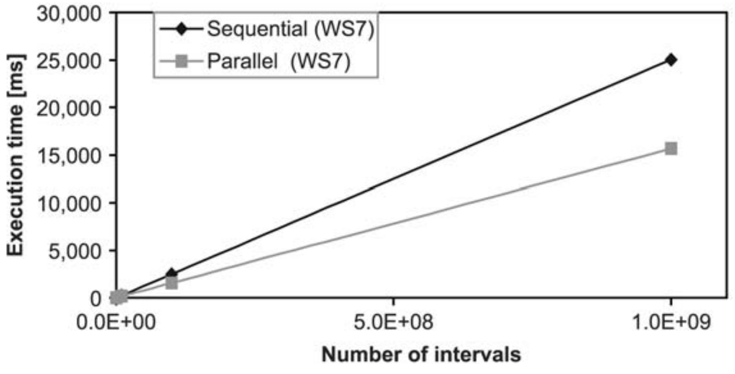


Figure 16. Comparison of the sequential and parallel π calculation

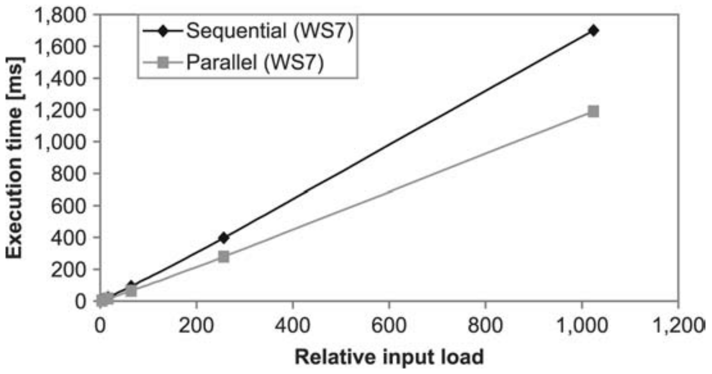


Figure 17. Comparison of the sequential and parallel 2DFFT calculation

6. Conclusions

Distributed computing was reborn as a kind of "lazy parallelism". A network of computers could team up to solve many problems at once, rather than one problem higher speed. To get the most out of a distributed parallel system, designers and software developers must understand the interaction between the hardware and the software parts of the system. It is obvious that the use of a computer network based on personal computers would be principally less effective than the used typical massively parallel architectures in the world, because of higher communication overheads, but a network of workstations based on powerful personal computers, belongs in the future to very cheap, flexible and perspective asynchronous parallel systems. We can see such a trend in dynamic growth just in the parallel architectures based on the networks of workstations as a cheaper and flexible architecture in comparison to conventional multiprocessors and supercomputers. But the principles of these in the world realised multiprocessors are implemented to this time in modern symmetric multiprocessor systems (SMP) based on the same processors which are realised on a workstation's motherboard. Unifying of both approaches (NOW's implementing both simple and SMP workstations) open the new possibilities in HPC computing.

The next steps in the evolution of distributed parallel computing will take place on both fronts: inside and outside the box. Inside, parallelism will continue to be used by hardware designers to increase performance. Intel's new SIMD or MMX instructions technology, which implements a small/scale form of data parallelism, is one example. Out-of-order instruction execution by super-scalar processors in all latest powerful processors is another example of internal parallelism.

Therefore, in relation to our achieved results we are able to do better load balancing among used network nodes (performance optimisation of parallel algorithm). For these purposes we can use calibration results of individual network nodes in order to divide the input load according to the measured performance power of used network nodes. Second we can do load balancing among network nodes based on modern SMP parallel systems and on network nodes with only single processors. Generally we can say that the parallel algorithms or their parts (processes) with more communication (similar to analysed 2DFFT algorithm) will have better speed-up values using modern SMP parallel system as its parallel communication overheads (similar to analysed π computation) we can use the other network nodes based on single processors.

Queueing networks and Petri nets models, simulation, experimental measurements, and hybrid modelling have been successfully used for the evaluation of system components. Via the form of experimental measurement we illustrated the use of this technique for the complex performance evaluation of parallel algorithms. In this context we presented the first part of achieved results. We would like to continue in these experiments in order to derive more precise and general formulae (generalisation of the used Amdahl's and Gustafson's laws at least in a limited way for typical used decomposition strategies or similar application problems) and to develop suitable synthetic parallel tests (SMP, NOW) to predict performance in NOW for some typical parallel algorithms from linear algebra and other application oriented parallel algorithms. In future we will also report about these results.

References

Andrews, G.R. (2000), *Foundations of Multithreaded, Parallel, and Distributed Programming*, Addison-Wesley Longman, Glen View, IL, p. 664.

Banks, J. and Dai, J.G. (1997), "Simulation studies of multiclass queueing networks", *IEEE Transactions*, Vol. 29, pp. 213-9.

Basoglu, Ch., Lee, W. and Kim, Y. (1997), "An efficient FFT algorithm for super-scalar and VLIW processor architectures", *Real Time Imaging*, Vol. 3 No. 6, pp. 441-53.

Fodor, G., Blaabjerg, S. and Andersen, A. (1998), "Modelling and simulation of mixed queueing and loss systems", *Wireless Personal Communication*, Vol. 8, pp. 253-76.

Greenberg, D.S., Park, J.K. and Schvabe, E.J. (1996), "The cost of complex communication on simple networks", *Journal of Parallel and Distributed Computing*, Vol. 35, pp. 133-41.

Hanuliak, I. (1999), *Parallel Computers and Algorithms*, ELFA Press, Košice, p. 327 (in Slovak).

Hanuliak, I. (2001a), "Buffer management control in data transport network node", *The International Journal of Systems Architecture*, Elsevier Science, Amsterdam, Vol. 47, pp. 529-41.

Hanuliak, M. (2001b), "To the behaviour analysis of mobile data networks", *Proceedings of 7th Scientific Conference*, TÂRGU – JIU, Romania, 9-10 November, pp. 170-5.

Hanuliak, I. (2002), "On the analysis and modelling of computer communication systems, Kybernetes", *The International Journal of Systems & Cybernetics*, Vol. 31 No. 5, pp. 715-30.

Harrison, P.G. and Patel, N. (1993), *Performance Modelling of Communication Networks and Computer Architectures*, Addison-Wesley, Reading, MA, p. 480.

Hsu, W.T. and Pen-Chung, Y. (1997), "Performance evaluation of wire-limited hierarchical networks", *Parallel and Distributed Computing*, Vol. 41 No. 2, pp. 156-72.

Hesham, EL-Rewini and Lewis Ted, G. (1997), *Distributed and Parallel Computing*, Manning Publications, Greenwich, CT, p. 467.

Hwang, K. and Xu, Z. (1998), *Scalable Parallel Computing: Technology, Architecture, Programming,* McGraw-Hill, New York, NY, p. 802.

Kumar, V., Grama, A., Gupta, A. and Karypis, G. (2001), *Introduction to Parallel Computing*, 2nd ed., Addison-Wesley, Reading, MA, p. 856.

Marinescu, D.C. and Rice, J.R. (1995), "On the scalability of asynchronous parallel computations", *Parallel and Distributed Computing*, Vol. 31, pp. 88-97.

Nancy, A.L. (1996), *Distributed Algorithms*, Morgan Kaufmann Publishers, San Francisco, CA, p. 872.

Sveda, M. and Vrba (2001), "Sensor networking", *Proceedings of Eight IEEE Int. Conf. on the Engineering of Computer - based Systems*, IEEE Computer Society Press, Washington, DC, pp. 262-8.

Williams, R. (2001), *Computer Systems Architecture – A Networking Approach*, Addison-Wesley, Wokingham, p. 660.