Interaction and Advances
High Performance Computing in Embedded Systems: From industrial requirements to practicable concepts

Gerhard Peise

Teichstr. 4, D-52224 Stolberg, Germany

Abstract

HPC is required for the next generation of industrial production and products. The application areas and the requirements in I/O and processing are shown. Parallel processing is the cost effective solution, so the concepts and methods used for specification and design of parallel systems are critically reviewed. A communication architecture for industrial real-time applications will be described.

1. HPC in industry?

1.1 State of the art and markets of the near future

Today information technology is able to deliver super computer power almost as a commodity. This is not happening the expected way like it is in science, where super-computers are used. Microprocessor technology used in parallel and distributed applications generates the processing power of current super-computers and is scalable and adaptable to solve problems in an economical way.

Cost-effectiveness is one pre-condition for industrial use. The second is time-to-market. Since problems tend to become more complex, appropriate methods and tools are required to develop adequate and marketable solutions. Parts of the industry is used to work with multi-processor systems for solving the problems, so the basic know-how is available.

When these two pre-conditions are met the industry can address potentially existing and new market niches. The next chapter describes a few examples of new applications. All these are real-time applications. Real-time does not mean fast, but right and in time. This 'right in time' is different for different applications.

1.2 Examples from industrial research targeting the near future products

1.2.1 Intelligent vehicles

Here cars, trucks, trains, planes, helicopters, space vehicles and servicing robots are addressed. This includes in the near future driver support, safety (e.g. distance keeping), improved engine management. In the long term more autonomy is given to the vehicles so that finally these systems have to navigate and operate in known and unknown environments, co-operate with other autonomous systems and co-operate with hierarchical guidance systems (e.g. global knowledge data bases). They operate 'intelligent' by individual recognition, classification, decision making, planning and learning. The estimated processing power is medium term 100 GFLOPS and 1 TFLOPS in the long range. The I/O bandwidth has to scalable up to 10 GByte/s.

Example: The projects PROMETHEUS and TRAFFONIC

Objective: Driver support in cars

Functionality: lane keeping
obstacle detection and avoidance
traffic sign recognition and speed control
distance keeping
automated driving

Status:

First experiments required a bus for the electronic equipment. The second generation fit into a small transporter. A completely transputer based version required only a van. Today's PowerPC version is in a small box in the Mercedes S-class car.

In begin of 1993 the transporter was cruising autonomously on a normally frequented highway for approx. 200 km from Stuttgart to Ulm and back. The operation included overtaking of trucks, distance keeping lane keeping while passing exits. This year the van was cruising from Stuttgart to Berlin (approx. 500 km) without driver interaction. The maximum cruising speed was 160 km/h (100 mph).

Short term system requirements for practical use in a car: 5 GFLOPS in 5 litre volume running at 500 Watt.

Example: Hovering support in helicopters

Hovering is required in rescue situations. It is an unstable operation mode of a helicopter and requires highest skill of the pilot. Beside this the effects of wind have to be compensated. Vision and radar processing will be used to control the system

A prototype system had a successful test flight this year. It automatically followed a truck which was driving in circles.

1.2.2 Transport management

Examples are automatic railway systems or road traffic management. For the railways this includes train protection (e.g. emergency braking), train operation, supervision of the network and interlocking. This requires real-time operation with reaction times of less than 0.1 s, fault tolerance and availability of more than 99,9%. The network size ranges up to 10 km, scalable and maintainable (MTTR < 1h), where a self diagnosis identifies the failed components. The road traffic management will lead to better use of existing roads and optimisation of traffic flow using visual monitor techniques.

Example: Car plate recognition for restricted access areas or automatic highway toll systems

The number plate is read and the car identified for access to restricted areas, parking areas or tunnels. In automatic toll systems the cars have to be identified, the number plate has to be located and read even when the car is driving at 200 km/h and the next car is following at a distance of 5 m. Rain and fog are limiting factors.

1.2.3 Quality control in industrial production

The control of prints on products has to cope with the speed of the production, while the quality of the recognition has to be compared with perception of a customer looking at the product. This requires high resolution (e.g. 5k x 5k pixel in cameras with shutter speed in the order of milliseconds. Standardised pre-processing and data reduction has to be integrated into the camera while the intelligent part will be in an external system. Today's prototypes have approx. 1 GFLOPS.

Example: Quality Control for Textile Production:

During its production textile is checked up to 5 times. Defects from weaving and dying have to be detected and marked. An experienced human inspector can inspect up to 10 cm/s. The targeted system shall operate at 1 m/s with a resolution of 0,1 mm at a 1.4 m wide textiles. The system requirements are: data distribution of 140 MB/s, pre-processing of 10 to 20 GOPS, intelligent post processing of a few 100 MFLOPS to one GFLOP.

1.2.4 Network based video service and video on demand

These systems operate at a data rate of 2 Mbit/s to the subscriber, 200 to 10,000 users supported, 1000 to 10,000 hours of video stored with 1 hour video is approx. 1 GByte of data. Each user should have short access time to his video.

1.2.5 High power generation and control

To monitor and control high voltage substations the following functions are required: confine transient failures, fast time response for recovery and restart, redundancy. the time constants for sampling rates of I/O validation are in the order of few milliseconds to 100 microseconds. Time redundancy for I/O (repeated reading of a value, up to 6 times before accepting) and a number of I/Os in the order of 2000 to 3000 is required.

1.2.6 Vision simulation

On the simulator market a quality standard of the visual animation is required to be close to photo realistic images. The requirement can be met by the use of texture mapping, which results in high communication bandwidth demands. The display operates in true colour mode at a resolution of 1280 x 1024. Each second 60 new frames have to be generated and the latency has to be less than 80 ms to avoid simulator sickness. The bandwidth to the display is 320 MB/s, texture data has to be supplied at 400 to 600 MB/s. The RAM store for the texture data requires a few 100 MByte. The intermediate amount of object and shadow data is 150 to 200 MB/s. To process the information a performance of a few GFLOPS is required.

1.2.7 OCR for check and voucher processing

A total of 210,000,000 transactions of international eurocheques and credit card vouchers have been processed in 1993 in the central German clearing agency, - that are 0,5 million per day or 6,6 per second, non stop. This will increase significantly in the near future. The main task is to read hand written numbers. the future requirements are: credit card on-line authorisation, intelligent systems for fraud prevention, 'learning' of the system 'on the job'.

1.2.8 Postal automation (letter sorting)

Postal services deal essentially with sorting and distribution of mail. Standard letter sorting machines, which are already successfully used all over the world, process printed addresses at a typical rate of 10 envelopes per second. The basic functionality of the system is the automatic reading of the addresses. On each letter the following tasks have to be performed: scanning the envelope, finding the address field, reading the address, understanding the address, verifying the address by on-line dictionary look-up and printing the bar code.

In the future it will be important to develop recognition systems which can integrate human reading capabilities especially when applied to complex addresses (reading of hand written addresses). For this challenging task the following HPC requirements are anticipated: data distribution at 240 MB/s, scalable general purpose processing of up to 5 GFLOPS and beyond, incorporation of special hardware and on-line RAM store of >> 1 GByte.

A prototype of an address reader for letter size envelopes consists of 2 racks, 250 T800 transputers, 2 GByte RAM and has 80 MByte of executable code. In a field test the average throughput over many hours of operation was measured as 25,000 envelopes/h. The target throughput is 90,000 envelopes/h. This requires a 1000 transputer system.

An alternative solution with the PowerPC MPC604 fits into 1 crates, uses 20 PowerPCs with in total 320 MByte of RAM (which is not sufficient) and operates also at 25,000 envelopes/h.

Flats (magazines) are larger and have frequently pictures on the front page. This requires more processing power. Also sorting of parcels is more complex. There are six surfaces which have to be scanned for the address.

Nevertheless, to meet the computational requirements in all these application examples parallel processing has to be used.

2. Parallel processing, what is that?

2.1 Parallel processing in everyday life

We are so used to parallel processing that we are not aware of it any more. Some examples will show parallel processing in everyday's life:

One of the oldest examples of parallel processing is theatre. A story book is a set of communicating sequential processes, or a parallel program. It describes a real time application with several processes, synchronised by communication, verbal and non-verbal. These systems are deadlock-free.

Even in ancient times wars were indeterministic real-time activities, where many people (processes) were active on both sides. The commander-in-chief (programmer or operator) tried to manage the situation by the use of commands (communication). Here we find a natural selection: the better or more powerful programmer wins.

The planning of a production procedure is managing parallel processes by planning the two parameters resource and time. A production process is a deterministic parallel real-time application: functional units are assembled simultaneously in different places, they are transported to their destinations and at the end they are assembled to build the final product. Indeterministic producing companies are meanwhile bankrupt. The planning of this production process is nothing more than planning of material flow and resources for the assembly procedure. It resembles a data flow machine where data is replaced by components.

Project management uses parallel technologies. Resources have to used in an optimal way, the total project time has to be minimised, interactions and interdependencies are known in

advance. The complete project is optimised. The critical path method (CPM) helps to plan the activities and control the progress.

Soccer is a clear parallel application, and most people manage to play. Tennis is a clear sequential sport.

Beside these examples: the biggest machine on planet earth is the telephone network. It is a pure communication system which follows a very simple rule: local calls are cheap, long distance calls are expensive.

What can we learn from those examples for our problem: practical parallel processing.

3. Parallel Processing, communication and management

3.1 Marketing first: what is 'Massively Parallel'?

Today people live in an environment of superlatives which describe the banalities of every days life. We are used to talk about superstars, megastars, etc. With this way of thinking we look at our own environment and ask: what is next? More than parallel? Paralleler? Real parallel? So how to market the next result of research? It's massively parallel! Whatever that is, this word will get a meaning by its use. My interpretation as follows:

In the 70´s we programmed the mini computers, in most cases in assembler, due to restricted memory and availability of other languages. The programmer developed a personal relation to his work, his program. He knew each variable and each procedure call. This changed when we got more memory and started to use high level languages: we reached a higher level of abstraction.

In today's method of parallel processing the programmer is in a similar situation: He has a very limited amount of processors and low functionality of tools, - and due to this he develops a personal relation to each process and processor. The programmer is not forced to rethink his method of working because he has hardly an other chance. And with his intelligence he is even able to deal with higher degrees of freedom: using more than one processor. The programming technology is not very different from that years ago: mastering the complexity with assembler mentality.

What finally is required is something I would like to call 'Pascal-mentality', that means a higher level of abstraction which handles functionalities: processors like memory cells. The result of this is, that each individual processor becomes more or less meaningless, like an individual ant or bee is meaningless, but integrated in a community with the right method of information exchange they become very powerful. These are real massively parallel applications.

In terms of information technology this means: the communication system is the key element, and this is valid for nature as well as for computer science.

3.2 Today's use of communication and the results:

Communication is the new element, in software and in hardware. When we have to deal with something we do not completely understand we frequently ask for a maximum of freedom and do not accept any restriction, just for our own security. This results in the request for communication of each to any. In small systems it could make sense, but limits scalability. For bigger systems it has some curious effects:

1) direct connections between all processors

n processors have n*(n-1)/2 connections

each cable shall have a length of 1 m

	price	weight
node:	1000 ECU	1000 g
cable:	1 ECU	20 g

Assume a 1000 processor system:

- each processor has 999 connectors (board space?)
- there are 1000*999/2 m = 499,5 km cable for communication
- cable weight 9,99 t
- processor weight 1 t
- cable costs 499,5 kECU
- processor costs 1000 kECU

Who is willing to pay for this flexibility?

How to maintain 500 km of cable in a big box?

2) a central communication switch

- each processor communicates with 10 MB/s
- bandwidth in the central switch is n/2 x 10 MB/s
- with 1000 processors we need 5 GB/s

If the technology is available to build this switch the same technology would speed up the processor too. The gap in bandwidth is still there.

So far the hardware. But the software is not much better off. There are also curious effects.

The programmer (as the master of complexity) dealt with many algorithms and was finally always successful. So the new feature 'communication' will be easily integrated. The way is: divide and conquer. Communication is reduced to its minimum: I/O. The experience from procedure calls is that well defined protocols can help significantly. So use it. One does not know what will happen - and does not know what is needed, so basically no restriction in communication is accepted. The experience from a few weeks of work helped to identify what is believed to be the basic laws of parallel processing:

- communicating parallel systems are indeterministic
- we need automatic parallelising tools

Great! - But this is simply nonsense. Here the programmer acts like a physicist, who tries to understand a natural system, e.g. the universe. The programmer or the computer scientist tries to explore a computer system: an artificial, man made system, - in most cases even made by himself.

So indeterministic behaviour is not a law of nature, but a result of an incomplete specification. It is a function of the individual implementation, and due to this it is completely within the responsibility of the programmer.

As an example: the ALT construct as commonly accepted indeterministic function can be used in a deterministic way. In theory, the ALT is indeterministic when we do not know what will happen. This is the bottom-up view. First was the computer, and now we look for the appropriate problems. In practice we have a real problem, we solve it by using appropriate tools like programming languages and computers. Their flaws and bad features are neutralised by programming. Indeterminism only occurs when the programmer is not aware

or does not care what will happen in different situations. But these problems have to solved anyhow. So why not planning it from the early beginning?

Now let's have to look at automatic parallelising tools. The basic idea is that these tools take a sequential program and make it parallel. This is similar to vectorising compilers which take code and look for things to vectorise, like loops etc. This is possible because within this code everything is completely specified. The vectoriser operates on instruction level.

Parallel machines operate on process level. e.g. each processor has a small and complete program which communicates with other programs. To parallelise a sequential program it is necessary to understand what is going on there, on what kind of information the processing depends and how much time it needs. Quite frequently even the author of the program is not able to specify this. I think today due to the lack of this information a tool can hardly do this work, and if, hardly with acceptable efficiency.

The main reason for this is that most applications are inherently parallel. As programming we understand to serialise a parallel task in a way, that it can be executed efficiently on a von-Neumann architecture. We are used to this way of thinking that we do no reflect consciously what we are doing, it is more a reflex, a function of the spine. After spending all the effort to serialise a task we expect to have a tool which parallelises it again, e.g. makes our previous work obsolete. Additionally this tool has to integrate the exchange of information: communication. But what is communication at all?

3.3 What is communication?

To understand the functional requirements for a communication system of a parallel machine we have to understand what is communication and why it is used.

The expression is taken from our everyday life and describes the standard method of our personal interaction. It is used for pure information transfer as well as to initialise or synchronise activities.

So what types of communication do exist in human interactions? The following but incomplete list gives some examples:

type	goal
1) discussion	directed message for immediate reaction and interaction
2) talking to the people	directed message without immediate feedback (politics, advertising, lecture)
3) law	directed message without feedback, but with control and sanctions
4) somebody talks and nobody listens	undirected message, end in itself

Which of these communication methods are useful for parallel processing and distributed systems? The reason for information transfer is that this information can not be processed at this place in time or that the message shall initialise other activities. The following table describes three communication methods used today on computers:

type	goal	result
a) point-to-point synchronising	exchange of information	organising, synchron. of processes
b) point-to-point without synchronisation	transfer of information	indeterministic, acknowledge unimportant
c) broadcast	information to all	no acknowledge, no feedback

Let's again compare the communication structure of big parallel systems with communication in our human society. In case of discussions we try to come close together. That is easy to understand when looking at a situation with many independent conversations taking place in one room. Talking to distant people disturbs the others. People group together.

The same effect we have using communication on parallel machines. Long distance communication has to be handled by intermediate partners or processors and uses their resources, e.g. disturbs their work. Is it not possible to arrange the communicating partners in a way we naturally do it in our everyday life?

Planning the flow of a parallel program is primarily planning the communication. This is the central task when developing parallel software. Communication has to be planned like a working plan or a production schedule for a producing company. The only difference is: the product is information.

On a computer a software engineer can implement a program in a way, which, compared with a production schedule, would ruin the producing company. But many programmers ask for exactly this freedom to be able to do their job.

In general we can say the following for parallel systems:

- communication synchronises the processes
- communication between neighbours is of importance
- global and random communication is only of theoretical interest and rarely exists in practice, only in badly planned and inefficiently operating programs
- load balancing is meaningless for real-time applications because the real-time programmer is the only one who has to know in advance what he is doing

Up to now it was a description of communication, - but how to use it? We have seen there are examples in our everyday life. The only problem is: people who do 'parallel processing' since centuries can't program, and the programmers don't know their methods.

What can we learn from these different fields for our problem: parallel programming?

3.4 Planning of communication and interaction in parallel systems

The key word is planning. And planning means understanding in advance. This might be new for many programmers, but it is the same as the planning of the execution of an algorithm (first * and /, then + and -). We are used to do this, because we have learned it at school, and we knew it before we started to learn how to program. Handling the execution of processes, their communication and their interdependencies brings a new parameter into the game: time. To deal with this is common use and called real-time programming. So the integration of communication in standard software integrates real-time features into the program. This has to be recognised by the programmer. A good method to deal with these new features can be found in the techniques of project management.

3.5 Project management and parallel programming

What is the difference between project management and parallel programming? Not so much. Both are multiple resource problems. Management uses old techniques to organise resources and their interactions to solve a problem. This is the top-down view. Parallel programming is believed to be a new technique, but it is finally the same. The only difference is that most programmers have a bottom-up view, concentrate on individual resources and later on establish their interactions.

Let's look at a factory and see what we can learn for parallel programming: There are many tasks to be executed. The duration depends on the resources spent, e.g. people working on it. If it lasts too long we have to put more people on to the job, as long as it makes sense. One job depends on results of an other. It cannot start before the previous job terminates or produces results. Results might be transported from one place to an other, e.g. the engines for a car are produced in a different place than the final assembly takes place. There is a clear interdependency. The describing parameters are resources and time.

To plan this each task is estimated in terms of load and duration. Interdependencies are identified. The flow of material is specified in terms of time and quantity. All this is information is put into a bar chart which represents the logical flow of work as a function of time. These charts are called Gantt-Chart (H. Gantt, 1861-1919).

In more complex and parallel situations we see that a series of tasks is active all the time while others are waiting for the results. This series of tasks is called the critical path. Other tasks are waiting for results from others before they can proceed. This waiting time is called free buffer time. Optimising the total job means:

- identification the critical path and the tasks with free buffer time
- change of resources to minimise the free buffer time

This critical path method (CPM) is an excellent planning method to describe parallel activities like production or programs (which are basically the same). The programmer should be able to estimate the execution time of a process as well as the load and duration of communication. This information is a complete description of the communication structure of the system, the communication architecture.

3.6 Quality, - or design for testability

Debugging a parallel system influences the system behaviour because it has effects on execution time and communication load. Since there is no interaction-free debugger we have to plan debugging as well as processing. This means that debugging has to be part of the specification of a parallel system. It has to be interleaved with the communication in a way that debugging messages will change the timing of the program flow, but not the functionality.

The principal integration of debugging is shown in the following example:

Basic function: communication is synchronising

message 1: sender: announcement of communication to receiver
sender waits
time for debugging messages from sender

answer 1: receiver: acknowledge
receiver waits

message 2: sender: information transfer to receiver
sender waits
time for debugging messages from receiver

answer 2: receiver: acknowledge
both processes continue

4. Practical considerations

An architectural concept for a scalable real-time system for industrial applications has to cope with scalability in communication and processing, cost efficiency, short turn-around times. So the whole process of design has to be addressed: from specification and simulation to implementation and validation. This covers hardware, system software and design tools.

This concept was developed for parallel real-time applications in industrial environments. It is based on real market and application requirements and implemented within the ESPRIT project 6290 HAMLET.

4.1 The Hardware and Architecture

The key element in parallel processing is the handling of communication. Communication has to be the base of a processor concept, not an add-on. Until now there were three processors with integrated communication brought to the market:

Inmos: transputer
Texas Instruments: C40
Intel: iWarp[1]

The most advanced communication concept is definitely in the T9000 transputer (3rd generation transputer) with the C104 communication switch. It is enhanced with the PowerPC for processing and an communication backbone for broadcast and data gathering.

[1]We understand that the production of the Intel iWarp processor has been cancelled

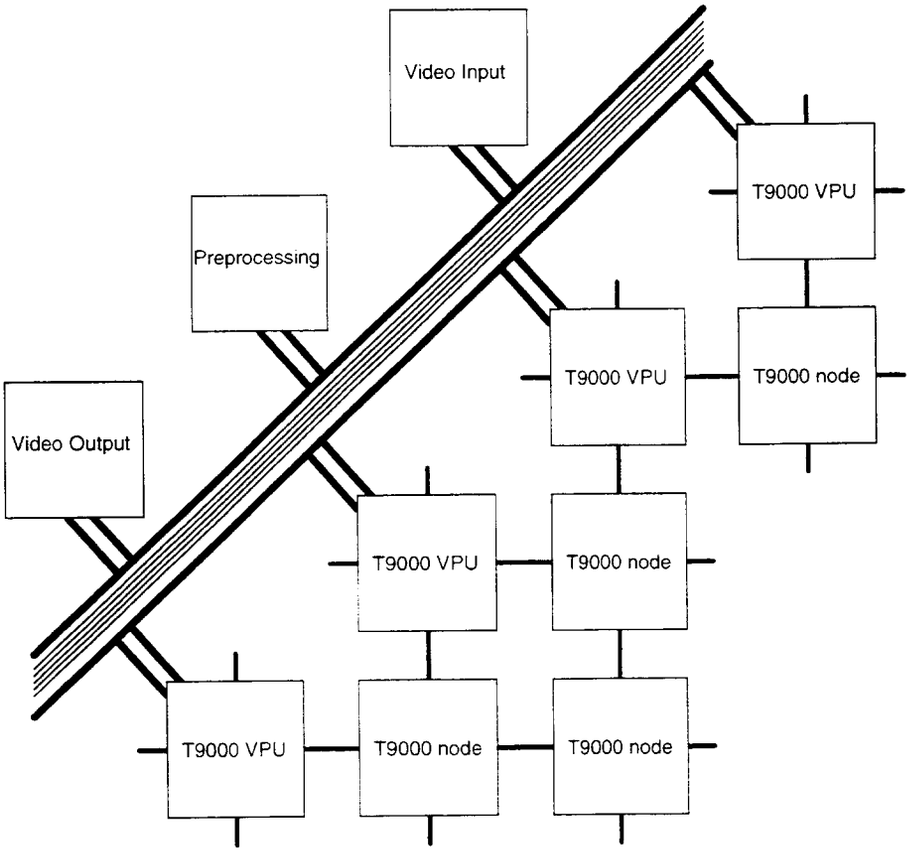

Fig. 1: The communication architecture

The communication backbone is a data distribution system operating at 300 MB/s. Data is transferred from the sources directly into the memory of the target components. These can be dedicated functions (convolvers, look-up tables) or the memory of standard processors (T9000 transputer) for further data reduction or post processing. The data transfer can be point to point, one to many, one to all or many to one. Communication between processsing nodes is point to point using the transputer links.

The operation of the communication backbone is synchronous. The access to the backbone is described in tables containing information on sender, receiver(s), length of transfer and channel number. These lists are executed in a round robin way.

4.2 The Development Environment:

4.2.1 A Concept for the Planning of Real-time Programs

The environment for the development of real-time programs should bring the programmer in a position where he/she can completely describe and specify the behaviour of the final system. This includes response time, execution time, load, data rates, etc. This specification (the design) is the only creative part of the job. The final implementation is limited by these early design decisions. If basic decisions have to be taken at the implementation phase the specification was incomplete in the first place.

The external parameters for the application program are defined in the requirement profile. The internal structure of the solution is the result of the system design and may consist from some tens to hundreds of processes. After this design phase the programmer needs early feedback on the correctness of his assumptions. Usually this can only be given by testing the executable code, - after finalising the major part of the implementation.

To avoid this delay the HAMLET approach is to simulate the system based on the specification information. Since all behavioural parameters like load, time and delays are specified, this behavioural simulation will give information on whether the assumptions are right or not. Early in the design cycle a test on the correctness of the assumptions can be made. This is the first revision loop.

When these tests show, that the simulation with the internal parameters meets the external requirements the implementation can start. Each individual process, as described earlier with its behaviour in time and its communication, can be tested after coding to check whether the implementation of each individual process meets the specification. So a second revision loop will help to optimise the implementation, or should that not be feasible, the specifications have to be changed. All this is possible on individual processes or groups of processes without having a complete implementation of the system.

This method of developing parallel real-time programs adopted by HAMLET is shown in Fig. 2. The integrated design environment consists of the following tools:

- the design entry system: DES
- the specification simulator: HASTE
- the debugger and monitor: INQUEST
- the trace analysing tool for HASTE and INQUEST data: TATOO

Their use will significantly shorten the design and implementation time. It can also be used by project partners for an early estimate of the functionality and cost of a software job.

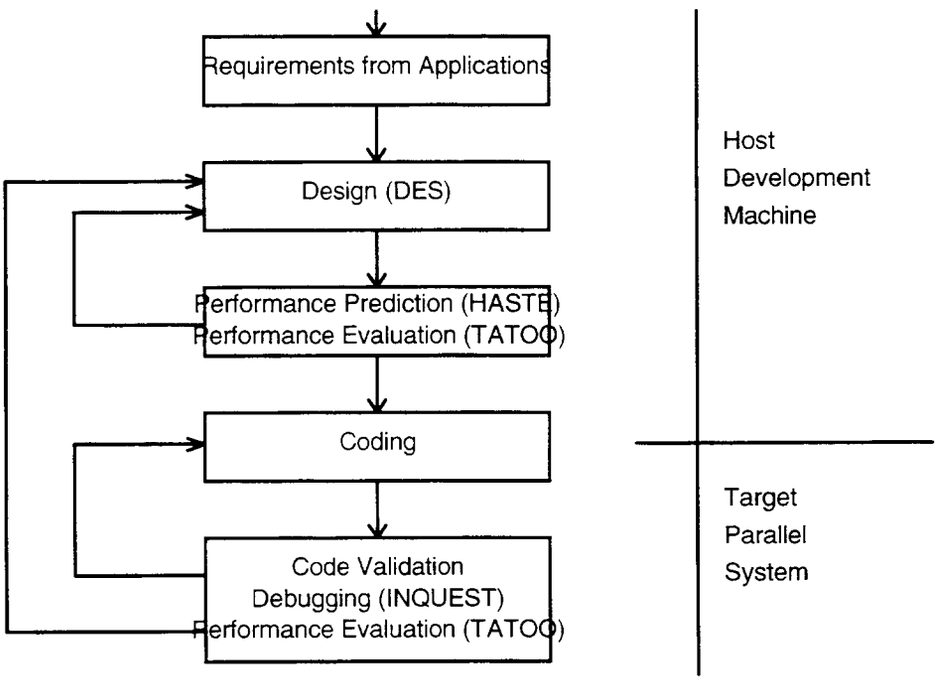

Fig. 2: Principle of the HAMLET application development system

4.2.2 The Design Entry System:

A design entry system is used for system *design,* that is planning and creation of the internal structure of software and hardware which will fulfil the external requirements. These requirements are described by parameters like data rates and timing. For the design additional information has to be given on the internal structure of the solution: how the task is split into processes and how these processes co-operate or communicate. Here also the parameters load (processing and bandwidth) and time (execution and communication time)

are used to describe this interaction. The result is a behavioural description of the system. When this description is correct the final coding should not lead to unexpected results.

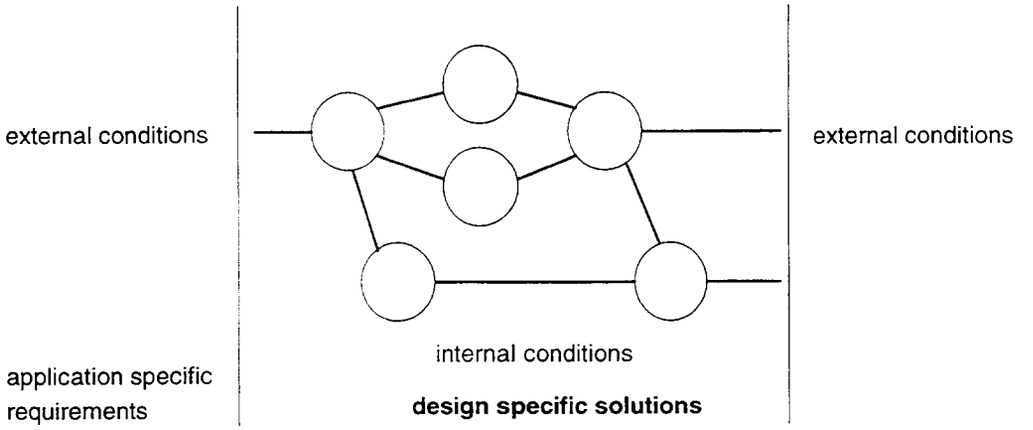

Fig. 3: external conditions / internal solutions

4.2.3 The Specification Simulator:

The structure developed in the DES with the parameters of load and time are fed into the HASTE simulator. Based on these parameters and the I/O requirements the operational behaviour of the planned system is simulated. There is no simulation or execution of application code. This early behavioural simulation gives a feedback on the real co-operation of the planned processes and their interaction with the external world. Traces are produced which can be analysed with the TATOO trace analysing tool. All this happens before the programmer starts with time consuming coding.

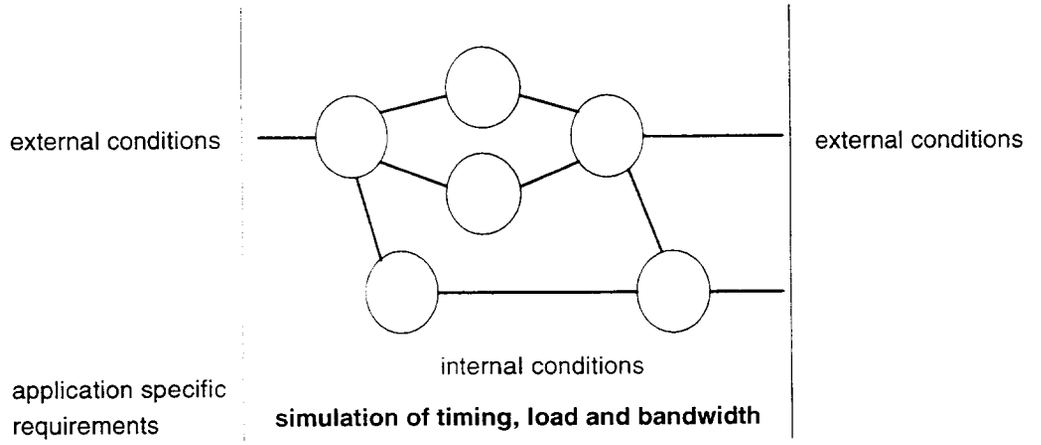

Fig. 4: Behavioural simulation based on specification data

4.2.4 The Debugger and Monitor:

The implemented code of individual processes or groups of processes is executed and the performance is measured. Traces are produced which analysed with the TATOO tool. A comparison of execution time, I/O behaviour etc. with the specification from the DES gives early feedback on how close the implementation is to the specification or how realistic the assumptions were. The debugger is specially designed for testing parallel programs.

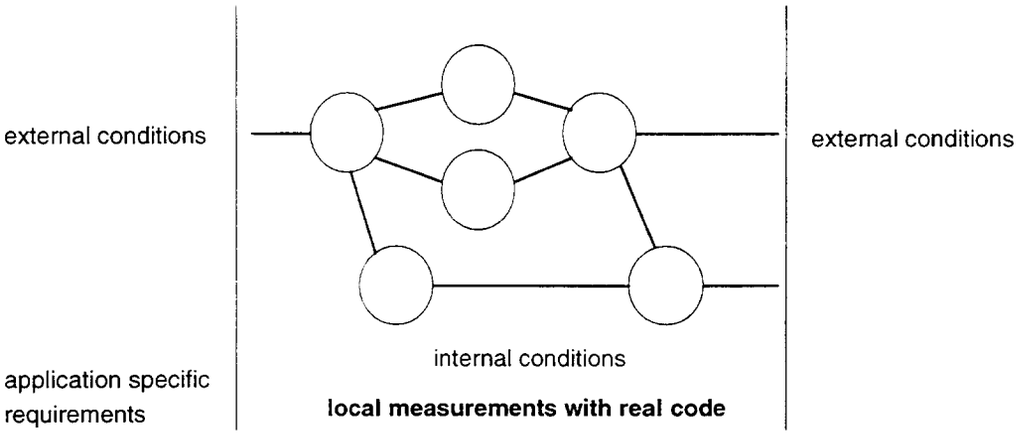

Fig. 5: Reference data from executed code

4.2.5 The Trace Analysis:

Here the results of the simulation as well as from real execution are visualised and can be compared. Data will be displayed as numbers, in bar charts, Gantt charts, colours for load representation, histograms, etc.

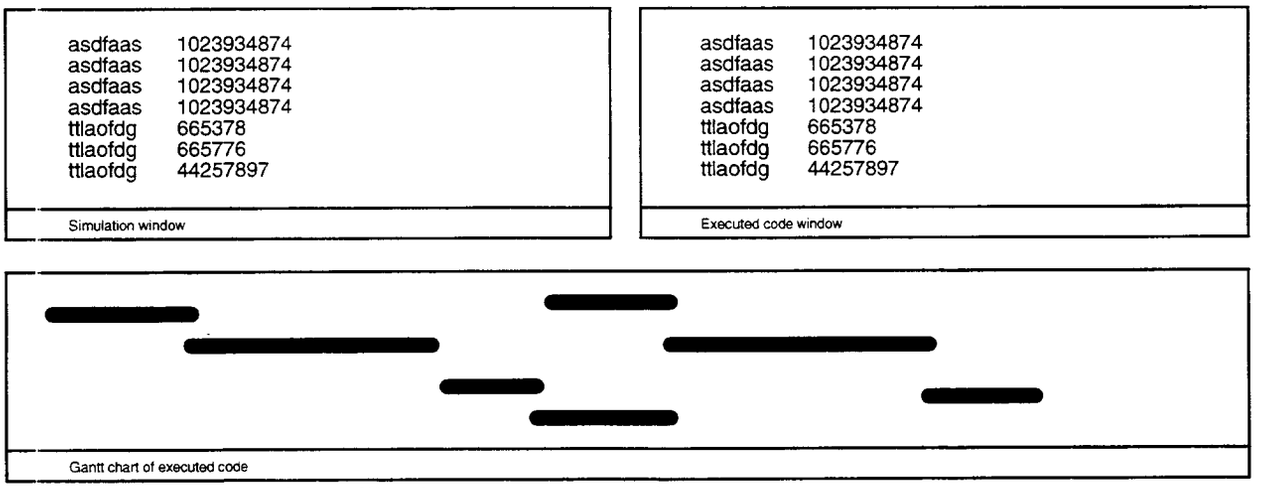

Fig. 6: Comparison of specifications with real results

4.3 Algorithms

Parallelising existing sequential algorithms could become a doubtful task. There is always more than one method to solve a problem. Most algorithms are selected because they produce in time good results on sequential machines. There is absolutely no reason that the same algorithms are useful for parallel execution. There might be other algorithms which give better or earlier results in parallel implementations. Parallelising algorithms means more than re-coding: it requires the understanding of the problem and the knowledge of different methods to solve it, not only modifying the existing solution by modifying instructions. The problem is not the computer, it is the implementation.

A good parallel implementation of an algorithm will have certain properties. One is scalability. This is required to be able to finally debug and use the program. Looking at the development cycle for parallel programs we see that we need algorithms which can be used the following way:

a) editing on one: 10^0 processors
b) debugging on few 10^1 processors
c) producing on many 10^n processors

The step from a) to b) is development work, the step from b) to c) can only be a specification. Scalability means the same program behaviour on one as on many processors. It has only an effect on execution time, not on the result.

5. Conclusion

This architectural concept solves the communication problem in industrial parallel real-time applications. It is scalable and cost-efficient. It is supported by a design entry system, a simulator, debugger and the languages C and occam.

Acknowledgement: The technical implementation of this concept is partly funded by the European Commission within the ESPRIT project 6290 HAMLET.