Carlstedt Elektronik AB
computer research and development

Resource theory for multiprocessors

by Gunnar Carlstedt

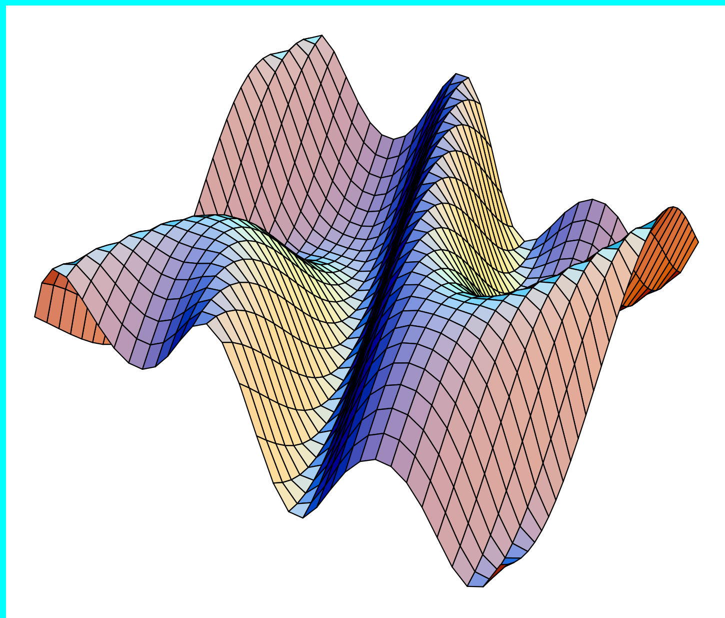
Abstract
A theory on the resource utilization in microprocessors is presented.

The theory shows that automatic placement, duplication and other control can be used to implement microprocessors.

Usually implementing any program on a microprocessor without knowledge of its architecture is possible.

Latency time and memory volume can be computed for any program using a number of characteristic figures known for the program.

The theory can be used to optimize the design of microprocessors. The processors can be of fine or raw type.



se.hylab.1:se.hylab.2:Bibliotek:dokument:91-89546 Vik:91-89546-07-5 Resource theory:91-89546-07-5 Resource theory
pages=31, changed=2001-10-26 18:06:30

1 INTRODUCTION

Traditional synthesis has developed from single processors to multiprocessors. The characteristics have been inherited. Based on the Turing machine the mechanism behind execution has been very simple. The mechanism of the multiprocessors has been based on different architectures as shared memory or message transferring. No real theories have been developed on the resource utilization, because of their large complexity. Instead a lot of experiments have been done, among which are [Hey 90].

There has been work considering special models for parallel execution. [Eager 89] has formulated a theory for communicating tasks where the size of the graph is constant. It has been shown that task allocation is crucial and could not be fixed [Burton 94].

This report is intended to present

- that under certain circumstances executing any program on general, arbitrarily small/large microprocessor systems are possible. The processors are not required to be tuned for the programs or vice versa.
- a model where latency time and storage volume used can be computed for a general executing program. The model uses three sets of parameters describing the program, the multiprocessor architecture and their speed.
- that microprocessors can be optimized using the presented model. It can be shown that in certain cases the granularity of processors can be very important for performance.

In this paper a general theory on execution is presented. This theory is based on the three main execution characteristics: storage, transport and execution. It presumes that such resources are used and is therefore different from the traditional sequential mechanism.

The theory is general and can be applied to most of the existing semantics such as the imperative or declarative and also indeterministic semantics. This fact makes the theory applicable to conventional multiprocessors and new ones.

In this theory there exist several allocating (or mapping) functions. These are specific to different architectures. In this paper very general functions are used. They are based on random allocation. More specific algorithms can be introduced enabling the random algorithms to be used as references.

The theory of imperative semantics is straight, but can be hard to implement in detail. To solve this, the implementation can be based on coarse grain. Then granularity grows, approximately corresponding to objects of object-oriented semantics. For declarative semantics the smallest details can be even analysed.

Several mathematical expressions are set up from the theory, modelling the execution. The model presents execution frequency (corresponds to an execution rate), the load on processors and links, and storage volumes. When the implementation of the units is known, architectures can be optimized using these parameters. Such optimization will be presented in a subsequent report.

The theory is the base for the computer concept rp8601.

2 EXECUTION GRAPHS

Data structures describing program and thread of executions are required to analyse execution. Therefore, execution graphs are introduced.

All programs (=applications) can be described by dependencies. The programs consist of nodes. Each node can be a simple expression or a more complicated calculation. Every such node is depending on other nodes or the node in question is a leaf. The classical example is $a+b$, where the + sign depends on the values a and b.

The dependencies can be implicit or are explicitly defined in programs. In certain cases only temporal dependencies are defined. In classical imperative programs, as C, Pascal and Fortran, a sequence is described. The programs consist of clauses, all representing nodes. These define dependencies: and expression depends on earlier defined expressions. The dependencies can also be visible directly in the syntax, which is the case for the declarative programs.

In this report graphs are used as a definition of dependencies. Three different graphs will be described. These are the program, the execution graph and the universal execution graph.

2.1 Program

A program is a graph. The nodes are expressions and clauses of a traditional program. The edges bind referred expressions. For the theory, the particular graph is of no interest but how it is used. Programs are characterized of their contents of alternatives[1] and applications[2].

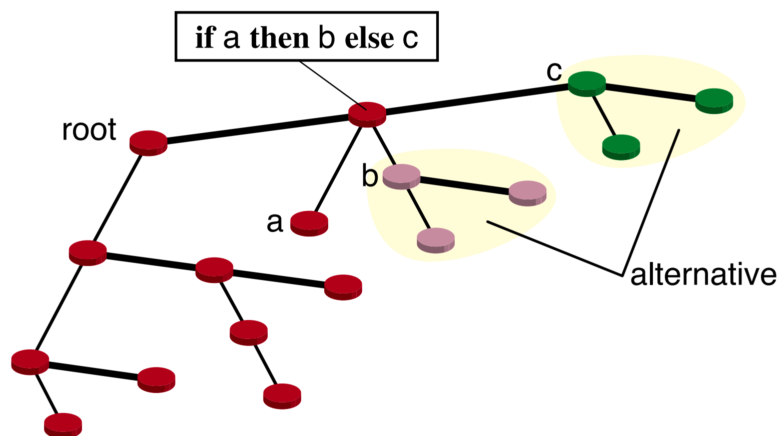


Figure 1: A program with several alternatives. Red nodes are unconditional. Green and lilac nodes are conditional. An if-expression chooses between the alternatives.

2.2 Execution graphs

The execution graph is intended to describe exactly one case of executions. Such an execution is valid at a specific state, a certain input and a certain choice of alternatives at indeterminism. For this case a graph, describing how different expressions depend on each other, exists.

The constraint on semantics is that when analogous conditions and choices of alternatives are present, the dependencies should be equal.

A graph contains nodes. Edges describe how they depend on each other. From the theory point of view the focus is explicitly on the dependency and not on the possible values being transported along the references of

[1] conditional clauses such as if-then-else and pattern matching.

[2] function calls or other subroutine calls.

the dependency graph. The nodes are conditional, unconditional or indeterministic. For the conditional and the indeterministic nodes only the ones used, are included.

The graph also contains instantiations of applications and calculations. In both cases a subgraph is rewritten from a value to another. In the first case the graph is expanded with the body of the function/procedure.

The whole program is expanded. Unused subgraphs are not included. They are replaced with unused[1].

Some of these nodes are executions. Such an execution can be strict with respect to its subgraphs. The strictness belongs to the semantics of the graph. It will not be discussed here. However, the strictness property specifies that some subgraphs must be computed before the execution of the node can be carried out. The subgraphs, however, can be computed in arbitrarily order. This leads to large degrees of freedom to choose order of execution between the subgraphs. Choosing the order of execution to achieve a high parallelism at low consumption of resources is possible.

The theory is based on the ability to do such choices in every node. In that way the subgraphs are allowed to execute in parallel, independently of each other. In this report the order of execution of the subgraphs[2] will not be discussed. However the report shows in which way the computer should be organized to use the parallelism in the best way.

To simplify visualising the execution, a mental picture of the graph is helpful. The graph is described in a three-dimensional space. In a horizontal plane a subgraph is described. Placed vertically are subgraphs, rewritten in executions. Higher the subgraphs are reduced more than lower. The uppermost plane is the final result. In the lowest plane are the function bodies found. Though, these are not drawn.

If the result of an execution is a permutation or a selection in a subgraph, the common parts in the result and the argument are not duplicated. Rewritten nodes are drawn on top of each other.

For every application there is such an execution graph. It contains as many nodes as there are executions and arguments. The graph contains all the information required to do the execution. The projected, horizontal area of the graph corresponds to maximal storage volume.

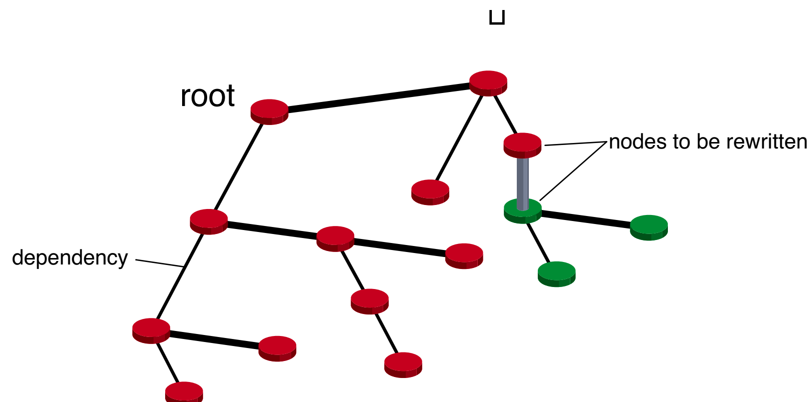


Figure 2: The execution graph shows two levels. The upper level (red) contains only one node that is to be rewritten. This node is illustrated in the second level (green). A part of the program (see figure Figure 1: on page 3)is not drawn (see figure Figure 3: on page 5).

The main usage of the execution graph is analysis of latency time and usage of temporary storage volume.

---

1. a token representing an arbitrary expression with features allowing execution of its father

2. it cannot be avoided that mechanisms for such selections must be studied.

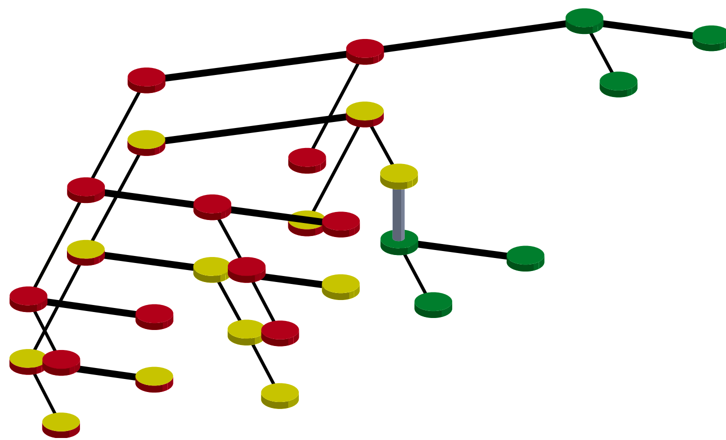
2.3 Universal execution graphs

A program can execute depending on many different states, ideas and choices of indeterministic alternatives. Thus many execution graphs – but only one for each such case. The number of execution graphs can be very large and may be infinite. Some of these execution graphs are more probable than others. Each execution graph has a finite size.

A “Universal execution graph” consists of all execution graphs of an application. Corresponding parts in the different alternatives are on top of each other. A Universal execution graph is very large, and may be infinite. The graph contains all the information to control a processor.

The projected area corresponds to the maximal graph required to store a practical execution. A Universal execution graph is used to estimate the volume of used memory.

Figure 3: A Universal execution graph. Two alternatives, red and yellow are shown. The yellow alternative contains an expression to be rewritten.



3 PERFORMANCE - latency time

An application has several execution graphs. Only one of them is studied here. It can be any of these execution graphs. For this graph the latency time is calculated. It is the time between the start of the execution and the traversal of the complete execution graph.

The different elements of the execution graph are associated with a basic computer element. A processor is a unit storing and performing executions. A node is associated with one processor. A transport is done by links. Each reference is associated with a transport[1]. Executions and transports takes time[2]. There are several embodiments of these basic mechanisms.

The graph of a typical application is shallow, maybe 20-100 nodes, and its size can be very different, maybe within the range 1k - 1G. There exist applications having much larger depth. Usually, the number of nodes is much larger than the depth[3]. Therefore, the parallelism is considerable.

The latency time can be calculated from selecting the longest time to execute one path between the root and a leaf. The time is the sum of the time for executions in nodes and transport times for references. Here the times are given including possible waiting times in queues.

---

1. the transport can be performed between processors or in the simplest case, between memory cells and registers in a processor. The mechanism here is completely general: all moves are transports.

2. some executions and transports are implemented in pipelines. Please ignore this for the moment. This will be explained further on in this report.

3. recursively defined applications, for instance text scanning, can be transformed to a parallel application.

$$t_L = \max_{t \in \text{paths}} \left(\sum_{e \in \text{path}_t} t_{\text{expr},e} + \sum_{r \in \text{path}_t} t_{\text{ref},r} \right) \quad (1)$$

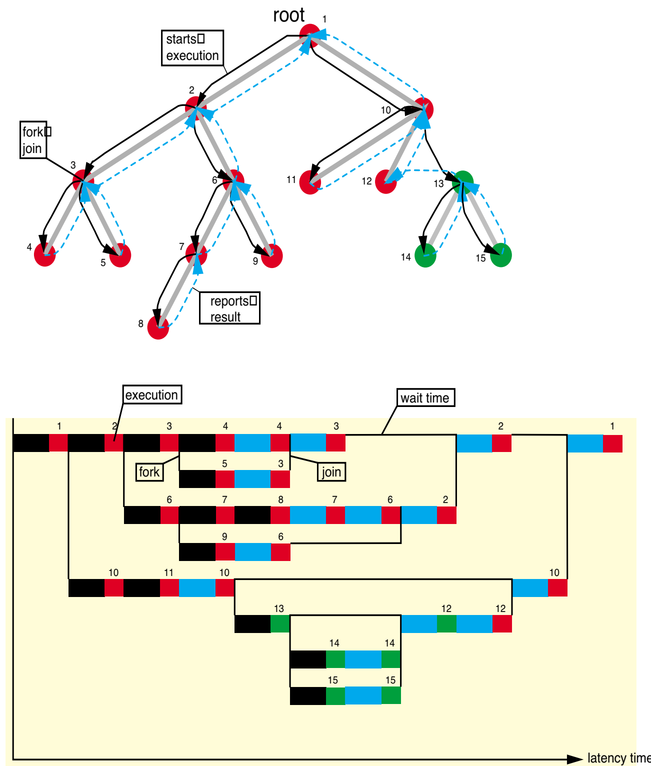


Figure 4: Latency time calculated with a view from the execution graph. Traversing towards the leaves are marked with black colour. Traversing in the opposite direction is marked with blue colour.

Processors and links execute by time multiplex of queued tasks. Tasks are allocated to processors and links. A view of this can be obtained from studying the particular execution graph during execution.

The execution normally executes from the root towards the leaves. Bodies of subprograms are instantiated. Then the execution turns back towards the root and computed values replace the instances. On the way towards the leaves, forks are done and on the way back towards the root, joins are done. At a join, the paths, not taking the longest time to execute, wait.

3.1 View from a processor

Instead of studying the execution graph from its root one can study it from every processor. Then the latency time is more easily estimated.

A normal computer has much fewer number of links and processors than the number of references and nodes of the graph, respectively. Maybe 1k-1G executions are multiplexed by each processor. Of these, the processor only stores the ones created still living. At most one of them is executing. The others are queuing. Correspondingly this holds for links.

The times for transport and execution consist of a dominating waiting time, and of a basic transport time and a basic execution time, respectively. The size of the waiting time depends on the number of nodes allocated on the unit. Waiting time consists of idle periods, when no work is done, and time periods, when other work is carried out (i.e. queue time).

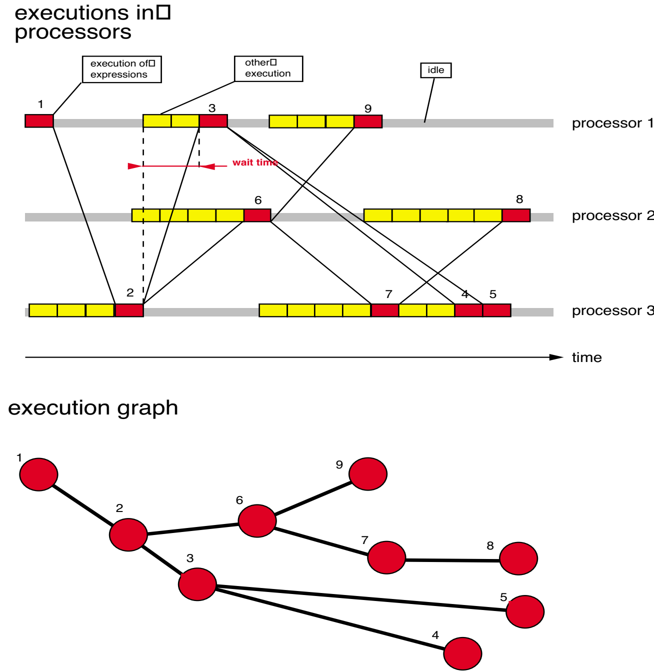


Figure 5: The execution viewed from the processors. One execution graph (red) is executed on three processors. Other executions (yellow) cause waiting times.

The time of arrival for an execution is stochastic and can be approximated by a Poisson distribution. Every processor and link can be viewed as a Markov process. It does an execution or a transport. The queues consist of stored unexecuted and untransported expressions, respectively. They cause the demand for temporary memory space. The size of these queues will be discussed further on in this report (see paragraph 8 on page 17).

The probability p_k for a queue of length k at the load ρ is:

$$p_k = (1 - \rho) \bullet \rho^k \qquad (2)$$

A *massive execution* is the execution of a lot of expressions of a graph ignoring dependencies. A large number of expressions are queuing for execution. Sometimes the queue is empty. The total latency time for a processor is given by the sum of the execution time t_{unit} for all elements in the queue, and the time when the queue is empty. Every such expression is an instance i of a certain template e.

The latency time for massive execution is:

$$t_{mass} = \frac{1}{\rho} \bullet \left(\sum_{e \in EXPR_p} \sum_{i \in expr_e} t_{unit,i} \right) \qquad (3)$$

However, in the real execution graph, the order of the executions must be considered. The latency time can be estimated to be within an interval settled by the following conditions:

- the latency time is larger than or equal to the latency time for a massive execution. The latency time is given by the unit with the largest execution volume V_{unit} and the load ρ, where *unit* stands for expr or ref.

- there exists a longest path between the root and a leaf. This path has length D. The longest latency time is obtained when this path is executed latest, i.e. after all other executions have been carried out. Usually, a corresponding part of the particular path has been computed as done for all other paths. The remaining nodes and references of the particular path are sequentially executed.

This can be summarized in the following relation:

$$\frac{1}{\rho} \bullet V_{unit} \bullet t_{unit} \le t_L \le \frac{1}{\rho} \bullet \left(V_{pr} - \frac{D}{N_{unit}}\right) \bullet t_{unit} + D \bullet (t_{expr} + t_{ref}) \qquad 4$$

Generally the depth D is much smaller than the volume of a unit. Thus, the last term is very small and can be ignored. The relation then transforms to an equality. Then the latency time equals the execution time for the most loaded unit.

There are some special cases:

- there is only one processor. Latency time is calculated (see equation 4) by summing the time for all expressions. This corresponds to the traditional estimate.
- there are enough processes to compute every expression in a unique processor. Then V_{pr}=1 and D/N_{unit}=1. The rightmost term settles the upper limit. The upper limit of the relation is correct latency time.

4 ALLOCATION OF EXPRESSIONS

The main semantic operations are instantiation, selection and rewriting.

instantiation

In both imperative and functional implementations a template is used to create an instance. Instantiation is accomplished through copying a template and join it with a state. Copying is carried out locally in any processor. An instance is created in the same processor.

For imperative implementations the template is a sequence of instructions. They control a sequence of copying operations (load instructions). Normally the state is the stack and some registers.

For functional implementations the template is an expression stored in a graph. The state is a reference to a data structure containing the arguments.

selection

At a selection, an operator accessing a value in a data structure is used. The operator has the data structure as an argument. The operator recursively traverses down in a graph and eventually finds the target element. If this element is an atom, it is copied. Otherwise, a pointer to the target element is created as a result. There are corresponding mechanisms for both imperative and functional applications.

For imperative implementations the operator consists of a sequence of instructions. This is a sequence of dereferences. Generally this is a particular load instruction not reading the value but returning the address. The particular access is often an address format of a more general instruction. Usually, compilers can convert a sequence of dereferences for static and known data structures to one.

For functional implementations the operator is a selection expression stored in a graph. The selection expression may be a list of indexes in the graph.

Rewrites

Rewrites are always carried out locally[1] in a processor. Rewrites can be of different types of permutations or numerical functions.

Such a typical function is Add. In imperative applications this is an instruction and in functional applications this is a built-in function.

During instantiation and selection there are four different cases for the locality. For each of the operator and the argument exists the cases that it is stored locally in a particular processor or elsewhere.

The execution graph can be changed to contain an extra execution locally, referring only to the non locally stored operator or argument. Therefore the allocation of the operator is always carried out locally. Here the template and the datastructure are chosen to be local for the instantiation and the selection cases, respectively. The argument and the operator, respectively, may then be local or global[1]. Analysing the case when nether operator nor argument is local is of no use, because this would only increase the number of transports.

4.1 Allocation

Allocation is the mechanism for placing an expression in a particular processor. In this section the communication between processors is ignored. Of this reason the allocation will be done to use the execution capacity in the best way.

Several allocation algorithms exist. They have different properties. Here a basic algorithm, based on random allocation, is chosen. Other algorithms can be related to this one. The random algorithm can always be carried out.

instances

The random algorithm is based on the existence of an address space identifying all processors. Every template expression is a pair of an expression and a processor address. The processor is randomly chosen.

The allocation results in placing of almost the same number of template expression in every processor. If the number of allocated templates is much greater than the number of processors, the distribution of templates in the different processors will be even. This holds for normal programs.

At random allocation there is no need for knowledge of the graph and its behaviour. Every set of templates of an execution graph is stochastic. Therefore even the templates of a universal execution graph are stochastic. In a graph with subgraphs, these are stochastic, and therefore the super graphs are also stochastic. The effect is that the distribution of the number of templates between the processors is even during all situations.

selections

Expressions are of normal form after they have been computed. Selections can only be done on graphs of normal form. No direct relation between the allocation of selections and templates exists. In the worst case only one template creates all graphs, from which the selections will be done. Therefore, single processors can be overloaded.

---

1. in an imperative machine this is performed giving an instruction code for how to carry out the rewriting. In an applicative machine the value of the expression settles the rewrite operation.

1. during a move, load on an additional processor occurs and the communication is loaded further. If there is more than one reference to a template/data structure, then it is not efficient to move the template/data structure.

Programs generally work in several phases. Every phase creates a data structure of its own. This can be a constant or a function to be accessed or instantiated. Only the first phase can be allocated during compilation. Other phases must use dynamic allocation.

Therefore assume that also for constants some random allocation methods exist. How it is implemented does not matter here. Only the property, that such data structures are evenly distributed, is used.

Rewrites

Because instances and selections are allocated, the expressions to be rewritten are already allocated. There is normally no reason for moving such expressions to another processor[1].

Assume that the execution graph contains a volume *V* of instances, selections and rewrites, and that there are N_{pr} processors. Then the maximal volume in a processor is:

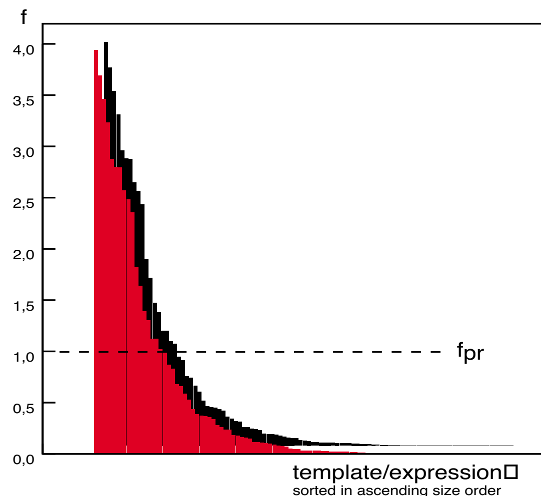
$$V_{pr} = \frac{V \bullet q_{alloc}}{N_{pr}} \qquad (5)$$

Here q_{alloc} is a measure of quality, used to specify the number of instances in the most loaded processor.

4.2 Duplication

Some graphs are often instantiated or selected. Therefore, processors, where these are stored, can be overloaded. To eliminate this problem, such a graph can be duplicated and allocated at other locations, in amount making their load not dominating.

Figure 6: An example of the frequency of instances of different functions and data structures. The frequency shows the number of instances or selections carried out per unit of time. f_{pr} shows the speed of execution for one processor.



Developing statistics on the frequency of used templates and data structures is possible. This can be based on the smallest element possible to allocate – an expression or a node in a data structure. The frequency can be described on the y axis, and the expressions can be placed sorted in decreasing frequency order along the x axis.

In the diagram above a typical such distribution is shown. A line has been added to describe the maximal execution frequency f_{pr} for a proces-

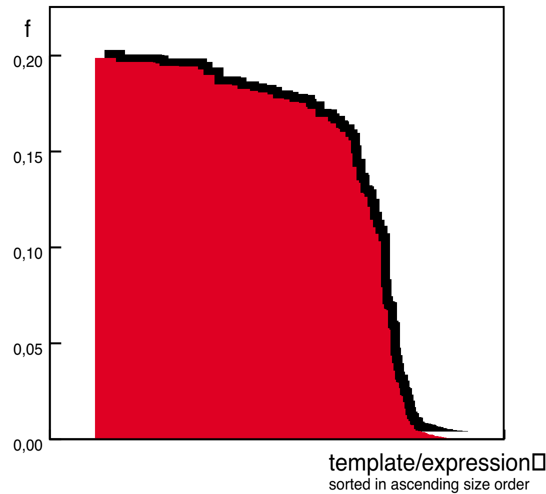
[1] in a further section the possibility to have processors, with different performance and insufficient functionality, will be discussed. This possibility implies that expressions in certain cases must be moved to another processor.

sor. All expressions with higher execution frequency will load a processor, increasing the latency time of the computer.

It is known that certain programs have such an uneven distribution that this is the case. This holds generally for signal processing and graphical programs. One example is the fast Fourier transform.

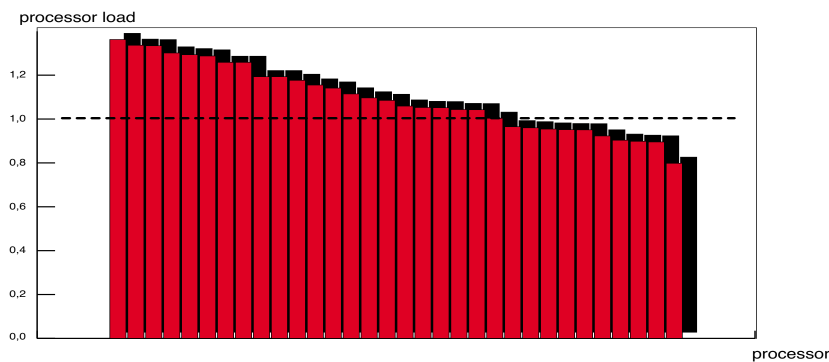
Duplicates can be created by different algorithms. Here a basic algorithm is used. It is based on the existence of a load f_{dub} constituting the largest quanta to execute on a processor. Therefore all expressions/nodes with a higher load will be duplicated so every duplicate has a load smaller than f_{dub}.

Figure 7: The result after duplication. f_{dub} is chosen to be 0.2. The statistics come from Figure 6:



After such duplication, the distribution of the load is much more even. It will not be perfectly even due to the finite number of expressions/nodes loading a processor. The number can from time to time be rather small.

Figure 8: The load in the different processors after duplication has been done. The load is based on 32 processors and 64 expressions. Load-limit is set to 0.25, giving an average of 8 expressions in each processor.



Every duplicate creates a demand for further storage volume. For every variant of the execution graph duplication is done. In certain cases, the super variant cannot use all the duplicates, since they have been allocated for some worst load on a specific node.

The load on a processor ρ_{pr} is the quotient between actual frequency of execution and the maximal frequency of execution. Execution is carried out traversing first towards the leaves and then back towards the root. Every execution in a node is assumed to take the time t_{exec}. Then the latency time for the most loaded processor is:

$$t_{Lexpr} = \frac{1}{\rho_{pr}} \bullet V_{pr} \bullet 2 \bullet t_{exec} \qquad (6)$$

5 ALLOCATION OF PATH

Usually two major alternatives for implementing communication are used: crossbar and network. These are discussed separately although a crossbar to some extent can be regarded as a special case of a general network.

5.1 Crossbar

The crossbar is a rich network with point-to-point connections between referred expressions. Only one link exists between executions[1].

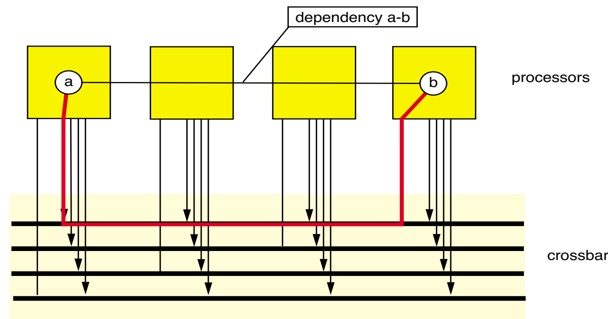


Figure 9: A topology based on a crossbar. The number of multiplexers increases with the square of the size of the system. Only one link is implementing a reference.

5.2 Network

The network is a general purpose transporting mechanism with nodes and links. The nodes are processors[2]. Here the network is assumed to be able to connect arbitrary expressions stored in different processors. Between executions there are one or more transits of links.

A reference represents a path of processors separated with links. To model this, the execution graph is adjusted to contain several relays. They are executions that only transport. In that way the size of the execution graph increases. After that, the execution graph can be directly mapped on the processors.

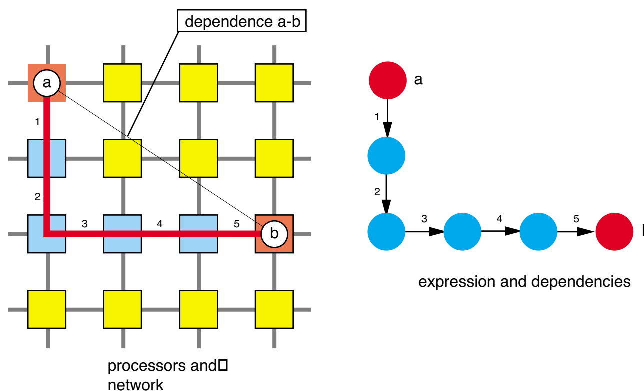


Figure 10: A computer with processors and network (to the left). The expressions a and b are stored in two processors. a depends on b. In the network the links 1-5 are connecting the processors to a and b, and four processors lying between. They are equivalent to four extra expressions acting only as relays.

For the resulting execution graph the performance estimate expressions are valid. The life times of threads increase[3].

---

1. corresponds to a network with m dimensions, where m=the number of processors
2. here the processor is assumed to be a general purpose. Further on in this paper special cases will be discussed, where the processor is considered to have limited functionality. In the simplest case it can only transport expressions.
3. threads will be discussed further on

6 PLACEMENT

A topology is defined. It is based on a space containing processors. A processor is assumed to have a certain position. The topology defines how these processors are connected with links. The topology is characterized having a dimension and parallel paths. These properties will be further discussed (see paragraph 7 on page 15). Moreover, every transport in the network is characterized by the need to traverse a certain number of links.

Placement is to place expressions in particular processors in the topology. Thus, the expressions are given a position. The representation of the position is of no interest here.

A reference is implemented as a transport along a path. It consists of a sequence of links, separated by processors. A path is normally placed to be short, to reduce the system load.

The load on each link depends on what placement and how paths are chosen. The load corresponds to the number of paths for an execution graph passing the particular link. An exact load on each link can be hard to estimate. The average load for links can be estimated well.

Assume that an execution in the execution graph is referred by k_{ref} references. A reference has a path consisting of h links. Every reference transports pr messages depending on the protocol implementing the reference. On the average every processor has k_{link} links. Within a certain volume the number of executions and the number of transports can be summarized. After that, the average number of transports on a link can be calculated.

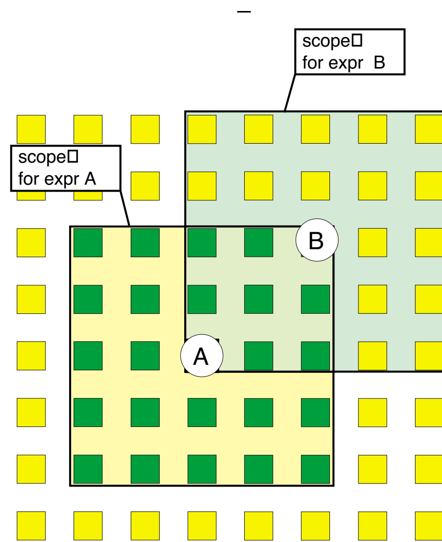


Figure 11: A unit, processor or link, is influenced by accesses from expressions nearby. An application has a globality defining the size of a scope. Here all expressions within A's scope influence the expression A. Such an expression B is marked.

Assume that the number of executions V_{pr} is constant. Then the load on a link increases with k_{ref}, h and pr, but decreases with k_{link}. The load ρ_{link} on a link will be defined further on (see equation 27).

k_{ref} depends on the application. pr depends on the low-level implementation. h depends on how well the placement is done. In the best case it can become a smallest value given by the globality of the application. The latency time caused by a link in the network is:

$$t_{Lref} = \frac{1}{\rho_{link}} \bullet \frac{V_{pr} \bullet k_{ref} \bullet h}{k_{link}} \bullet pr \bullet t_{msg} \qquad 7$$

The average load can be considerable different from the actual load in a link. This is due to that even for spacious located executions many paths can pass a certain point, forming a “hot spot”.

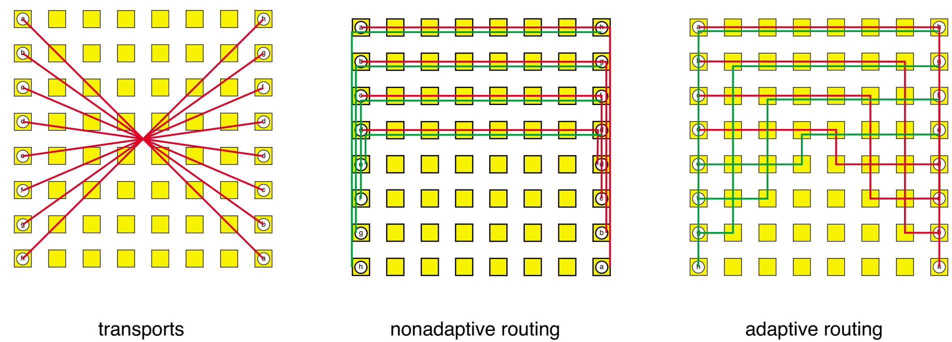


Figure 12: Hot spot caused by transports. One column (to the left) with expressions depends on expressions in another column (to the right) having a different order.
References cross each other in the middle and cause 8 references in the same point.
In a 2-dimensional mesh with clockwise Manhattan routing the references are gathered in two vertical streets.
Using adaptive routing, these vertical streets can be split.

The total latency time can be calculated considering both executions and transports. The longest of them gives the latency time:

$$t_L = max\begin{cases} t_{\text{Lexpr}} \\ t_{\text{Lref}} \end{cases} \quad (8)$$

If the computer is execution limited, then placement does not affect the latency time. In the complementary case the latency time increases proportionally to h, which entirely depends on the placement.

For a certain link the goal is to reduce h for all transports passing the particular link. Doing this minimizing is extremely difficult. An approximate method can be to divide the problem into two levels: one macro and one micro problem:

1 place to reduce the average load over an arbitrarily chosen subspace.

2 using a short path, route to reduce link loaded.

This division of this problem has been used for routing and placement of VLSI circuits and printed circuit boards. The technique has been proven to be useful.

Routing is an adaptive algorithm. It is based on the analysis of the transport load around an imaginative path and allocates on not overloaded links. The goal is to reduce the load on the maximally loaded link. At a low load, the path may be defined.

The placement algorithm allocates nodes close to each other, to make the load of the processors equal. The goal is to reduce the average path length of all paths. This corresponds to reducing the total path length.

Within the technique for integrated circuits and printed circuit boards suitable algorithms exist. Some such algorithms are based on “simulated annealing”. It has been proven that such algorithms always solve the problem to certain accuracy. For the placement and routing algorithms, several different algorithms are possible. It is not the purpose to select an algorithm here. It should only be understood that such an algorithm causes h links to be traversed for each reference.

7 TOPOLOGY

In this section general purpose networks are studied. These networks can connect all nodes. There exist other networks, which are more limited and do not connect all nodes.

The topology states how nodes (i.e. processors) are connected. The connections can be parallel or in different dimensions.

7.1 Globality

A definition of locality is introduced. Call it globality. The reason for this is that a measure *G* will be used. The lesser this value is the better locality. The globality is:

1 Start from the Universal execution graph. Divide it into a hierarchy of regions. For each reference chose the smallest region containing all the executions referred by the reference. The total number of expressions in this region is the globality of the particular reference. Due to large subspaces, this value can be high.

2 Calculate the average globality for all references.

3 Repeat step 1 and 2 for all possible region divisions. Then select a division having smallest average globality. This globality is the globality G.

The globality G expresses the average number of expressions present in the scope, in which a reference does an access. Because the regions are arbitrarily constructed, the definition of locality is not topology dependent but program dependent.

7.2 Dimensions

A change in topology affects the number of links a message transits for a particular reference. The topology is mainly given by the number of separate axises used. An axis is a direction of a transport not loaded by transports along other axis. The number of axises is called the dimension. This can be 0, 1, 2, 3, 4 etc.

A reference is implemented as a path between two processors containing the two communicating expressions. A smallest scope contains the two communicating expressions. The size and form for this scope are topology-dependent.

Using a n-dimensional space, the edge *L* of the scope is proportional to the n:th root of the globality *G*:

0 - bus
A 0-dimensional space is a bus. All transports pass the only link of the network. Because all references pass this single link, the globality does not affect the load.

1 - chain
A one-dimensional space is a chain of processors with links between. The closest L processors communicate with each other.

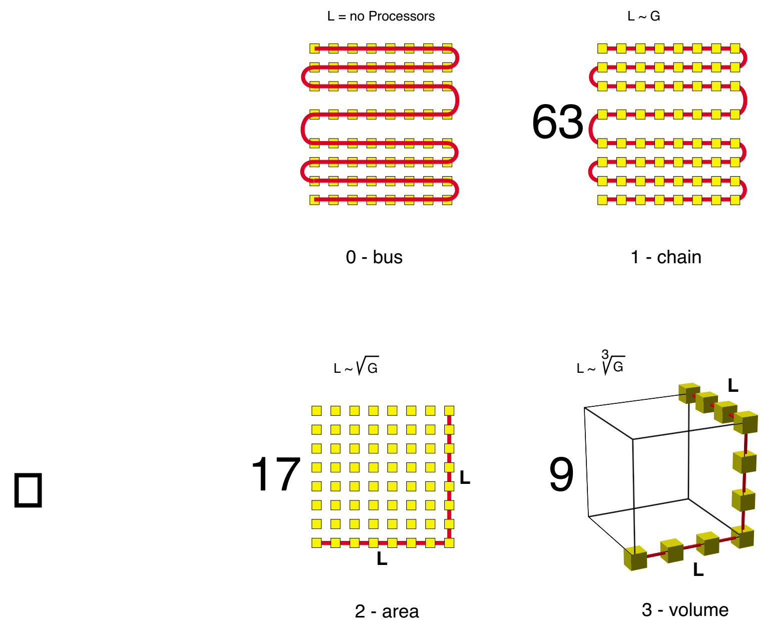
2 - surface
A two-dimensional space is a mesh in a plane. The edge of the scope L, corresponds to the square root of the size of the space.

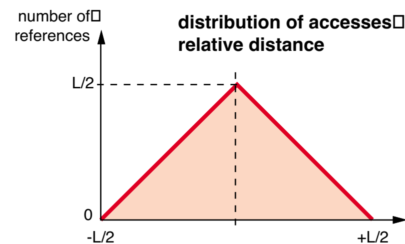
3 - volume

The corresponding holds in three dimensions as in two dimensions.

Placing nodes in the space in an appropriate way, one can map the regions of globality. Actual placement cannot do this perfectly. A scope with edge L becomes q_G times greater than the optimal space G.

Figure 13: Different topologies. Variation of the dimension between 0 and 3. The transport length (large figures) substantially decreases when the dimension increases. Above, the load on a link in the scope is shown.





The edge L can be calculated considering the factor of quality q_G and the average number of messages in each processor[1]:

$$L = \sqrt[\dim]{\frac{G \bullet q_G \bullet N_{pr}}{V}} \quad (9)$$

On the average the number of traversed links in each dimension is half the edge of the scope. The average load on each link within the scope is:

$$h = 0.25 \bullet (L - 1) \quad (10)$$

Thus, the globality and the load are almost proportional. In a communication restricted application, the load on the links is strongly dependent on globality and dimension. On the other hand, in an execution limited application, the latency time is not depending on the dimension or the globality.

[1] V_{pr} is the maximum number in a processor, while V/N_{pr} corresponds to the average.

A crossbar network is a specialization. The length of the path is one. Each processor has N_{pr}-*1* links to other processors. The placement will imply an even load between the N links. On the average the load becomes the total load divided by the number of links:

$$t_L = \frac{1}{\rho_{\text{link}}} \bullet \frac{V \bullet k_{\text{alloc}} \bullet k_{\text{ref}}}{N_{\text{pr}} \bullet (N_{\text{pr}} - 1)} \bullet \text{pr} \bullet t_{\text{msg}} \qquad 11$$

The load on a link decreases approximately with the square of the number of nodes. This is one dimension more than how the load on the processor decreases. Consequently, the transport load can be substantially reduced using a crossbar, however, to the cost of more hardware.

7.3 Parallel links

An increase in the number of parallel links, reduces the number of messages send per link. The links can be connected in a different topology or just parallel. Earlier expressions model this in a correct way (see equation 7)[1].

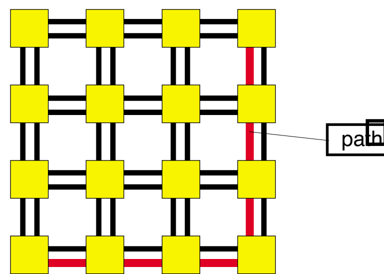


Figure 14: Parallel links are used to increase the transport capacity. This does not change the transport length.

8 STORAGE VOLUME

The processors are storages. Depending on the granularity, they can be modelled as separate memory cells or as more conventional processors, with a local memory containing several memory cells. The difference between these models is the introduction of another network. The memory cells can communicate with the processor through a bus. We ignore this more fine modelling, because it does not influence the general discussion.

The storage is the only device storing values. Therefore, the memory is a part of the processors. The storage volume is of paramount interest, is a major part of a computer cost. There is no sharp limit between the different usages of the storage. It can be used for:

program code - V_{prog}
: The program code contains the templates to be instantiated. The program code is completely application dependent. It contains data structures of normal form. These can be duplicated.

state - V_{state}
: This is the state gradually changing in the pace with input. The state is completely application dependent. It contains data structures of normal form. Parts of the state may be duplicated.

queue from input, queue to output - V_{in}, V_{out}
: These are queues from input ports and queues to output ports. They contain data structures of normal form. Parts of the queues

[1]. Generally the number of links depends on how many wires there are room for. An increase in the number of links implies lower capacity for an individual link. Thus, the total capacity is almost constant. The total capacity increases with the number of available wires.

may be duplicated. The size of the input and output queues are given by the application. They can to some extent be limited by an intelligent control of the execution. Often the queues are small and in that way marginally affecting the total volume.

temporary data structures - V_{tmp}

These are data structures with a short life time, used for the execution of an output or caused by an input. Parts can be of normal form. Only parts of normal form can be duplicated. Please ignore for the moment that such duplication is possible.

All the data structures belong to the universal execution graph. k_{dup} is the average factor of duplication for everything but the temporary data structure. Normally the program code becomes heavily duplicated. The total memory volume is:

$$V_{\mathrm{mem}} = (V_{\mathrm{in}} + V_{\mathrm{out}} + V_{\mathrm{prog}} + V_{\mathrm{state}}) \bullet k_{\mathrm{dup}} + V_{\mathrm{tmp}} \qquad 12$$

The temporary memory volume strongly depends on how the execution is done. Therefore, it is to be control in the right way. A bad control may imply an explosion of size – which can become dominating over the other data structures or become too large for the computer.

The size can be approximated studying threads (see paragraph 9 on page 18). A thread exists for each execution. The leaves have a thread depth of 1. Other nodes have a finite depth. The volume of temporary storage corresponds to the volume of the number of open threads. This corresponds to the number of threads N_{thr} multiplied with its average depth D_{thr}:

$$V_{\mathrm{tmp}} = N_{\mathrm{thr}} \bullet \overline{D_{\mathrm{thr}}} \qquad 13$$

The average depth is given by the thread algorithm (see paragraph 9.3 on page 21) and the application. The number of threads is controlled by a main control for the computer (see paragraph 10 on page 22).

9 THREADS

Most applications contain a large amount of parallelism. Every parallel execution requires a certain temporary storage space. If all executions were to be carried out in parallel, enormously large memories are required. Normally, no such memory capacities are available. Consequently a mechanism to limit the parallelism is required.

The limitation is done by threads. This feature might be a specialization. This is not so. However, the text below is written as to some extent touch an implementation.

Parallelisms are of different kinds. Therefore, note the difference between:

1. The parallelism in an application. This is the quotient between the entire volume of the execution graph and its depth (V/D).
2. The parallelism in the present computable expressions.
3. Parallelism in the present schedule and queuing expressions.
4. The number of parallel processors.

In order of appearance the ratios between these can be several orders of magnitude. By controlling the possibility to execute an expression, the parallelism can be limited. Between the parallelisms mentioned above a control method, screening the expressions to be used on the next level, is used.

9.1 The Tread

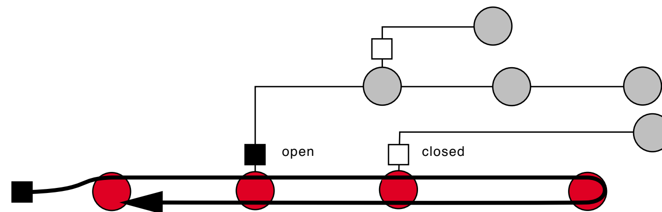
During the execution of a graph, the graph is traversed from the root towards the leaves and then back towards the root. During the traversal towards the leaves, the expression is screened by the control as mentioned above. Two cases exist in this situation – there is only one son or there are several.

In the general case each son could be decided to be allowed or not allowed to execute. In that case an execution could be temporarily suspended in a node. There is no mechanism, improving performance or price, benefiting from such an action[1]. Consequently it is assumed that at least one son is allowed to execute. Thus, there is at least one path between the root and a leaf executing. Such a path is a thread.

A thread has a root. There is a path in the graph between this root a leaf. All nodes and references along this path are included in the thread. Nodes in the thread may have branches. Some of these are used. Such a branch is a subgraph. It has a thread starting in its root.

A thread can be closed, open or finished. A closed thread is a thread not allowed to execute. An open thread is executing and has one node allowed to execute. A finished thread is of normal form.

Figure 15: A thread. Circles are expressions being instantiated and executed. Read circles belong to a thread. Grey circles are other threads. A black rectangle is the root.



The total number of threads is controlled. The memory volume of a thread corresponds to the number of nodes between its root and its leaves. A thread is not finished until all subthreads are finished. Therefore, all nodes under the root of a thread, must be traversed before the thread is finished. The sum of the volume of all threads is the temporary storage.

If the execution graph does not contain any executions rewriting the execution graph to become lesser, the temporary memory volume would be constant.

Normally most of the branches contain executions such that rewrites increase and decrease the depth of the thread. In that way the temporary memory volume varies. Using control, the total temporary memory volume can be kept below an acceptable level.

Generally parallel threads execute completely asynchronously. Therefore, there must be memory space reserved for the sum of their maximal memory volume. The total memory volume corresponds to the sum of the memory volume of all open threads. Therefore, when there are a fixed number of threads, these should be as shallow as possible to reduce the demand for memory.

The sum over all threads gives the total memory volume. This can be divided by the number of threads to receive an average depth per thread. This average depth is a measure on how ”depth first” the schedule is:

---

[1]. if several executions are performed concurrently to generate data to ports with different priority (low priority often implies later output), the start up of such a parallel execution can motivate an interrupt. However this is not the normal case and it must be treated separately.

$$\overline{D_{\text{thr}}} = \frac{1}{N_{\text{thr}}} \bullet \left(\sum_{t \in \text{threads}} D_{\text{thr},t} \right) \qquad 14$$

9.2 The thread mechanism

It is known that depth first control results in an acceptable and low temporary memory volume, not always the least but often close to it. The depth first mechanism, uses only one thread at a time. Therefore, no parallelism exists. Does a similarly featured mechanism with many parallel threads exist?

The control of the threads may be designed to mimic the depth first traversal. Instead of sweeping a path between root and leaf through the whole graph, sweeping a branch consisting of many such paths is possible. These paths do not have to be synchronized, but must fulfil some criteria to reduce the demand for memory volume.

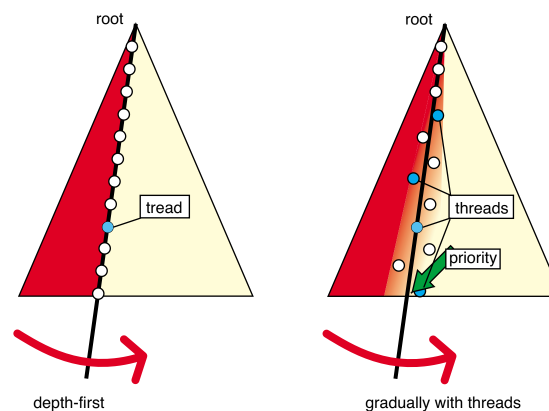


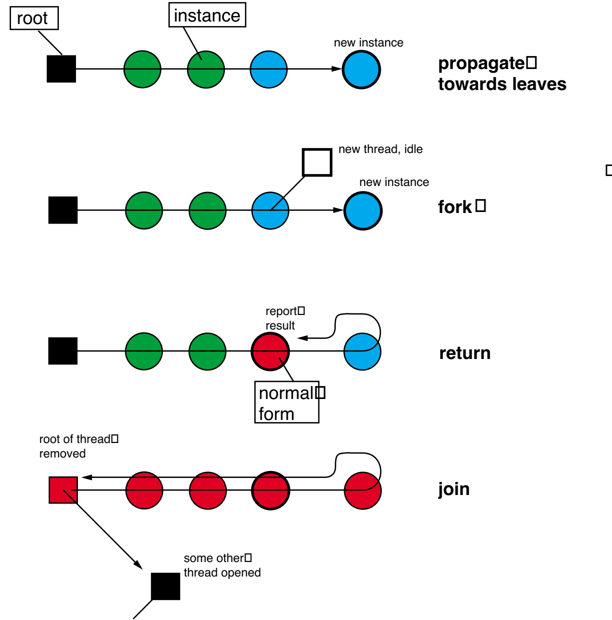
Figure 16: The classical Depth-First control (to the left) has only one thread. This thread sweeps through the tree from left to right. The thread algorithm, described here, has a segment with an almost constant number of threads. This segment is swept in the same way.

The control for this mechanism is here called the *thread mechanism*. It only works on threads and controls in this way:

- an open thread traverses towards the leaves, and then traverse towards the root.
- when a branch is found, one branch is selected to continue the traversal. The selection depends on the thread mechanism and to some extent to the expression stored in the node. The other strict branches are marked as new, unopened threads.
- when a leaf is reached, execution turns back towards the root.
- when a thread has finished executing, it is finished. Unopened threads are opened. Normally only one thread is opened. The number of open threads may be variable and is to be kept within certain limits. Guaranteeing that at least one open thread is left or threads are spontaneously opened, is important.

Largely the selection of a new thread settles what the execution tree looks like. Giving priority to the latest created threads, belonging to the oldest threads, the depth first mechanism is mimicked. Here too, the algorithm is not required to be absolute but can be approximate.

Figure 17: The thread mechanism has five main functions: propagation towards/from leaves, fork, return from leaves, and join.



9.3 A thread's lifetime

The lifetime of a thread is the latency time for traversing towards the leaf and returning to the root. For each node in the subgraph the latency time can be calculated in the same way. Then a recursive algorithm is obtained:

$$\tau = \max_{b \in \text{branches}} \tau_b \tag{15}$$

Assuming a massive execution being in progress, the latency time for an arbitrary subgraph can be estimated from its size V_b and available execution frequency f_b for the subgraph:

$$\tau = \frac{V_b}{f_b} \tag{16}$$

Assume that total available executing rate is f. Then the sum of all executing rates of all of the subgraphs equals F. The different subgraphs can start executing after each other or to some extent overlapping. This overlap causes a branch not to finish execution before the other branches start executing. Therefore, a branch is not given all available resources. This causes the branch to work longer.

Here a rough approximation of an ideal case is treated, where N_{thr} threads are used. This execution consists of a thread from the root, sharing the depth D with the execution graph, and also N_{thr}-1 additional threads at the leaves, forming a perfect tree. In each node the perfect tree has a branching factor β. These threads are started and finished executing before the next corresponding tree is executed.

There is a thread from the root of the execution graphs to the outer tree. If the parallelism is high, this part of the thread can be ignored. The volume of the outer three is:

$$N_{\text{thr}} \bullet \overline{D_{\text{thr}}} = 1 + \beta + \beta^2 \ldots \beta^{D-1}$$

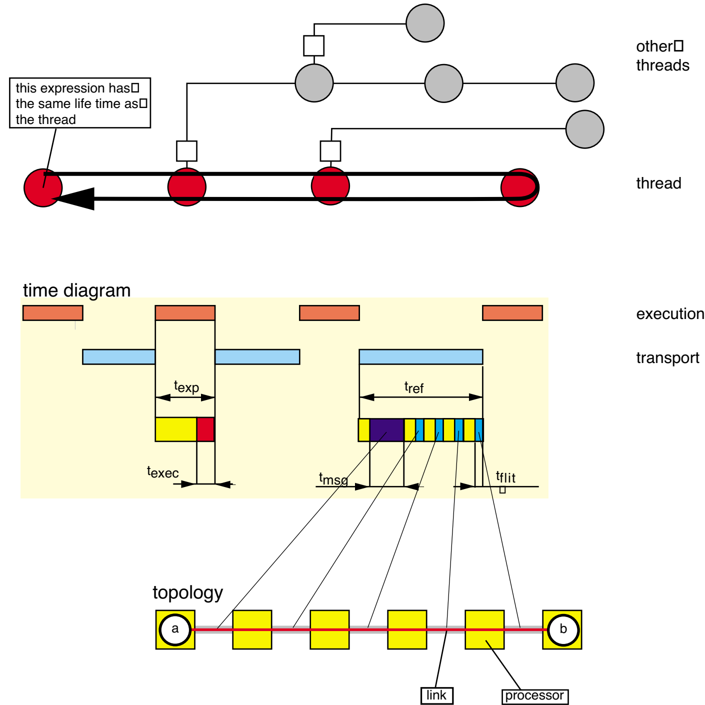
$$N_{\text{thr}} = \beta^D \tag{17}$$

Then the volume per thread is:

$$\overline{D_{\text{thr}}} = \frac{\beta}{\beta - 1} \bullet \left(1 - \frac{1}{N_{\text{thr}}}\right) \bullet q_{\text{thr}} \tag{18}$$

When the number of processors grows the number of threads N_{thr} also grows. Therefore, the right term in the right factor becomes neglectable. The result is then the left factor. At a branching factor of 2 and 3 the depth is 2 and 1.5, respectively. A factor q_{thr} is added compensating for the actual implementation.

Figure 18: Execution of a thread. The thread is propagated towards the leaf and back towards the root. Each execution consists of a waiting time and an actual execution. The references consist of a message transport time and many durations for hops in the network. Each such message transport time consists of a waiting time and the actual time. The waiting time depends on the load.



The latency time for a reference is the total time of traversing all the links corresponding to that reference. The message is sent as flits over *h* links in a pipeline in *dim* dimensions. Every reference must be sent *pr* times over the path to carry out the reference. The load on the links increases the latency time.

$$t_{\text{ref}} = \frac{1}{1 - \rho_{\text{ref}}} \bullet \text{pr} \bullet (t_{\text{msg}} + h \bullet \text{dim} \bullet t_{\text{flit}}) \qquad 19$$

The corresponding holds for the processors.

$$t_{\text{expr}} = \frac{1}{1 - \rho_{\text{expr}}} \bullet 2 \bullet t_{\text{exec}} \qquad 20$$

The lifetime can be rewritten to consist of a mean transport and execution time multiplied with a thread depth, representing the number of executions along the thread.

$$\tau = \overline{D}_{\text{thr}} \bullet (t_{\text{expr}} + t_{\text{ref}}) \qquad 21$$

This function has divided the dependency in one architecture dependent $(t_{expr}+t_{ref})$ part and one execution graph dependent part D_{thr}. The average depth is a measure on how serial the execution is. This depends on the form of the execution graph and the thread mechanism. The average depth is wanted to be as small as possible.

10 CONTROL

The performance of a computer must be regulated. Otherwise, resources may be consumed in large quantities. An execution frequency f is adjusted by regulating the number of available threads. The latency time then becomes:

$$t_L = \frac{V_{\mathrm{pr}}}{f} \qquad (22)$$

A maximal execution frequency is depending on whether the computer is transport limited or execution limited. An execution limited computer has an execution frequency f_{pr}, given by the processor load and the execution time of a basic execution. A transport limited computer has an execution frequency f_{link}, given by the maximum transport rate and the number of transports per execution.

The execution frequency f is given by the temporary storage volume (the product of all open threads N_{thr} and the thread depth) and the thread lifetime length τ:

$$f = min\begin{cases} f_{\mathrm{pr}} \\ f_{\mathrm{link}}/k_{\mathrm{ref}} \\ V_{\mathrm{tmp}}/\tau = N_{\mathrm{thr}} \bullet \overline{D_{\mathrm{thr}}}/\tau \end{cases} \qquad (23)$$

where

$$f_{\mathrm{pr}} = N_{\mathrm{pr}} \bullet \frac{1}{2 \bullet t_{\mathrm{exec}}} \qquad (24)$$

$$f_{\mathrm{link}} = N_{\mathrm{pr}} \bullet k_{\mathrm{link}} \bullet \frac{1}{h \bullet pr \bullet t_{\mathrm{msg}}} \qquad (25)$$

This execution frequency results in a certain load on processors and links:

$$\rho_{\mathrm{expr}} = f/f_{\mathrm{pr}} \qquad (26)$$

$$\rho_{\mathrm{ref}} = f \bullet k_{\mathrm{ref}}/f_{\mathrm{link}} \qquad (27)$$

The load on the processors corresponds to a certain number of open threads. In that way the threads are given a certain lifetime and a certain temporary storage volume arises. The number of open threads can be controlled depending on, for instance, used storage volume, processor load and transport load.

Some properties:

1 In an execution limited computer: an increase in the number of threads increases queue length and degree of usage. If no problems using too large storage volume arise, latency time decreases.

2 In a transport limited computer: if the number of threads is increased above a certain limit, the network cannot transport any more messages. The execution capacity is saturated. An additional increase only implies more storage consumed. Here a control system must consider the use of storage. If not, the demand for storage explodes.

11 OPTIMIZATION

An architecture with only one type of processors and links can be optimized. Here is described the way different parameters affect performance.

11.1 Granularity

Equation 23 shows how the execution frequency is settled. Its upper limit is given by the capacity of processors and links. The following cases can be identified:

1 only the number of processors N_{pr} is reduced by the factor x. The processor capacity is correspondingly altered. The link capacity decreases less because h depends on N_{pr}. The net effect is that the capacities of all system resources decrease.

2 use fewer large processors. The number of processors N_{pr} is reduced x times and the number of external references k_{ref} is reduced $x^{0.5}$ and $x^{0.67}$ times for 2 and 3 dimensional topologies, respectively. Thus, h is altered as $x^{0.5}$ and $x^{0.33}$ respectively. The net effect is that the processor capacity is decreased x times and the transport capacity is unchanged. In the 0- and the 1-dimensional cases the transport capacity is constant.

3 use fewer and more powerful processors. As well the number of processors as t_{exec} is reduced by x. The rest is as in the preceding case. The execution capacity and the transport capacity are constant.

The three examples mentioned above, showed three main dimensioning possibilities. Getting improvements by only altering the number of processors is not possible. However, if one assumes that the programs become more local in each processor and that the number of external references k_{ref} decreases, then the keeping the transport capacity and decreasing the processor capacity is possible. If one also scales the basic execution speed, performance can be kept constant. The third case shows that one can increase granularity. This requires that:

1 the number of external references k_{ref} decreases more in each processor than the figures mentioned above. This means that the placement algorithm must have very good properties finding the locality.

2 the network shall use 2 or more dimensions to compensate for the loss of transport capacity.

3 the processor rate increases correspondingly. This means that the memory in the processor must increase both storage capacity and speed to the same extent. Thus, the requirement on the memory is considerably increased.

Of this reason, one understands that, during some favourable conditions, increasing the granularity may be possible. In the general case this is probably hard to do.

11.2 Architecture

11.2.1 Topology

The following relation states that the computer is transport limited:

$$\frac{2 \bullet k_{\text{link}}}{k_{\text{ref}} \bullet h \bullet pr} > \frac{t_{\text{msg}}}{t_{\text{exec}}} \qquad (28)$$

The number of links k_{link} per processor varies generally inside the range 1-10. Generally considerably fewer are used. Large amounts require many wires.

The number of references per expression *ref.* is approximately 1 in massive applications. In large nodes with coarse granularity k_{ref} can decrease in the pace with the size of the node. If localities cannot be obtained then the value of granularity approaches the value in the massive case. There the factor is heavily program dependent.

The load increase on the links h depends on the globality G and the dimension *dim*. The dimension strongly influences, where the values can be 1, 2 and 3. Larger values are hard to implement. For a globality of 1000, h becomes 1000, 31, and 10 respectively. This is the absolute largest influence that can be established in the relation.

The number of references per reference pr is protocol dependent. The only known value is 3.

To summarize, it can be concluded that the dimension of the network is crucial. Other parameters are of marginal interest. The massive case is transport limited. In a massive, 3-dimensional topology, with 1000 processors without a locality the left side is 0.4. Increasing the locality can raise the value 1-5 times.

11.2.2 Placement, duplication and thread algorithms

The placement algorithm affects performance by evenly distributing the executions over the processors. At the same time the load on links is to be reduced. These two features are almost independent of each other.

In an execution limited computer the only valid requirement is evenly to distribute the executions. Simple algorithms as random placement can do this with a good result.

In a transport limited computer, the transport load is to be reduced on the links. Random allocation creates no locality. Equation 25 describes how to do the optimization. h should be reduced, i.e. the locality will be improved. For the reason of the dependency being the third root (the network has dimension 3), the rate is only affected by a factor 2 if the globality is altered by a factor 8. This means that the placement function cannot contribute much. For dimensions less than 3, the placement function matters a lot.

The latency time is directly depending on the worst loaded unit. For that reason spreading the load is important. The concentration in certain nodes directly affects the latency time for execution. However such hot spots do not affect the link load that much. Such a load is integrated from h processors. Thus in a massive transport limited computer, the influence on the latency time is low. However, the influence on the storage volume may be high.

The thread algorithm does not directly influence the latency time. In a storage limited computer the lifetime of threads should be as short as possible. The lifetime is proportional to the thread depth. The q_{thr} of the thread algorithm therefore directly controls the latency time. No extreme variations are to be expected.

If one summarizes this one find that the discussed algorithms should be dimensioned to do a decent job. The influence on performance cannot become dramatic.

11.2.3 The storage volume

The storage volume depends on the lifetime and the thread depth. The first one depends on the load increase h, which depends on the globality and the dimension. The other one only depends on the quality of the thread algorithm. Low latency time in the network and god locality are desirable. The locality dependence is not large (see paragraph 11.2.2 on page 25).

The storage volume increases especially when the execution frequency f approaches the maximal frequency. The pace of an increase is dramatic when the load of the network or the processors is approaching 1. For

that reason the loads ρ is to be kept much lower than 1. A value of <0.5 is probably preferably.

11.3 Implementation

The implementation in VLSI only affects the basic time delays. The costs are not discussed here. Increased granularity requires more powerful processors. This leads to the requirement of very powerful memories.

11.4 Program

The size of the execution graph V does directly influence the memory volume and the execution capacity.

Many frequently used expressions require duplication. The duplication k_{dub} sets the additional demand of storage.

The globality G influences the transports and by that the speed. In a 3-dimensional topology, the globality is only affected by the third root, i.e. relatively weak.

The average number of references to an expression k_{ref} directly affects the transport rate. Generally the variations of k_{ref} are relatively small for massive programs. However, at large granularity, considerable differences may occur.

12 IMPLEMENTATIONS

One major feature of computer architecture has been the technique to mix different implementations, in a way obtaining optimal price and performance. Therefore a few examples on different implementations will be studied, seen from the perspective of theory.

12.1 Different price/performance

For the basic units hold that

1. memories have always been used with different characteristics. The reason for this is that there are very fast, but expensive memories, and large, but cheap memories. A proper mixture results in very good price and performance for computers.

2. there are several different communication links. They have different latency times and transport capacities. Cost and distance control the selections. Generally the selection of a link has been a consequence of another dimensioning.

3. processors are available with different performances. Generally they are selected for a certain application. Many different processors are generally not included in a single application.

The theory is general. Therefore we can adjust it so all basic times such as t_{exec}, t_{msg} and t_{flit} can be related to different units and have different values. Then the analysis of performance becomes more complicated, but does not influence the comprehension of the mechanisms.

The placement function, however, becomes important. A proper mixture of placement must be done to reduce latency time. Normally, this means that fast units should be heavily loaded. This may, but not always, lead to an increase of local storage volume. Different topologies with varying units may be proper. However, this is beyond the objectives of this report to discuss such matters.

12.2 Insufficient capacity

There are also possibilities to implement the different basic units with insufficient features. This means that they are unable to carry out the function they are intended for. Some common such features are:

1. the memory is considered to consist of several levels. The addresses for the memory are in some way permuted between the levels. The single memories can seem to lack the ability to store all expressions. Addressing must sometimes access other memories.

   This structure can be modelled as a network of memories. Some placement algorithm places the expressions into a proper memory. This can be done dynamically or planned.

2. the processors are separated into memories with memory cells and controlled arithmetic units. Therefore the memory cells are very simple processors, which can only be read and written. The actual processors have a few registers for storage, but are otherwise full-fledged.

   This structure can be modelled using processors for memory cells and for the actual processors. Between them is a network, which is the memory bus. Many possible structures exist.

3. the processors are not completely implemented. General purpose and more specialized processors exist. All expressions can be executed in the general purpose processors. Specialized processors must move certain executions to other processors.

   Here are many implementations used in today's processors, such as a) instruction decoders: this is a special processor, which cannot execute the expressions but distributes them to other processors, and b) many executions are serialized and placed to be executed in a pipeline.

4. the network cannot communicate among all nodes. Some examples are hierarchical networks, pipelines and dedicated networks. These can implement arbitrary algorithms if properly allocated.

Generally speaking, the insufficient capacity raises demand for relocation. Therefore, extra transports are done. The placement algorithm must place expressions in a way making desired transports always executable. For performance this means that more transports are carried out and by that the links are more loaded, and more links are traversed on the threads, increasing the average depth of a thread. The consequence is longer latency time and greater demand for storage, respectively.

Normally the insufficient capacity is used as a way reducing the cost and increasing the performance. To make this feasiblely, the network is enriched with extra parts, often in completely new dimensions, and extra storage capacity. The extra networks can be specialized, as for the examples instruction decoder and pipeline above.

The placement must be done so communication is always possible. Storage volume must be available locally. If this is fulfilled then the general theory, discussed in this report, is valid.

12.3 Insufficient resources

In this report free storage space for all expressions has been anticipated. If this is not true then the conditions for the theory are not fulfilled.

Lack of storage space can imply deadlock or starvation. These problems are normally taken care of on a different level, and are thus ignored here. However, the local lack of memory must be solved. Mainly, two solutions exist:

- the expression, which is to be stored, is placed on another location than planned. This implies extra communication and to some extent also extra executions.
- the writing is postponed in time until a suitable space is free. This implies a changed order of execution.

As long as these two measures only occur to a small extent, they affect the theory marginally.

12.4 Dynamical reconfiguration

The placement algorithm places expressions in processors. Because execution is a dynamical course of events, periods exist when different optimal placements could be used. It is possible that different parts of the execution graph have different courses of time and need different placements. It is also possible that these discrete changes in placement can grow into a continuous course of events.

What these courses of events looks like are not being studied here. Solely assume that there is a placement algorithm varying dynamically. In the theory discussed earlier, the result of an execution has always been placed locally. A switch of placement algorithm causes transportations. After that it works as in the general case.

These transportations increase the load on the network[1]. There are some special cases to annotate:

- when transporting from a slow memory to a fast memory, at least one access of the slow memory and two of the fast memory are required. If only one memory access is to be done, there is no use of moving the expression.
- cache memories move blocks containing several expressions. To benefit, such a block must be accessed many times. Modern programming writes to each memory cell only once, but can read from them several times. Either an expression must be accessed many times or several expressions must be accessed in the particular block. With gains in performance up to 100 times, it is unrealistic that a single expression is accessed that many times. Instead it is the case that many expressions are being accessed.

  This requires the placement algorithm to be able to find a locality for a block, to make it contain many parts being accessed[2].

13 SUMMARY

The theory presented here, showed that under certain circumstances, arbitrarily large multiprocessors can be used to execute arbitrary programs. Good performance can always be obtained. Programs are not required to be adjusted to the architecture. Performance depends on the program and the dimensioning of the architecture and implementation.

---

1. common cache memories can be assumed to have dynamical placement functions. The transport between cache memory and primary memory is the very mentioned transport.

2. if it is the case that the single expressions are accessed only a few times, accesses to the cache memory and the primary memory are of the same magnitude. Conventional processors use extra communication bandwidth to create a shorter access time, because performance is directly depending on the access time. Therefore, this can be accepted. In a massive multiprocessor such an access time is of no significance!

The VLSI implementation only affects the rate in proportion to basic transport and execution times.

Using built-in functions, duplication and controlling threads for placement is possible. The features for these functions influence performance, but are not critical.

The conditions mentioned above are the possibilities to implement the more basic functions such as routing/addressing, low-level protocols and language interpreters.

The results of the theory can be described using several measures. These depend on the program and the architecture. Several additional measures depending on the quality of the implementation are also used.

The program related measures are the execution frequency *f*, the size of the execution graph *V*, the depth of the graph *D*, the number of references to an expression *ref.*, the duplication factor k_{dup}, the branching factor in the graph β and the globality *G*.

The architecture related measures are the number of processors *NGr.*, the number of links per processor k_{link}, the dimension of the network *dim* and the number of messages per reference *pr.*

The quality related measures are the uniformity of the distribution of expressions on the processors q_{dup} caused by the duplication algorithm, the globality q_G caused by the placement algorithm and routing algorithm, and the thread depth q_{thr} caused by the thread mechanism.

The implementation related measures are the time of an execution t_{exec}, the time of transporting a message over several links t_{msg} and t_{flit}.

The theory can be used to analyse more special architectures, among other things architectures with units of different rates and functionalities.

References

Burton 94 F Warren Burton, J Rayward-Smith, *Worst case scheduling for parallel functional programs*, Journal of Functional Programming, Vol 4, No 1, pp 65-75, (1994)

Eager 89 Derek L Eager, John Zahorjan, Edward D Lazowska, *Speedup versus efficiency in parallel systems*, IEEE Transaction on Computers, Vol 38, No 3, pp 408-423

Hey 90 Antony J G Hey, *Experiments in MIMD parallelism*, Future Generation Computer Systems, Vol 6, No 3, pp 185-196, (1990)

Shahookar 91 K Shahookar, P Mazumder, *VLSI Cell placement techniques*, ACM Computing Surveys, Vol 23, No 2, pp143-220, (1991)

Symbols

Symbol	Meaning
D	the mean depth of the execution graph
D_{thr}	the mean depth of a thread
G	the globality of a reference. Corresponds to the number of expressions to execute within the scope of an access
f	the controlled execution frequency
f_{pr}	the maximal execution frequency
f_{link}	the maximal transport frequency
h	the number of additional hops in a transport in a network
k_{dup}	the multiplication constant due to duplication
k_{link}	the number of links per processor
k_{ref}	the number of references per execution in the execution graph
L	the edge length of the allocated scope
N_{pr}	the number of processors
N_{thr}	the number of open threads
q_{alloc}	the ratio between the maximal number and the mean number of executions, averaged among all processors
q_G	the ratio between actual and theoretic globality G for a particular program
q_{thr}	the ratio between actual and theoretical thread depth
pr	the number of messages transported on a link for one single reference
t_{expr}	the latency execution time for a single execution. The time includes waiting time.
t_{exec}	the basic execution time for a processor
t_{flit}	the transport cycle time
t_L	the latency time for the application
t_{msg}	the base time delay for a transport. It includes routing and transport of the first flit.
t_{ref}	the latency time for a reference. The time includes waiting time.
t_{mass}	the latency time for a massive application, i.e. when the dependencies are ignored
V	the volume of all executions in the execution graph.
V_{pr}	the maximum volume of executions allocated to a single processor
V_{in}, V_{out}, V_{prog}, V_{state}	the volume of expressions for the input and output queue, the program and the state
τ_p	the life time of a thread
β	the branching factor for a graph
ρ	the load on a unit
ρ_{pr}, ρ_{link}	the load on a single processor and link, respectively

The author

Gunnar Carlstedt has 25 years of experience in computer architecture. Most of his time has been devoted to research and development in this field, on consultancy bases, mainly to the Swedish industry, but also to other European and US based companies. He has designed 23 different computer architectures, of which about every second have been produced. Among them, computers have been designed for Ericsson APZ telephone switching systems, the Swedish Defence air crafts JA37 Viggen and JAS 39 Gripen.

Among his interests were hardware-near languages, including various general-purpose languages and graphical languages of imperative and applicative type. His main concern has been the semantics and its implementation.

His interests also include the processor micro-architecture, its formal definition, way of mapping language interpreters onto this structure and the performance estimating of such interpreters. He probably designed the first RISC processor the year 1974.

His interests also include the hardware components of processors as memories, buses and arithmetic units of various kind. He has a special interest in associative memories.

His interests also include the methods for synthesis of computer architecture and processors, including hardware description languages, automatic synthesis on high and low level, and special silicon compilers.

He was very early (1975) interested in distributed parallel computer systems, with an emphasis on languages, problem allocation, protocols, memory systems, and computer topology and this interest remains.

Gunnar Carlstedt has also been researching the Swedish JAS 39 Gripen project on the behalf of the board to find the performance of the programs, computers, tests, tools and project.