This paper appeared in: 7th International Parallel Processing Symposium, Proc. of 2nd Workshop on Heterogeneous Processing, pages 65–70, Newport Beach, CA, April 13–16, 1993.

Triton/1: A Massively-Parallel Mixed-Mode Computer Designed to Support High Level Languages

Christian G. Herter, Thomas M. Warschko, Walter F. Tichy, and Michael Philippsen
University of Karlsruhe, Dept. of Informatics
Postfach 6980, D-7500 Karlsruhe 1, Germany

Abstract

We present the architecture of Triton/1, a scalable, mixed-mode (SIMD/MIMD) parallel computer. The novel features of Triton/1 are:

- *Support for high-level, machine-independent programming languages;*
- *Fast SIMD/MIMD mode switching;*
- *Special hardware for barrier synchronization of multiple process groups;*
- *A self-routing, dead-lock-free perfect shuffle interconnect with latency hiding.*

The architecture is the outcome of an integrated design, in which a machine-independent programming language, optimizing compiler, and parallel computer were designed hand-in-hand.

1 Introduction

The goal of adequate programmability of parallel machines can best be achieved by tightly coupling the design of machine-independent programming languages, compilers, and parallel hardware. In the past, the development of parallel computers has been mainly driven by hardware considerations, without regard to the law of the weakest link:
The overall performance of a parallel computer is determined by the performance of its slowest part with respect to the requirements of the software.
Ignoring software requirements has resulted in unsatisfactory performance of these machines on machine-independent parallel programs. To avoid these shortcomings and to show that high-level parallel programming does not necessarily lead to poor performance, we specifically analyzed the requirements of programming languages and compilers before designing the hardware.

The following section outlines the parallel programming language Modula-2* and derives general requirements for parallel computers. In section 3 we describe the architecture of Triton/1. We emphasize those features which arose from software requirements.

2 Modula-2*

Modula-2* (pronounced Modula-2-star) is a small extension of Modula-2 for massively parallel programming. The programming model of Modula-2* incorporates both data and control parallelism and allows mixed synchronous and asynchronous execution.

Modula-2* is problem-oriented in the sense that the programmer can choose the degree of parallelism and mix the control mode (SIMD- or MIMD-like) as needed by the intended algorithm. Parallelism may be nested to arbitrary depth. Procedures may be called from sequential or parallel contexts and can themselves generate parallel activity without any restrictions. Most Modula-2* programs can be translated into efficient code for both SIMD and MIMD architectures.

2.1 Overview of language extensions

Modula-2* extends Modula-2 with the following two language constructs.

1. Parallelism can be created in Modula-2* programs only by means of the *`FORALL` statement*. There are two versions of this statement, a synchronous and an asynchronous one.
2. The distribution of array data may be optionally specified by so-called *allocators* in a machine-independent way. Allocators do not have any semantic meaning; they are hints for the compiler.

Because of the compactness and simplicity of the extensions, they could easily be incorporated in other imperative programming languages, such as Fortran, C, or Ada.

A synchronous `FORALL` statement creates a set of synchronously running processes. The number of the processes is determined by the `FORALL` statement and is not limited to the number of the PEs of the machine. As long as there is no branch in the control flow of the statements inside a `FORALL` statement the semantics of the execution is equivalent to a SIMD machine executing those statements. If there are branches in the control flow that define alternatives, like IF THEN ELSE or CASE, the processes are partitioned into several groups. Each of those groups is executing one branch in the control flow. The processes belonging to one group execute synchronously in SIMD style,

but groups are allowed to execute concurrently with respect to each other. There is no assumption about the relative speeds of two processes in different groups.

In contrast to the synchronous `FORALL` statement, an asynchronous `FORALL` statement creates a set of independent processes running at an unspecified relative speed. Common to both variants is that the termination of a `FORALL` statement is determined by the termination of the last process created by the `FORALL` statement. The end of a `FORALL` statement always defines a synchronization barrier. For further details about the language, see [7], for detailed discussion of compilation techniques and optimization, see [5].

2.2 Software requirements for parallel computers

On distributed memory machines, the distribution of array data over the available processors is a central problem. Two conflicting goals, (1) data locality and (2) maximum degree of parallelism, must be reconciled. Data locality means that data elements should be stored local to the processors that need them to minimize communication costs. Perfect data locality could be achieved by employing a single processor. Parallelism, which reduces runtime, may unfortunately reduce locality and increase communication costs. Additional goals for data distribution are: (3) exploiting special communication patterns supported by hardware and (4) generating simple address calculations to prevent addressing from becoming a dominant cost.

Even with optimal layout of the data, there will still be communication in the general case. In fact, in most massively parallel applications that are not trivially parallel, communication is almost as frequent as computation. Therefore,

the network MIPS measure must approach the CPU MIPS measure

A second performance-oriented recommendation is an

independently operating network with asynchronous message delivery

since it allows the delivery of packets concurrently with computation. That would enable the compiler to interleave computation and communication and thus to hide some of the communication latency.

On the other hand we recommend a

shared address space

A shared address space does not imply shared memory; it only means that every processor can generate addresses for the entire memory in the system. System-wide addresses are especially important for pointers, because otherwise they would have to be simulated quite inefficiently in software. Even the memory of the control processor, e.g., the frontend of a SIMD machine, should be part of the shared address space.

Furthermore and similar to the above, we call for a

uniform memory access instructions

Many parallel machines today provide a set of instructions for accessing local memory, a second one for accessing memory in neighbors, and a third set for accessing distant memory units. The differences in speed are significant and, therefore, require that the compiler detect the faster cases. However, it is often impossible to know statically for which case to optimize. For instance, we found that in many cases it was impossible to determine in the compiler whether a procedure would access local or non-local memory. The generated code thus has to check all three cases at run-time. Such simple, frequent, and dynamic analyses could be done more efficiently in hardware.

Barrier synchronization is extremely frequent. Basically, every communication requires a barrier synchronization. The reasoning is as follows. Communication sends or receives data. Unless communication is redundant, there must be a write between two successive communication calls to the same cell. It follows that a synchronization operation must be placed somewhere between one of the communication calls and the write in order to avoid race conditions. If many processes communicate at once, as in massively parallel machines, this type of synchronization amounts to a barrier: No process may proceed until all processes have completed either their write or communication operation. Because of its frequency, we need

fast barrier synchronization.

In principle, the communication network could be used for barrier synchronization. However, communication networks usually have high latency which make them too slow for fast barrier synchronization. These nets are optimized for transporting data, while barrier synchronization requires the transport of only one or two bits but must also implement a reduction or scan operation on these bits.

An additional complication is that there are usually several groups of processes that need to communicate among themselves, necessitating multiple, non-overlapping barriers. Consider, for instance, a pipelined architecture in which one set of processes passes data to another set. Each set may have to synchronize internally, independent of the other. Similarly, an IF-statement within a FORALL divides a set of processes into two subsets which may have to synchronize independently. Thus, we need

barrier synchronization for multiple, independent sets of processes.

3 Triton/1

The poor programmability of today's parallel machines is a consequence of the fact that the design of these machines has been driven mostly by hardware considerations. Programmability seems to have been a secondary issue, resulting in languages designed specifically for a particular machine model. Such languages do not satisfy the needs of programmers who need to write machine-independent applications.

Triton/1 matches most of the recommendations of the previous section. Looking from a general point of view, Triton/1 is determined by the following statements.

General Architecture. Triton/1 is a SAMD (synchronous/asynchronous instruction streams, mutliple data streams) machine: it runs in SIMD mode where strict synchrony is necessary; it can switch to

MIMD mode where concurrent execution of different tasks is beneficial. It is even possible to run a subset of the processors in SIMD mode, the other in MIMD. Thus, Triton/1 is truly SAMD, i.e, mixed-mode, not just switched-mode. Only a few research prototypes of mixed-mode machines have been built: OPSILA, TRAC, and PASM [1, 6]. Triton/1 provides support for switching rapidly between the two modes and a high-level language to control both modes effectively.

Fast barrier synchronization is supported by special hardware. The usage of synchronization hardware is possible in both operating modes. Synchronization with hardware support overcomes the necessity of coarse grained parallelism.

Network. We chose the De Bruijn network for Triton/1, because it has several desirable properties (logarithmic diameter, fixed degree, etc.), is cost-effective to build, and can be made to operate extremely fast and reliable. In section 3.3 we present performance figures.

Scalability and balance. Parallel machines should scale in performance by varying the number of processors; furthermore, the performance of the individual components (processor, memory, network, and I/O) should harmonize. Scalability is mainly a property of the network. The most popular networks today, hypercubes and grids, do not scale well: hypercubes are too expensive because they have variable degree, while grids cause high latency because of large diameter. Triton/1's De Bruijn net has none of these problems and scales well. It is also well matched to the speed of the processors.

I/O capabilities. I/O must also scale with the number of processors. Few parallel machines today provide for scalable I/O. Triton/1 implements a massively parallel I/O architecture: one disk per processor. For large sets of disks, we have extended the traditional notion of a file to what we call a **vector file**. Massively parallel I/O also provides the basis for research in parallel operating systems, such as virtual memory, parallel paging strategies, and true multiuser environments. Results in these areas are required to bring parallel machines into wide-spread and general purpose use.

3.1 Architecture of Triton/1

Triton/1 is divided in a frontend and a backend portion. The frontend typically consists of a UNIX workstation with a memory-mapped interface to connect via the instruction and the control bus to the backend portion. The backend portion consists of the processing elements, the network, and the I/O system.

The Triton/1 prototype will be built up of a Intel 486 based PC running BSD UNIX as frontend. The prototype will contain 256 + 4 PEs of which 72 are supplied with a disk. 256 of the PEs are provided for computation and 4 PEs are for hot stand by. These PEs can be configured under software control into the network, if other PEs fail. The reconfiguration involves changing the PE numbers consistently and recomputing the routing tables in the network processors. The 72 disks are logically organized in 8 groups of 9 disks, where each group contains 8 data and one parity disk. RAID level 3 [4] is used for error handling. Figure 1 gives an overview on the logical organization of Triton/1.

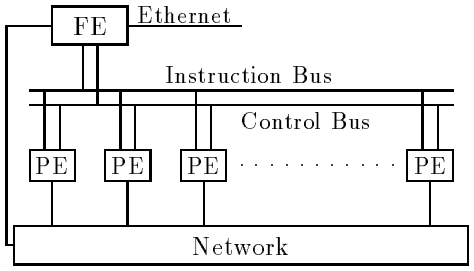

Figure 1: Triton/1 Architecture

In SIMD mode the frontend produces the instruction stream and controls the backend portion at instruction level. In MIMD mode the frontend is responsible for downloading the code and the initiation of the program. The instruction bus is 16 bits wide. For reasons of decoupling frontend and backend in order to reduce the time of the frontend waiting for the backend to become ready, or vice versa, the instruction stream is sent through a fifo. The handshake signals necessary to control the instruction stream are part of the control bus.

For reasons of debugging it is a good idea to have direct access from the frontend to all parts of the machine, especially to the main memory distributed among the processing elements. As a direct consequence of the common address space of Triton/1 this is possible via the so called *analyze mode*. To support the analyze mode the control bus includes 40 address lines and several dedicated control signals, the instruction bus is used for data transport. While being in analyze mode, all PEs release their local busses, to enable frontend access via direct memory access.

The processing elements are designed as universal computing elements, capable of performing computation as well as service functions. Each PE consists of a Motorola MC68010 micro-processor, a memory management unit (MC68541), a numeric co-processor (MC68881 or MC68882) , 2 MBytes of main memory, a SCSI interface, and a network-processor. Figure 2 gives an overview. No extra controllers for mass storage access or any other I/O are necessary.

The network of Triton/1 is built up of the network-processors included in the PEs, the interconnection lines, realized with flat cables, and fifo buffers for intermediate buffering of data packets. The network is able to route data packets from their source to their destination without interfering with the PEs. Non-interference permits latency hiding techniques to be applied. Again for reasons of decoupling, the interface between a PE and its respective network processor is implemented with fifos.

Parity checking of main memory, network links, and mass storage implements error detection. Periodic signature tests locate malfunctioning elements.

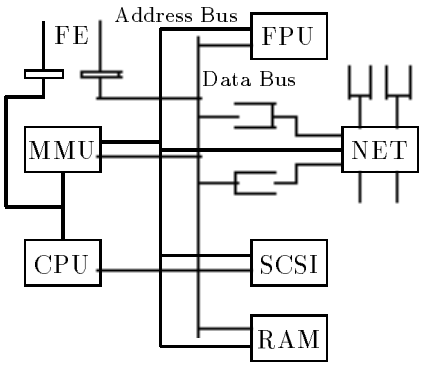

Figure 2: Triton/1 Processing Element

3.2 Detailed discussion of selected hardware aspects

In order to get a better idea on the architectural features of Triton/1 it is necessary to look into some implementation details of the hardware.

The implementation of the **instruction bus** and the **control bus** is quite naturally done by a hierarchy of bus drivers, for signals from the frontend to the backend For the opposite direction, *global-wired-or* lines are emulated by explicit OR-combining the signals from the single PEs. In SIMD mode all PEs execute the same instruction at a time (or idle), including reading the instruction at the same time. The reading of instructions by the PEs is controlled via three control signals: *Instruction strobe* signals the frontend that all PEs currently listening to the instruction stream are ready to read an instruction. The frontend then asserts the instruction and answers with *instruction transfer acknowledge*. If all PEs currently listening accepted the instruction *instruction fetch done* is asserted, which signals the end of an instruction transfer. The three-way handshake introduces a non-negligible amount of delay due to the signals traversing the complete bus hierarchy several times. To reduce that delay we introduced an instruction buffer at each driver level in the bus hierarchy, reducing the delay for the instruction fetch by two clock cycles in the normal case. Thus the handshake described above is executed in between every two hierarchy levels, rather than between the frontend and the PEs.

As mentioned above the **global address space** constists of a 40 bit address. The least significant 23 bits are used to select the memory and the memory mapped I/O in the PEs. Another 16 bits are used to identify the PE to be accessed, and one bit is used to distinguish frontend and backend. The identification of the PEs is twofold. Each PE has a hardware identification, which is selected by a switch setting. The hardware identification is used to select the PE in the analyze mode for debugging. Additionally, each PE has a software identification, which is used while computing. The software id is initially set to the same value as the hardware id, but can alter for reasons of hardware error handling. However, implementing a concept with a 40 bit address space does not automatically imply computing with 40 bit addresses all the time. In the majority of the cases computing with 32 bit addresses suffices, reducing the time spent with address calculation.

Another point of interest is **mode-switching**. Though the MIMD mode is more natural to the processor the system is started in SIMD mode. This is done to save additional hardware for the startup code. In SIMD mode, the function codes of the processor are used to determine whether the processor intends a data or a program access. According to that, the processor bus is connected to the local memory or the instruction bus, respectively. If a PE is selected not to execute an instruction, the local signal *listen to instruction stream* is turned off and the processor of that PE is not notified of instructions, except if the instruction is unconditional. The value of the processors program counter is completely ignored in SIMD mode. In order to switch to MIMD mode the program to be executed has to be downloaded to the memory of the PEs. This is done via the instruction stream in SIMD mode. Thus, the distribution of code is, in contrast to many other MIMD machines, done in a time proportional to the length of the code, independent of the size of the machine. The switch from SIMD to MIMD mode is performed by two instructions: With the first instruction, the program counter is set according to the location of the program to be executed in MIMD mode by a JMP instruction. With the second instruction, the SIMD request bit in the command register local to the PE is deactivated. The PE then switches to MIMD mode at the end of the current cycle and commences execution of the local code without delay. To switch from MIMD to SIMD mode, the SIMD request bit in the local command register simply is activated, which causes the PE to switch to SIMD mode at the end of the current cycle. The next instruction is then expected form the instruction stream.

While some PEs are executing in MIMD mode, the rest of the PEs may execute in SIMD mode. This is achieved by activating the instruction transfer handshake in the case of MIMD operation. If there is no PE left to execute in SIMD mode but still some instructions remain in the instruction fifo, the handshake signals automatically empty the buffer.

Data transfer is an important point in every parallel computer. There are several different data paths to consider. The most important point is the data transport between the PEs. That task is performed by the network which is described later in detail. Another important point is the transport of data from the frontend to the backend and vice versa. There are different possibilities for each direction: To transport data from the frontend to the backend, the easiest way is to send the data as *immediate data* via the instruction stream in SIMD mode. With that possibility, any subset of the PEs can be the destination of the data. Unfortunately only unidirectional access is possible. The second possibility of transferring data is the direct memory access within the analyze mode. Herein data can be transferred in both directions. The drawback of the analyze mode is that no computation can

take place and not more than one PE can be accessed concurrently. The third possibility of data transport is via the network. There is one dedicated network node which is connected to the frontend. This is especially useful in the case that more than a few bytes have to be transported from different PEs to the frontend, e.g. picture data. Another advantage of a network node included in the frontend is that computation can commence while data is transported.

3.2.1 Fast barrier synchronization in MIMD mode

An important problem is the realization of barrier synchronization in the case that several different sets of processes are distributed randomly over the PEs. A set or group of processes is defined as executing the same part of code (e.g. procedure) and therefore sharing common variables. If there is only one set of processes requesting synchronization, the barrier synchronization is easily done by the usage of a global-wired-or line. Each PE sets its ready bit on the line to true as soon as it reaches the synchronization point. Approximately one clock cycle after the last PE sets its bit, the frontend is able to recognize the result and the PEs are notified by the result line.

The problem of using synchronization with a single global-wired-or line in MIMD mode is that a global-wired-or line cannot be partitioned randomly. In the general case, more than one group of processes exists. Each of these groups share common variables to which accesses have to be regulated. In most cases the groups of processes are distributed randomly over the set of PEs so that they cannot be partitioned by partitioning the backend.

To enable the usage of hardware supported synchronization with several groups of processes running in MIMD mode, the global-wired-or line is administrated by the frontend as a synchronization resource in the following way. Each group of processes is identified by a unique process group number. Initially, the synchronization line is not used, and each PE is allowed to request it on behalf of a group. The request is performed by the first PE reaching a barrier. That PE signals the frontend by the service request line and sends the group identification via the analyze circuits. If more than one PE reaches a barrier at once, the analyze circuits will select one randomly. The frontend then knows which group demands the synchronization line. Next, the frontend interrupts all PEs and forces them into SIMD mode to perform a barrier setup. The PEs not belonging to the requesting group are prohibited to request the sync line themselves. They also turn on their ready bits. The PEs belonging to the requesting group set their ready bit to true if they already reached the sync point, otherwise to false.

After this setup phase, the PEs return to MIMD mode. All PEs continue computation, independent of their group membership. As soon as the last ready bit is turned on, the group owning the global-wired-or line synchronizes and then releases the sync line. The frontend then releases the request prohibition in order to enable other groups to synchronize.

This discussion glossed over the difficulties that arise if a PE virtualizes, i.e., executes several threads or processes which may belong to different groups. In this case, the ready bits have to be virtualized as well. The details depend on the virtualization strategy, which may be a mixture of looping and context switching.

3.3 Communications Network

The Triton/1 network is based on the generalized De Bruijn Net [2, 3]. The number N of nodes in the network is not limited to powers of two. The (maximum) diameter is $\lceil \log_d N \rceil$. The average diameter is well below $\log_d N$ and in practice quite close to the theoretical lower bound, the average diameter of directed Moore graphs.
In our implementation we use degree $d = 2$, which makes our net a perfect shuffle (see figure 3). In comparison with other frequently used networks, this design has the benefit of a constant degree per node and a small average diameter. Data transport is done via a table-based, self-routing packet switching method, which allows wormhole-routing and load dependent detouring of packets. Every node is equipped with its own routing table and with four buffers: two for intermediate storage of data packets coming from other nodes and two to communicate with its associated processing element. The buffering temporally decouples the network from local processing.

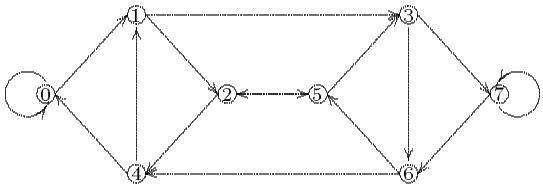

Figure 3: De Bruijn Net with 8 nodes

The communications processor is able to route the packets without interfering with the local processing element. Optimal routes are stored in a routing table per communications processor. The network can thus transport data in parallel with the operation of the processing elements. This feature can be used by the compiler to overlap communication and processing time by rearranging code.

In order to analyze the behavior of the network, we built a simulator based on the measured performance of a single communications processor. We simulated the overall performance of the network in various modes. The number of nodes ranged from 32 to 8192. Figure 4 presents the results of a series of experiments with a random communication pattern (random H-permutations).
Both the sender and the receiver were chosen randomly, with the restriction that the number of data packets to be transported is the same as the number of nodes in the network. The simulation shows that the network scales well: the delay introduced by the network is within $O(\log N)$, with N denoting the number of nodes and messages.

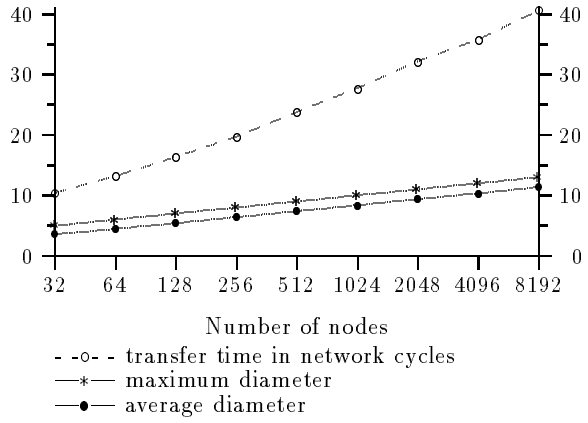

Figure 4: Performance on random communication

The robustness against overload is surprisingly good. Even if all processing elements send a great number of packets simultaneously, the overall throughput of the network does not decrease. Irregular permutations are performed especially fast. All "hard" patterns known from literature, e.g., transposition of a matrix, butterfly, and bit reversal perform well too. The delivery time for those is equal to or lower than the delivery time for random permutaitions.

4 Conclusion

The integrated approach of designing language, compiler, and hardware together has lead to a parallel architecture that supports higher-level languages adequately. Fast barrier synchronization for multiple process groups, SAMD mode, shared address space, and a fast, independently operating network should make parallel computers run efficiently even when programmed in a machine-independent fashion.

Status and schedule of Triton/1

The fully functional prototype of a PE board was completed in October 1992. The individual components (communication processor, processing element and control processor interface) are tested and are running according to specifications. The manufacturing of the printed circuit boards is in progress. The final assembly of Triton/1 will be completed early in 1993.

References

[1] M. Auguin and F. Boeri. The OPSILA computer. In M. Consard, editor, *Parallel Languages and Architectures*, pages 143–153. Elsivier Science Publishers, Holland, 1986.

[2] N. G. De Bruijn. A combinatorial problem. In *Proc. of the Sect. of Science Akademie van Wetenschappen*, pages 758–764, Amsterdam, June 29 1946.

[3] Makoto Imase and Masaki Itoh. Design to minimize diameter on building-block network. *IEEE Transactions on Computers*, 30(6):439–442, June 1981.

[4] David A. Patterson, Garth Gibons, and Randy H. Katz. A case for redundant arrays of inexpensive disks (RAID). In *Proc. of the 1988 ACM-SIGMOD Conference on Managenment of Data*, pages 109–116, Chicago, 1-3 June 1988.

[5] Michael Philippsen, Walter F. Tichy, and Christian G. Herter. Modula-2* and its compilation. In *First International Conference of the Austrian Center for Parallel Computation*, pages 169–183, Salzburg, Austria, September 1991. Springer Verlag, Lecture Notes in Computer Science 591.

[6] H.J. Siegel, T. Schwederski, J.T. Kuehn, and N.J. Davis. An overview of the PASM parallel processing system. In D.D. Gajski, V.M.Milutinovic, and H.J.Siegel and B.P. Furht, editors, *Computer Architecture*, pages 387–407. IEEE Computer Society Press, Washington, DC, 1987.

[7] Walter F. Tichy and Christian G. Herter. Modula-2*: An Extension of Modula-2 for Highly Parallel, Portable Programs. Technical Report No. 4/90 (Interner Bericht), University of Karlsruhe, Department of Informatics, January 1990.