On the Distributed Realization of Parallel Algorithms

Klaus-Jörn Lange*

Fakultät für Informatik, Universität Tübingen

Abstract. This paper discusses some aspects of implementing parallel algorithms on distributed computer systems like a LAN–connected set of workstations. The notions of parallel and distributed computing are represented by their interrelation. The possibility of distributed simulations of parallel models is discussed. Finally, the complexity theoretical consequences will be addressed.

1 Introduction

The efficient and correct use of distributed computing systems is one of the most important and challenging topics of the present computer science. The abundance both of distributed problems and of distributed models can be roughly distinguished into two main branches: in the problem area of mastering distributed systems and in that of making efficient use of them. The latter case includes the task of speeding up computations by using parallelism. In this context, only few and rather easy synchronization problems like race analysis or dead-lock avoidance occur. But the process of speeding up cannot totally avoid to take these matters into account since every physical realization of a parallel model necessarily exhibits concurrent phenomena. The topic of this paper is to discuss certain aspects of the distributed realization of parallel algorithms on rather weak "parallel" systems like a set of Ethernet connected workstations. This research has been carried out in the projects KLARA and KOMET at the Technische Universität München and the Universität Tübingen.

The paper is organized as follows. We will first relate the notions of distributed and parallel computing and compare their properties. Then, the possibilities and limitations of bridging the gap between parallel and distributed models are discussed. Finally, some remarks concerning parallel complexity theory are given.

2 Parallel and Distributed Computing

This section considers distributed and parallel computing in their interrelation. First, it deals with the comparison of distributed vs parallel problems, models, and programming. Then, it considers the usefulness of parallel models as intermediate steps between parallel problems and distributed models.

* Supported by the DFG, Project La618/3-2

2.1 Parallel vs. Distributed

The aim of this subsection is to contrast and to compare the notions of parallel and distributed computing. Things will be (over)simplified in order to exhibit the characteristic distinctions of these two notions. A related and very interesting discussion referring to aspects of concurrency has recently been given by Panangaden [12].

Parallel problems, which ask for the possibility of decreasing running time by using parallelism, are a sub-area of *distributed problems* which also include questions of concurrency, nondeterminism, or liveness. While central distributed problems, like for instance correctness, are rather simple and uninteresting in the parallel case, issues like efficiency, the main concern of parallel computing, are up to now nearly irrelevant in the distributing setting. Efficiency is an objective and very crucial measure of success; there is no sense in any parallel solution, as correct or beautiful it might be, which gives no speed-up. While *concurrent problems* ask for the taming of distributed systems, parallel problems simply want to exploit them.

In the following, we use the term *distributed* problems in the general sense as the task of managing systems involving several processing units including coordination of the components of distributed systems. In contrast, we speak about *parallel* problems if we are solely interested in speeding up the computation. We remark here that every "physical" or realized system exhibits concurrent behavior. *Reality is not parallel, but inherently distributed.*

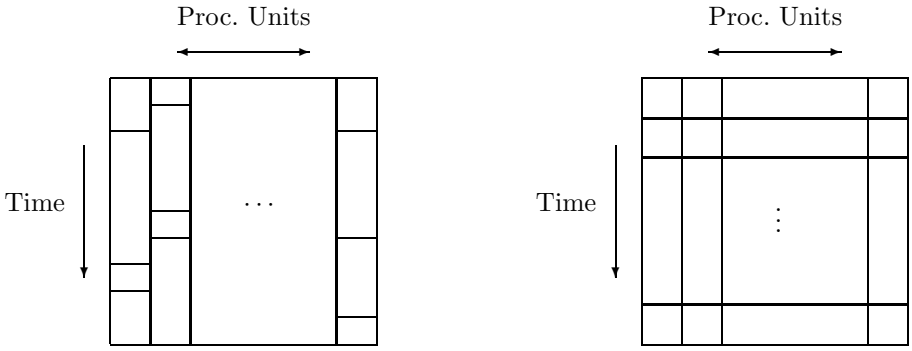
The confrontation of distributed vs. parallel also pertains to models and algorithms. *Distributed algorithms* are designed for distributed models, i.e., for systems composed of several more or less closely coupled processors exchanging messages. There is broad variety of distributed models. Throughout this paper we will consider rather weak distributed systems.

Also within parallel computing we find a large variety of models ranging from PRAMs with global memory to systolic nets with distributed memory. Which model should one choose for one's purposes? It depends, since there is sort of trade-off between ease of software development and hardware availability.

- The stronger the model, the easier to design and express parallel algorithms. In addition, it might be easier to give proofs of correctness.
- The weaker the model, the higher the chance to find an existing parallel system coming close to the chosen model.

Throughout the paper the following notation will be used in connection with parallel algorithms. The number of steps performed by a parallel program is called the parallel running time. Often we denote or bound it by $T(n)$ if n is the size of the input. When there is a related sequential algorithm we denote or bound its running time by $t(n)$. The *speed-up* is then the quotient $t(n)/T(n)$. The number of processors used by the parallel algorithm is denoted by $P(n)$. An algorithm is called *optimal* if the number of processors is linear in the speed-up or, equivalently, if the time-processor-product of the parallel algorithm is linear in the sequential time.

There are essential differences in programming parallel and distributed systems. A *distributed program* for a distributed system consists in a set of sequential programs which communicate with each other by the help of mechanisms like semaphores or message passing. On the other hand, a *parallel program* consists in a sequence of steps which are executed in a synchronized way by all processing units. These differences are indicated in the following diagram:



Distributed Computing **Parallel Computing**

Let us consider an example. The exchange of two register values a and b could be done in parallel by performing $a := b$ on one processor and $b := a$ at the same time step on another one. Running the same pieces of program in an asynchronous way leads to unforeseeable results. To put it in a more picturesque scene, consider acrobats throwing and catching dishes "in parallel". Without a perfect synchrony, a pile of broken fragments would result. Alternatively, a protocol of exchange messages like "Are you ready?" – "Yes. Please throw now!" – "Did you get it?" – "Yes, I got it." ... would preserve the dishes – or not ⋯ "Did you get it?" – "No. Did you throw it?" – "CLINK!".

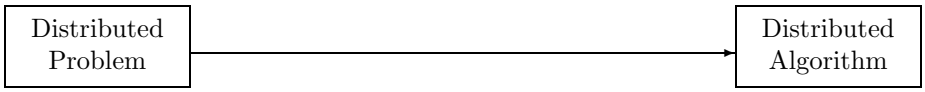
2.2 Distributed Solutions of Parallel Problems

Now let us discuss the process of solving parallel speed-up problems on available parallel computing systems. Since these necessarily show distributed behavior, typical problems of concurrency theory have to be solved during this process despite the fact that the original question of speeding up the running time needed to compute some function is free of these distributed aspects.

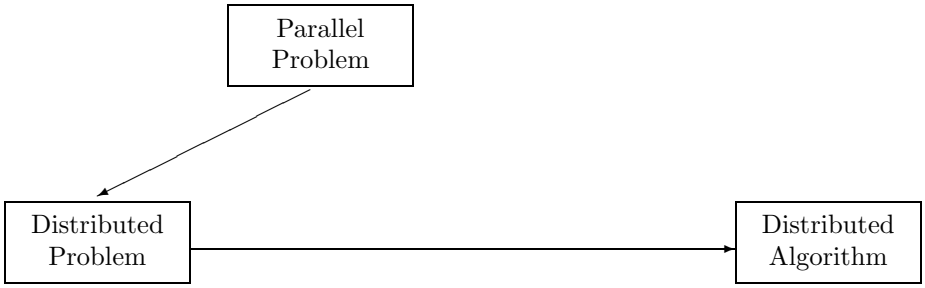
In the following we will assume the available hardware to be rather weak for instance a bunch of workstations coupled by an Ethernet. An appropriate formal model for this situation would be a set of *nodes* or active elements partially connected by a set of directed edges. There is no global clock; instead each node has its own program counter. The nodes communicate by sending and receiving messages along the edges. Typically, communications are very slow compared to local operations.

Any use of a machine or model like that involves the solution of intricate distributed problems. Let us symbolize this long way to go when solving distributed

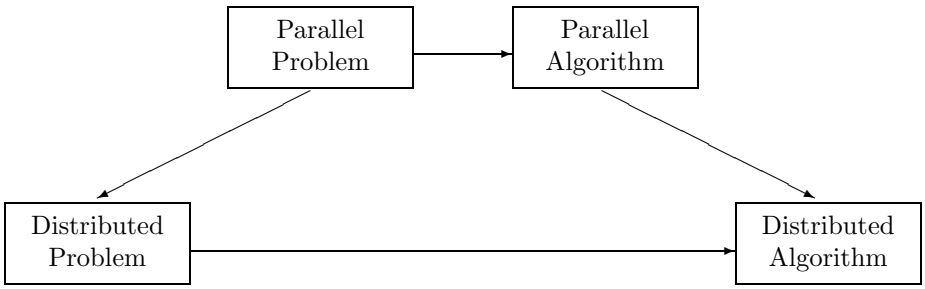
problems by constructing algorithms on distributed models by the following diagram:



The use of distributed systems in order to speed up computations is a subarea of the realm of all distributed problems. The nature of this kind of distributed problems exhibits fewer concurrent aspects and the process of finding a solution on the distributed system should be easier than the general case. In some sense it is a detour to treat speed-up questions simply like arbitrary distributed problems. This might be indicated by the following diagram:



The restricted amount of distribution in parallel problems should offer easier solutions. Further on, efficiency which is most important when trying to speed up running time is usually a rather unimportant and not formalized aspect of distributed problems. These are the reasons to use parallel models which serve as a platform to formalize and express parallel algorithms. But these models leave us with the task of efficiently transforming the parallel algorithm onto a distributed system. This situation is depicted in the following diagram:



Thus the problem of speeding up the running time needed to solve some computational problem is now divided into two steps: first, construct an algorithm on a parallel model and, second, simulate the parallel model on the distributed system. The first step of coming from a problem to a solution is much easier in

the parallel setting than in the distributed one. Since there are no concurrent problems involved the whole process is similar to the sequential case. But the more powerful the chosen parallel model is the easier is the step of finding a parallel solution to a speed-up problem. This is the reason for the abundance of algorithms designed for powerful parallel models like PRAMs or hypercubes.It is significant for these models that in general there are no efficient transformations known leading down to realistic distributed models.

It is the aim of this paper to discuss this "missing" second step under contradicting aspects like efficiency, correctness, ease of handling, or independence of the underlying distributed system.

3 Model Discussion

In this section parallel and distributed models are discussed and their main differences brought out.

3.1 Constants and Functions

First let us consider the degree of parallelism, i.e. the number of processing units. Off course, every real, existing machine has a bounded number of processors. Even if this number may vary in time it surely doesn't grow with the input size. Hence this number, at first sight, seemingly has to be modeled by a constant. But in order to get a both general and robust (i.e. model independent) theory constants are to be avoided.

On the parallel side of the gap the degree of parallelism will be treated as a resource measured as a function in the input length. This might be compared with the opposition of finite vs infinite memory. Although every existing computing system is finite, the appropriate models are infinite like Turing automata or register machines. In some sense even the simplest algorithm like a matrix multiplication can be regarded as a uniform family of algorithms: For each possible dimension of the input matrix there is one member of the algorithm family solving the matrix multiplication for matrices of this size.

There is no specific algorithmic theory for memory size of 16MB in contrast to 32MB and in the same way there is no specific theory for parallelization on machines with 16 processors in contrast to 32 processors.

It should be added that the danger of abusing this feature is higher for the degree of parallelism than for the memory size, since space is bounded by time which is in general not true for the degree of parallelism.

Thus on the parallel model the degree of parallelism is treated as a function of the input length.

On the other hand, on the distributed side, the number of processing units is modeled by a constant. This makes the first crucial difference between the parallel and the distributed side of the gap. The other differences of parallel and distributed models, described in the following subsection, point to fundamental difficulties in the process of simulating a parallel model on a distributed one.

But this is not the case for the number of processors, since it is no problem to simulate a larger number of processors by a smaller one without increasing the time-processor-product. In the contrary, the high degree of parallelism in many parallel algorithms could be regarded as a resource which can be exploited in order to get efficient implementations on distributed systems.

3.2 Local and Global Properties

There is an abundance of platforms for parallel programs reaching from systolic arrays to PRAMs. The common feature of all these models is that they assume *global time*: all participating processors are assumed to perform their steps in a synchronized way; no processor start its $i+1$st step before all processors finished the execution of their ith step. Global time is the basis for *global space*: all processors share a common address space and the access time to the memory is independent of the accessed address.

In the following, when speaking of the parallel model being the platform for some parallel algorithms, we refer to a model like a PRAM with global time and global space using an input dependent number of processors and programmed in the parallel way.

The world of distributed models is an unintelligible jungle, as well. Probably the weakest, but also most common form of an existing distributed computing system is a set of workstations connected by some local area network. Inspired by this the distributed model will have *local time*, i.e. each processing unit will have its own clock or program counter which is not adjusted by any "master clock". Further on, the model has *local space*: each unit has its own local memory and cannot access directly the memory of other unit but has to simulate that by message passing over the network. In addition, the number of units is bounded and independent of the size of the actual input to be solved. On the other hand, this number is not regarded as fixed since it might change by technical faults or successful applications for grants every day. Finally, the distributed model is used by a distributed program.

The gap to be bridged between the parallel algorithm and the distributed algorithm, depicted in the last diagram, can now be given in more detail:

4 On the distributed realization of parallel Models

A parallel model like a PRAM with all its properties cannot be realized in a scalable way for several physical reasons [18,19]. But even with a bounded number

of processing elements this seems to be at least expensive if it is to be done in a general way. One possibility is to use the fact that the degree of parallelism of a parallel program is much larger then the actual number of processing elements. This *slackness* leads to situation that every processor has to serve a large number of the parallel jobs. Using a powerful interconnection network it is possible to hide the cost for latency of the distributed machine behind the computation time for the many jobs [13,17]. This scheduling approach has the advantage that independently of the parallel program to be executed the running time of the distributed system will with sufficiently high probability be proportional to the parallel time-processor-product. On the other hand, it should be observed that this solution uses randomization and is not deterministic. Further on, its interconnection network is logarithmic and hence this approach is not scalable. In addition, this interconnection construction is intricate and cannot be simulated by a simple network like an Ethernet, not even in the case of a small number of participating processing elements. Remarkable effort in building a PRAM has been made by W Paul and his group [1]. Their activities even include the implementation of a PRAM-specific parallel programming language [6].

A totally different way of bridging this gap is to use the fact that in most existing interconnection networks it is cheaper to send one large message than to send many small ones. This leads to the exploitation of *locality*. The disadvantage of this approach is that it is not as general as the scheduling approach. Not every parallel algorithm provides enough locality and probably not every parallel problem allows for a parallel algorithm with enough locality. This gives the (complexity) theoretical issue to say, exactly which problems are efficiently solvable using locality, and the practical issue to say, how the efficiently solvable problems can be efficiently solved. In the following, we will not consider the strategy of *caching* to try to find implicit locality by working with large neighborhoods of single addresses. Instead we will follow the approach to handle locality explicitly in the parallel algorithm.

4.1 Using Locality

In this subsection we consider some aspects of using locality and propose strategies to use it[1]. They are presented a bit more detailed and with some few examples in [5].

Let us shortly review the gap between the parallel Model and the distributed one. We have a parallel algorithm, i.e. a sequence of parallel steps, working with a large number of processors which communicate via a shared memory at no extra cost. We want to execute or simulate this program on a distributed machine consisting of a smaller number of processing units which communicate by exchanging messages via a network. The time needed to exchange a message through the network is much larger than that to access local data.

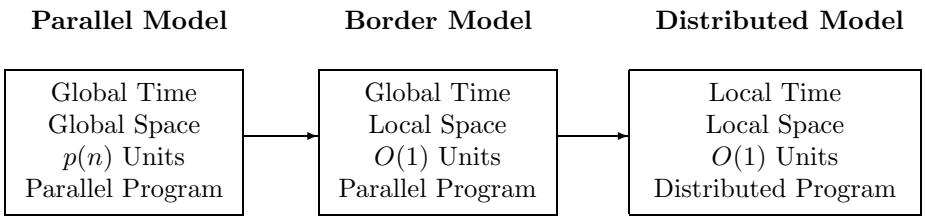
[1] Some of these theoretical results and their implementations can be found under http://www-fs.informatik.uni-tuebingen.de/forschung/komet.html.

Since the degree of parallelism is much larger than the actual number of processors, each node of the distributed system will simulate many of the parallel processes. Those of them which are laid on on the same physical node share an ideal "PRAM atmosphere": there are no synchronization problems and no differences between local and remote communication; they share global time and global space. Problems are caused by those processors which want to interact but are laid on different nodes. One way to cope with these problems is simply to consider only very restricted parallel algorithms.

- Admit only regular algorithms which allow to block many small communications into few large ones.
- Admit only simple algorithms for which the transfer from global time to local time can be done without too much overhead

Off course, this is not possible for all algorithms and is not clear which problems possess such algorithms. Nevertheless, there are simple and regular algorithms for many relevant problems like sorting or matrix multiplication.

Thus, for adequate algorithms we are faced with two major steps: the *distribution of space* and the *distribution of time*. It is a central dogma of this contribution that these two steps have to be performed in this order: first go from global space to local space and afterwards go from global time to local time. The first step asks for the solution of parallel problems and cares for efficiency. The distribution of time poses distributed problems. Its main objective will be the solution of concurrency problems. Hence we propose to work with an intermediate model with local memory but global time. The situation can be depicted as follows:



The Border model marks the frontier between parallel and concurrency problems. It is used by a *designer* (of space) for parallel algorithms, who cares for efficiency, knows properties of his algorithm, but doesn't know too much about distributed problems or tools offered by concurrency theory. The border model is implemented by an *expert* (of time) dealing with distributed problems. The expert gets formalized, explicit information about synchronization aspects of the parallel algorithm. This information is given in terms of declarations in the parallel program by the designer. The expert implements the parallel model on a distributed one and guarantees the correctness of his simulation as long as the conditions declared by the designer are fulfilled.

Distribution of Space: Virtual Topologies In this first step the gap from a purely parallel model to the weaker border model has to be bridged. Every parallel algorithm exhibits in its access to the global memory some pattern which we will call *virtual topology*. This doesn't pertain solely to the arrangement of the processors. For instance, both a typical 2D-grid algorithm like a red-black pebble game and Warshall's algorithm would name their processors by two indices. But while the grid-algorithm uses communications only between neighbors, Warshall's algorithm exhibits a clique communication on each column and on each row. Thus, the virtual topology is given by naming scheme of processing units together with a collection of *multicasts*. A multicast expresses the parallel movement of data in global memory possibly including concurrent read and write features. It consists in one or more sets of names of processing units acting as senders, one or more sets of units acting as receivers and information regarding which data to be send. Typical examples would be *row-multicast*, the sending of entry in a matrix to all element in the same row, or *par-row-multicast*, the execution of row-multicast for all rows in parallel.

The algorithm designer now writes his program by explicitly declaring the virtual topology of his algorithm. This is done by choosing an adequate virtual topology including its multicasts from a predefined set of topology types. The set of available topologies might differ between different implementations of the border model and depends in the topology of the distributed system to be used. This *real topology* might range from clique realized by a slow bus to a quick hypercube network or a two dimensional grid.

The virtual topologies available within an implementation of the border model are not implemented by the algorithm designer. The field of embedding and simulating topologies into other ones is too large to expect every algorithm designer to know all about it. He is only to use the virtual topologies for expressing his parallel algorithms.

The implementation of multicasts, i.e.the translation into communications on the distributed system follows the properties of the real topology. If the real system only allows sending messages from one sender along a connection to one receiver the simulation will transform every multicast statement into a collection of point-to-point communications. If, for example, the real topology provides broadcasting this would be used for the simulation of concurrent reads.

The efficiency of the simulation of a virtual topology off course strongly depends in the real topology and will be different on different implementations of the border model. These efficiencies are represented by *border parameters* which are values expressing the time consumption of the simulations of multicast statements in the actual implementation of the border model. These values can be accessed by the designer or even by his parallel program in order to select the appropriate available virtual topology. Assume, for instance, that for some subtask there is a very good parallel algorithm using a hypercube topology and a slower one using a simple grid. If now the border parameters of the hypercube topology are bad since the actual real topology is inadequate to simulate a hypercube, it might be better to switch to the grid algorithm to solve the subtask.

This decision could even be made at runtime by the parallel program reading the border parameters. Thus the text of the parallel program would be independent of the implementation of the border model. In this way we also regard the number of processing units to be a border parameter.

Thus it is the task of the algorithm designer to select depending in the actual border parameters an appropriate algorithm and to decompose it into as many pieces as there are processing units by dividing the parallel processors appropriately onto the processing units of the border model. This could be connected with some local modifications of the original parallel algorithm in order to increase the possibility to combine many small accesses to memory into one large data movement. Sometimes it can be useful to repeat computations and thus to increase the time processor product but at the same time to decrease the number of communications [5].

Distribution of Time: Synchronization Types The second step has to link the border model to the distributed system. Their main difference is the use of global and local time. Even if the program on the border model simply exchanges data via directed channels, this difference is crucial and in general it is not possible to execute directly this parallel program on the distributed machine without further synchronization. This is demonstrated by the following example. Assume the real topology to consist in two units P_1 and P_2 which are connected by two directed channels. Let now the parallel task be the exchange of the value of two cells of global memory G_1 and G_2. Than this could be done by the single line OROW-Program (owner read owner write, see [15]) where P_i executes $G_i := G_{3-i}$. Here P_1 is the read-owner of G_2 and the write-owner of G_1 while P_2 is the read-owner of G_1 and the write-owner of G_2. If this program would be executed without global time with high probability the content of one of the two cells would be lost.

A conceptionally simple method to provide global time is *barrier synchronization* where a node can only start a new step if all other nodes have performed their corresponding steps, which requires an enormous overhead of synchronization messages on a distributed machine. In addition, this method forces some processors to wait, even if they actually do not need any results from other processors at this point. Waiting for input values should be the only acceptable delay. This can be realized generally but inefficiently if each node stores each computed value of each step in a local list and sends such a value to another node whenever it receives a corresponding request. The transformation from global to local time should yield more efficient distributed programs in the case of a more restricted communication structures of the corresponding parallel program.

The idea to achieve this is to let the designer to give explicitly information relevant for synchronization of his program. This (s)he does by making *synchronization declarations*. These pertain either to elements of global memory or to single communications in terms of multicast statements. By assigning a *synchronization type* to a (global) variable certain conditions are guaranteed to be fulfilled by the parallel program. Thus a synchronization type is a condition

which when fulfilled by a variable or by a communication makes it possible to implement this variable or this communication with a certain expense of synchronization overhead. In general one might say that the weaker the condition of a certain type the more complex and expensive will be the protocol used in implementing the type. This information is then used in the transformation from global time to local time with the aim of minimizing synchronization overhead. It should be observed that the designer isn't confronted with any distributed Problem. He works and thinks in a non distributed world with global time where matters like liveness or correctness are comparable to the sequential case.

Assume, for example, the communications accessing a cell of global memory are *2-cyclic*, that is if a processor P writes into this cell a, then each processor Q reading a has to write something into another cell b which has later to be read by P followed later by P writing into a before Q may again read a. Then this cell could be declared as being of synchronization type *cycle* (together with the information of the cycle length, in this case 2). Then the access to this cell simply can be simulated by FIFO-queues of length 2 (See [5]). Examples of 2-cyclic algorithms are convolution or red-black-algorithms. Other examples of synchronization types are *write–determined* (each processor when reading from global memory knows the time the data it is going to read has been written [11]) and *zippered* (between any two writings into a cell at different global times there is a reading from that cell and vice versa).

If a global variable doesn't fulfill the conditions of a certain synchronization type in all multicasts accessing this cell it is nevertheless possible to give guarantees for some of them by declaring those which fulfill certain restrictions and thus need less synchronization overhead.

The task of the expert who bridges the gap between border model and distributed system is to implement virtual topologies in an efficient and correct way. To preserve efficiency he tries to find or construct the best possible embedding into the real topology and to use the given synchronization information in an optimal way. His transformations have to result in correct executions on the distributed system as long as the synchronization properties stated by the designer are fulfilled. Thus correctness proofs and the resulting distributed questions occur only for every implementation of a virtual topology and not for every parallel program using this implementation.

5 Complexity Theory

This section deals with complexity theoretical aspects of parallel and distributed computing. First, a short overview of current parallel complexity in terms of classes like NC and P is given. This is followed by a discussion of the disadvantages of this approach. Finally, we discuss some possibilities to avoid these problems.

The reader of this section is assumed to be familiar with the basic facts of sequential and parallel complexity theory as they are contained for instance in [2,3].

5.1 Parallel Complexity Theory

The original aim of complexity theory is to classify computational problems according to the amount of time or of other resources needed for their solution. The problems of proving matching upper and lower bounds remain in general unsolved and led to the comparison of complexities and to notions like hardness and completeness.

In the the parallel case it is not useful to consider just the time needed for a solution. It seems necessary to bound simultaneously the degree of parallelism. Observe, that complexity classes defined by simultaneous resource bounds are more difficult to handle than the usual ones. For instance, they lack sharp hierarchy theorems.

Using the complexity theoretical tools which yielded the successful notion of *NP*-completeness the class *NC* of all problems possessing efficient parallel algorithms has been introduced. While the *P*-completeness of a problem stands for its inherent sequential nature, membership in *NC* is regarded as an indication that the running time of a problem can be decreased dramatically using parallelism. The class *NC* is extremely robust against modification of the model. It is defined by polylogarithmically time or depth bounded PRAMs, circuits, or alternating machines. This uniform picture changes if logarithmic instead of polylogarithmic time and depth bounds come into consideration. This results in three levels. The most powerful one, *unbounded fan-in parallelism*, is represented by CREW and CRCW PRAMS, circuits of unbounded fan-in, or depth bounded alternation and contains as a typical class AC^1. The second, weaker level is characterized by space bounded *sequential computations* and is represented by space bounded sequential models with or without recursion. A typical representative is the class $DSPACE(\log n)$. The lowest level, *bounded fan-in parallelism*, is characterized by circuits of bounded fan-in or time bounded alternation and leads to the class NC^1.

5.2 Problems of NC-Theory

As explained above, the use of reducibilities led to the opposition of *P*-completeness regarded as a sign of being inherently sequential vs membership in the class *NC* of all problems which have "efficient parallel algorithms". But this classification has the clear drawback that it looks for the reduction of polynomial running time down to polylogarithmic one, i.e. for exponential speed-up, without caring for the time-processor-product and optimality. Vitter and Simmons were able to construct a *P*-complete problem which has an optimal parallel algorithm with polynomial speed–up [20]. This phenomenon is caused by the use of reducibilities which allow polynomial padding when defining the class *NC*. In order to avoid this it seems necessary to restrict the growth of the reducibilities by linear or nearly linear functions. There are some attempts to define classes capturing polynomial speed-up [20,7]. There is even a "hardest sequential problem" in *P*, which has polynomial speed–up if and only if this is true for every member of

P [14]. But these notions lack the existence of an adequate notion of reducibility and didn't gain the relevance of a notion like NP-completeness.

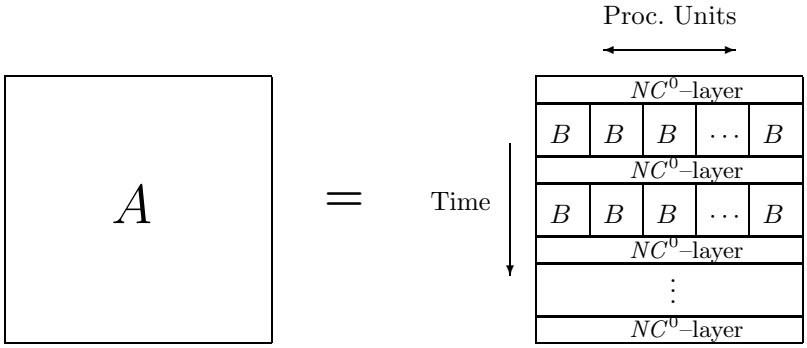
It should be remarked here that the many approaches to define more realistic models as for instance the LogP model [4] are adequate in order to get good estimations of running times, but they don't seem to be appropriate for structural classification of being inherently sequential in contrast to being efficiently parallelizable.

5.3 Classification by criteria

A way out of this dilemma could be to classify not by criteria that are based on reducibilities like hardness and completeness but instead to find other properties of problems or algorithms which may serve as sufficient or necessary criteria for the existence or nonexistence of efficient parallel solutions.

A striking example is the notion of obliviousness or *data–independence* [8]. It was inspired by the observation that most of the algorithms which showed an acceptable speed-up on a net of workstations exhibited a static pattern of access to data. More formal, assume that we work with a CRCW-PRAM algorithm. Then we can consider the read- and the write-access of this algorithm to the global memory formalized as a graph. If these structures are independent of the actual input data but only depend in the size of the input we speak about data-independent read and data-independent write. If both read and write are data-dependent we have no restriction at all and get the full power of unbounded fan-in parallelism. For instance, in logarithmic time we can solve exactly the problems in AC^1. If, however, we restrict the write to be data-independent and let the read data-dependent we get space bounded sequential classes! In logarithmic time we now get the class $DSPACE(\log n)$. And if we work with static communication patterns where both the read and the write is data-independent, we end up with bounded fan-in parallelism. In logarithmic time we get exactly the class NC^1. This might be interpreted in the way that parallel models are related to unbounded fan-in parallelism while distributed models are related to bounded fan-in parallelism. Since space bounded sequential computations seem to be more powerful than bounded fan-in parallelism (unless $DSPACE(\log n)$ and NC^1 coincide) it is no surprise that a property like data-independence cannot be treated by traditional complexity theory: Their reducibilities are originally based on sequential devices which cannot preserve data-(in)dependence.

In this connection the notion of a problem being *recursively divisible* introduced by R. Niedermeier is very interesting [10,9]. He calls a problem A divisible if there is an algorithm solving A in time $T(n)$ using $P(n) = n^\epsilon$ processors on inputs of size n which consists in $O(1)$ alternating layers of simple and regular communications followed by layers consisting in $P(n)$ independent applications of some algorithm solving a problem B on inputs of size $n/P(n)$ as indicated in the following picture. (Here simple and regular is formalized as being NC^0–computable.) Obviously, an algorithm like that can be easily implemented on a distributed system with acceptable efficiency.



Algorithm for A

If now B is identical to A we can repeat this process of decomposition. In this case A is called *recursively divisible* [10,9]. Many important computational problems like sorting or matrix multiplication are recursively divisible. All known examples of recursively divisible problems are members of NC^1, a class defined by bounded fan-in parallelism. This supports the impression that distributed models are closely related to bounded fan-in parallelism.

6 Discussion and open Questions

One aim of this paper was to relate and link rather separated areas. Much of this contribution is unfinished and speculative. Only few constructions have been implemented until now. One reason for this are the many open question of these areas.

A very important task to settle is to handle recursion properly. On the theoretical side a proper definition of benign recursion is missing. Recursion is necessarily dynamical and not static. Hence we have to deal with data-dependence. While there are well-known examples of malign cases of data-dependence like pointer jumping there also many cases of dynamic data-dependent algorithms which exhibit a very local communication pattern like many approximation algorithms. Neither there is a theoretical treatment of this dynamic case nor there seems to exist an adequate language to express and formalize this phenomenon. (An interesting approach to express (non)locality was given by Sabot [16]).

Another weakness of the concept of virtual topologies is that often algorithms exhibit virtual topologies which only come close to a more regular one offered in an implementation of the border model. For these cases something like an editor for topologies would be helpful which could be used by the algorithm designer to tailer an existing virtual topology to something more appropriate. The difficulties to keep efficiency and correctness of the implementation of the modified virtual topology are incalculable. One further problem would be that there would be no adequate border parameters for the modified topology.

One task of the designer is to embed his parallel algorithm using n processors where n depends in the input size into a border program using a much

smaller number of k processing units. Once he did this, it is in many cases no problem to modify his construction to be executed with a different number k' of units as long as the topology stays similar. It should be possible to do this computer aided. Questions are here, how to express and formalize this process on the syntactical level and, second, can those problems which are adequate for this (semi-)automatic lay out, be characterized in complexity theoretical terms. These question are also interesting for the case of a weakly dynamic real topology where the set of available nodes may change day by day.

While these problems concerned the distribution of space there are also many open questions in connection with the concept of synchronization types. First, there is the question for more reasonable synchronization types which are general enough to be useful and which are restrictive enough to be implementable without much synchronization overhead. Unclear is the general relation between the different synchronization types and their structural behavior. Is it possible to characterize synchronization types in terms of complexity theory? Are there connections to concepts like completeness or to the other criteria mentioned in subsection 5.3?

A specific feature sometimes found in parallel algorithms is that they are robust against certain differences in local time. Consider for example a monotone algorithms like Warshall's algorithm. The participating memory cells a_{ij} carry either the value 0 (there is no path from i to j) or 1 (there is a path from i to j)) and never a 1 is overwritten by a 0. That means that the algorithm could tolerate results which are too young, i.e. come out of the future (wrt global time). What has to be avoided is to receive data which is too old. It should be possible to exploit this behavior and to introduce a synchronization type named, may be, *best before*.

Acknowledgment

I like to thank Henning Fernau, expert for rewriting and formal languages, for rewriting the introduction.

References

1. F. Abolhassan, J. Keller, and W. J. Paul. On the cost-effectiveness and realization of the theoretical pram model. SFB Report 09/1991, Universität des Saarlandes, Saarbrücken, 1991. revised and extended version of SFB Report 21/1990.
2. J. Balcázar, J. Díaz, and J. Gabarró. *Structural Complexity Theory I.* Springer, 1988.
3. J. Balcázar, J. Díaz, and J. Gabarró. *Structural Complexity Theory II.* Springer, 1990.
4. D. Culler, R. Karp, A. Sahay, K. E. Schauser, E. Santos, R. Subramonian, and T. von Eicken. LogP: Towards a realistic model of parallel computation. In *Proc. 4th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 1–12, 1993.

5. D. Gomm, M. Heckner, K.-J. Lange, and G. Riedle. On the design of parallel programs for machines with distributed memory. In *Distributed Memory Computing, Proc. 2nd European Conference, EDMCC2*, volume 487 of *LNCS*, pages 381–391. Springer, 1991.
6. T. Hagerup, A. Schmitt, and H. Seidl. FORK: A high-level language for PRAMs. Technical Report 22/90, Universität des Saarlandes, Fachbereich 14, Im Stadtwald, 6600 Saarbrücken, 12 1990.
7. C. P. Kruskal, L. Rudolph, and M. Snir. A complexity theory of efficient parallel algorithms. *Theoret. Comput. Sci.*, 71:95–132, 1990.
8. K.-J. Lange and R. Niedermeier. Data-independences of parallel random access machines. In *Proc. of 13th Conference on Foundations of Software Technology and Theoretical Computer Science*, number 761 in LNCS, pages 104–113. Springer, 1993. Accepted for publication by JCSS.
9. R. Niedermeier. Recursively divisible problems. In *Proc. of the 7th ISAAC*, number 1178 in LNCS, pages 183–192. Springer, 1996.
10. R. Niedermeier. *Towards Realistic and Simple Models of Parallel Computation*. PhD thesis, Universität Tübingen, 1996.
11. N. Nishimura. Restricted CRCW PRAMs. *Theoret. Comput. Sci.*, 123:415–526, 1994.
12. P. Panangaden. Does concurrency theory have anything to say about parallel programming? *EATCS Bull.*, 58:140–147, 1996.
13. A. G.. Ranade. How to emulate shared memory. *J. Comp. System Sci.*, 42:307–326, 1991.
14. K. Reinhardt. Strict sequential P-completeness. In *Proc. of 14th STACS*, number 1200 in LNCS, pages 329–338. Springer, 1997.
15. P. Rossmanith. The owner concept for PRAMs. In *Proc. of the 8th STACS*, number 480 in LNCS, pages 172–183. Springer, 1991.
16. Gary Wayne Sabot. *The Paralation Model*. MIT Press Cambridge Massachusetts., 1988.
17. Leslie G. Valiant. A bridging model for parallel computation. *Communications of the ACM*, 33:103–111, 8 1990.
18. Paul M.B. Vitányi. Nonsequential computation and laws of nature. In *VLSI Algorithms and Architectures*, number 227 in LNCS, pages 108–120. Springer, 1986.
19. Paul M.B. Vitányi. Locality, communication, and interconnect length in multicomputers. *SIAM J. Comp.*, 17:659–672, 1988.
20. J. S. Vitter and R.A. Simons. New classes for parallel complexity: A study of unification and other complete problems for P. *IEEE Trans. on Computers*, 35:403–418, 1986.