On the Design of Parallel Programs for Machines with Distributed Memory *

Dominik Gomm, Michael Heckner, Klaus-Jörn Lange, Gerhard Riedle
Technische Universität München, Institut für Informatik, Arcisstr. 21, D-8000 München 2

1 Introduction

The efficient and faultless use of parallel systems, in particular problems concerning the programming of parallel computers belong to the most important issues of computer science. These problems may be classified into two main categories: "*Distributed Parallelism*" deals with mastering concurrency and "*Speed-up Parallelism*" tries to improve on computation time by employing many processors to solve one task. Whereas the former treats questions of correctness, the latter — which is what we are concerned with — is determined by matters of efficiency. Problems of both areas are even becoming more difficult by the desire for an easy and clear way of programming parallel systems — if possible, similar to conventional programming.

These needs are fulfilled more easily on shared memory systems, where "synchronous" programming with a global clock yields both simplicity and good efficiency. On parallel systems with distributed memory (like the iPSC/2 Hypercube, Transputer Systems and others) both aims have at least to be reached independently or may even be considered as conflicting aspects. Programming distributed systems usually leads to rather coarse grained parallelism as the complexity of synchronization and communication prohibits a more fine grained implementation. Although this coarse grain approach is supposed to lead all the way back to conventional programming, we are then left with severe synchronization and consistency problems. In addition, this approach seems to work with algorithms that are characterized by a minor communication load, only. But, increasing the number of processors in a parallel system leads to a situation where technological constraints forbid "clocked" systems with a shared memory (e.g. [Vit86]).

The aim of this paper is to indicate a procedure for a convenient and machine-independent design of efficient programs for a large class of algorithms. This will be accomplished by hiding problems concerning concurrency and restrictions due to the communication topology of the underlying machine as far as possible. We want to point out that we do not think of an efficient implementation of a "universal" PRAM, but of developing and studying classes of problems or algorithms. These classes can be implemented in an efficient and convenient way. The expectation that these classes will nevertheless contain very relevant problems is supported by the fact that at present most computation-intensive "numerical" algorithms are of very regular structures allowing efficient partitionings.

2 Shared Memory Programming and its Desynchronization

In parallel complexity theory, one of the best known models describing parallel algorithms is the PRAM — the parallel random access machine. (See [Coo85] for an overview.) A PRAM can be

*This work was carried out within the scope of DFG-SFB 342 by subprojects A3 and A4

thought of as a collection of processors (each of which having local memory) working in a synchronous way controlled by a global clock and communicating via *global memory*. There is a variety of PRAM-models depending on how memory-conflicts are resolved. While the most general model, the *concurrent read concurrent write* CRCW-PRAM is a system with shared memory, each global memory cell of a *owner read owner write* OROW-PRAM is the conception of a directed communication channel. Hence, an OROW-PRAM is a (clocked) network of processors with distributed memory (see [Ros90]). We decided to consider the intermediate model, the CROW-PRAM, which already has some of the features of a real distributed memory computer.

As indicated above, it is relatively simple to design efficient algorithms on a synchronous parallel system with shared memory (i.e. the speed-up is proportional to the number of processors involved) and it is relatively simple to show their correctness. However, the PRAM-model makes some drastic and totally unrealistic assumptions: (A1) PRAMs work synchronously, i.e. the processors execute statements of unit length in a clock-controlled way. (A2) Processors have unbounded communication, i.e. each pair of processors can transfer data in unit time via a *global* (shared) memory. Thus, there is no difference in the time delay between local and remote communication. (A3) PRAM-algorithms use an unbounded amount of hardware, i.e. the number of processors involved in a PRAM computation is not fixed but depends (polynomially) on the length of the input.

Since the number n of virtual processors required by an PRAM-algorithm is usually much bigger than the number k of real processors of any existing parallel computer, there is the need to execute many virtual steps on one real node. This implies however that there is an ideal "PRAM-atmosphere" for these combined PRAM-processors in that there is no synchronization problem and no difference between local and remote communication, i.e. assumptions A1 and A2 are trivially fulfilled for PRAM-processors laid onto the same physical node. The problem is, of course, how to handle communications between PRAM-processors which are laid on different real nodes.

There seem to be two major ways to cope with this inhomogenity. According to one conception, all of the n virtual PRAM steps, which are to be executed in parallel, are distributed over the k real processors in a randomized way hoping for an equal load of all remote communications. Then each processor performs sequentially $\frac{n}{k}$ steps that were assigned to it. Each remote–step is handled by a communication–unit, such that (in the average) no processor has to wait for the execution of a remote–step, but instead can begin to execute the next of its $\frac{n}{k}$ steps [Val90]. This approach ends up in a flooding of the underlying network with a huge amount of small messages and in addition seems to pay off only if the ratio of remote to local communication time is not too far beyond 1. The other approach uses the fact that in most systems with distributed (nonuniform) memory the high cost (in terms of time) for a remote communication consists of a comparatively high *startup time* added to the usual transmission cost, which is linear in the length of the transmitted information and roughly corresponds to the cost of a local communication. Furthermore, it is possible to save in the cost of the startup time by *building blocks of communication*. Thus, it is very important within this approach to arrange and cut the n virtual steps in a way, that both the number of such blocks of communication and their sizes are balanced and equally shaped for all k processors. This is only possible if the communication structure of a parallel program is determined before run–time, i.e. we can consider algorithms with *data–independent communication structure* only. Thus, this second approach differs from the first one by providing very efficient programs for only a certain subclass of algorithms and by involving the user in the procedure. This is the most essential step of partitioning w.r.t. efficiency. The task is presumably unsolvable in an asynchronous environment.

This is why we treat the design of parallel programs in two major steps, *partitioning* and *desynchronizing*; this has to be done in this order. At first, in the *partitioning step* an appropriate PRAM algorithm is laid out on a synchronous PRAM with a small (i.e. realistic) number of processors in a way that the cost of inter-processor communication is reduced to a minimum. In order to avoid confusions we will call this machine in subsequent sections of this paper a *synchronous superstep machine* (SSM). Then, in the *desynchronizing step* the synchronous program is to be transferred onto an *asynchronous distributed memory machine* (ADMM) with the same number of processors using message passing mechanisms. As the global clock-pulse is missing in a distributed system we

have to find means and ways to replace it. It turns out, that even in the case of complete information about how to build communication blocks optimally, this results in inefficient programs, unless we restrict ourselves to a small class of parallel algorithms with a very restricted communication structure. We intend to solve this inefficiency problem by allowing the user to give information concerning the asynchronous behaviour of certain parameters of his program. The succeeding *desynchronizing transformation* desynchronizes correctly and as efficiently as possible according to the given information the program resulting from the partitioning step. (In a certain way, data–independency of communication structure is just one example of this kind of information). These two steps shall now be presented in more detail.

2.1 Partitioning

Partitioning consists of three substeps: *cutting*, *building blocks of communication* and introducing *subbroadcasts*. The basic assumptions of the target machine, the SSM, that have to be met are: (S1) All processors work synchronously, (S2) global memory is still assumed but the OROW quality now required actually gives it the character of distributed memory and (S3) finally, the number of processors is restricted to a small (compared to n) value of k with respect to existing parallel architectures. Given this SSM architecture, we now want to present the three substeps that will carry PRAM programs onto SSM "hardware".

2.1.1 Cutting — Building Blocks of RAMs

Naturally, the partitioning starts with a reduction in parallelism. However, this goal is not achieved by some sort of multitasking of a certain amount of RAM programs on each SSM processor. [Val90]. Our basic idea is to cut the whole PRAM program (usually consisting of an enourmous amount of n RAM programs) into parts in a way that each SSM processor obtains one program simulating the work of many RAMs. We want to stress that many RAM programs are merged in one program for each SSM processor. Consequently, within each piece of the original program there is still the ideal PRAM athmosphere as described in assumptions A1, A2 and A3. Only the connections crossing the frontiers of SSM processors will cause severe problems with respect to that model, i.e. communication and synchronization. We leave synchronization problems essentially to the second major phase, the desynchronizing step. Hence, the following criteria are of primary concern to achieve an efficient cutting: a) Equal loads for all SSM processors have to be produced and b) the pattern of communication, that is the algorithm's characteristic communication structure (which is pratically never a fully interconnected one!) has to be considered in order to minimize external communication.

With this substep we introduce the concept of a superstep. A superstep in the program of a SSM processor comprises the execution of exactly one computational step of each of the RAMs that it has to simulate.

Having selected one cut of the PRAM algorithm a suitable sequential order of simulation has to be found. This is the second task that has to be solved within the substep cutting. Again, it may have considerable influence on the feasibility of building blocks of communication. (See convolution example).

A third task to be tackled by the cutting substep is the correct handling of non-monotonous algorithms. The following statement is a typical element of non-montonous parallel algorithms: "$a :=: b$". Sequentializing this parallel statement correctly, entails that additional variables are introduced that protect variables a and b from being overwritten too early. However, given monotonous programs, the cutting could support a more efficient sequentializing of the PRAM algorithm due to some declaration given by the user indicating montony.

2.1.2 Building Blocks of Communication

the second substep deals with the analysis of external communication. Within each superstep communication with external sources should be analysed. This is aimed at finding a number of elementary (i.e. derived from single RAMs) read statements that access the same source SSM processor (but possibly different RAMs within that SSM program). This allows for reading these values within one communication setup before the execution of that superstep actually begins. Obviously, a clumsy cutting prohibits a successful building of blocks of communication. Therefore, it is realistic to conceive the cutting and the building of blocks as the two constituents of a feedback cycle.

This substep can be seen as a kind of local optimizing. For each SSM processor its read access to other, i.e. external SSM processors has to be optimized. It is the aim to give each SSM program the most efficient provision with external data (i.e. in blocks of messages).

2.1.3 Subbroadcasts

In contrast to the previous substep the subbroadcast step deals with the global aspect of communication. As we started with CROW PRAMs and, until now, did not resolve the concurrent reads, a certain, although reduced, amount of concurrent reads among the SSM processors is retained. To get in control of this concurrent access, the concept of subbroadcasts is introduced. To put it clear, the user does not resolve the concurrency. He simply "marks" it by using the subbroadcast statement. The actual resolution is done within a library function that is called by the subbroadcast statement.

Before having a more detailed look at subbroadcasts, we have to elaborate on the idea of a decomposition topology. [Fea88]

Keeping in mind the presumptions of hardware-independent programming on the one hand and the restricted interconnection topology typical of all large-scale parallel systems on the other hand, we propose standard decomposition topologies. Already indicated by the name this aid offers standard communication topologies for our decomposed PRAM algorithm, i.e. the algorithm after cutting and building blocks of communication. These topologies should guarantee the best mapping from this decomposition topology into any communication topology of real hardware systems. During the partitioning stage the user can select any of the decomposition topologies given in some sort of a library suitable for his algorithm. We think especially of grids, hexagonal meshes, hypercubes and trees. For example, matrix computations usually will be well represented on grids.

A subbroadcast enhances the expressive power of decomposition topologies as follows. Instead of restricting communication to the direct neighbours within the topology, communication is always possible in well-formed, distinguished subsets of the topology, e.g. within a 2-dimensional grid it is possible to address all menbers of a row or a column — the "dimensions" of the topology. The communication pattern of the algorithm has to fit into the expressiveness of subbroadcasts on the selected decompostion topology.

Resuming the discussion of concurrent reads, this third substep of partitioning finds all concurrent reads (that is all one-to-many communication links) and embeds them in subbroadcasts. Please note, that in our understanding the term subbroadcast does not imply any assumption about the direction of information flow, i.e. one sender, many receivers. It simply denotes the pattern of communication.

2.1.4 Comments on the Substeps of Partitioning

Considering the sequential order of substeps it is obvious that the partitioning stage has to start with cutting. But why is the building of blocks prior to subbroadcast treatment? The inverse order would not only inflict a concurrency of $O(n)$ to be dealt with where n is the enourmous number of RAMs instead of $O(k)$, the small number of SSM processors but also prohibit a successful building of blocks of communication in some cases.

All substeps are supposed to be executed prior to any compilation and especially prior to execution of the program.

It is evident that no data-dependent algorithm such as any kind of pointer jumping will render this approach useless. There is no basis on which a sound building of blocks could rely. Furthermore, restriction to subbroadcasts cannot possibly be efficient as communication connections are unpredictable.

2.1.5 Examples

Partitioning will be described by two examples. Not every detail mentioned above will be demonstrated. The two examples differ in the time-dependence of their communication pattern. The first example, convolution, is marked by an invariable pattern as time passes on.

Convolution

P_i: **repeat** $g(t)$ **times**
⌈ ≪ step ≫
processorswitch i
case 1 : $A_1 := f(0, A_1, A_{i+1})$
case n : $A_n := f(A_{i-1}, A_i, 0)$
default : $A_i := f(A_{i-1}, A_i, A_{i+1})$
endswitch
⌊ ≪ endstep ≫

Figure 1: PRAM Convolution Algorithm (for any PRAM i)

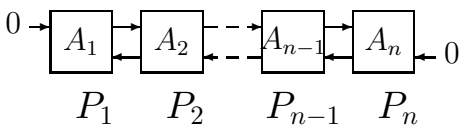


Figure 2: Communication Structure of the PRAM Convolution Algorithm

Let us assume that we have n RAMs and only k SSM processors (nodes), where $n \gg k$. There is one obvious way to cut this algorithm. It simply combines $\frac{n}{k}$ neighbouring RAM programs in one SSM processor program, thus giving us the k programs needed for the SSM. The result of this simple cutting can be seen in figure 3. Each node simulates the $\frac{n}{k}$ RAMs laid onto it from left to right within each superstep. Obviously, there are no elementary read access statements within each SSM processor Q_j that could become blocks because they address the same source SSM processor.
It seems worth mentioning that the change in sequential order of the simulation of RAMs would yield a possibility of blocking:
Choosing **snakelike** execution means that each node first simulates the $\frac{n}{k}$ RAMs within a superstep from left to right, followed by a superstep reversed in order. Thus the bordering memory location at the right or left end, respectively is updated twice in two successive computation steps. Both values are retained and combined in one block of communication. Thus computation only takes place after each "megastep" comprising to supersteps. This change in the sequential order of events cuts communication frequency in halves and doubles the length of each communication block which will make up to a considerable performance increase.
The **overlap** idea is based on redundancy. An overlap of o can be explained as follows: Each node starts with simulating not only its own $\frac{n}{k}$ RAMs but also a certain number of RAMs , precisely $2o$ RAMs, that belong to its right and left neighbouring nodes — o of them to each. Thus we can execute o succeeding computation steps without any communication by simulating a decreasing number of overlapping RAMs. Therefore we create a "megastep" on each node comprising o computational steps on $\frac{n}{k}$ RAMs plus the additional computation necessary for the overlapping RAMs. The message length is increased from 1 element to o elements, message frequency however is curbed drastically by factor of o.
In fact, the standard partitioning proposed above is a special case of overlapping: $o = 1$.

topology grid(**dim** 1, **proc** k)
Q_j: **repeat** $g(t)$ **times**
⌈ ≪ superstep ≫
$A_l :=$ **subbroadcast** (**row** , **sender** $\forall_{i \in \textbf{row}}\ Q_i$, **receiver** Q_{i-1}, **value** $A_{\frac{n}{k}}$)
$A_r :=$ **subbroadcast** (**row** , **sender** $\forall_{i \in \textbf{row}}\ Q_i$, **receiver** Q_{i+1}, **value** A_1)
for $i = 1(1)\frac{n}{k}$ **do**
switch i
case 1 : **if** $j = 1$ **then** $A'_1 := f(0, A_1, A_2)$ **else** $A'_1 := f(A_l, A_1, A_2)$
case $\frac{n}{k}$: **if** $j = k$ **then** $A'_{\frac{n}{k}} := f(A_{\frac{n}{k}-1}, A_{\frac{n}{k}}, 0)$ **else** $A'_{\frac{n}{k}} := f(A_{\frac{n}{k}-1}, A_{\frac{n}{k}}, A_r)$
default : $A'_i := f(A_{i-1}, A_i, A_{i+1})$
endswitch
end
for $i = 1(1)\frac{n}{k}$ **do** $A_i := A'_i$
end
⌊ ≪ endsuperstep ≫

Figure 3: Obvious Partitioning of the Convolution Algorithm

Warshall

The Warshall algorithm is the most popular algorithm for solving the problem of the transitive closure. As you can see from figure 5 the Warshall algorithm is characterized by a permanent change of communication partners. However, the communicating pairs of processors can be anticipated — they can be derived from the number of the computation step.

P_{ij}: **for** $r = 1(1)n$ **do**
⌈ ≪ step ≫
$A_{ij} := A_{ij} \vee (P_{ir}.A_{ir} \wedge P_{rj}.A_{rj})$
⌊ ≪ endstep ≫
end

Figure 4: PRAM Algorithm

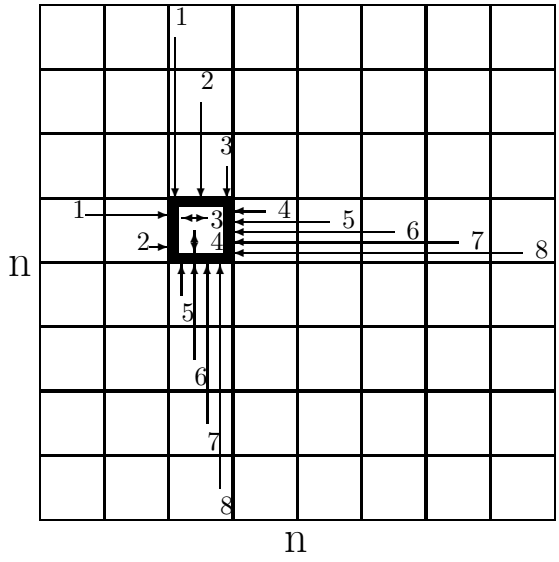


Figure 5: Warshall's Algorithm on an 8×8 Matrix (on RAM P_{43})

We will present two possible cuttings that can be ranked by the ability to build blocks of messages. For the time being, we will assume that we have n^2 RAMs but only k nodes. In the PRAM case every matrix cell has got a RAM doing the n computation steps necessary to determine the resulting value of this cell.

Partitioning the matrix by the **row**
$\frac{n}{k}$ rows of n RAMs each are merged in one node program. Since each superstep needs the values of one distinct row of the matrix these values can be transferred in one block and thus increase

communication efficiency.
Note that the case of partitioning by the column is symmetric.

Partitioning the matrix by the **tile**
One node simulates a square of $\frac{n}{k'} \times \frac{n}{k'}$ neighbouring RAMs where $k'^2 = k$ nodes. This time each superstep operates on two message blocks: successive values of one section of a row and successive values of one section of a column. The section size equals the length of a tile. Note that this way of partitioning reflects the basic structure of the Warshall algorithm. The result is shown in figure 6.

It will turn out that this partitioning by tile results in an even more efficient SSM-algorithm with respect to communication than the partitioning by the row. (c.f. table 1). The total amount of information that has to be exchanged is smaller.

topology grid(**dim** 2, **proc** k)
Q_{rs}: **for** $t = 1(1)k'$ **do**
for $u = 1(1)\frac{n}{k'}$ **do**
$\lceil \ll$ superstep $\gg$
$(A_{in_{row}1}, \ldots, A_{in_{row}\frac{n}{k'}})$:= **subbroadcast** (**row** , **sender** $Q_{\mathbf{row}\ t}$,
receiver $\forall i \in \mathbf{row}\ Q_{\mathbf{row}\ i}$, **value** $A_{1u}, \ldots, A_{\frac{n}{k'}u}$)
$(A_{in_{col}1}, \ldots, A_{in_{col}\frac{n}{k'}})$:= **subbroadcast** (**col** , **sender** $Q_{t\mathbf{col}}$,
receiver $\forall i \in \mathbf{col}\ Q_{i\mathbf{col}}$, **value** $A_{u1}, \ldots, A_{u\frac{n}{k'}}$)
for $v = 1(1)\frac{n}{k'}$ **do**
for $w = 1(1)\frac{n}{k'}$ **do** $A_{vw} := A_{vw} \vee (A_{in_{row}v} \wedge A_{in_{col}v})$
$\lfloor \ll$ endsuperstep $\gg$
end

Figure 6: Partitioning by the Tile

Partitioning	messages		
	amount of messages	length	information total
by the tile	$2kn = 2k'^2n$	$\frac{n}{\sqrt{k}} = \frac{n}{k'}$	$2\sqrt{k}n^2 = 2k'n^2$
by the line	$kn = k'^2n$	n	$kn^2 = k'^2n^2$

Table 1: Comparison

2.2 Desynchronizing

In the second major step of transformation, called **desynchronization**, programs from the synchronous SSM stage are transfered onto an asynchronous distributed memory machine ADMM. Thereby, all access to global memory has to be replaced by message passing mechanisms. Read and write operations in a SSM superstep are replaced by **asynchronous send and receive operations** on the ADMM level.

As the ADMM is a model of a distributed system we can no longer rely on a global clock signal. Therefore, we have to use suitable synchronization mechanisms to simulate the missing global clock pulse such that every node operates on the correct data in each computation step.

The result should always be an efficient program where nodes can perform their local steps as independently as possible. Therefore, we are not interested in solutions like barrier–style synchronization [Val90] where a node can only start a new step if all nodes have performed their corresponding steps, which in addition requires an enormous overhead of synchronization messages on a distributed machine. 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. Desynchronization should yield more efficient distributed programs in the case of a more restricted communication structure of the synchronous program. If the communication structure does not depend on input data, it is predictable, such that requests for values and local lists can be omitted. Simple submission of values which must be indicated by the step they are computed at is sufficient to guarantee that all nodes are working on the correct data. Furthermore, if the communication structure is constant in time, i.e. the same in every step, indication can be omitted if we assume FIFO buffers.

Besides cheaper synchronization mechanisms, faster message passing system calls can be used to increase efficiency. In the case of a **cyclic** or even **symmetric** communication structure, the algorithm itself has enough inherent synchronization to make precise assumptions about worst case buffer sizes. Knowing this, we can always find implementations which rely on more efficient plain system calls than the ones usually provided, because free buffer space can be guaranteed – thus excluding the sender of having to wait.

We will use *Petri nets* to analyse *runs* of non–clock–controlled systems to find correct transformations using minimal synchronization mechanisms. Petri nets are adequate for the modelling and analysis of distributed systems. They can be nicely represented as bipartite graphs of passive (*conditions* which hold iff they are marked — drawn as circles), and active elements (*events* which may happen, thereby changing conditions they are related to — drawn as boxes), and by the notion of the runs of a net system they have a well-defined partial order semantics. Due to the lack of space we refer to [Rei85] for all technical details.

Convolution

In the case of the systolic convolution algorithm it can be shown that no additional synchronization mechanisms are necessary for a correct transformation because variables are accessed **symmetrically**: Each inner node has two communication partners that never change. In each superstep there is a send and receive operation — thus forming two communication cycles of minimal length (i.e., of symmetric shape). The convolution algorithm based on message passing can be modelled as a Petri net consisting of the behaviour of the k nodes and buffers — drawn as conditions — for communication as follows:

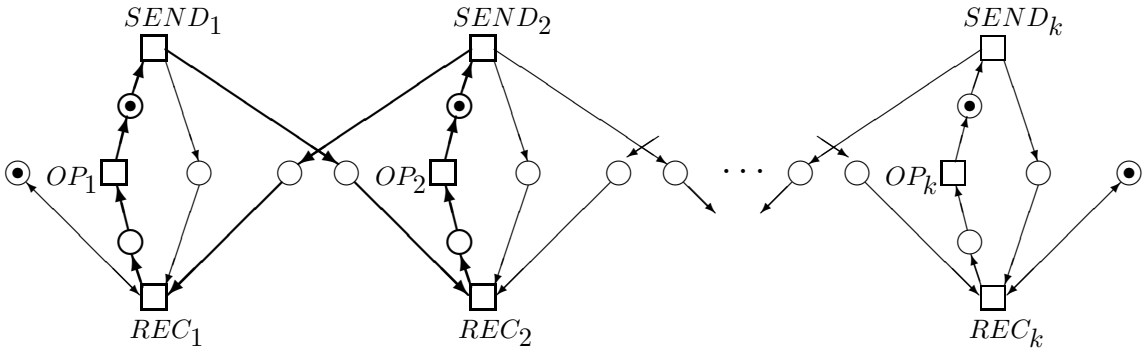


Figure 7: Petri Net Representation of the Partitioned Convolution Algorithm

In doing so, Petri net analysis methods can be applied. The thicker lines in Figure 7 form a *place invariant* over the buffers of two neighbouring nodes. The number of tokens on all these conditions remains constantly 2 under each reachable marking M of the system. Thus, in any system state there will never be more than two values in any buffer, such that a buffer capacity of 2 is sufficient. To guarantee correctness (each node operates always on the correct data) it is necessary that these buffers are having FIFO property. We will show the realization of FIFO-buffers of capacity 2 between two neighbouring nodes as a Petri net in Figure 8.

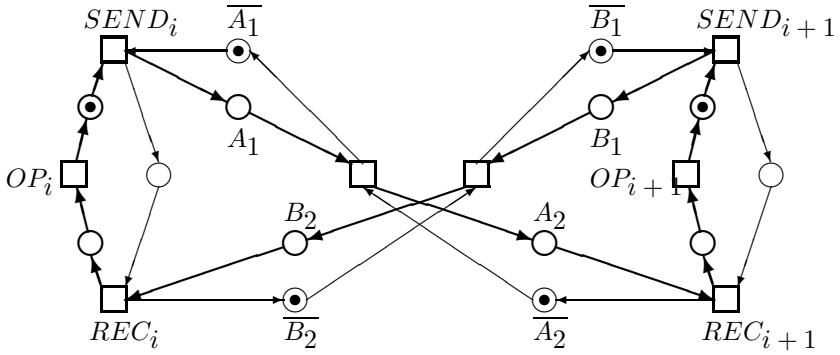


Figure 8: FIFO Buffers with Capacity 2 between two Neighbouring Nodes i and $i+1$

The basic idea is to model buffers consisting of two conditions. If all the buffer conditions are *complemented* we can now find place invariants for all conditions p in the channel, e.g. A_1 and its complement $\overline{A_1}$, such that $M(p) \leq 1$ which implies FIFO property.

To show the correctness of the system with buffers of capacity 2 we are going to have a look at the runs of the system, which are acyclic Petri nets: conditions and events are partially ordered. Here, we will determine on which value — received from neighbour $i+1$ — the node i is operating on in superstep t.

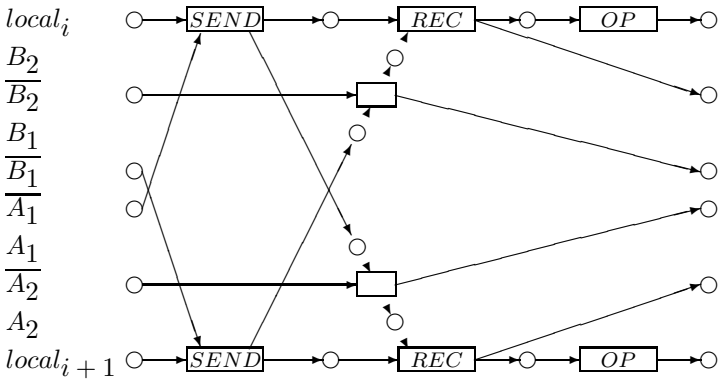


Figure 9: Run of the System Depicted in Figure 8

Figure 9 shows the unique minimal run of the system in which each node computes at least one value, i.e. an event "op" occurs at least once in both nodes. This run ends in the state it started with. The one and only maximal run of the system is never-ending. This means that no deadlock will occur. Within clockcycle t node i computes a new value including the value of the neighbouring node $i+1$ from clockcycle $t-1$. The following relations between events hold:

1. $\mathrm{OP}_{i+1}^{t-1} < \mathrm{SEND}_{i+1}^{t}$ due to the cyclical order of local events
2. $\mathrm{SEND}_{i+1}^{t} < \mathrm{REC}_{i}^{t}$ due to the causality in the run of the system
3. $\mathrm{REC}_{i}^{t} < \mathrm{OP}_{i}^{t}$ due to the order of local events within each superstep
4. OP_{i}^{t} and OP_{i+1}^{t} may occur concurrently, as they are not causally ordered within the run of the system

It follows from 1 – 3 that $\mathrm{OP}_{i+1}^{t-1} < \mathrm{OP}_{i}^{t}$ and from 4 that $\mathrm{OP}_{i+1}^{t} \not< \mathrm{OP}_{i}^{t}$, guaranteeing that node i computes its t^{th} value by using the $t-1^{st}$ value of neighbour $i+1$.

This desynchronizing step is optimal with respect to efficiency because we do not need any additional synchronizing mechanisms. The clock pulse is simulated solely by the values that have to be exchanged. This idea can be extended to systolic algorithms with **cyclic** access to variables: The

maximal amount of values in a buffer equals the length of the shortest circle the buffer is located at. Being aware of that, the implementation of big enough buffers allows the usage of cheaper plain system calls for message passing. In the case of an implementation with one incoming buffer per node, messages from two senders will be merged which requires indication of the values by their senders' names for correct identification.

Warshall

If we consider the Warshall algorithm, access to variables is weaker synchronized. Although the communication behaviour changes dynamically in time, it is nevertheless predictable, i.e. does not depend on the input data. So, messages can be sent by nodes without receiving any requests. But because a node can receive subsequent messages from several other nodes, FIFO channels are not sufficient to guarantee the correct order of incoming messages. Therefore, each submitted value needs indication of the step it is computed at. For this class of algorithm it is sufficient to enhance the basic model for a superstep of a node at the ADMM level in Figure 7 as follows:

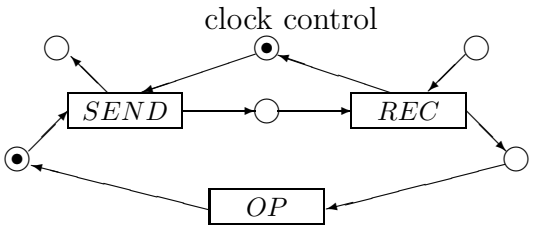


Figure 10: Petri Net that Models the Node Behaviour Including a Clock-Control

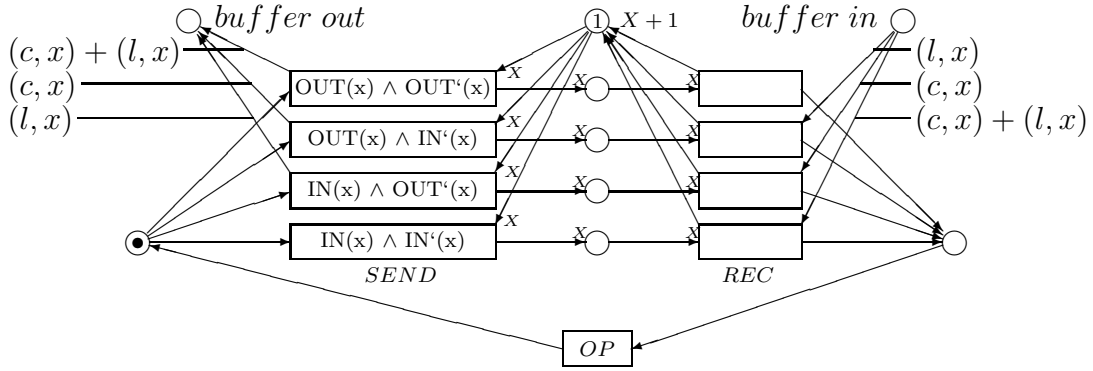
In the case of tile partitioning, each node has two communication partners: one within its line and one within its column. Both partners may change in any superstep. Furthermore each of these two communication events can be either a send or a receive event depending on the superstep number, but never both. Therefore, the clock control is realized as a counter (a natural number, initially 1) in Figure 11. Send and receive transitions are refined by predicates. Thus, a transition is only activated if the corresponding predicate is true, i.e. the counter lies inside a previously defined interval. Depending on x the *conflict* between the four communication patterns is solved deterministically, i.e. for any x exactly one of the four predicates is evaluated to true.

Each time a section l of a line (or a section c of column) is sent, it is indicated by the corresponding superstep number x. The message l (or c) is **broadcasted** in the column (or line, respectively) of the sender. Indication guarantees that the receiver will operate on these values in the correct order. After each communication event the local counter is increased by 1.

Analogously to Figure 9 the runs of the system can be analysed to show that the clock pulse is simulated correctly.

The capacity of the buffers has to be $n - \frac{n}{k}$ in the worst case, because during performance of the algorithm values are sent only from upper to lower tiles and from left to right, respectively. In particular there is no cycle synchronizing the algorithm by itself. Summarizing, again cheaper plain system calls for message passing can be used, but only if buffers with their size depending on n are implemented. In this case, indication of submitted values with the step number is sufficient for correct desynchronization.

Worst case buffer size needed for plain system calls can be decreased for the price of additional synchronization. After computation of some values, bottom and right nodes may send reply messages to upper and left nodes in the matrix, thereby constructing cycles of the desired length. So, buffer size and explicit synchronization messages can be regarded as two parameters which can be tuned reciprocally to yield an optimal result for the implementation.



x: representing the superstep counter
c: representing the values of the column
l: representing the values of the line

For a node Q_{rs} it holds that:

- $\mathrm{OUT}(x) = \text{true} \Leftrightarrow x \in \{(r-1) \cdot \frac{n}{k} + 1 \ldots r \cdot \frac{n}{k}\}$
- $\mathrm{OUT}'(x) = \text{true} \Leftrightarrow x \in \{(s-1) \cdot \frac{n}{k} + 1 \ldots s \cdot \frac{n}{k}\}$
- $\mathrm{IN}(x) = \text{true} \Leftrightarrow \mathrm{OUT}(x) = \text{false}$
- $\mathrm{IN}'(x) = \text{true} \Leftrightarrow \mathrm{OUT}'(x) = \text{false}$

Figure 11: Communication of the Warshall Algorithm Partitioned by the Tile

3 Discussion

Our approach simplifies the design of programs for highly parallel, asynchronous computers by providing a synchronous view of the system. This is achieved by separating the relevant problems in two steps — partitioning and desynchronizing. This procedure is only applicable to a certain class of algorithms. Hence, a classification of algorithms is indispensable. At present parallel complexity theory does not meet this requirement satisfactorily i.e. the following question is not answered adequately: Is there an efficient parallel solution for a given problem w.r.t. to the restrictions of real parallel computers? Within the scope of SFB 342 which funded this work the subproject A4 tries to develop a new complexity theory. This new theory should take into consideration the desynchronizing types (e.g. periodic, cyclic, symmetric, etc.) and especially characterize data–independent communications. Based on this model the construction of a stable notion of reducibility which preserves the characteristic features mentioned above is a primary aim. In a later stage, these reductions could eventually provide transformations on the synchronous level useful in the partitioning stage.

It is not yet well understood up to which extent partitioning can be automatically supported. Obvious standard problem structures could lead to a fully automated partitioning. If this is not the case there should be at least some assessment information provided by the system evaluating partitioning proposals of the user (e.g. table 1). These questions will determine our future work.

References

[Coo85] Stephen A. Cook. A taxonomy of problems with fast parallel algorithms. *Information and Control*, 64:2–22, 1985.

[Fea88] G. Fox et al. *Solving problems on concurrent processors, Vol. I: General Techniques and Regular Problems.* Prentice Hall, 1988.

[Rei85] Wolfgang Reisig. *Petri Nets*, volume 4 of *EATCS Monographs on Theoretical Computer Science*. Springer, 1985.

[Ros90] Peter Rossmanith. The owner concept for PRAMs. Technical Report TUM-I9028, SFB-Bericht Nr.342/15/90 A, TU München, 8 1990. To appear in STACS 91.

[Val90] Leslie G. Valiant. A bridging model for parallel computation. *Communications of the ACM*, 33:103–111, 8 1990.

[Vit86] Paul M.B. Vitányi. Nonsequential computation and laws of nature. In *VLSI Algorithms and Architectures*, pages 108–120, 7 1986.