Primitives of Sequential and Parallel Computation

Martti Forsell
Department of Computer Science
University of Joensuu, FINLAND

Ville Leppänen
Department of Computer Science
University of Turku, FINLAND

Martti Penttonen
Department of Computer Science and Applied Mathematics
University of Kuopio, FINLAND

December 3, 1998

Abstract

This paper addresses the question: 'Are there some primitive concepts that can be used to explain various aspects of computers and computations?'. We claim that many concepts and design principles can be understood in terms of *parallelizing/sequentializing*. In the case of physical resources, such as time or communication, these terms translate to *multiplexing/demultiplexing*, while in the context of algorithmic control, we rather speak of *splitting/joining*.

1 Introduction

Computations can be presented by directed flow graphs consisting of arcs and nodes, as in Figure 1. The nodes represent elementary operations whereas arcs mean data transmission. The transfer of control is a special case of data transmission, namely transmission of the message "you may now continue the computation". Solving a problem means designing a computation graph by using a set of elementary nodes (available in a computer or in an abstract model of computation).

Occasionally, an output of a node is needed in several places. It is natural to assume in the description of computations that the outputs of elementary computations are reusable. Often, it is also seen desirable to divide, or *split*, the future processing of the output of a node (or nodes) to several – more or less – independent parts. Respectively, by *join* operation we refer to a situation, where several nodes direct their arcs to a single node. The split operation is one side of the commonly used *divide-and-conquer* design method and the join operation is the other side. We discuss algorithmic design methods more in Section 2.

The *implementation* of such a description of the computation can be seen as a *wave* that sweeps over all the nodes and arcs (see Figure 1). The implementation is determined by specifying, which arcs the wave touches at each moment. If at one moment a wave is at an incoming arc of a node, at the next moment it can be at the same arc, or at the outgoing arc of the node. The only precondition of a legal implementation is that the wave can proceed over a node only after it has arrived at all incoming arcs of the node, which means that the operation at the node can be executed. There are a wide variety of ways how the wave can sweep over the nodes and arcs. The largest number of nodes between two successive waves expresses the degree of *parallelism*. If there is only one node

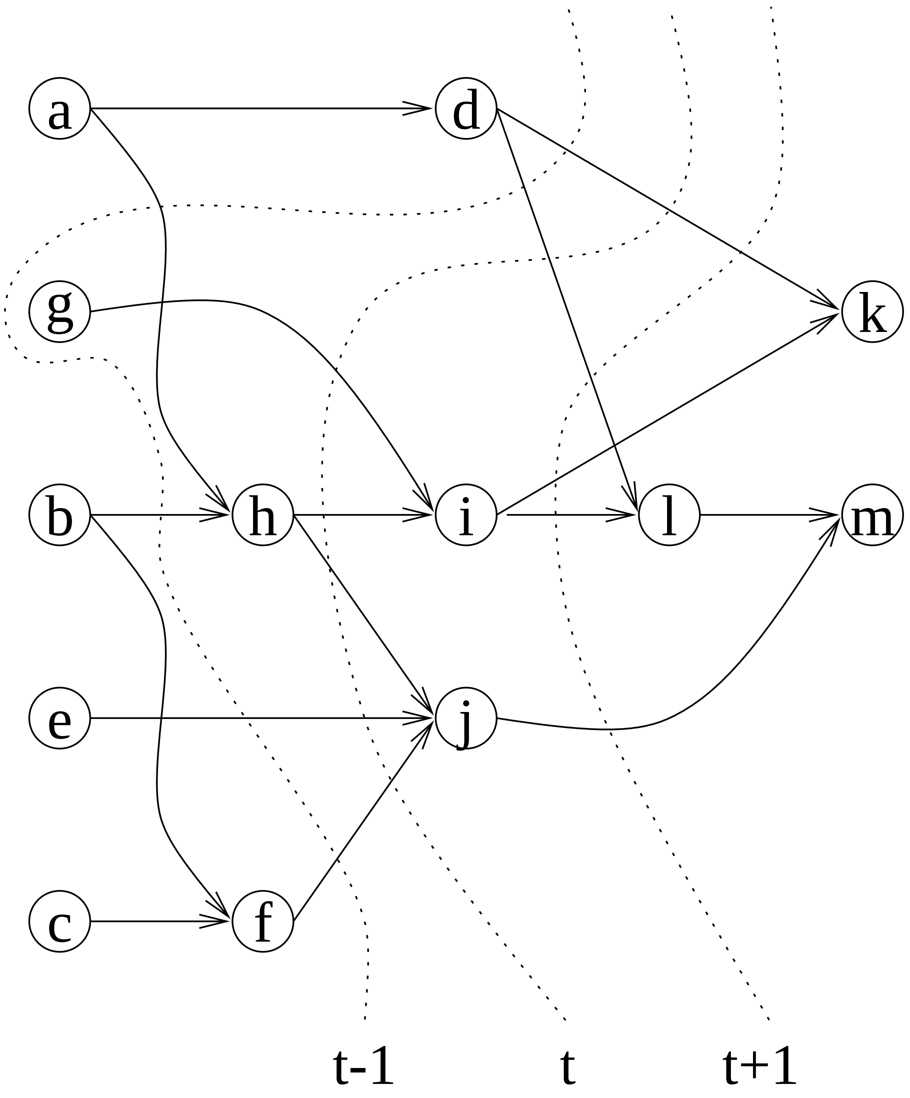


Figure 1: A flow graph and time waves.

between two successive waves, the implementation is *sequential*. Figures 2 and 3 illustrate two implementations of the flow graph of Figure 1.

Note that the number of nodes in the longest path of the flow graph determines the *minimum time* of any implementation. By allocating a different *processor* for each node of the flow graph, and assuming that each elementary operation can be executed in unit time, we can execute the implementation in minimum time. Such an implementation can be very wasteful, if processors are only waiting most of the time. The *minimum number of processors* required is the number of nodes that the wave passes in one step of computation. However, it would be advantageous that the nodes at both ends of an arc refer to the same processor, because otherwise the arc would mean interprocessor data transmission. Obviously, finding an efficient parallel implementation that uses the resources of underlying physical parallel machine in a balanced manner, is a challenging task. Later in Section 3, it will turn out that the methods to balance the use of resources are various forms of *multiplexing*.

Observe that the description of computation itself is parallel by nature. Often a se-

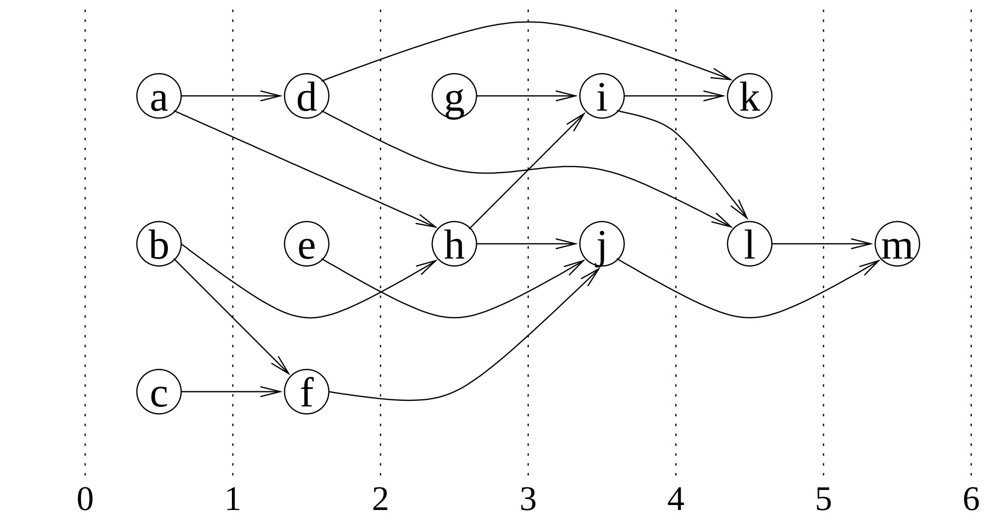


Figure 2: A parallel implementation of flow graph with 3 processors in time 6.

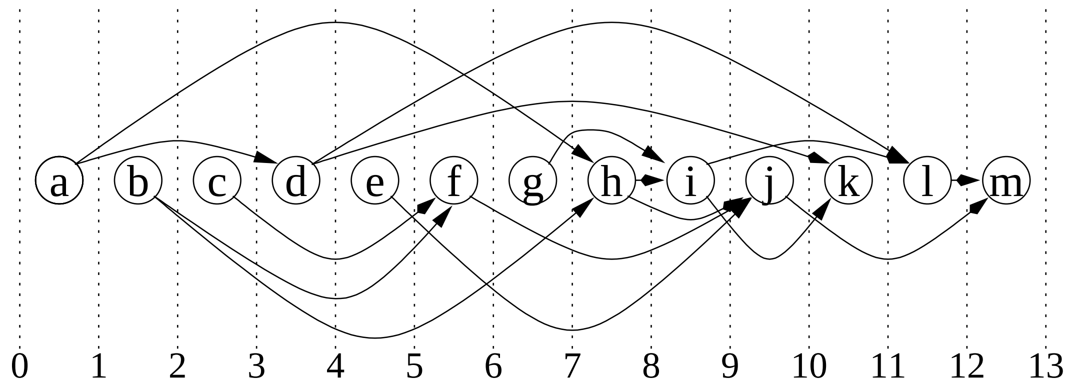


Figure 3: A sequential implementation of a flow graph in time 13.

quential description of computation can be very cumbersome and odd. At worst, choosing (somewhat arbitrarily) one of the possible sequential execution orders of a description can make it difficult to comprehend the meaning of the computation. The sequential programming can also be criticized of hiding parallelism found in the problem from the compiler since parallelism is a resource which can lead to efficiency, even in sequential execution [13, 19].

What objectives should be set for the design of algorithms and programs? Let us assume that producing correct solutions is not the problem, but rather we are concerned about the "efficiency" of solutions. An objective can be to find a graph that has minimal number of nodes, i.e. *minimal work*). Another objective could be to minimize the length of the longest path in the graph, i.e. the *fastest parallel* algorithm, or to minimize the length of the longest path while keeping the number of nodes close to the minimum, i.e. *work-optimal parallel* algorithm.

Unfortunately, the optimization of the programs cannot be based solely – perhaps, not even mostly – on the of the above objectives. Often the knowledge about the implementation stage has a great influence on the design – for example, the number of nodes of certain type should be minimized, or the arcs should be drawn so that the locality of communication can be maximized. Moreover, programming is a creative work whose success with respect to the goals depends on the skills of the programmer as well as on the properties of the programming language used to express solutions.

The rest of this paper follows the scheme of Table 1. First we study concepts and goals (in terms of our primitives) related to the design of descriptions of computations in Section 2. In Section 3, we do a similar study concerning the implementation of the descriptions. In essence, Section 3 discusses ways of balancing the use of available resources in the execution of algorithms. Section 4 continues in the same spirit, but now the subject of study are various low level components (which computers are made of). We present conclusions in Section 5.

	Algorithm	**Architecture**	**Hardware**
Problem	inventing the algorithm	mapping the computation and communication in computer	structure of physical components of computer
Goals	minimize time or work or time and work	efficient use of resources	efficient use of components
Means	split / join	multiplex / demultiplex	multiplex / demultiplex
Implementation	programming or computing models	architectural models	processors, memories, and communication devices

Table 1: Schematic view of the studied subjects.

2 Designing algorithms

In order to successfully solve algorithmic problems, it is essential that they can be solved at right abstraction level, close enough to the concepts of the problem. Unless the problem is more or less trivial, it is not advisable to write the algorithm directly for a certain parallel or sequential computer, or even for a certain programming language. Low level technical details can prevent from seeing the high level complexities. A further advantage of high level design is the portability of the algorithm to different systems. The algorithm should, however, be ultimately efficiently executable in the target architecture. In this section, we study what role parallelizing / sequentializing plays in algorithm design.

2.1 Algorithms as flow graphs

To keep the presentation simple, we choose flow graphs as the model for presenting algorithms. In the early phases of program design the nodes do not correspond to atomic execution elements directly executable in a computer. Instead, they are rather like "black boxes" or "molecules" or "modules" that implement a subalgorithm of suitable size. In top-down design methodology, the challenge of algorithm designer is to identify (or invent) useful modules. This kind of modularity makes designing easier and promotes the reusability of modules (eventually, program code). Also the arcs need not correspond to moving elementary data but, in a way, they can be seen to have some thickness.

What is essential, the description of computation can be seen to consist of splitting and joining of algorithmic modules as described in the previous section. Obtaining exact descriptions of computations requires implementation of the black boxes, i.e., refinement of them to submodules. This refinement process is studied via algorithmic techniques in Section 2.2.

During the refinement process one should not invent entirely fancy black boxes, since the goal of design process is to reduce the abstract description to use only certain kind of (simple) nodes and connections between nodes. Guidelines for the target constructions are given by a *programming model* (associated to some programming language) – or more generally by a *computational model*.

2.2 Design techniques

Traditionally algorithms have been designed to be sequential. However, we would like to see them written in parallel (as in [13, 19]). One reason for this is seen in Figures 1 and 3. In sequential representation extra effort is needed in fixing the order of execution, while parallel representation focuses at the logical structure of the algorithm only. Furthermore, a parallel algorithm can be automatically sequentialized. Hence, parallelism is an extra resource that allows for the use of multiple processing units, when such a computer is available. As an example, we consider a couple of design techniques from the point of view of split and join.

Perhaps the best-known algorithm design technique is the *divide–and–conquer* technique. The problem is split in two or more, more or less, independent subproblems, which are solved more or less independently, and the solutions of the subproblems are joined as the solution of the whole problem. Figure 4 illustrates, how the maximum of an array is computed by divide and conquer technique. Note that Figure 4 can be implemented parallelly or sequentially as in Figures 2 and 3.

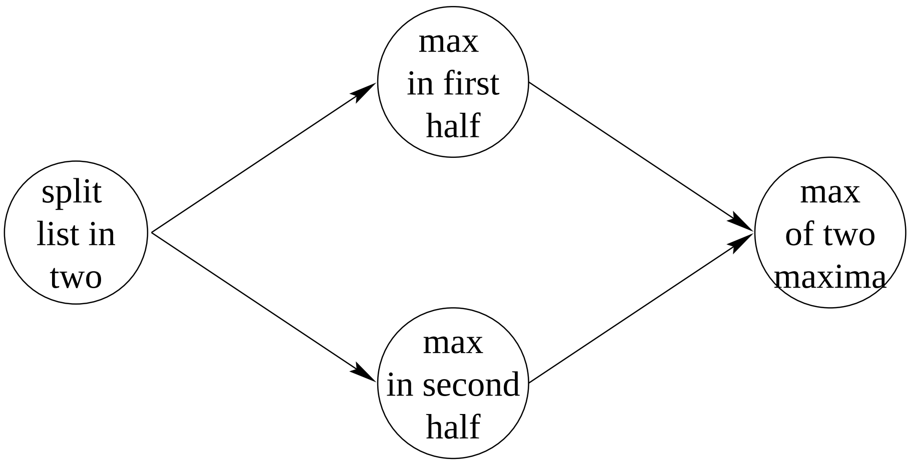
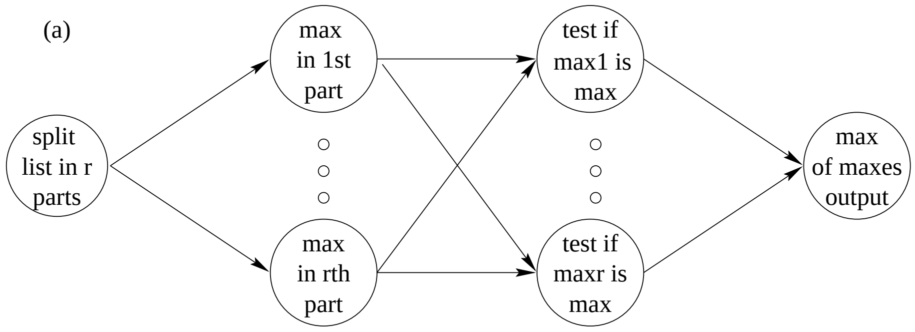


Figure 4: Computing maximum with binary divide–and–conquer.

For greater speedup, parallelization can be increased. In Figure 5, the n input keys are divided in $\sqrt{n}$ segments of $\sqrt{n}$ keys, their maxima are found recursively, and finally, the maximum of all maxima is determined by direct comparisons. The algorithm can be executed sequentially in time $O(n)$. If n (CRCW PRAM) processors are available, time $O(\log \log n)$ is achieved [15], which is seen by the recurrence $T(n) = T(\sqrt{n}) + O(1)$.

(a)



(b)

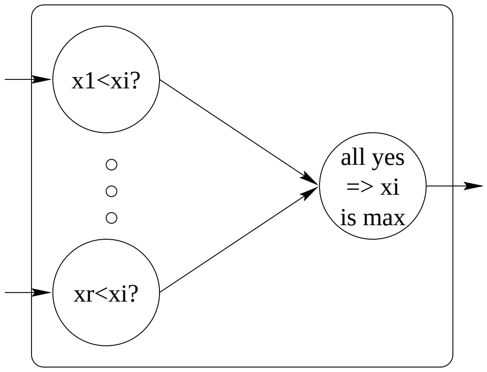


Figure 5: (a) Maximum with $\sqrt{n} = r$– divide–and–conquer.
(b) Subalgorithm for maximum testing.

A recursive algorithm, such as the recursive maximum algorithm in Figure 4, often leads to recursive calls of fine granularity. It is reasonable to cut the recursion when the problem size decreases below a threshold, and use another algorithm for small problems. In Figure 6 on the top the maximum is computed by a recursive algorithm (in parallel), while the boxes are small problems solved by another algorithm (sequentially). The *blocking* can be used top-down or bottom-up, applying blocks at the end or at the beginning. A reason for using such *hybrid* algorithms is to gain work-optimality. By simple divide-and-conquer the maximum of n numbers can be computed with $n/2$ processors in log n levels of recursion. If blocks of size log n are calculated sequentially in log n time with one processor, the $n/log\ n$ subresults can be calculated in log n time with $n/log\ n$ processors, leading to a work-optimal algorithm running in log n time. If splitting and joining in the target architecture are costly, it may be worthwhile to start the sequential phase a little earlier than by pure algorithmic grounds.

In the above example, the degree of divide-and-conquer is two whereas the maximum finding algorithm uses degree $\sqrt{n}$. In a way, the blocking technique can be seen to utilize the divide-and-conquer technique with degree 1 (sequentialization). The divide-and-conquer technique can be applied with wide variety degrees for dividing. The problem size n can act as the degree (fully parallel problems), and even using e.g. the degree n^2 has applications (sorting in constant time on SUM CRCW PRAM). Moreover, some algorithms may divide in numerous stages, where the algorithmic principle may or may not change from stage to stage, and the degree of parallelism may increase or decrease from stage to stage. In general, the divide-and-conquer provides a very flexible method to design the calculation while controlling the width and depth of calculations and even grouping subsolutions in a desired way.

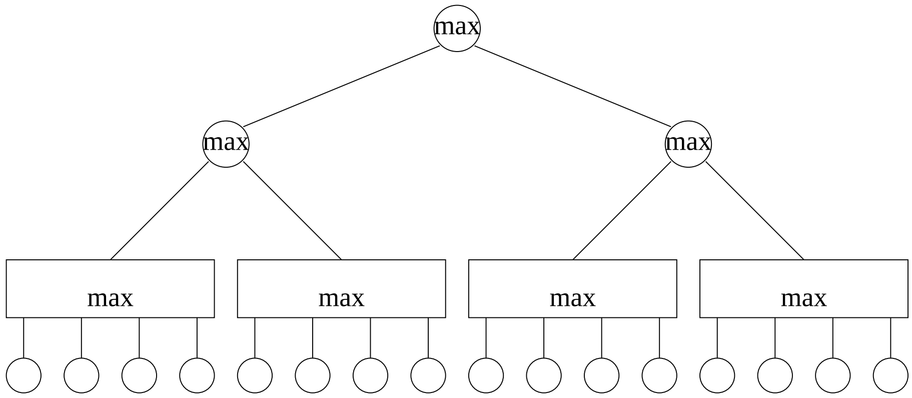


Figure 6: Hybrid algorithm: parallel recursion and sequential blocks.

What do algorithmic techniques mean in terms of exact descriptions of computations? Besides the refinement aspect, the idea of algorithmic techniques is to define a structure (a pattern) of nodes and arcs. Often such a structure is large and homogeneous, though the description of the technique is simple and compact. Since our basic primitives of descriptions of computations are split and join, it should be no surprise that basic techniques utilize splitting and joining in somehow repeating manner. It is tempting to claim that most of the algorithmic techniques are forms of the classical divide-and-conquer technique, where the variation comes from the degree and stages of splitting and joining.

3 Execution of algorithms in computers

When an algorithm is executed, it should run efficiently in the (parallel) computer at hand. It means that all parts of the computer are used most of the time and they are used meaningfully. In this section we focus on mapping the computation into the computer in such a way, that the computer is used efficiently.

We see algorithm design as defining the output data in terms of input data and the operations of the computational model. We focus at the dependency relations of data and operations and implicitly assume that the data is immediately available everywhere and in all times as soon as it has been defined as the result of an operation.

Algorithm, when expressed in form of a flow graph, does not explicitly determine in which order the operations are executed. The programmer, who knows how many processors his computer has, may wish to express that some parts of the program should be executed in parallel. We assume that the programming language has some facilities to express the degree of parallelism.

In our opinion, the algorithm designer and the programmer should concentrate at the logical dependencies of the problem and say as little as possible about the use of computer resources. There are two reasons for this. First, algorithm design and programming are difficult enough tasks and the human genius should be saved for the essential. Second, as soon as the theory of parallel computation is developed enough, the machine can optimize the use of resources much more efficiently than the programmer. Unfortunately, the parallel computers and their compilers are not yet at a satisfactory level and programmers have to play the role of the compiler.

3.1 Conditions of the efficient use of resources

The condition of efficient implementation of an algorithm is that right data is at right place at right moment. If data arrives too late, processing power is wasted for waiting. If data comes too early, it must be stored somewhere, which requires extra effort and storage. If data is at wrong place, moving it takes time and requires communication capacity. Therefore, mapping the computation in the computer is of primary importance.

For the execution of the algorithm, each processing node of the algorithm must be allocated a processor or a number of processors for some period of time. This node mapping implies mapping of arcs to interprocessor communication. Also, it may be necessary to allocate some storage for arcs, because the processor mapped for the target node may not be available at the moment, when data corresponding to the arc arrives. Therefore, the mapping must meet the following constraints:

- Processors must be multiplexed in space (i.e. have a sufficient number of processors) or in time (i.e. share the time of a processor between many processes or threads) so that each processing node of the flow graph is allocated physical computing resource.
- Communication links corresponding to the arcs in the flow graph must be multiplexed in space (to increase "bandwidth") for immediate availability at the target node, or multiplexed in time (i.e. serialize the communication).
- When multiplexing of processors in time is used (i.e. always), arcs of the flow graph are implemented as a communication link from source node to memory, and another communication link from memory to target node. Memories can be considered as processors that are only capable of receiving, keeping, and sending data.

When implementing a description of computation (defined implicitly by a program and its data), we are forced to operate under the physical limitations (or properties) of the underlying machine. Since the descriptions of computations do not need to obey these restrictions, we must somehow *multiplex* with respect to *time* the usage of limited machine resources. On the other hand, the parallelism in the description of computations may suggest that it would be wise to *multiplex* some components in *space* – e.g., to have more processors or to widen the communication links or to increase the length of (arithmetic) pipelines. This implementation problem can also be seen as a problem of designing machines with efficient multiplexing properties.

3.2 Controlling the use of resources

The big problem is, how should the flow graph of the algorithm be mapped on the computer so that the computation is fast and efficient. It depends on the mapping of nodes, how close to the optimal work, time, or work $\times$ time one gets. The degree of parallelism can be controlled by the

for `processor_set` **pardo** `subcomputation`

statement.

Mapping the processing nodes implies communication of data. It would not be a problem, if there were a *shared memory*, where all data is uniformly accessible. At current level of technology, large and fast shared memories do not physically exist. Therefore, one has to assume that the physical memory is *distributed*. It is obvious that interprocessor communication is more costly than retrieving data from local memory, which is the case when two successive nodes of the algorithm are mapped to the same processor. Unfortunately, there is no general method to maximize the locality in a computation, and finding a good mapping case by case is painful.

A general solution for the mapping problem, if no better strategy is known, is to map the processing nodes to the processors and data to the memory at random, and solve the general communication problem (i.e. any processor to any processor) by an efficient routing algorithm. Clearly, randomized memory mapping is not local, and therefore it implies the memory *latency* problem, i.e. getting data takes time. Randomized memory mapping alone would be inefficient due to latency, but with so–called *slackness* [17], execution of algorithms can still be made work–optimal. When slackness is used, each physical processor works for a number of independent nodes of the algorithm. Even if the communication cannot be made fast enough by multiplexing the communication link in space, execution can be made work-optimal by multiplexing the processor in time. This technique is used to simulate the abstract PRAM (Parallel Random Access Machine) on distributed memory machines [5, 8, 10, 11, 14, 17].

3.3 Directions in parallel computing

In each approach to parallelism something is assumed about the multiplexing level of the components of the underlying machine. Similarly, something is assumed about the multiplexing properties of the programs to be executed. Thus, bottlenecks can be identified and solutions to the observed problems can be suggested. In the following, we make a quick tour over some approaches to parallelism and attempt to understand them in terms of multiplexing / demultiplexing.

3.3.1 Shared memory abstraction on distributed memory

Consider implementing a shared memory abstraction on a distributed memory machine. Hashing means mapping the shared memory locations to the (sequential) memory modules of the distributed memory machine. The objective of hashing is to distribute the memory references made by the P processing units in a given time period as evenly as possible. At best, the time multiplexing level of any memory module is required to be only $\approx 1/P$'th of the time multiplexing level of the shared memory. Clearly, the *hashing* technique is an algorithmic attempt to trade time multiplexing to space multiplexing.

As was explained, work-optimal implementation of the PRAM model is based on parallel slackness, hashing and a high-bandwidth routing machinery. Slackness is property of parallel programs at the implementation stage. If p processors are used to implement a description of computation having width $s \times p$ (at some point), the program is said to have slackness s (at that point of the program). The slackness represents the fraction of width that must and can be easily multiplexed in time. Another view is that s virtual PRAM processors are demultiplexed to a single physical processor.

The idea in hiding latency ℓ (measured in unit steps) with slackness s is roughly that the implementation of shared memory accesses is pipelined (multiplexed in time) and routing machinery has bandwidth and capacity to move $s \times p$ requests to their targets and back in time $O(s)$. This requires ability to move $\Omega(\ell \times p)$ packets at each step. This routing capacity is achieved by multiplexing routing machinery nodes and connections in space while preserving latency ℓ.

Rather than attempting to adapt to the properties of some specific parallel machine, the PRAM approach sets various multiplexing requirements for the PRAM programs, processors, memory modules, connections, and routing machinery. Requirements concerning routing machinery and memory modules (normalized with respect to the performance of current processors) are troublesome at current level of technology but not hopeless: experimental implementation exists [1, 4].

3.3.2 BSP and LogP approaches

The BSP [2, 12, 16] and LogP [3] (as well as many similar) approaches identify the time multiplexing properties of routing machinery and memory modules with respect to that of the processors. For example, the BSP identifies latency ℓ and parameters h and h_0 – the routing machinery can realize any h-relation (where $h \geq h_0$) in time gh with high probability. An *h-relation* is a routing problem, where each processor is the sender and receiver of (at most) h messages.

The idea is that the identified properties are exposed to the programmer as parameters, which enable to design computations that can be implemented efficiently using the available time multiplexing properties. Thus, the programmer's burden is increased by forwarding him at least partially the responsibility of finding a successful solution to the mapping problem!

3.3.3 Data-parallelism

The basic idea in *data-parallel computing* is to express parallelism through parallel variables (vectors, matrices) instead of parallel control streams. Homogeneous structures are run through a homogeneous processing set (processors, vector processors, arithmetic pipelines). Thus, high space multiplexing of several computing units is often assumed.

Data-parallel programming is, perhaps, best seen as an automatically parallelizable extension of sequential programming. Typical data-parallel languages (Fortran 90, HPF, NESL) do not even have the concept of processors or processes. The data-parallel approach leans on clever compiling techniques and data-parallel language constructs, which enable implementation to advance existing space multiplexing properties efficiently.

Often data-parallelism is applied in context, where processors are fast and have highly sophisticated arithmetic pipelines but whose inter-processor communication mechanism is rather slow (low time multiplexing properties). Then the main problem of a compiler is to locate data on the processors so that the locality of references is thus maximized.

3.3.4 Dataflow computing

In *dataflow computing* [9, 18], the idea is to see the whole computation as a directed graph, where nodes represent small processes (of approximately machine instruction size) and arcs tell the input/output dependences of the processes. The equivalence with our description of computations is evident. There is no implicit synchronization mechanism controlling the execution of processes, but each of them is executed as soon as possible (that is, when all the inputs are available). There is no program counter guiding the execution of processes. In principle, all the nodes are participating the computation from the beginning of the computation. A node gathers information (packets) from incoming arcs and executes the instruction(s) once all the required inputs are available – then it passes the results via outputs arcs to the receivers.

In practice, sets of nodes are collected to sequential processes and the problem in making processes out of graph nodes is how to maximize locality of references, while creating a suitable amount of parallelly executable processes. Dataflow computers (Monsoon, EM-X, *T) are also often multithreaded computers unless their lean on the ability to eliminate cache misses by having large cache memories. Multithreading means that each processor is able to execute several independent threads simultaneously. The dataflow computation leans on clever compiling techniques and language constructions (that support compiling). As in the PRAM case, the multithreading is used to hide the latency of data transfers (see Figure 9) and it is compiler's (rather than programmer's) duty to advance the multithreading properties of the machine in a balanced manner.

4 Parallelism on hardware

The computer hardware consists of processors, memories, and communication devices. In the flow graph representation of Fig. 1, the nodes will be ultimately implemented by processors (and memories), while arcs correspond to communication. A processor is a device that transforms an input to an output. A memory can be seen as a processor whose output is identical to the input, still the characterizing feature of the memory is that it multiplexes a datum in time so that it is available when it is needed. Like processor time, the memory time should be seen as a cost, counted in *bit seconds*, for example. Communication is multiplexing of the datum in space, so that it is available to (an)other processor(s). The fact that communication takes time, sets some constraints to the mapping of the algorithm to the computer.

Due to physical speed, size, and connection limitations of semiconductor (or optical) components, the number of components taking part in processing of the same data should be kept small, otherwise significant slowdown occurs. This is why a large number of components usually is divided into mostly independent *blocks* of high locality, e.g., separate processors. Parallelism inside such processing blocks is called *instruction level parallelism* or *chip level parallelism*. Parallelism between blocks is called *thread level parallelism* or *machine level parallelism*. Thus, due to certain physical constraints it is wise to apply space multiplexing at several levels.

4.1 Processor

There are two basic architectural alternatives to speed up execution of instructions in a processor using parallelism at instruction level: pipelined execution and superscalar execution. *Pipelined execution* divides the execution of an instruction into s parts so that different parts of subsequent instructions can be executed in different (sub)units simultaneously corresponding to multiplexing in space and time. Figure 7 illustrates, how a sequence of 13 instructions is executed in a processor using 3-level pipeline in 20 (shortened) time units. For simplicity only the execution parts of instruction execution are shown. Observe that pipelining changes the size of the blocking and therefore shortens the time between consecutive waves. In theory, the pipelined execution gives an s-fold performance increase with small hardware costs, if the instructions are independent of each others. Unfortunately, this in normally not the case (see Figure 7), and some loss of performance occurs. An extreme form of pipelined execution is used in *vector processors*, where so called vector instructions generate a set of homogeneous and independent (sub)operations which are then executed in a very deep pipeline.

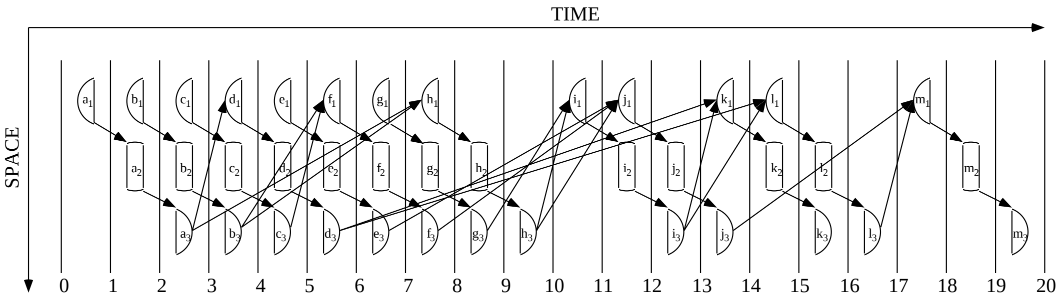


Figure 7: Execution of 13 instructions in a processor using 3-level pipeline.

In *superscalar execution*, at most f instructions are executed in f functional units simultaneously corresponding to multiplexing in space. There are two types of superscalar execution — dynamic and static. Dynamic superscalar execution is used in *superscalar processors*, where delays caused by data dependencies are eliminated by buffering instructions in processors and dynamically selecting what instructions are executed in parallel. Buffering increases the length of execution pipeline remarkably, which causes a lot of control dependency delays. This is why complex delay elimination techniques, like branch prediction, are used extensively in superscalar processors. Static superscalar execution is used in *VLIW* (Very Long Instruction Word) *processors*, where instructions to be executed simultaneously are selected in compile time by an advanced compiler. Figure 8 illustrates, how a sequence of 13 instructions is executed in a VLIW processor using three functional units in 6 time units. VLIW processors offer potentially better performance than superscalar processors, because compile time selection of parallel instructions is more efficient than dynamic runtime selection. Moreover, certain VLIW constructions succeed in avoiding unnecessary dependency delays typical to superscalar processors [6]. In theory, the superscalar execution provides an f-fold performance increase. In reality, however, dependencies prevent full utilization of functional units and some loss of performance occurs (see Figure 8).

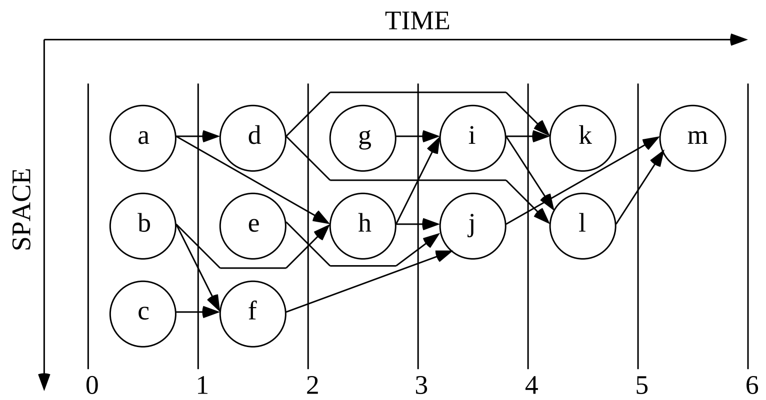


Figure 8: Execution of 13 instructions in a VLIW processor using 3 functional units.

If slackness is used to hide latency of communication, processors must be able to handle multiple threads efficiently. This kind of thread level parallelism can be realized by a *multithreaded processor*, which shares its time to multiplex threads in turn (see Figure 9). Fast context (i.e. thread) switching is implemented by multiplexing registers in space and using special architectural techniques. Thread level parallel constructions may also exploit instruction level parallelism, i.e., multiplex the execution of instructions in space and time, as described in [7].

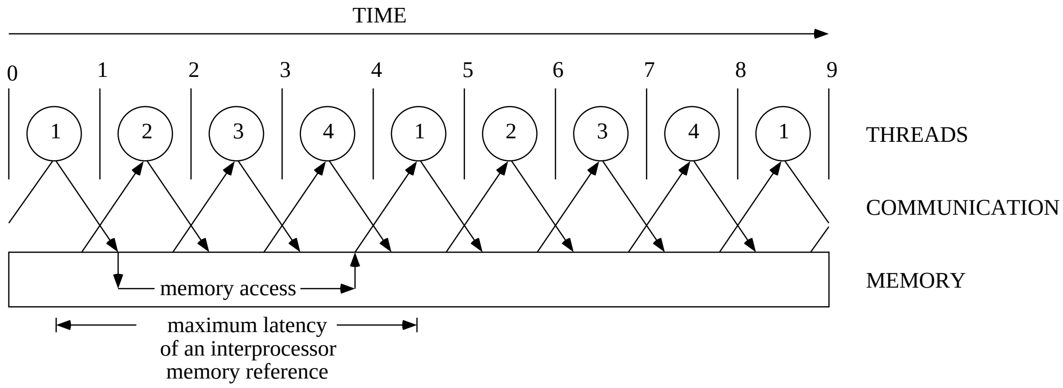


Figure 9: Execution of four threads (1,2,3 and 4) in a multithreaded processor.

4.2 Memory

Memories are sequential devices that enable time multiplexing of communication. Assume that a memory can satisfy x memory requests in some time period called reference time. Consider situations, where a design (of a computation or a processor) suggests that y memory requests should be fulfilled within the reference time, and $y > x$. A technique called *banking* means multiplexing the memory in space to z *memory banks*, where each bank can satisfy x requests (simultaneously) in the reference time. If $y < z \cdot x$, the banking can provide a solution to the memory congestion problem. Unfortunately, the banking requires separate memory system communication lines for each bank, which may effectively multiply the complexity of the system by z. *Memory interleaving* is an extension of the banking, where only one communication channel is needed due to multiplexing of memory requests and replies in time. Thus, both the memory banking and interleaving provide a solution to trade time multiplexing for space multiplexing. The interleaving uses pipelining (of requests and replies), i.e., it is based on both time and space multiplexing.

In a parallel computer, a memory module can be accessed by several processors. Considering a memory module, the role of routing machinery between processors and the memory module is to demultiplex the requests, if a memory module is a sequential device. In a *multiport memory*, the demultiplexing role of routing machinery is pushed into the memory module (in extreme case: in front of each memory cell).

Also *hashing* is used for trading time multiplexing for space multiplexing. Hashing is used, for example, when shared memory is simulated by distributed memory. By mapping the address space of a (shared) memory with a good hash function to distributed memory modules, it is not probable that a lot of memory accesses are targeted to the same memory module and thus delayed, but they are dispersed over the space.

4.3 Communication

Communication has a very important role in parallel computation, because it makes possible the multiplexing and demultiplexing in space, i.e. propagating subcomputations to other units and collecting the results.

By common terminology, communication can be *parallel* or *serial*, where parallel means sending several bit streams in parallel along several physical lines, while serial means timesharing a single line. In our terminology, parallel communication uses multiplexing in space (and time), while serial communication uses multiplexing in time.

There are more sophisticated uses of multiplexing/demultiplexing in communication, though. If there is a fast enough bus connecting memory or processing units, it can be timeshared so that the effect of a complete network is achieved. On the other hand, routing algorithms can be speeded up by increasing the bandwidth. For example, if processors in a complete network send randomly addressed packets to each other under 1-collision assumption (where all colliding packets fail), increasing the number of links connecting a pair of processors from 1 to m increases success probability from $1/e$ to $1/e^{1/m}$. In *optical* communication, there is a new means of multiplexing. In addition to *space division multiplexing* (SDM) and *time division multiplexing* (TDM), there is also *wavelength division multiplexing* (WDM) offering more bandwidth.

In general, a routing machinery is formed by multiplexing communication lines and routing nodes in space. Within the nodes, buffers play a similar role as pipelining in processors. The role of routing machinery nodes is to demultiplex incoming data streams to outgoing data streams (which means space multiplexing). Since outgoing streams are typically capacity constrained, the demultiplexing of parts of incoming streams to a single outgoing stream is implemented by applying time multiplexing. The whole routing machinery represents space multiplexed routing capacity (the amount of packets that can be moved in unit time), but the machinery is also often made flexible in the sense that it can dynamically time multiplex the movement of packets (i.e. delay them when necessary).

Another technique to compensate slow communication, *slackness*, was described at the end of Section 3.3.1. From the routing machinery point of view, the idea is to have routing capacity at least $\Omega(s)$ per each source and average path length, if s is the slackness factor.

5 Conclusions

The purpose of our study has been to provide an affirmative answer to the question: 'Can one identify some basic elements of computations that can be used to explain various concepts related to design of computers as well as algorithms?'. We made a tour to computer science concepts appearing at various levels of abstraction from algorithm design principles to hardware design. Surprisingly, it seems to us that these can be understood in terms of splitting / joining and multiplexing / demultiplexing. The same ideas, concepts, and design methods are used over and over again in various settings, but under a different name. Especially, we find fascinating the application of the various forms of multiplexing in the implementation stage. Besides providing meta-level understanding of the nature of sequential and parallel computation, our study can also be used to reveal new solutions to a given problem setting by comparing it to a similar but well-studied setting.

Efficient use of computational resources is one of the main goals in computer science. Efficient use means using (a) all available resources (b) in the most efficient way. It should be noted that point (b) depends on point (a) as all computational problems are not equally parallelizable.

The basic idea of parallel computation is to speedup the computation by multiplexing it in space. However, when mapping such parallel descriptions directly to machine, this may turn out to be too expensive, take too much time, or be otherwise unfeasible. Then it may be useful to multiplex some part of computations in time, since time seems to be much more flexible resource than space when computations are implemented. Multiplexing / demultiplexing in time / space together provide tools to balance computation to achieve maximum efficiency.

Forming efficient computations has three sides: algorithmic description; machine, where the algorithm is executed; and mapping of the algorithmic description into the machine. Descriptions are two-dimensional: time multiplexed by nature, and suitably parallel (space multiplexed). However, computers provide more dimensions for multiplexing and demultiplexing parallelism within processors (pipelines, several instruction streams, ...), communication mechanism (SDM, TDM, WDM, ...), and memory modules (interleaving, ...). The problem in forming an efficient mapping is, how to utilize the other dimensions of machine, while saving the time dimension as much as possible.

References

[1] F. Abolhassan, R. Drefenstedt, J. Keller, W.J. Paul, and D. Scheerer. On the Physical Design of PRAMs. *The Computer Journal*, 36(8):756 – 762, 1993.

[2] T. Cheatham, A. Fahmy, D.C. Stefanescu, and L.G. Valiant. Bulk Synchronous Parallel Computing – A Paradigm for Transportable Software. In *Proceedings of the 28th Hawaii International Conference on System Sciences (HICSS)*, pages 268 – 275. IEEE Press, January 1995.

[3] D.E. Culler, R. Karp, D. Patterson, A. Sahay, K.E. Schauser, E. Santos, R. Subramonian, and T. von Eicken. LogP: Towards a Realistic Model of Parallel Computation. In *Proc. of Fourth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 1 – 12, San Diego, CA, May 1993.

[4] A. Formella, J. Keller, and T. Walle. HPP: A High Performance PRAM. In *EuroPar'96, Lecture Notes in Computer Science 1124*, pages 425 – 434, 1996.

[5] M. Forsell, V. Leppänen, and M. Penttonen. Efficient Two-Level Mesh based Simulation of PRAMs. In *Proceedings of International Symposium on Parallel Architectures, Algorithms and Networks, ISPAN'96*, pages 29 – 35. IEEE Computer Society, 1996.

[6] M.J. Forsell. Minimal Pipeline Architecture-an Alternative to Superscalar Architecture. *Microprocessors and Microsystems*, 20(5):277 – 284, 1996.

[7] M.J. Forsell. MTAC - A Multithreaded VLIW Architecture for PRAM Simulation. *Journal for Universal Computer Science*, 5(3):100 – 114, 1997.

[8] A. Kautonen, V. Leppänen, and M. Penttonen. Constant Thinning Protocol for Routing h-Relations in Complete Network. In *Euro-Par'98, LNCS 1470*, pages 993 – 998, 1998.

[9] B. Lee and A.R. Hurson. Dataflow Architectures and Multithreading. *IEEE Computer*, 27(8):27 – 39, August 1994.

[10] V. Leppänen. *Studies on the Realization of PRAM.* PhD thesis, TUCS, Department of Computer Science, University of Turku, November 1996. TUCS Dissertation, No 3.

[11] V. Leppänen and M. Penttonen. Work-Optimal Simulation of PRAM Models on Meshes. *Nordic Journal on Computing*, 2(1):51 – 69, 1995.

[12] W.F. McColl. Scalable Parallel Computing: A Grand Unified Theory and its Practical Development. In B. Pehrson and I. Simon, editors, *Proceedings, 13th IFIP World Computer Congress. Volume 1*, pages 539 – 546. Elsevier, 1994.

[13] R. Orni and U. Vishkin. Two Computer Systems Paradoxes: Serialize-to-Parallelize, and Queuing Concurrent-Writes. Technical Report CS-TR-3586, University of Maryland Institute for Advanced Computer Studies Dept. of Computer Science, Univ. of Maryland, September 1995.

[14] A.G. Ranade. How to Emulate Shared Memory. *Journal of Computer and System Sciences*, 42(3):307–326, 1991.

[15] Y. Shiloach and U. Vishkin. Finding the Maximum, Merging and Sorting in a Parallel Computation Model. *Journal of Algorithms*, 2(1):88–102, 1981.

[16] L.G. Valiant. A Bridging Model for Parallel Computation. *Communications of the ACM*, 33(8):103–111, 1990.

[17] L.G. Valiant. General Purpose Parallel Architectures. In *Algorithms and Complexity, Handbook of Theoretical Computer Science*, volume A, pages 943–971, 1990.

[18] A. Veen. Dataflow Machine Architecture. *ACM Computing Surveys*, 18(4):365 – 396, December 1986.

[19] U. Vishkin. Can Parallel Algorithms Enhance Serial Implementation? *Communications of the ACM*, 39(9):88 – 91, September 1996.