JPRS-EST-89-026
7 SEPTEMBER 1989

FOREIGN
BROADCAST
INFORMATION
SERVICE

JPRS Report

Science & Technology

Europe

FRG's 'Suprenum' Supercomputer

19980511 113

Science & Technology

Europe

FRG's 'Suprenum' Supercomputer

JPRS-EST-89-026 CONTENTS *7 SEPTEMBER 1989*

36980245 Amsterdam SUPERCOMPUTER in English Mar 89

Concept *[Ulrich Trottenberg; SUPERCOMPUTER, Mar 89]* 1
System *[Wolfgang Giloi; SUPERCOMPUTER, Mar 89]* 4
User Interface *[Karl Heinz Werner, et al.; SUPERCOMPUTER, Mar 89]* 7
Programming *[Karl Solchenbach; SUPERCOMPUTER, Mar 89]* 10
Parallel Programming *[Bernhard Thomas, Klaus Peinze; SUPERCOMPUTER, Mar 89]* 13
Application Software *[Karl Solchenback; SUPERCOMPUTER, Mar 89]* 19
SUPERB Parallelizer *[Heinz-J. Bast, et al.; SUPERCOMPUTER, Mar 89]* 22

Concept

36980245 Amsterdam SUPERCOMPUTER in English Mar 89 pp 5-12

[Article by Ulrich Trottenberg of Suprenum GmbH, Bonn: "Suprenum—The Concept"]

[Text] The Suprenum 5-Gflops supercomputer is the result of a national German project which includes an MIMD [multiple instruction, multiple data] multivector processor hardware with distributed memory and the development of all software layers that guarantee comfortable and efficient exploitation of the scalable hardware. Also, a large amount of application software has been developed on the basis of superfast parallel algorithms. The applications cover all typical models in scientific computing.

Suprenum is an unusual combination of a widespread, long-range research-oriented activity and a strictly product-oriented development. The research idea was to bring together:
— the users of supercomputers representing the know-how for the grand challenge problems ("superproblems") in scientific computing and numerical simulation;
— the computer architects representing the know-how on parallel architectures, parallel languages, and tools for parallel computing;
— the numerical analysts representing the know-how on fast numerical algorithms (like multigrid and multilevel approaches) and their "superfast" parallel versions.

Suprenum combines these three fields. In its product-oriented part, it consequently develops a system that integrates:
— hardware;
— the operating and the run-time system;
— programming environment;
— parallelization (partitioning and vectorization) tools;
— basic and advanced application software.

About a third of the manpower in the Suprenum project is devoted to the hardware, another third to system software, programming environment and tools; and the last third is used in application software.

The Suprenum Hardware Essentials and Software Developments

The Suprenum prototype system which will be operational by the end of 1989 is, with respect to its hardware, characterized by the following essentials (see Figure 1):
— highly parallel MIMD architecture with a peak performance of 5 Gflop/s;
— 256 computing nodes aggregated into 16 clusters;
— each node with 8 Mbyte private memory (giving an overall main memory of 256 x 8 Mbyte = 2 Gbyte);
— each node with a vector floating-point unit (20 Mflop/s, if chaining is used);
— flexible two-level (intra- and inter-cluster) interconnection network on the basis of very fast busses.

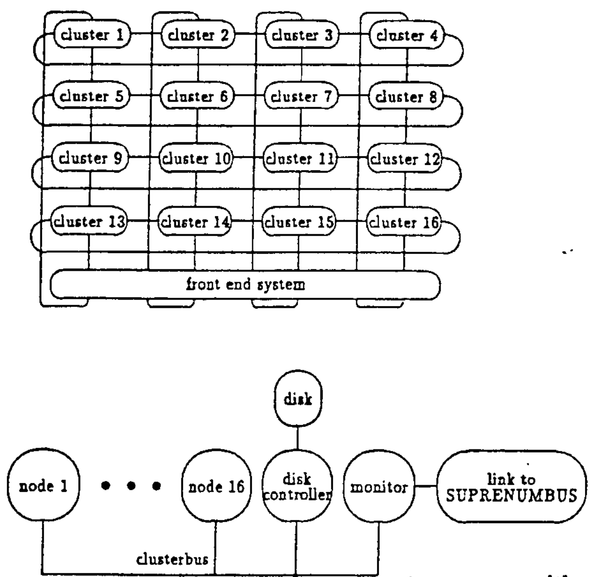

Figure 1. Structure of the Suprenum 1 prototype with 256 nodes in 16 clusters.

The timetable for the hardware development is as follows: After the first ideas in 1984, most of the essential architecture decisions were made in 1985. In 1986, Suprenum GmbH was founded, and work essentially commenced. A preliminary system (10 nodes in two clusters) with nearly the full functionality was available in 1987. The final node with full performance was running in 1988.

This year, 1989, a 32-node-system (two full clusters) will be in operation at the Hannover fair in April, and the final 256 node prototype (16 full clusters) will be operational by the end of this year.

The development of system software, programming environment, compilers, tools, and application software was carried out essentially in parallel with the hardware development. Actually most of the software will already be finalized before the hardware is fully operational. In order to be able to achieve this, it was extremely important to make *simulators* available that allow, for example, Suprenum application software (in Suprenum-Fortran) to be developed and tested on other computers. Furthermore, an essential part of the project was devoted to analyse the system behavior and its influence on the overall system performance for each single hardware and software component. On the basis of these tools, very precise performance predictions were made which, in turn, allowed tuning and optimizing the system essentials and the software.

Suprenum in the Supercomputer World

In order to embed Suprenum into the supercomputer world, we distinguish 8 classes of architectures (see Figure 2).

1. *SIMD* [single instruction, multiple data] *versus MIMD* (vertical line). SIMD operation mode means that parallel or pipelined functional units execute the same instruction sequence on different data.

The MIMD principle is the favorite operation mode for multiprocessors based on independent complete processors. Each processor may execute a different instruction stream within the same application.

2. *Shared versus distributed memory* (horizontal line). One of the central problems to be solved in the design of multiprocessor systems is the memory access. Basically, there are two possibilities to organize this:
— shared memory (sm) guarantees fair access to a global memory for each processor;
— distributed memory (dm) means that each processor has direct access only to its own private memory.

Often both memory organization types are combined in hierarchical memory systems.

Our classification reflects the user's view of the memory organization rather than its hardware realization.

3. *Scalar versus vector floating-point units* (dashed horizontal lines). Presently, scalar floating-point units seem to be restricted to a floating-point performance of less than 10 Mflop/s. The most cost-effective way to achieve higher floating-point rates is vector processing. Therefore, the most powerful architectures today are mixed MIMD/SIMD multiprocessor systems (class 4 and 8). The efficient use of these architectures requires parallelism on two levels: the coarse grain parallelism related to the global MIMD structure and the fine grain parallelism which ensures efficient vector processing locally.

In the following we briefly describe the 8 classes of supercomputer architectures and name typical representatives of them.

Class 1: Scalar Computers

The "traditional" Von Neumann computer architecture (SISD [single instruction, single data]) is the basis for mainframes, minicomputers, and microcomputers. Using the current hardware technology, the floating-point performance of this architecture seems to be limited to 10 Mflop/s, which is much less than current supercomputer performance.

Class 2: Vector Computers

Historically the first machines to be called supercomputers were vector computers. Their hardware architecture is based on very fast arithmetic pipelines which support the rapid execution of vector instructions operating on all components of vector operands simultaneously. Vectors in that sense consist of components which can be processed independently. Hence, vector processing is a special form of parallel processing based on fine grain parallelism. Application codes have to be vectorized (i.e., operations are defined on vectors and certain data dependencies between operations are excluded) in order to exploit the potential speed of the hardware. The need for vectorization resulted in new vector algorithms and in special compiler tools (vectorizers) for the automatic vectorization of existing codes.

Examples for vector machines are Cray-1, Cyber 205, Fujitsu VP, NEC-SX, Hitachi S-810, and the IBM 3090-VF.

Due to technological progress in VLSI chip development, vector computer architectures today can be realized in standard microcomputer technology. These systems are smaller, somewhat slower and considerably cheaper than the classical vector computers and therefore called minisupercomputers. The vector-minisupercomputers take advantage of the existing software and tools for vector machines; some systems are even Cray-compatible. Examples are Convex C1 and SCS-40.

Class 3: Scalar SM-Multiprocessors

Another way to increase the computing performance is the combination of several single processors to a multiprocessor system and to replace the sequential processing by parallel processing. The optimal degree of parallelism (fine or coarse granularity) depends on the number and the power of the single processors as well as on the memory organization. The shared memory concept restricts the number of CPUs to less than or equal to 8 today (e.g. the Alliant). If the memory is accessed via a network, a larger number of CPUs can be connected at the cost of longer access times. Examples are the IBM RP-3 and the Cedar project (= clusters of Alliant systems). Further examples in this class are the Sequent, Flexible, Encore, and Concurrent Computers machines.

Class 4: Vector SM-Multiprocessors

The step from a single processor to a multiprocessor system (class 1 to class 3) is, of course, also possible and obvious for vector computers (class 2). Similarly as for scalar multiprocessors, the performance is increased by composing several vector CPUs to multiprocessor systems with the same memory access problems. The shared memory concept limits the number of vector processors (today less than or equal to 8). The parallelism on these systems is often used to increase the throughput of the systems (running different jobs on different CPUs). MIMD parallel as well as SIMD-like processing is also possible (e.g. on the Cray X-MP using macrotasking or microtasking constructs). Representatives of this class are the Cray X/Y-MP, Cray-2, and the ETA-10.

Class 5: Scalar Array Processors

The era of parallel computers started with array processors which perform one instruction simultaneously on an array of operands (in SIMD mode). Recently these systems have been upgraded to massively parallel multiprocessors (with many thousands of processors). Each processor is relatively small and weak but the enormous degree of parallelism may result in supercomputer performance. Typically, these systems are used for a restricted class of special applications. We mention here the historical Illiac IV, the Goodyear MPP, the ICL DAP, and the Connection Machine 1.

Class 6: Vector Array Processors

The combination of (SIMD) array and vector processing has been realized in the Connection Machine 2, which presently is the system with the highest floating-point performance rate for special applications on very regular data structures.

Class 7: Scalar DM-Multiprocessors

Today, multiprocessor systems with a large (and principally unlimited) number of processors require that the memory units are physically associated with the processors (distributed memory). The basic unit of such a system (a "processing node", or shortly a "node") consists of the CPU, the arithmetic coprocessors, the memory, and the communication unit. The first prototypes of this class were based on hypercube topologies and were built up at the Californian Institute of Technology. Intel's iPSC was the first commercial product, followed by Ametek and Ncube. Recently Intel came out with its second generation, the iPSC-2. Multiprocessor systems with transputer nodes have also entered the market (Meiko, Parsytec).

Class 8: Vector DM-Multiprocessors

These systems combine the advantages of the vector and the parallel processing concepts. The multiprocessor architecture is derived from the class 7 machines, whereas the node architecture is taken from low-cost vector computers (class 2). The basic idea is to combine powerful vector nodes having an advantageous cost/performance ratio with a multiprocessor system. Due to the size and the cost of a single node, their number is—although principally unlimited—today practically limited to several hundreds. The computational speed of the nodes, of course, imposes strong requirements on the speed of the communication. If the communication problem is solved satisfactorily, these machines are the most powerful supercomputers existing today. Systems currently entering the market are Suprenum, the Intel iPSC-VX, and the Ametek 2010.

The classification of parallel computers in Figure 2 is by no means unique and complete. An important classification category which is not taken into account in Figure 1 is the hardware technology. Systems based on very high-speed technology hardware (like the Cray and ETA systems) are much more powerful (and expensive) than systems based on microcomputer technology (like the Alliant), although they belong to the same class.

	SIMD	MIMD	
scalar floating-point units	class 1 scalar computers (SISD)	class 3 scalar sm-multiprocessors	shared memory
vector floating-point units	class 2 vector computers	class 4 vector sm-multiprocessors	
scalar floating-point units	class 5 scalar array processors	class 7 scalar dm-multiprocessors	distributed memory
vector floating-point units	class 6 vector array processors	class 8 vector dm-multiprocessors	

Figure 2. Classification of parallel and supercomputer architectures.

The Suprenum Project Organization

The Suprenum project was conceived—during its definition phase by the *Gesellschaft fuer Mathematik und Datenverarbeitung mbH* [Company for Mathematics and Data Processing]—as a big national joint venture. From the beginning of the project, the initiators were aware of the fact that only by means of a concentrated cooperation of the leading experts could Germany catch up in the international supercomputer developments. Users, mathematicians (numerical analysts), system software experts, and computer architects had to be brought together and be committed to a uniform, clear development goal.

Thirteen partner institutions were recruited from industry, national research laboratories, and universities, being involved in the Suprenum project:

- German Research and Experimental Institute for Aeronautics and Astronautics (DFVLR);
- Dornier GmbH;
- Company for Mathematics and Data Processing mbH;
- Nuclear research center in Julich;
- Nuclear research center in Karlsruhe;
- Siemens AG (Power Plant Unit);
- Krupp Atlas Elektronik GmbH;
- Stollmann GmbH;

- Institute of Advanced Technology in Darmstadt;
- Brunswick Technical University;
- University of Bonn;
- University of Duesseldorf;
- University of Erlangen-Nuremberg.

The contributions of the partners are sponsored by the Federal Ministry of Research and Technology (BMFT).

Suprenum GmbH is the fourteenth partner. It was founded in 1986 due to an initiative of the BMFT by the main development partners Krupp Atlas Elektronik GmbH, Stollmann GmbH, and Gesellschaft fuer Mathematik und Datenverarbeitung mbH. It is funded by BMFT and the Ministry for Economy and Technology of the State of North Rhine-Westphalia. The primary tasks of Suprenum GmbH are:
- — coordination and management of the Suprenum project;
- — integration of the hardware and software components which are developed in the project;
- — fundamental research and development;
- — marketing of individual results, especially the Suprenum systems;
- — conceptional responsibility for the further Suprenum development.

Perspectives

With the realization of the Suprenum concept, an attempt is made to set a standard in the promising area of parallel processing. In order to ensure the long-term realization of this chance, it is important to conduct a permanent development.

The next Suprenum generation, Suprenum 2, is being conceived. The general Suprenum philosophy is:
- — to use a large number of processor nodes and
- — to make each node as powerful as possible on the basis of VLSI technology.

In order to achieve an optimum cost/performance ratio, this will not be changed.

Generally, the trends in supercomputer developments are characterized by an increase of the number of nodes for "conventional" supercomputers like the Cray and an increase of performance per node for the massively parallel computers like the Connection Machine. Thus, a convergence of architectures can be expected, with Suprenum in the middle of these trends.

The general concepts like the abstract Suprenum machine, the programming model, etc. will be maintained; some of them will be extended. One development goal of the system software, for example, is—among other things—the automatization and dynamization of process assignment (process migration, etc.), automatic optimization and dynamization of load balancing, etc.

The main emphasis in the field of applications software will be placed on the development and parallelization of new, even faster algorithms (e.g. on dynamic and self-adaptively modifying grid structures) and further numerical and also non-numerical application classes. Of course, all application software that runs on the Suprenum 1 will also be usable—with correspondingly higher performance—on the Suprenum 2.

A long-term research and development goal in the field of parallel computing should be to overcome the division of the parallel world into computers with a global shared memory and computers with distributed local memory units.

On the hardware side, these differences will disappear if multi-level memory hierarchies (the more local, the faster) are used. Such developments combine both concepts and anyway include what a forward-looking memory technique requires.

On the software side, the concepts for the presently pursued and future architecture lines should be standardized so that portability of the application software is ensured generally and not only within certain architecture classes. Several areas which are treated in the project (communication library, semi-automatic parallelizer) offer natural and promising approaches for those developments.

References

1. Giloi, W., *Suprenum—the system,* Supercomputer, this issue.

System

36980245 Amsterdam SUPERCOMPUTER in English Mar 89 pp 13-19

[Article by Wolfgang Giloi: "Suprenum—the System"]

[Text] The Suprenum supercomputing hardware consists of a scalable number of clusters each containing 16 vectorprocessors with local memory for high-speed computing and several special nodes for services within a cluster (disk controller, diagnosis, external links). Interprocessor communications based on a hierarchical bus concept, a parallel high-speed bus within a cluster, and a toroid system of multiple serial busses between clusters. This multiprocessor kernel is handling by a dedicated distributed operating system (PEACE) which provides—based on teams of light weight processes—fast services for message passing, resource management and all other functionalities within a multiprocessor kernel.

The Suprenum Node Architecture

A Suprenum supercomputer consists of up to 256 "Processing Nodes" (PN). Each PN is a complete single-board "vector machine" running its own operating system PEACE and communicating with other PNs. A PN consists of the following major resources (see Figure 1):

- node CPU (Motorola MC68020, 20 MHz) with Paged Memory Management Unit (PMMU) (Motorola MC 68851) and Scalar Arithmetic Coprocessor (MC 68882);
- 8 Mbyte of Node Memory (DRAM, 35 nsecs static column access time);
- pipeline vector processor (IEEE double precision) with 2 x 64 Kbyte of vector memory (SRAM, 20 nsecs access time);
- DMA/Address Generator for block transfer of data-structure objects;
- communication coprocessor for internode communication.

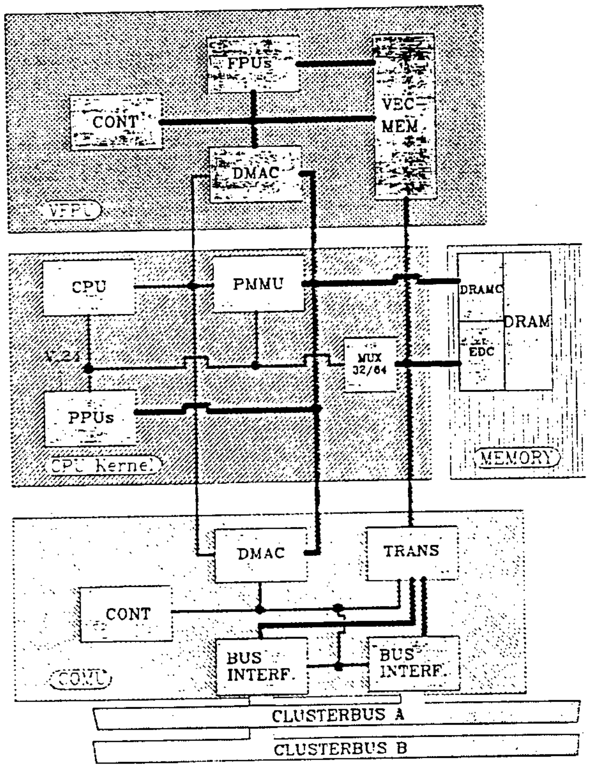

Figure 1. Internal structure of the suprenum processing node.

The node CPU performs the operating system tasks and interprets the instructions of a program. For the sake of a secured operation, access to the code and the data objects of the operating system and user tasks residing in the node memory is protected by the node PMMU. As the name suggests, the node memory is paged; however, not in the sense of virtual memory. Rather, paging is a means of protection and of providing fast block transfer DMA to the data elements in a page in a static column mode of operation. Since the PMMU adds 45 nsec to a memory access, it is employed only on the first entry into a new page, in order to exercise access right control, and from then on bypassed for all the other accesses to the same page. This is feasible since a page boundary violation would be detected "on the fly" by special page boundary watchdog logic.

The Pipeline Vector Processor (PVP) uses the Weitek WTL2264/2265 chip set in connection with a microcoded controller accommodated in one of the ASICs. At 20 MHz clock frequency, the Pipeline Vector Processor has a peak performance of 10 Mflop/s for the single operations (IEEE standard double precision) and 20 Mflop/s for the chained operations (e.g. vector dot product). The Vector Memory (VM) ensures a sustained performance close to the peak performance. The PVP performs also the common scalar floating-point arithmetical operations. The Scalar Arithmetic Coprocessor (SAC) (MC68882) provides additional floating-point functionality such as conversion, trigonometric and trancendental. The DMA/Address Generator (MAP) allows for a high-speed block transfer of data-structure objects (DSO):

i. between Pipeline Vector Processor and Vector Memory,

ii. between Vector Cache and Node Memory, and

iii. between Node Memories of different nodes.

Its microcoded address generators support all required access functions for the data-structure types "vector" and "matrix" [2,3]. The MAP functions are performed by an ASIC.

The Communication Coprocessor (CC) performs the functions of formatting, sending, and receiving of messages by hardware in the microsecond range. It is realized by an ASIC.

All four coprocessors utilize the same unique coprocessor interface of the MC68020, whose functionality has been expanded by a special coprocessor interface ASIC. In addition to the ASICs already mentioned, there are also ASICs for other functions such as memory error detection and correction (EDC), address decoding, data path multiplexing and bus protocol handling. All ASICs are realized by CMOS gate arrays from LSI Logic Inc.

The node memory utilizes DRAM SMDs with 1 Mbit capacity each, mounted on SIPs. The various static memories (Vector Memory, microprogram control stores) as well as the CMOS bus drivers are packaged as hybrid modules.

The Suprenum Cluster

A Suprenum cluster consists of 20 nodes accommodated in one 19 inch rack; 16 nodes are the "Processing Nodes" as described in the preceding section. The other 3 nodes are: the Cluster Disk Controller Node (DCN), the Inter Cluster Communication Node (CCN), and the Cluster Diagnosis Node (CDN).

The nodes of a cluster communicate via the cluster bus system, which consists of 2 message switching parallel buses with 64 date lines each. Doubling the cluster bus including its controller logic renders the cluster bus system fault tolerant and, by the same token, doubles the interconnection bandwidth in the cluster to a total of 320 Mbytes per second (arbitrated). Several nodes can communicate simultaneously via the cluster bus system.

The remaining question how to interconnect the clusters is discussed in the following section, which presents the rationale for the rather unique interconnection structure of Suprenum.

The Cluster Interconnection Structure

The interconnection structure of a high-performance MIMD/SIMD system with a large number of nodes must be blocking-free in order to avoid the performance degradation.

For many computer architects, the "classical" answer to the inter-node communication problem is the use of a multi-stage interconnection network (IN), realized in the form of either a circuit switching network, where a physical connection is provided directly from the source node to the destination node, or a packet switching network, where a logical connection is provided between source node and destination node.

There exist a large variety of IN structures, and many papers have been published dealing with the interconnection properties of the various network types. However, a few papers address the issues of technical feasibility, interconnection bandwidth obtainable, packaging problems including the severe pin limitation problem one may be running into, driving power limitation, cost, and other mundane technical problems [4]. Here we briefly discuss the dichotomy between the solution that exhibits ideal interconnection properties at the cost of an unfavorable (N^2)-complexity—the crossbar switch network—and on the other hand the favorable (*N log N*)-complex networks which have unfavorable interconnection properties.

The network type that provides total point-to-point connectivity without the danger of blockings is the crossbar switch. However, the quadratic complexity of the crossbar solution limits its size for reasons of pin limitation and packaging complexity to 32 x 32 or 64 x 64 at most. The network types that provide an optimal trade-off between interconnection properties and circuit complexity are the (*N log N*)-complex networks.

There exists a way out of the dilemma of the technical non-feasibility of large crossbar switch networks and the unfavourable interconnection behavior of the INs with (*N log N*)-complexity. The solution is a two-stage approach in which either a crossbar network or an equally fast packet switching network is used to interconnect *clusters of nodes* rather than the nodes themselves. Consequently, instead of having to interconnect a large number of nodes only a much smaller number of clusters needs to be interconnected.

Thus, the first stage of the two-stage interconnection network consists of the *intra-cluster interconnection structures,* while the second stage is formed by the *inter-cluster interconnection structure.* The advantage of this solution is that as long as the cluster size is kept sufficiently small there exists an extremely fast and economical solution for the intra-cluster interconnection structure, given in the form of the common *parallel bus.* Parallel buses can be made rather wide, e.g. 64 data bits, a measure that alone already guarantees a high interconnection bandwidth. In addition, as long as a parallel bus is kept short, it can be made quite fast. A parallel bus can be made fault tolerant, e.g. by adding a number of redundant bus lines. However, if one wants to combine fault tolerance with the highest possible interconnection bandwidth, the better approach is to simply double the parallel bus and have a bus arbiter that allocates to a requesting node either one of the two buses, whichever is free next. In order to keep the length of the cluster buses sufficiently short, we restricted the number of circuit boards to a maximum of 20, assuming 20 mm spacing.

In the Suprenum supercomputer, the clusters are interconnected via the torus structure. The torus structure is formed by a matrix of bit-serial ring buses which transmit data at a rate of 2 x 125 Mbits per second on the basis of the token-ring protocol. The net data rate, which the clusters of a torus must share, is about 20 Mbyte/s. This is the reason why not more than 4 clusters are inserted in each ring, so that there remains enough interconnection bandwidth per cluster. Doubling the torus structure by having row rings and column rings not only doubles the interconnection bandwidth but also renders the structure fault tolerant: should a ring fail, there is still the possibility of reaching the clusters of the ring through alternative routing. Alternative routing is provided by the CCN in each cluster (one of the specialized nodes).

The Node Operating System (PEACE)

Suprenum has been designed as a message-based, loosely coupled system, the rationale for this design decision being twofold. Firstly, "hot spot contentions" that may easily arise in memory sharing systems are avoided. Secondly, a high degree of fault tolerance had to be designed into the system, and thus, availability. Designing Suprenum as a fault tolerant architecture implies all the characteristics of a distributed system.

Therefore, centralized resources—including a centralized "global operation system" had to be avoided. Consequently, Suprenum has a local "node operating system" in each node, while a global operating system exists only virtually, its functions being performed in reality by the collective of node operating systems.

Major tasks of the node operating system are:
— local resources management, including access right control to memory;
— local process management;
— interprocess communication.

In performing its tasks, the node operating system may request services from other node operating systems; by the same token, it must be willing to provide services for other node operating systems.

There are strong reasons for designing the node operating system as a multitasking operating system. At the system level, a highly modularized design of the operating system enhances its efficiency, security, and fast implementation. At the user level, multi-tasking is a prerequisite for constructing application software independent of the specific configuration of the machine, i.e., the number of the nodes: the application program is partitioned into a number of cooperating processes which are then distributed over the number of nodes of a particular configuration.

PEACE (Process Execution And Communication Environment) [5] is the Suprenum node operating system especially designed to meet the requirements outlined above. Specifically, PEACE supports the following features:
— remote access of resources (files, devices) in other nodes;
— remote monitoring of system components in other nodes;
— dynamic reconfigurability of the system after the detection of faults and dynamic reconfiguration for load balancing and service migration in user programs.

The architecture of PEACE is based on the *team concept,* resulting in a highly modularized, hierarchically structured system. Means of structuring in PEACE are: processes and teams.

Processes are lightweight processes representing system components that render services to other such components; they are subject and object of access rights, and they allow readily the construction of dynamically reconfigurable systems. A team is a group of lightweight processes that share common access domains to intrinsic system objects such as files, memory segments, and processes.

A process requests a service from a remote server process by issuing a *remote procedure call* (RPC) message. Message passing is based upon a synchronous communication mechanism of maximal efficiency.

PEACE is hierarchically structured, its core consists of:
— PEACE kernel
— process server;
— name server;
— memory server;
— team server.

Functions of the PEACE kernel are:
— interprocess communication (supported by a specific communication coprocessor);
— process and address space switches;
— propagation of traps and interrupts as messages;
— message routing (send, reply).

Functions of the name server are the issuing and monitoring of name spaces and service access points (SAP). Functions of the process server are the issuing of unique process identifiers (PID) and the dynamic process administration. Functions of the memory server are the issuing of segment identifiers and the dynamic management of memory objects. The team server handles a variety of specialized teams that function as *administrators* such as: name administrator, team administrator, memory administrator, panic administrator, signal administrator, clock administrator, device administrator, and file administrator, the names of the administrators indicating their role in the system. Each administrator team usually encompasses several server processes (e.g. the memory administrator comprises the memory server and the MMU trap server, etc.).

PEACE has been designed in MODULA-2 and was rewritten in C for performance reasons. PEACE has been optimized and fine tuned to render its basic function as fast as possible. At present, PEACE is believed to be the fastest message-passing operating system currently existing.

References

1. Giloi, W., *Suprenum: A trendsetter in modern supercomputer development,* Parallel Computing 7/3, 1988.

2. Giloi, W. and H. Berg, *Introducing the concept of data structure architectures,* Proc. 1977 International Conference on Parallel Processing, 44-51, 1977.

3. Giloi, W. and R. Guth, *Concepts and realization of a high-performance data type architecture,* Internat. J. Comput. Inform. Sci. 11(1), 25-54, 1982.

4. Ermel, W., *Untersuchungen zur technischen Realisierbarkeit von Verbindungsnetzwerken fur Multicomputer-Architekturen,* Ph.D. Thesis, Technical University of Berlin, FB Informatik, 1985.

5. Schroder, W., *A distributed process execution and communication environment for high-performance application systems,* in: Nehmer, J. (ed), Experiences with Distributed Systems, 1987.

User Interface

36980245 Amsterdam SUPERCOMPUTER in English Mar 89 pp 20-24

[Article by Karl Heinz Werner, Ulrich Brass and Ernst Thomas: "The Suprenum User Interface"]

[Text] From the front-end the Suprenum multiprocessor along with all its resources and file system is accessed simply as a common Unix-device to the user. Server like the job manager, the Suprenum kernel manager, and a file server system can be invoked by Unix commands and take care for mapping and execution of jobs on a requested partition of the kernel, for file management and access rights, job security and I/O. The user interface is implemented on top of PEACE and Unix.

Very early in the conceptual phase of the Suprenum project it was clear that the global architecture of the system will be divided into the host system (one or more Unix machines) and the Suprenum kernel (sk), consisting of several independent nodes organized in clusters and performing high speed numerical programs.

An interesting question is how to use a high performance numerical computing device such as the Suprenum kernel within a Unix system from a user's point of view. A Suprenum system is expected to run in different environments: computing centers, university institutes, industrial laboratories and others.

Impacts on the solution of this question come from the major usage of the system as a numerical supercomputer, specific decisions in the architecture of the system (cluster structures, distributed disk system, etc.), properties of both operating systems (Unix on the host and PEACE [1] on the Suprenum kernel), the abstract programming model and the user expectations.

The programming model is based on independent tasks exchanging messages. There is always an initial task and a set of node tasks. Usually the node task is based on the same program, but with different data. The communication model is asynchronous. This means for instance, a "send"-operation performs without explicit blocking and without an explicit acknowledgement.

The PEACE node operating system provides lightweight processes, organized into teams, a variable number of teams on a node, a rendezvous mechanism for interprocess communication, hardware-supported high-volume data transfer and a remote procedure call mechanism based on interprocess communication as basic primitives. Remote procedure call is embedded in a distributed name space concept. The various name spaces are connected to each other like directories in the file system. PEACE is optimized for fast process switches and fast network wide communication ([1,2]).

A task from the programming model is mapped to a PEACE team consisting of the application process, a mailbox server and some other servers like a name server and signal server. During the design process of the user interface questions came up both in connection with the systems as well as from the users point of view.

Typical questions were:
— In which way should jobs be executed in the sk, what is an acceptable granularity with respect to the jobs/node ratio?
— Host and node operating systems have to be connected. How should this be done?
— In which way are file-accesses handled in the distributed system?
— How can user-identification and other security mechanisms be extended to the Suprenum kernel?
— What is a good collection of tools for a system administrator to manage the system with respect to disk usage, handling of user jobs and so on?

Since Suprenum is a distributed project, different project teams worked together in order to solve these and related questions. A common forum, the "Suprenum user interface circle", was established in the spring of 1988. In this paper we collect the design decisions as results from the "Suprenum user interface circle" and experiences from the implementation of prototype systems. Currently, major design decisions are done and the implementation process is going on. Still there is no experience with a complete system.

Basic Design Decisions

The basic design decision is as follows:

- *The combined system is represented as a homogenous Unix-system to the user, with the Suprenum kernel as a specific (high-performance computing) device.*

Modifications in the underlying Unix system produce costs by adopting new releases. Also, there exist different Unix systems (System V, Berkeley-Unix, etc.). For Suprenum, the target host system is a System V (V.3) machine (the MPR 2300), but in the process of development, other Unix machines were used.

- *Search for portable solutions with respect to software running on the host.*

As a practical consequence we try to minimize the number of software components that run on the host system. Most servers are prepared to run in the Suprenum kernel. There is a distributed disk system, consisting of the host disks and the cluster disks. On each cluster disk there is a Unix file-system. Clearly the expectation is:

- *There is a unique logical file-system.*

The different cluster file-systems are mounted in the host file system. A typical path could be:

/sk/clu5/user/joe/pdesolver/euler/data44

Access to files in the Suprenum kernel from the host system is permitted in a command set including the usual file handling commands ([3]). The Suprenum system should run efficiently in different environments. Since we were not able to predict what would be the best way of system management for each possible environment, the decision was to:

- *Provide tools, so that the system can be configured to specific needs.*

Last but not least the following principle was adopted:

- *Increase overall system throughput.*

Single/Multi-Tasking System

The decision which type of single/multi-tasking system should be used is difficult. A typical Unix user would expect a timesharing system for the Suprenum kernel. Technically this is not impossible because the node operating system supports multi-tasking. On the other hand, supercomputer users tend to expect a batch system for performance reasons.

In a parallel system, one can be middle of the road by partitioning the set of nodes for several users. What is the smallest unit available for a job?

In the Suprenum context, this can be a single computing node or a cluster. Clearly a single node allows a more flexible handling of user requests, but also the management overhead increases. Currently, the smallest unit available for a user job is a cluster.

If a partition is assigned to a job A, then no other job is allowed to use the computing nodes of this partition during execution time of job A. On the other hand, a user working on the host system is allowed to move data in and out of the clusters which are assigned to job A. Also all types of system processes may run in the partition assigned to user A.

This decision increases overall throughput by a (usually) minor decrease of system performance for a single job.

Job Management

User jobs on the Suprenum kernel are started on the host by the skx command ([4]). By using options to this command, the user can request a certain number of clusters and other resources.

The skx part runs under Unix, it interprets the command line, initializes data structures for the job spooler, reads the (Unix-)environment, and controls the usage of files. Then, the skx command sends the job request to the job spooler.

The job spooler controls a fifo-queue of sk requests. There is a server called sk-manager ([5]). This component manages the Suprenum kernel in terms of actually available clusters, nodes and communication paths. By requesting information from the sk manager, the job spooler obtains information on which parts of the Suprenum kernel are available for job execution. If there is a job waiting and the resources are available the spooler forks itself and starts job-execution.

A job can be in one of three states: active, waiting or frozen. Frozen means that a distributed job is stopped and swapped out of the Suprenum kernel. There is a package of tools allowing a single user to manage owned jobs. The system administrator has his own access party to the configuration of the job manager. In this way, it is possible to constrain resources of the Suprenum kernel for specific users.

Access Rights

The usual identification and access mechanisms are extended to the Suprenum kernel. Files on cluster disks are owned by a specific user; this yields read, write and execution rights for the owner, the group and others. By starting a job, the job-executer initializes the name space of the job. There is a security-server for this job, which is connected to the job-name space. Also a file server is included in the name space. The file server may issue a getuid remote procedure call, which is replied by the security server with the user identification.

Environment

The Unix environment is read and sk-specific data are added by the skx-command. For each task this environment is at hand. By routines like getenv, putenv the environment can be read into user programs, manipulated and put back.

Finding Files in the Suprenum Kernel

It may occur that initially a task with file I/O will run on, say, cluster 1. Then for the second time, the task will run on, say, cluster 2. Now the problem is how to find the old file(s) from the first run. Our solution to this question is that logically a user is always at the same place in the file system during task execution, i.e. the current user directory of the host.

All files in the Suprenum kernel, newly created or modified during task execution, are mapped by symbolic links in the current user directory by the full path names. This is self-documenting and enables the file server system to find the files. By the environment mechanism a user can enter a *local* flag, which makes the local file server responsible for moving *remote* data to the local disk ([3,6]).

Connecting the Host and the Suprenum Kernel

Physically, there are special VME-boards that fit into the host system. This system is called CAC and it consists of a CAC/processor board and a CAC/Suprenumbus board. The latter connects the different clusters with the host system via the Suprenumbus, a 125-Mbit/s serial link. On the CAC/processor a special version of the PEACE operating system iruns (with special device drivers). Logically, parts of the PEACE environment are emulated in the Unix environment. This allows the initial task to run on the host system. On the other hand, the initial task can run in the Suprenum kernel. Then, only a few servers run in the Unix environment providing for file access, graphics support and so on, while most of the servers run in the Suprenum kernel. This is faster, but limitations like a fixed amount of money on a node in the Suprenum kernel become important.

Implementation of the User Interface

A system of servers is used to realize the user interface as described above. At the outer level there are sk and job manager for execution of jobs, the file server system including the various disk servers (one per cluster disk) and tty servers for atomic I/O operations on tty's. Included in the run time system there is one security server per userjob, individual file server per job name space, mailbox server for asynchronous communication, name and signal server and optional server for mapping, performance data delivery, accounting and related tasks. A distributed debugging system is available, which can be initiated during job execution.

References

1. Schroder, W., *Concepts of a Distributed process Execution and Communication Environment (PEACE),* Technical Report, GMD FIRST an der TU Berlin, 1986.

2. Schon, F. *Hochvolumen-Datentransfer in PEACE,* Technical Report, GMD FIRST an der TU Berlin, 1987.

3. Lange, D., *Wiederauffinden von Files in Suprenum Rechner,* Stollmann GmbH, Max-Brauer-Allee 79-81, D-2000 Hamburg 50, FRG, 1988.

4. Schaffler, G. and Janke-Martinez-Santelices, T., *Linking Suprenum Programs,* Stollmann GmbH, Max-Brauer-Allee 79-81, D-2000 Hamburg 50, FRG, 1988.

5. Janke-Martinez-Santelices, T., *Hochleistungskern-Manager Aufgabenstellung und Losungsansatz,* Stollmann GmbH, Max-Brauer-Allee 79-81, D-2000 50, FRG, 1988.

6. Lange, D., *Inpout/Output in MIMD Fortran 8x Programs,* Stollmann GmbH, Max-Brauer-Allee 79-81, D-2000 Hamburg 50, FRG, 1989.

Programming

36980245 Amsterdam SUPERCOMPUTER in English Mar 89 pp 25-30

[Article by Karl Solchenbach: "Suprenum-Fortran—An MIMD/SIMD Language]

[Text] Scientific codes for SUPRENUM are programmed in SUPRENUM-Fortran, a superset of standard Fortran 77. The SUPRENUM-specific language extensions are directly related to the SUPRENUM architecture. Process handling and message passing constructs support comfortable MIMD programming. In order to program the SIMD hardware of each node efficiently, array constructs (conforming to the coming Fortran 8x standard) can be used.

Fortran is still the most widely used programming language on supercomputers. This is mainly due to the fact that an enormous amount of technical and scientific software libraries are written in Fortran and nobody can afford to rewrite them. Several attempts have been made to replace Fortran (PL/1, Algol) but their success was limited. Fortunately, Fortran is evolving and has changed considerably compared to old Fortran 66. With the advent of the new Fortran 8x standard—which is currently heavily discussed—Fortran will be an up-to-date language.

Since Suprenum is focussing on technical and scientific applications it was an absolute necessity to provide Fortran for the Suprenum users. It was also clear that standard Fortran had to be extended by different features to support the Suprenum MIMD/SIMD architecture optimally.

The Suprenum-Fortran Language

The Abstract Suprenum Machine as described in [1] is characterized by MIMD parallel processing and SIMD vector processing within each process. In order to formulate parallel programs (in terms of processes) and to exploit the Suprenum hardware efficiently, standard sequential Fortran 77 turns out to be inadequate. A special Suprenum-Fortran language has been defined and a new compiler has been developed. Suprenum-Fortran is based on Fortran 77 [2] and extended by MIMD constructs, SIMD constructs, and miscellaneous extensions; these three are treated here.

The advantages of language extensions as compared to run time system subroutine calls are:
— They provide a greater flexibility in formulating the message passing objects. Subroutine calls require continguously stored messages.
— Fewer copy steps are required since the Suprenum node hardware can access and transfer vectors with constant increments (> 1).
— The compiler can do a better optimization. Overlapping of communication and calculation can be achieved by instruction scheduling.

The Suprenum approach of language extensions may be disadvantageous if portability of codes were an important issue. Portability of codes to other distributed memory systems with a simpler message passing interface, however, is guaranteed either on the level of communications libraries [1,3] or on the basis of the so-called Argonne message passing macros [4].

The complete and correct definition of Suprenum-Fortran is given in [5]. In the following sections, the most important extensions are described in a rather informal way.

MIMD Extensions

Process creation. A process or a set of processes is created by the NEW-TASK function call:

pid = NEWTASK (progname, cpuid)

progname is the name of a TASK PROGRAM which is written as a usual Fortran main program including a TASK PROGRAM progname header, and contains the code to be executed by the new process. cpuid is a node number or an array of node numbers which the new process(es) are loaded on. The nost numbers may be specified by the user or can be determined by the mapping library (see [1]). NEWTASK returns a TASKID (see below) value or an array of those, respectively. Analogously to the EXTERNAL statement, the progname has to be specified in a TASK EXTERNAL statement if it is passed as argument, e.g. in NEWTASK.

Process termination. A process is terminated if it terminates itself (STOP or END-statement), or if the initial process is stopped. Termination of the creating father process does not terminate the child process.

TASKID **data type.** Each generated process is associated with a 32-bit identifier of data type TASKID. Variables of that data type are used as addresses in SEND/RECEIVE-statements (see below) and they can only be defined by a NEWTASK function call. TASKID variables can be passed to subroutines and may occur in COMMON-blocks if they are not mixed with other data types. They must not occur in formatted I/O statements. The null value of this data type is the constant .NOTASKID.

Message passing. The basic message passing primitives are SEND and RECEIVE. The syntax is

SEND ([TASKID=] pid, [TAG=] tag) iolist

The brackets [ ] indicate optional keywords. pid is the process identifier of the addressed process. Multicast and broadcast is possible by specifying a set of process identifiers or ALL, respectively. The tag is an additional information (integer number) which is associated to the message. In addition to the address and the tag, the user can specify an error label and a status variable.

The iolist is a list containing expressions, implied do-loops (similar to those in READ and WRITE statements), arrays, and array sections (see below).

The syntax of the RECEIVE statement is similar:

RECEIVE ([TASKID=] pid, [TAG=] tag,
SENDER= sender) iolist

The parameters are similar to those in the SEND statement. The TAG specification is obligatory. The optional "SENDER=" specification returns the process identifier of the sending process which might be useful to know if the "TASKID=" specifier has been omitted. Error labels and status variables can also be specified.

The Suprenum message passing model is asynchronous:

— The sending process continues immediately after execution of the SEND and can overwrite data in the iolist. The sending process does not wait for the execution of the corresponding RECEIVE by the receiving process.
— Each process has a mailbox which contains all messages sent to this process and which have not been RECEIVEd yet. If a process wants to receive a message with a specified tag and (optionally) a specified sender and no message with matching data is in the mailbox, the process blocks until a matching message has arrived in the mailbox.
— Messages do not preserve temporal order. Messages from the same sender can be distinguished ony by the tag.

Selective receive. In order to avoid possible blocking of a receiving process, several expected process identifier and tag combinations can be specified in the WAIT statement

WAIT ([TAG=] tag, [TASKID=] pid,
COND= condition,
LABEL= label$_k$,...)
CONTINUE= label$_0$

The tag/pid/cond/label combination may be repeated and several labels label$_1$,..., label$_n$ may be specified. If a message is in the mailbox which matches to one of the tag/pid combinations and if the corresponding condition is true, the program branches to the associated label$_k$. If none of the tag/pid/cond combinations is fulfilled the program continues execution at label$_0$.

Inquiry functions. The following inquiry functions are provided:

— MYTASKID () gives the process identifier of the calling process;
— MASTER () gives the process identifier of the initial process;
— TESTTAG ([TAG=]tag) is a logical function which is .TRUE. if a message with the specified tag is in the mailbox, otherwise it returns .FALSE.;
— TESTMSG ([TAG=] tag, [TASKID=] pid) is a logical function which is .TRUE. if a message with the specified tag and pid is in the mailbox, otherwise it returns .FALSE.

TESTTAG and TESTMSG can be used to avoid blocking processes.

SIMD Extensions

SIMD vector processing within each process (and node) is supported by the array constructs as they are part of the new Fortran 8x standard [6], meanwhile also optimistically called Fortran 88 [7]. Although the standard has not been accepted finally by the responsible ANSI and ISO groups, some essential parts of the new Fortran including most of the array processing constructs can be regarded as stable. The formulation of an application program in terms of arrays and vectors instead in terms of (nested) loops is an important advance in programming vector computers. Using the new array notation, programming becomes more "object-oriented", the codes look clearer, and automatic vectorizers are more or less superfluous since in the array notation vectors are expressed explicitly.

In the following, only some important features of the array notation will be mentioned (see [8] for a detailed explanation).

Array properties. An array is a named set of contiguously stored data entities. All elements of an array must be of the same data type. Subsets of arrays are called array sections. Each array has a data type, a rank (the number of dimensions, less than or equal to 7), a size (total number of elements), and a shape (defined by the rank and number of elements in each dimension). Depending on when they are defined, arrays can be declared as explicit-shape, assumed-shape, assumed-size, or deferred-shape arrays.

Array subscripts. An array or an array section is referenced with one or more subscripts in a subscript list. The subscripts may be triplets (lower bound : upper bound : increment) or vector subscripts. Example:

```
REAL, ARRAY (20) :: A, B, C         ! new declaration statements
INTEGER IND(10)
IND = [4,3,2,1,9,10,8,7,6,5]        ! array constructor
A(2:20:2)=B(IND)+C(10:1:-1)         ! triplets and vector subscripts
                                    ! A(2)=B(4)+C(10),
                                    ! A(4)=B(3)+C(9),...
```

Dynamical allocation. Arrays can be declared without specified bounds. Execution of an ALLOCATE statement specifies the bounds and makes the array definable. The dynamic allocation can be used e.g. in parallel matrix or grid applications when the size of the submatrices or subgrids depends on the number of processes. This number is usually an input parameter and not known at compile time. Example:

```
REAL, ARRAY, ALLOCAT-      ! deferred shape
ABLE (:,:) :: GRID
...                        ! declaration
RECEIVE (...) NPROCX,      ! receive process
NPROCY
                           ! configuration
ALLOCATE GRID (0:NX/       ! dynamical allocation
NPROCX, 0:NY/NPROCY)
...
DEALLOCATE (GRID)          ! deallocation
```

Assignments. Before an assignment variable=expression is made, the expression is evaluated completely. Example:

```
A(1:10) = A(10:1:-1) ! reverses elements of array A
```

Conditional operations. Array assignments can be masked using the WHERE-statement. Example:

```
REAL A(10), B(10)
WHERE (A > 0.0) B =       ! the assignment is evaluated
LOG(A)                    only
                          ! where the elements of A are
                          > 0.
```

WHERE can be regarded as "vectorized" IF and can similarly be used as block statement together with ELSEWHERE and ENDWHERE.

An array assignment can be specified in terms of array elements using a FORALL statement. Example:

```
REAL GRID (0:M, 0:N)
GRID = 0.                        ! set all grid points
                                 ! to 0.
FORALL (I=1:M-1, J=1:N-1)        ! set interior grid
GRID(I,J) = 1.
                                 ! points to 1
```

Intrinsic functions. Fortran 8x provides new intrinsic functions related to array processing. Here are some examples:

```
DOTPRODUCT (VA,VB)     ! dotproduct of two 1D-
                       vectors
MATMUL (MA,MB)         ! matrix product of two
                       matrices
MAXVAL (A)             ! maximum value of all ele-
                       ments of A
MAXLOC (A)             ! location of maximum ele-
                       ment
SUM (A)                ! sum of all elements of A
ALL (MASK)             ! determines whether all ele-
                       ments
                       ! in MASK are true
```

Some functions can be called with optional DIM and MASK arguments. The use of the new intrinsic functions is recommended since they are implemented very efficiently on the Suprenum node either as assembler programs or in microcode.

In order to exploit the vector floating point unit of the Suprenum node efficiently, special vector instructions have to be generated. The compiler can generate these vector instructions only, if

1. the vectors are already formulated in the Fortran source using array notation as described above or

2. if a loop-based code is transformed by a vectorizer. Since not only new Fortran 8x programs but also "old" Fortran 77 programs should run on the Suprenum node, the Suprenum-Fortran vectorizer has been developed which works either as an integrated vectorizer in combination with the Suprenum-Fortran compiler or as a source-to-source transformer which generates readable Fortran 8x code.

Miscellaneous Extensions

Suprenum-Fortran covers most of the Vax and IBM extensions as DOUBLE COMPLEX, INTEGER*2 and BYTE data type and additional numerical intrinsic functions as COTAN, GAMMA, ERF. As soon as the BIT data type is finally defined in the Fortan 8x standard, it will be supported by Suprenum-Fortran.

References

1. Thomas, B. and K. Peinze, *Suprenum comfort of parallel programming,* Supercomputer, this volume.

2. *Fortran 77,* ANSI X3.9-1978.

3. Solchenbach, K., *Application software for Suprenum,* Supercomputer, this volume.

4. Lusk, E., et al., *Portable programs for parallel processors,* Holt, Rinehart, and Winston, New York, 1987.

5. *Suprenum-Fortran, reference manual,* Suprenum GmbH, Bonn, 1989.

6. *Fortran 8x,* Draft S8, Version 104 of X3.9-198x standard, April 1987.

7. *Fortran 88,* ISO/IEC JTC1/SC22/WG5 - N335, December 1988.

8. Metcalf, M. and J. Reid, *Fortran 8x explained,* Clarendon Press, Oxford, 1987.

Parallel Programming

36980245 Amsterdam SUPERCOMPUTER in English Mar 89 pp 31-43

[Article by Bernhard Thomas and Klaus Peinze: "Suprenum Comfort of Parallel Programming"]

[Text] Program development for the Suprenum multiprocessor is based on an abstract machine concept. Languages such as Suprenum-Fortran and a rich choice of tools support, and considerably facilitate, the writing, testing, and analysing of parallel applications in the frame of this programming model.

System architecture and software components of Suprenum have been described in detail in preceding contributions [1-3]. But in fact the programmer of a Suprenum system does not have to be aware of all these details when developing and running his/her software. This is one important issue of the Suprenum software concept, which is depicted in Figure 1.

Programming for the Abstract Machine

Application software development for Suprenum can be based on a very general programming model, the Abstract Suprenum Machine. The model allows program design in terms of concurrently executing, communicating processes and can be mapped in principle, to a wide range of MIMD-parallel, distributed memory systems. Suprenum, in particular supports this view on both hardware and system software level as well as by dedicated language extensions.

The Abstract Supreneum Machine comprises the following concepts:
— an application consists of a dynamical system of processes that are generated from independent program units;
— there is one *initial process* that initiates the distributed application;
— each process can create other processes at any time, termination of a process is an internal event, e.g. by executing a STOP;
— termination of the initial process, however, will end the distributed application;
— processes have access to private data space only;
— inter-process data requests are handled strictly by message passing;
— the process system can be of arbitrary structure (with respect to creation and interprocess-communication);
— vector and array processing can be programmed for within processes.

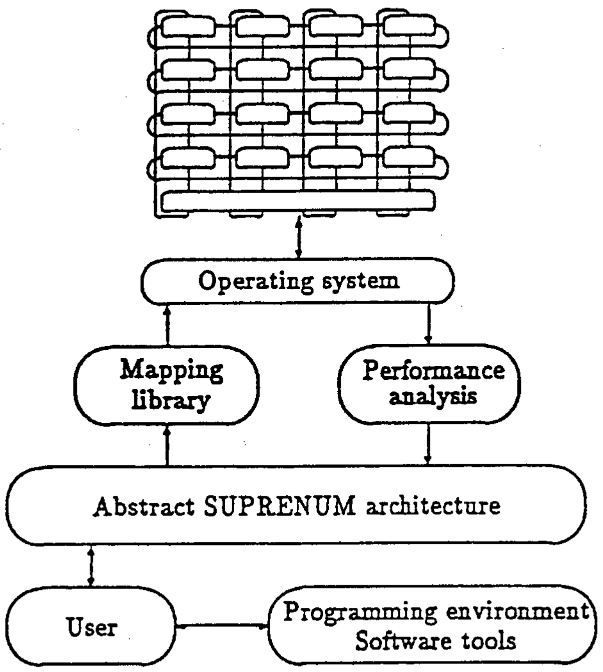
Operating system
Mapping library
Performance analysis
Abstract SUPRENUM architecture
User
Programming environment Software tools

Figure 1. The Suprenum software concept, a layered structure providing system transparency on various levels.

Obviously, configuration details such as the cluster structure or the interconnection system do not explicitly occur in the abstract machine context. There is, of course, a natural understanding of how these concept items might go together with a given hardware: processes will usually be thought of as executing on individual processors (or nodes), with the initial process being located on a front-end processor. Also, vectorized operations will be assigned to the vector floating-point unit in a node.

Yet there is no need to be concerned about these aspects in the course of program development. For example, an application designed to comprise a certain number of processes may actually run on a smaller number of processors (multi-processing on nodes), with lower performance, of course, but without any implications for the program development.

In a sense, the granularity of the individual processes should also be reflected, which indeed introduces a hardware aspect into the programming model. But again, this can be kept quite general, and may function merely as a guide to the complexity of the tasks to be performed in a process.

The layout of a parallel application in terms of the Abstract Machine concept is quite easy. The initial process is written as a single main program that will usually take care of creation and initialization of the process system (e.g. provide parameters and initial data) as well as for general I/O. All other processes are executable copies of one or several task programs, which are written according to the chosen parallelization strategy. In grid based applications, for example, the same task program usually codes for all necessary operations to be performed on a part of the grid (see the grid partitioning parallelization paradigm in [4]). In other applications there might be independent algorithmic components that can be programmed for concurrent execution. Creation of new processes and message based data exchange are programmed for whenever needed or wherever suitable in task programs of the initial program (dynamical process creation).

Within task programs attention should be paid to any portion of code that suits vector processing mode. Here the programmer may choose to utilize SIMD-parallelism within MIMD-parallelism by writing vector instructions explicitly.

The Abstract Suprenum Machine is a useful concept for mapping high level parallelism identified in an intended application onto an efficient process structure, before taking pains to code it in a particular programming language for a particular system. Nevertheless, a comfortable programming environment and suitable language primitives are needed to support this model.

Using Suprenum-Fortran

Suprenum provides two programming languages that directly provide this support by means of language extensions: Suprenum-Fortran and Concurrent Modula-2. Focusing on numerical applications, we will mainly consider Suprenum-Fortran throughout this contribution.

There are many reasons for choosing Fortran as a primary language in scientific computing, and these have been given in [3] along with a detailed description of the main extensions. Here we concentrate on the MIMD-oriented features.

The example below gives code fragments in Suprenum-Fortran that realize the following abstract model of a relaxation algorithm as it might be used in solving boundary value problems for elliptical partial differential equations:

Initial process
— get grid dimensions and problem parameters interactively;
— create 2-D array of processes executing the relaxation program (see node process) and send parameters of the application (e.g. identifications of neighboring processes, grid, subgrid and process array extensions) to either process as well as initial data on the corresponding subgrid;
— receive solution data from the processes and establish global results;
— stop.

Node processes
— receive parameters and initial data for local subgrid;
— perform computations, essentially the conventional relaxation routine;
— after each computational step, update values in points near inner boundaries by mutual exchange with neighbor processes;
— retrieve and send out global results (residual norms), preferably along a tree-like structure;
— send results to host.

The initial process is written as an initial task program whereas the node processes can be generated from one task program (called NODEPRG in this example). Note the ease of using tags (see [3]) to make asynchronous user data exchange between processes logically safe, and to ensure that computational parts are written in common Fortran 77 throughout and kept distinct from communication parts. A typical consequence of this strategy is that large parts of existing Fortran codes, e.g. written as subroutines, can be reused unaffected by parallelization requirements.

Host Program

```
C declarations of arrays etc.
        ...
C declaration of TASKID variables and arrays
        TASKID PID(..,..)
        TASKID SOUTH, NORTH, WEST, EAST
        ...
C initialize tags used in SEND's and RECEIVE's
        INTEGER TIN, TST, TSO, TRE
        DATA TIN/10/, TST/11/, TSO/12/, TRE/13/
        ...
C user input:
C process configuration  NPX x NPY, grid size NX x NY
C initial solution U, right hand side F
        READ(...) NPX, NPY, NX, NY, U, F
        NP=NPX * NPY
C compute size of subdomains
        IPSX=NX/NPX
        IPSY=NY/NPY
C create 2-D array of node processes from task program NODEPRG
        DO 10 IPX = 1, NPX
         DO 10 IPY = 1, NPY
          NEWTASK('NODEPRG', PID(IPX,IPY))
C compute index boundaries of subgrids
C and store to index arrays IX, IY
        ...
 10   CONTINUE
C distribution of parameters of the application
        DO 20 IPX = 1, NPX
         DO 20 IPY = 1, NPY
C identify neighbors of process in IPX, IPY
          SOUTH = .NOTASKID.
          IF(IPY.NE.1) SOUTH=PID(IPX,IPY-1)
          NORTH = .NOTASKID.
          IF(IPY.NE.NPY) NORTH=PID(IPX,IPY+1)
          WEST  = .NOTASKID.
          IF(IPX.NE.1) WEST=PID(IPX-1,IPY)
          EAST = .NOTASKID.
          IF(IPY.NE.NPX) EAST=PID(IPX+1,IPY)
C send parameters and process specific information
C to process in IPX, IPY
          SEND (TASKID=PID(IPX,IPY), TAG=TIN)
     &        IPX, IPY, IX(IPX, IPY, 1:2), IY
     &        SOUTH,NORTH,WEST,EAST
 20   CONTINUE
C send initial data  right hand side
        DO 30 IPX = 1, NPX
DO 30 IPY = 1, NPY
          SEND (TASKID=PID(IPX,IPY), TAG=TST)
     &        U(IX(IPX,IPY,1):IX(IPX,IPY,2),
     &        IY(IPX,IPY,1):IY(IPX,IPY,2)),
     &        F(IX(IPX,IPY,1):IX(IPX,IPY,2),
     &        IY(IPX,IPY,1):IY(IPX,IPY,2))
 30   CONTINUE
C receive solution in arbitrary order
        IP=0
        IF(IP.LT.NP) THEN
         RECEIVE (TAG=TSO) IPX, IPY,
     &       U(IX(IPX,IPY,1):IX(IPX,IPY,2),
     &       IY(IPX,IPY,1):IY(IPX,IPY,2))
         IP = IP+1
        ENDIF
C receive residual norms in arbitrary order
        IP=0
        RES = 0.D0
        IF(IP.LT.NP) THEN
         RECEIVE (TAG=TRE) RESLOC
         RES = MAX (RES, RESLOC)
         IP = IP+1
        ENDIF
C postprocessing
        ...
C end of host program
        ...
        STOP
        END
```

Node Program:

```
C declarations of arrays, TASKID variables etc.
        ...
        TASKID SOUTH, NORTH, WEST, EAST
C initialize tags
        INTEGER TIN, TST, TSO, TRE
        DATA TIN/10/, TST/11/, TSO/12/, TRE/13/
C receive parameters and initial  information
        RECEIVE(TAG=TIN) IPX, IPY, IX, IY,
     &        SOUTH, NORTH, WEST, EAST
C receive initial data and right hand side
        RECEIVE(TAG=TST) U(IX(1):IX(2), IY(1):IY(2)),

        F(IX(1):IX(2), IY(1):IY(2))
C iterative loop (pre-assigned number of passes: MAXIT)
        DO 10 IT = 1, MAXIT
C subroutine RELAX contains the usual sequential program
         CALL RELAX(U, F, ...)
C data exchange across inner boundaries by message passing
         TEX = 100 + IT
         SEND(TASKID=WEST,TAG=TEX) U(IX(1),IY(1):IY(2))
         SEND(TASKID=EAST,TAG=TEX) U(IX(2),IY(1):IY(2))
         SEND(TASKID=SOUTH,TAG=TEX) U(IX(1):IX(2),IY(1))
         SEND(TASKID=NORTH,TAG=TEX) U(IX(1):IX(2),IY(2))
         RECEIVE(TASKID=WEST,TAG=TEX) U(IX(1)-1,IY(1):IY(2))
         RECEIVE(TASKID=EAST,TAG=TEX) U(IX(2)+1,IY(1):IY(2))
         RECEIVE(TASKID=SOUTH,TAG=TEX) U(IX(1):IX(2),IY(1)-1)
         RECEIVE(TASKID=NORTH,TAG=TEX) U(IX(1):IX(2),IY(2)+1)
C end of iterative loop
 10   CONTINUE
C send local solution to host
        SEND (TASKID=MASTER(), TAG=TSO) IPX, IPY,
     &        U(IX(1):IX(2),IY(1):IY(2))
C send local residual norms to host
        CALL RESID(U, F, RES, ...)
        SEND (TASKID=MASTER(), TAG=TRE) RES
         ...
C end of node program
         ...
        STOP
        END
```

Clearly, within the computational parts, SIMD-parallelism can be exploited by using appropriate vector notations as provided by Suprenum-Fortran. Besides, the Suprenum-Fortran compiler includes an autovectorizer (see [3,5]).

Employing Libraries

As it was noted in the previous section, collecting and sending out global data from a grid of processes could be done most efficiently in a treewise fashion. This would imply treating the collection of processes as a tree-structure rather than a grid. On Suprenum, multi-structured process systems are supported, being an issue of the Abstract Machine model view.

For example the grid structure laid out in a 2-D TASKID array in the example program fragment above can easily be supplied with an additional tree structure by introducing a father/left-son/right-son scheme of TASKIDs in each process.

Structuring process systems and having process structures distributed across available processors is particularly facilitated by the *Mapping Library*. The library can be involved to either establish:

— for a particular process set, one of the elementary topologies (trees, rings, cubes, etc.) or a general graph topology specified by a (weighted) adjacency matrix;
— automatic process placement, where a new process is loaded onto a new processor taking current workload into account;
— semi-automatic process placement, where the programmer may specify a new process to be placed onto the same processor or same cluster or elsewhere.

Whereas the first two items are automatic mappings, the third one along with explicit process placement enables the user to directly control where a process has to go ([6]). It is noted that programming can use various levels of transparency with respect to the underlying hardware according to the special requirements of the programmer.

Thus, if the programmer does not want to take care of process structuring and placement, he might resort to mapping library routine calls. If he prefers not to worry about process creation and communication at all, he might even write programs completely in Fortran 77 (probably including Fortran 8x notations provided by Suprenum-Fortran) by relying on *Communication Library* calls.

For the current version of the communication library, this is only meant to work for problems where grid partitioning is the parallelization paradigm. However, they are numerous and occur in various fields of applications (see [4]). The table below lists example routines that would correspond to whole parts of the relaxation program above.

Initial Task	Task	Comment
crgr2d	grid2d	creation of 2-D grid of processes generated from specified task program; also generates logical tree structure; transfer of initial information
	supdt2	send/receive values in boundary area of specified width
	rupdt2	to specified neighbors (2-D problems)
gloph	glops	compute global values from local ones according to specified arithmetic operation; distribute and receive results (initial task: receive only)
	agglm2	agglomeration for 2-D process grids; maps logical process grid onto another one of different size

Essentially, 2-D and 3-D process grid and tree structures are created simply by subroutine calls. It is worth mentioning that all the necessary process identification management (by data of type TASKID) is completely hidden in the subprograms of the communication library. Communication across inner boundaries is done by calling boundary data exchange routines, where the depth of the boundary layer as well as an ordering of points can be respected. Several other services are accessible according to needs, including collection and sending out global values tree-wise, and agglomeration, a strategy applied in multi-processor implementations of multigrid methods.

Besides the aspect of merely programming in plain Fortran, there are several general advantages of employing the communication library in grid-oriented numerical software:
— programming is safer using well tested and optimized routines for data exchange across process grids;
— redundant coding is considerably reduced;
— software becomes portable for a wide class of multiprocessor systems, since only the library has to be adapted, which can be done safely, leaving the user program essentially unaltered, on date implementations have been done on iPSC, Ncube, and even to a shared memory system ([7]).

Writing Programs

Program editing can be done on the front-end system or, equally well, on remote machines like Unix workstations. In any case, the programming environment is Unix-based [2] and thus familiar to most numerical programmers.

Besides, there is a comfortable, language-dependent programming system which allows maintaining sources, keeping version control, facilitating code writing and doing syntax checks even on program fragment level. The programming system is operated through a window-based look-and-feel type surface and makes programming much easier and safer. Details are given in [8].

With a set of files constituting the parallel program (e.g. an initial task program relaxh.f and associated task programs, e.g. relaxn.f), ways may now temporarily diverge: the programmer might either hope for a fault-free, well-performing program and push it forward onto the Suprenum multiprocessor (see below). Or, he might take a more careful step in using the Suprenum simulator instead, which runs on the front-end or a Unix workstation.

Simulations With SUSI

SUSI, the Suprenum simulating system, is an implementation of the Abstract Suprenum Machine, and can be used to simulate a parallel application on a conventional architecture. SUSI consists of language dependent preprocessors, a hardware configurator, a user interface, runtime systems for both languages (Suprenum-Fortran and Concurrent Modula-2), and the actual scheduling and execution kernel.

The *preprocessor* is invoked for the Suprenum-Fortran example by:

```
mf77 [mf77-options] [f77-options] [ld-options]
relaxh.f relaxn.f
```

where the list of program files might be preceded by the common f77 and linker options as well as by special preprocessor options which include a reference to the mapping library. mf77 resolves MIMD-constructs in Suprenum-Fortran into subroutine calls, compiles them, and links them with SUSI's runtime system. The simulator kernel takes care of scheduling processes and object dispatching based on coroutine concepts from Modula-2. Another version of the simulator currently under development is based on an emulation of the basic distributed operating system PEACE [1].

The SUSI *user interface* provides a flexible and comprehensive control of the simulation run, and is able to extract all sorts of statistics and trace data on activities of such objects like user processes, CPUs, busses, mailboxes, etc. To begin with, the *hardware configurator* expects input of a hardware configuration, to which the application will be mapped. This is usually contained in a hardware description (HADES) file, which might be specified as in the following example, or be generated by a graphical tool (HADESGEN).

Systems liks SYS1 and SYS2 are "plugged together" from clusters, intercluster busses, CPUs and cluster busses to taste.

After starting the simulation run, SUSI's interface prompt will show up. The simulation may then be run for a specified period of simulation time by issuing commands like:

```
start simtime=0.2
```

It may be interrupted by CTL-C or after simtime elapsed and continued, eventually with new options set, or

```
TYPE
  Cpu1 - CPU        MIPS - 2.0;     TSLICE - 0.05 END (* cpu *);
  Bus1 - BUS         MBITS - 30.0; END (* bus *);
  Bus2 - BUS         MBITS - 300.0;END (* bus *);
  Icb1  - ICB        MBITS - 24.0; END (* inter-cluster-bus *);
  Icb2  - ICB        MBITS - 512.0;END (* inter-cluster-bus *);
  Clu1 - CLUSTER    Cpuj [8] : Cpu1;
                    Busj      : Bus1;END (* cluster *);
  Clu2 - CLUSTER    Cpuk [8] : Cpu1;
                    Busk      : Bus2;END (* cluster *);
  H1   - HOST       Cpul [1] : Cpu1;
                    Busl : Bus1;END (* host *);
  H2   - HOST       Cpum [1] : Cpu1;
                    Busm : Bus2;END (* host *);
  SYS1 - LATTICE        (* system configuration *)
           #    #  Icb1  Icb1 ;
           #    #  H1    #   ;
           Icb1 #  Clu1  Clu1 ;
           Icb1 #  Clu1  Clu1 ;
        END (* lattice *);
  SYS2 - LATTICE  (* another system configuration *)
           #    #  Icb2  Icb2 ;
           #    #  H2    #   ;
           Icb2 #  Clu2  Clu2 ;
           Icb2 #  Clu2  Clu2 ;
        END (* lattice *);
END.
```

stopped to terminate the simulation. Whenever appropriate, a variety of trace information on all kinds of events can be switched on or off. As an example, by giving

```
trace on msg all (up=system)
```

listings like the one below will be output from SUSI

```
0.003202 UP001CPU000: multicast No=  9 sent: PAT/TAG=1
0.003230       CPU001: multicast No=  9 deliv. to UP002CPU001
                                 OrigNo= 9 PAT/TAG=1
0.003230       CPU002: multicast No= 13 deliv. to UP003CPU002
                                 OrigNo= 9 PAT/TAG=1
0.003230       CPU003: multicast No= 14 deliv. to UP004CPU003
                                 OrigNo= 9 PAT/TAG=1
0.003230       CPU004: multicast No= 15 deliv. to UP005CPU004
                                 OrigNo= 9 PAT/TAG=1
0.003340 UP002CPU001: multicast No=  9 recv from UP001CPU000
                                 OrigNo= 9 PAT/TAG=1
0.003340 UP003CPU002: multicast No= 13 recv from UP001CPU000
                                 OrigNo= 9 PAT/TAG=1
0.003340 UP004CPU003: multicast No= 14 recv from UP001CPU000
                                 OrigNo= 9 PAT/TAG=1
0.003340 UP005CPU004: multicast No= 15 recv from UP001CPU000
                                 OrigNo= 9 PAT/TAG=1
0.003522 UP001CPU000: userdata  No= 41 sent to   UP002CPU001
                                 PAT/TAG=2
0.003609       CPU001: userdata  No= 41 deliv. to UP002CPU001
                                 PAT/TAG=2
0.003719 UP002CPU001: userdata  No= 41 recv from UP001CPU000
                                 PAT/TAG=2
0.003837 UP001CPU000: userdata  No= 42 sent to   UP003CPU002
                                 PAT/TAG=2
0.003924       CPU002: userdata  No= 42 deliv. to UP003CPU002
                                 PAT/TAG=2
0.004034 UP003CPU002: userdata  No= 42 recv from UP001CPU000
                                 PAT/TAG=2
```

The complete range of commands and options exceeds the scope of this contribution; for further reference see [9,10]. Graphic tools (as described below) can be used to process trace data and facilitate the understanding of what the parallel application actually does as a process system.

SUSI is employed to do logical testing of the complete parallel application, to get a feeling for loads on CPUs, busses, etc., for the communication paths, inappropriate message scheduling, and deadlocks. It provides some rough estimates on resource utilization and efficiency.

If it is felt that the program is perfect, with respect to SUSI, one wants to see real performance by running it on the actual hardware. In most applications that promise good parallel efficiencies on theoretical grounds, the problem will be scalable. That is, to do real life calculations the user will only have to change some (interactively set) parameters that describe the size of the application, e.g. the total grid size and the size of the subgrids within one process. There will be no rewriting of any part of the code to switch from SUSI to Suprenum hardware in such cases.

Running Programs on a Real Configuration

The Suprenum user interface (see [2]) provides the necessary commands and services to let a user run his application on real hardware without having to bother about multi-processor specific details such as resource allocation, job control, downloading and the like in depth. Since the Suprenum system is perceived by the user as a homogeneous Unix-system with the Suprenum kernel hardware appearing as a specific device, conventions for e.g. file access, job executing, and querying the status of the system are quite familiar. In particular, details about the kernel operating system (PEACE, see [11]) are not visible on the user interface level.

To run a program that has been compiled and linked by the Suprenum-Fortran compiler (see [3]), the job manager can be invoked by the Suprenum kernel execute command skx as simply as e.g.

```
skx -n 4 -t 20:00 relax
```

Here, relax is assumed to be the executable file generated from relaxh.f and relaxn.f in the above example. Option -n claims 4 clusters to be needed for the run, and -t gives the time limit. A detailed discussion is given in [2].

Keeping Informed

Besides displaying the actual user program output which may make use of GKS and X/Windows based tools on the front-end there are several facilities that provide information about what is happening on the allocated part of the machine. Among these are:
a parallel debugger;
performance analysis and profiling;
state reporting,

which can be invoked together with the execution command.

The state reporting facility, in particular, will extract data on the process and message passing activities of a distributed application running on Suprenum hardware, similar to the trace data generated by SUSI's trace options.

A set of visualization tools operate on such data streams to produce a comprehensive graphical representation of the overall behaviour of the application: the *dynamic map*, the *time map*, and the *statistic map*.

All three of them take a stream of standardized event descriptions as input. A *filter* program extracts relevant information either from SUSI traces or from the state reporting facility and condenses it into this standard event format. What information is considered to be relevant as well as hints where to find it in a line of information can be specified by the user in a filter description file that acts as an interface between the various sources of process data and the filter program.

The dynamic map produces an animated view of the activities of processes, processors, mailboxes, and of the interconnection system along with information on sent-out or incoming messages. The animation can be slowed down to follow through the actions of the system, e.g. right before a deadlock situation.

The time map produces a comprehensive temporal overview of the behavior of the complete application. It displays a record of states of processes and associated mailboxes as well as of process creation and intercommunication over the full time-interval. The graphics provide a "first glance" of the communication schedule and active-to-inactive time ratio of processes. Analyzing the display in more depth can provide useful hints to inefficient organization of the communication and lead to improved tuning of the application.

The statistic map generates a final evaluation of the reported data and displays some statistics on the performance of a distributed application. It computes figures, and displays them graphically, for e.g. the load of processors, number of interprocess communication events, occupation of mailboxes, as well as time period spent on message passing activities in much details.

An overview of the complete visualization system is given in [12]. The visualization can be run concurrently (in the sense of Unix pipelined processes) with the application by establishing a suitable pipeline, including the filter. However, it usually does not make much sense to monitor behavior of an application with these tools on-line, unless it is run as a simulation. Instead, reported data will be written on file, either on cluster disks and/or on external mass memory, for further postprocessing by the visualization tools. Hard copies of the visualization can be made most easily with a suitable color graphics printer.

References

1. Giloi, W., *Suprenum—the system,* Supercomputer, this issue.

2. Werner, K., *The Suprenum user interface,* Supercomputer, this issue.

3. Solchenbach, K., *Suprenum-Fortran—an MIMD/SIMD language,* Supercomputer, this issue.

4. Solchenbach, K., *Application software for Suprenum,* Supercomputer, this issue.

5. Trottenberg, U., *Suprenum—the concept,* Supercomputer, this issue.

6. Kramer, O., *Suprenum mapping library—user manual,* GMD, St. Augustin, 1987.

7. Hempel, R., *The Suprenum communications subroutine library for grid-oriented problems,* Mathematics and Computer Science Division, Argonne National Laboratory, Argonne, Il, 1987.

8. Thies, Ch., *Anleitung zum Benutzen von PSG-Programmierumgebungen,* Report PI-R9/87, TH Darmstadt, FB 20, 1987.

9. Limburger, F., Ch. Scheidler, Ch. Tietz and A. Wessels, *Benutzeranleitung des Suprenum-Simulationssystems SUSI,* GMD, St. Augustin, 1986.

10. Tietz, Ch., *Das Benutzer-Interface des Suprenum-Simulationssystems,* GMD, St. Augustin, 1987.

11. Bast, H.-J., M. Gerndt and C.-A. Thole, *SUPREB—The Suprenum Parallelizer,* Supercomputer, this issue.

12. Thomas, B., E. Thomas and E. Truchet, *Suprenum visualization tools for distributed applications—user's guide,* Suprenum Report 12, Bonn, 1988.

Application Software

36980245 Amsterdam SUPERCOMPUTER in English Mar 89 pp 44-50

[Article by Karl Solchenbach: "Application Software for Suprenum"]

[Text] About one third of the SUPRENUM development resources has been spent for implementation of application software packages, mainly from scientific computing as CFD and statistical physics. Some of the codes are based on sequential versions which have been "parallelized", others have been written completely new. The paper gives an overview over the SUPRENUM application codes and sketches briefly the underlying parallelization techniques.

The availability of practical relevant application software is decisive for the scientific and commercial success of a new computer architecture like Suprenum. This software must make use of the specific advantages of the architecture and translate these advantages into gains of speed. It is, however, not desirable to use old numerical methods on advanced computer architectures. Only efficient numerical algorithms in combination with the parallel Suprenum hardware can provide the computing performance which is required for large scale scientific and technical simulations. Consequently, roughly one third of the Suprenum development resources have been spent for application software development which covers the implementation of new algorithms as well as the parallelization of existing codes.

The application software for Suprenum is based on the Abstract Suprenum Architecture (see [1]), i.e., the parallel programs are formulated in terms of parallel processes. The number of processes and their topology (defined by the message passing communication) are primariy prescribed by the numerical problem and they are independent of the actual hardware configuration.

The programming language for nearly all of the application packages is Suprenum-Fortran (see [2]). Before the hardware was available, the application software development was done on the Suprenum simulator.

Application Software Packages

Linear Algebra Package

Suprenum provides parallel algorithms for linear algebra computations including:
- vector and matrix operations;
- elimination methods for linear systems with dense matrices (Gauß, Cholesky);
- elimination methods for linear systems with banded matrices (reduction methods);
- solution of eigenvalue problems;
- iterative solvers for sparse systems (conjugate gradients, incomplete decompositions, ADI, block relaxation).

The interface to the dense matrices solvers is an extension of the LIN-PACK interface.

Multigrid Software

A library of multigrid solvers for elliptic boundary value problems is available on Suprenum. The partial differential equations are of the type

$$\nabla \cdot (D \nabla u) + cu = f$$

Different classes of coefficients and boundary conditions can be selected. The domain is a 2-D or 3-D cube. The solvers are based on highly efficient parallel multigrid algorithms. Due to their high degree of parallelism they can be parallelized (across the Suprenum nodes) and can be vectorized (within each node).

Computational Fluid Dynamics (CFD)

Potential solver. Because of its comparatively small computational work the potential equation is still an attractive model used very frequently for aerodynamical applications where only limited accuracy is required. It can be applied for subsonic as well as for transonic flow. It is, however, principally restricted to irrotational and inviscid flow.

Its mathematical formulation is a nonlinear scalar PDE which is elliptic in the subsonic flow areas and hyperbolic in the transonic areas. The problem is solved numerically by a parallel multigrid code which provides a special treatment of the sonic shock curve.

Euler solver. The Euler equations are the standard model for the description of inviscid flows. They are used for all simulations in aerodynamics where the viscosity can be neglected. Mathematically they form a coupled nonlinear system of PDEs for the flow-velocity components, the pressure, and the total energy.

Often it is necessary to take viscous effects into account near boundaries whereas they are negligible in zones far away from the body surface. This leads to a combination of the Euler equations with special boundary layer approximations.

Navier-Stokes solver. The most general CFD models are the (compressible) Navier-Stokes equations which principally describe all flow phenomena governed by macroscopic physical rules. Mathematically they form a system of PDEs similar to that of the Euler equations with additional viscous terms.

Suprenum offers the parallelized version of the established code Ikarus (by Dornier) for the compressible case and the completely new Navier-Stokes solver Liss (by GMD)—based on multigrid methods and designed for Suprenum—for incompressible calculations.

Grid generation. Most, interesting flow phenomena are related to geometrically complex domains with curved boundaries. Typical examples are the exterior space around wings, airplanes or cars or the interior space in pipes, turbines, etc. In order to discretize the mathematical model (i.e., the system of PDEs) by finite differences or finite volumes one needs a grid with the following properties:
— good resolution (especially near boundaries);
— simple and accurate discretization of boundary conditions;
— logically simple structure.

Suprenum offers 2-D and 3-D grid generators for boundary fitted grids (based on Thompson's method [3]) with graphical interfaces.

Other Applications

Besides the CFD applications many different application software packages have been adapted to Suprenum. Here we mention some of them:

- *Structural analysis.* A finite element code (PERMAS) is currently adapted for Suprenum. The sequential linear solver (Cholesky) is replaced by a parallel version.
- *Quantum chromodynamics (QCD).* These very time-consuming simulations can optimally be mapped on Suprenum. An SU(2) code is already running; the SU(3) code is under development and expected to run with nearly 2 Gflop/s.
- *Reactor safety.* In the framework of the Suprenum project, a thermohydraulic code for the simulation of a nuclear reactor core is developed. Another ongoing activity in this area is the parallelization of the Relap5 code which simulates the cooling circuit within a nuclear power plant.

Parallelization

Many of the different applications mentioned in the previous section are based on either matrix-vector or grid data structures. This is not surprising since the underlying mathematical model consists of PDEs and their discretization most naturally leads to grid structures or at least to large matrices.

Parallelization requires the selection of parallel algorithms and the distribution of the data structure to the local memory units. The data distribution should try to preserve *locality* and to achieve *load balancing.*

Matrix Based Applications

The basic data structures for linear algebra calculations are matrices and vectors. Depending on the particular algorithm, matrices are distributed in rows, in columns or—the most general method for dense matrices—in blocks (submatrices). The distribution is chosen according to the following requirements:
— minimal number of communication steps;
— minimal length of communicated data;
— maximal vector length in each process.

Vectors are distributed conforming to the matrix distribution. Complete redistributions (matrix transform) should be avoided whenever possible.

Parallel algorithms. The linear algebra algorithms can be parallelized on block level, i.e., submatrices are treated independently and simultaneously. In case of matrix multiplication this can be done in a straightforward way. In the case of Gaubian elimination the dependency on pivot elements has to be considered. Within each process the algorithms are based on vectorized BLAS routines.

Grid Based Applications

The Suprenum application packages support two classes of grid structures:
— *Regular grids* are characterized by direct addressing of the grid points and a rectangular or cuboid address space. Geometrical neighbors are also logical neighbors.
— *Block-structured grids* are composed of several regular grids. Each single block shows internally a regular grid structure; the block structure itself, however, is irregular (with certain restrictions).

In future, also codes based on irregular grids (as used by finite element methods) and locally refined grids will be implemented on Suprenum.

Parallel grid algorithms. A grid algorithm is a (usually iterative) method which calculates the value of a grid function at one point as a function of values defined at neighboring points (also called relaxation). The iteration can be characterized as Jacobi-type (the new iterate at a grid point is calculated using only old neighboring values) or Gaub-Seidel-type (using already calculated new neighboring values). Obviously, Jacobi-type methods are completely parallel since the calculation in each grid point can be performed independently (see Figure 1a). If the number of grid points is *N* the parallelism is also *N*.

The parallelism of Gaub-Seidel methods depends on the order in which the grid points are processed. Lexicographic ordering implies that only points on "wave fronts" can be calculated in parallel (see Figure 1b).

For Gaub-Seidel methods, a far better degree of parallelism, namely *N*/2, is obtained by "coloring" the grid points appropriately and processing all points of the same color simultaneously, e.g. the so-called red-black relaxation (see Figure 1c).

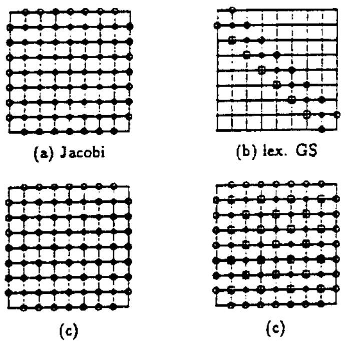

Figure 1. Jacobi- and Gaub-Seidel relaxation schemes.
● denotes grid points which can be calculated independently in parallel, o denotes grid points with old values, and □ denotes grid points with already calculated new values.

Although the range of grid algorithms for CFD applications varies widely they all can be regarded either as Jacobi-like (to these belong explicit time-stepping schemes) or as Gaub-Seidel-like or as a mixture of both. The parallelism of the grid algorithms in all cases is sufficiently high for highly parallel systems.

The same applies if instead of point relaxation schemes (as described above) line or plane block relaxations are performed, where the values of a whole line or plane of grid points are updated simultaneously. The implementation of parallel grid applications on Suprenum is based on the method of grid partitioning (often called domain decomposition).

Multigrid methods. Standard iterative multigrid algorithms process a cycle from the fine to the coarse grids and back to the fine grids sequentially, whereas on each grid level the actual problem is treated in parallel similarly to the parallel single grid algorithms.

The algorithmic and technical details of parallel multigrid algorithms are described in [4,5].

Communications library. For grid applications, the explicit programming of the communication can be hidden from the user. In the Suprenum project, for example, a library of communication routines has been developed [6] which ensures:
— clean and error-free programming;
— easy development of parallel codes;
— portability within the class of distributed memory computers, programs can be ported to any of these machines as soon as the communication library has been implemented.

The library supports regular and block-structured grids and is used by most of the Suprenum applications.

Performance

The quantities of interest in evaluating the performance of parallel algorithms are:
— time $T(N,P)$: time to solve a problem of size N on a multiprocessor system using P nodes;
— speed-up $S(N,P) := T(N,1)/T(N,P)$;
— efficiency $E(N,P) := S(N,P)/P$.

Note that on the Suprenum the utilization of the hardware capabilities is the product of the "multiprocessor" efficiency as defined above and the efficiency related to the vector processing unit. The total problem solving time—which is the only interesting number from the user's point of view—depends, of course, additionally on the numerical efficiency of an algorithm.

In practice *E* will be smaller than its ideal value 1, mainly because of communication (including synchronization), unbalanced load, and sequential parts in the algorithm (Amdahl's law). It is often claimed that the speed-up on parallel systems is limited due to Amdahl's law. This does, however, only apply if a constant problem is distributed to more and more processors. Realistically, the applications are scalable, i.e., the parallel fraction of the program increases as the problem size increases.

Since the sequential part of such a program is less dependent on the problem size, its fraction is not constant and the assumptions of the classical form of Amdahl's law are not fulfilled (see [7]).

Performance estimates. A simple analysis shows that asymptotically for matrix and grid based applications:

$$S(N,P) \rightarrow P,\ E(N,P) \rightarrow 1$$

if P is fixed and $N \rightarrow$ infinity. For many grid and matrix algorithms E depends mainly on N/P, i.e. the size of the submatrices or subgrids.

Estimated performance results for CFD applications are given in the next two tables.

3-D potential solver (parallel version of FLO22) with N approximately equals 200,000 grid points:

	$P = 16$	$P = 64$	$P = 256$
$E(200{,}000,P)$	0.98	0.97	0.85
Mflop/s	75	300	1040

The table shows that for realistic CFD problems a performance of more than 1 Gflop/s can be expected on Suprenum.

2-D incompressible Navier-Stokes solver:

$N =$	16384	65536	262144
$E(N,256)$	0.62	0.85	0.95

As predicted by the asymptotic analysis, the efficiency is increasing with growing problem sizes.

References

1. Thomas, B. and K. Peinze, *Suprenum comfort of parallel programming,* Supercomputer, this volume.

2. Solchenbach, K., *Suprenum-Fortran—an MIMD/SIMD language,* Supercomputer, this volume.

3. Thompson, J., Z. Warsi and C. Mastin, *Boundary-fitted curvilinear coordinate systems for the solution of partial differential equations on fields containing any number of arbitrary two-dimensional bodies,* NASA report CR-2729, Washington, D.C., 1977.

4. Thole, C. and U. Trottenberg, *A short note on standard parallel multigrid algorithms for 3-D problems,* in: Lichnewsky, A. and C. Saguez (eds.) Supercomputing, North-Holland, Amsterdam, 1987.

5. Solchenbach, K., C.-A. Thole and U. Trottenberg, *Parallel multigrid methods: implementation on Suprenum-like architectures and applications,* in: Houstis, E., T. Papatheodorou and C. Polychronopoulos (eds.), Supercomputing, 1st International Conference, Lecture Notes in Computer Science 297, Springer Verlag, New York, 28-42, 1988.

6. Hempel, R., *The Suprenum communications subroutine library for grid-oriented problems,* Report ANL-87-23, Argonne National Laboratory, 1987.

7. Gustafson, J., G. Montry and R. Benner, *Development of parallel methods for a 1024-processor hypercube,* SIAM J. Sci. Stat. Comp. 9, 4, 609-638, 1988.

SUPERB Parallelizer

36980245 Amsterdam SUPERCOMPUTER in English Mar 89 pp 51-57

[Article by Heinz-J. Bast, Michael Gerndt, and Clemens-A. Thole: "SUPERB—the Suprenum Parallelizer"]

[Text] Although automatic vectorization is a well-known technique, automatic transformation of sequential programs for MIMD execution on distributed memory architectures, like a Suprenum, is a research topic. The real problem is not the detection of MIMD parallelism but the detection of locality in the memory references. From the application point of view two basic kinds of locality for memory references are distinguished: matrix-type and grid-type. The basic task for automatic transformation tools for distributed memory architectures and computers with memory hierarchies is outlined. The interactive system SUPERB is oriented to the parallelization of grid-type problems. The design of this system for semi-automatic transformation of Fortran 77 programs into parallel programs for the Suprenum machine is given. The system is characterized by a powerful analysis component, a catalog of MIMD and SIMD transformations, and a flexible dialog facility.

The Challenge of Automatic Parallelization

Parallel programs for parallel architectures can be created by explicit formulation of the parallelism using special language constructs (e.g.Suprenum-Fortran) or special language semantics (e.g. functional languages). The application programmer would prefer the automatic detection of parallelism in sequential programs written in standard Fortran.

In the case of SIMD parallelism as it is used by conventional vectorcomputers the generation of vector instructions from loops is well known. Comparison of the automatic vectorizing compilers of different vendors shows the very high quality of their products [1].

For some architectures compilers supporting MIMD parallelism can be used (e.g. Alliant FX/xx, Convex 2xx, Cray-2 and Cray X/Y-MP systems). All of these architectures are shared memory computers. The automatic parallelization uses different levels of nested loops or strip mining to generate vector instructions to be executed in parallel. The work is assigned to processors either in portions of fixed size or dynamically in several smaller portions to improve the balance of the loads [2-4]. In [5,6] it is shown that most of the parallelism in scientific applications is due to parallel work on elements of the same large data structures. Parallel execution of substantially different threads of code only leads to a small degree of parallelism. This means that the approach sketched here for the extraction of parallelism is appropriate for systems with many processors.

The challenge of modern architectures is not the recognition of parallelism in the applications but the support for locality in the data references to make efficient use of memory hierarchies. This is in particular true for local memory architectures.

Several of the state-of-the-art supercomputers contain, besides local registers for the processors, small and fast local or global memory in front of the huge shared main memory with larger latency and smaller bandwidth. Examples of this kind of architecture are the ETA10, Cray-2 or the Alliant. Distributed memory architectures behave in a similar way. The local memory of a processor can be accessed very fast by the processor itself while the access to the memory of other processors has larger latency and smaller bandwidth. An essential difference between the shared and distributed memory approach is that in the first case the local memories of the processors enclose only a small fraction of the total amount of memory available.

Even for shared memory architectures, a compiler has to optimize code for this kind of memory structures by minimizing memory references to the non-local memory by exploiting the locality of memory accesses in the algorithm.

Scientific applications contain in principle two different kinds of locality of memory references: matrix-type and grid-type.

For multiplication of matrices of size *n* the number of operations is $2n^3$ while $3n^2$ data transfers are necessary. Furthermore, the matrix multiplication can be decomposed into a series of smaller ones of any size; and the favorable ratio of computations and memory transfers depends on the size of the submatrices.

Figure 1 shows a simple but typical small program for a grid-type computation. For a n x n grid, $5n^2$ computations but also at least $O(3n^2)$ memory transfers have to be executed. This number of memory references can only be achieved if each value of the array UOLD has to be loaded only once. This means that even UOLD(I-1,J) and UOLD(I+1,J) do not have to be reloaded when UNEW(I,J) is computed. Nevertheless, the ratio of computations emory transfers is small and depends only little on the size of the problem.

```
      PROGRAM RELAX
      PARAMETER (N=100)
      REAL UOLD(0:N+1,0:N+1),UNEW(0:N+1,0:N+1),
     &          F(0:N+1,0:N+1)
      ...
      DO 10 I=1,N
       DO 10 J=1,N
        UNEW(I,J)=0.25*( F(I,J)
     &          + UOLD(I-1,J) + UOLD(I+1,J)
     &          + UOLD(I,J-1) + UOLD(I,J+1) )
10    CONTINUE
      ...
      END
```

Figure 1. Example of a subroutine used for a simple grid-type problem.

On the other hand, the computation UNEW at a certain element of the array requires only values from neighboring points. This kind of locality can be exploited only if the respective parts of the data structure can be kept local to the processor over several executions of the code segment and only the boundary values of a subgrid have to be updated.

From both examples it can be concluded that a compiler, which shall optimize code for this kind of architectures, must be able to partition the data structures in such a way that the parts match each other for the desired computations. In some sense multidimensional strip mining has to be applied to nested loops to exploit full locality in the case of matrix-type problems and to yield minimal memory transfers in the case of grid-type problems if the code segment is executed only once.

The full locality in the case of grid-type problems can be exploited only for distributed memory architectures, because only this type of architecture has the feature that the entity of the fast local memory forms a significant amount of memory. In this case the parts of the data structures have to be assigned for a longer period of computation statically to a specific processor; and only boundary information of the partitioned grid structures will be passed to the memory of other processors.

As shown in the following section the interactive parallelizer SUPERB—result of a research activity at the university at Bonn—was designed according to these requirements.

Structure of the Parallelizer

SUPERB (SUprenum ParallelizER Bonn) is a semi-automatic source-to-source parallelization system. In contrast to existing parallelizers, SUPERB is designed to combine both MIMD and SIMD parallelization into one integrated interactive system that is oriented towards the Suprenum computer and its application for large-scale scientific computing.

As already mentioned in the first section, automatic SIMD-parallelization is a well-known task, but it is extremely difficult to detect parallelism for systems with distributed memory automatically. In SUPERB, data partitioning—the only useful way to extract enough parallelism for this kind of machine—has to be done interactively. The user assigns parts of the data domain, e.g. a grid or a grid hierarchy represented by the arrays of the program, to specific processes. Due to the inherent incompleteness of analysis information the system cannot automatically extract the global relationships between the program's arrays necessary to obtain efficient parallel code.

In principle, there are no restrictions on the kind of programs SUPERB can be applied to. However, to be successful in MIMD-parallelization, the programs should work on a mesh or mesh-like data domain, the computation at the mesh points should be local and the problems to be solved should be large.

The overall structure of the system is depicted in Figure 2. The main components and the overall parallelization process may be outlined as follows: the front-end, the core, the transformation catalog and the backend. (A detailed description of the structure can be found in [7,8].)

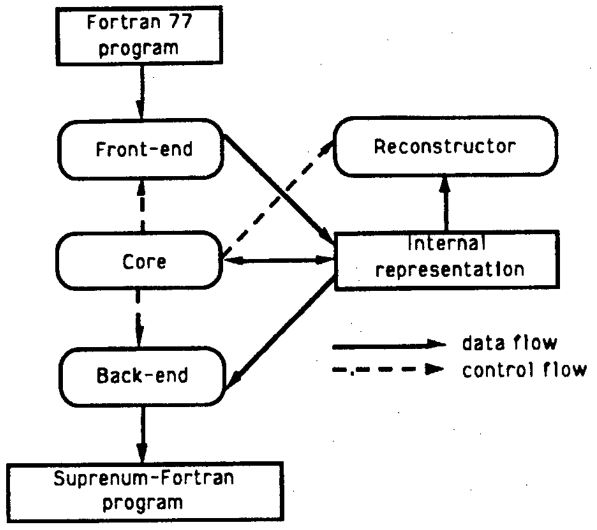

Figure 2. Structure of SUPERB.

The front-end first transforms a Fortran 77 program into an internal representation (attributed abstract syntax tree, symbol table, call graph). After splitting the original program into the initial task (running on the front-end machine and performing I/O) and the subtask (describing the actual computation), in both tasks, control flow (IF-conversion) DO-loops, expressions and special Fortran-features are normalized to facilitate application of other transformations and analysis of the program. Some initial analysis information, such as sets of variables used or defined in a statement (in particular if there are calls to other program units) and control flow relationships describing possible execution sequences of the program, are computed.

The core—the main part of the system—controls the execution of the other system parts, provides a catalog of transformations and analysis services, and contains the interface to the user.

The transformation catalog—organized in a hierarchical structured set of menus—offers a number of transformations (e.g. for MIMD-parallelization) the user can select from. The analysis component verifies the existence of preconditions necessary for the application of a transformation and supplies the user with details about his program. For example, information between statements in loops, interprocedural relationships, references which require communication between processes or conflicts caused by the currently selected data partition can be computed and displayed.

The back-end produces the final Suprenum-Fortran code. Vector code (corresponding to Fortran 8x syntax) is generated for all vectorizable statements. The information collected during the interactive parallelization process is used to insert correct send/receive statements. Some final optimizations are performed to increase efficiency of the generated code.

A Small Example

The interactive parallelization process is described below by using the small example program in Figure 1.

After applying the front-end to the program, the user determines a partition specification. This specification describes a set of partitioned arrays and the information for mapping segments of these arrays to selected processes.

Here the arrays UOLD, UNEW and F are partitioned into segments as shown in Figure 3. Using special analysis services offered by the core the user can look at the communication overhead resulting from this partition.

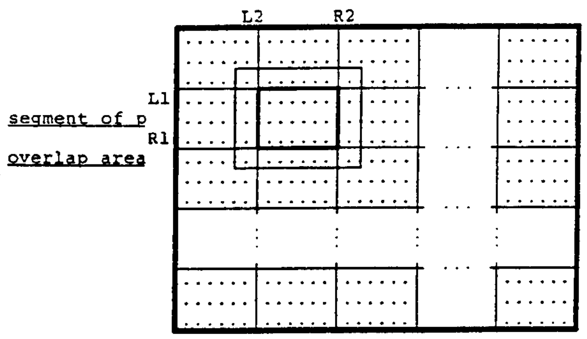

Figure 3. Partitions of UOLD, UNEW and F.

In the second phase the user identifies critical code sections, i.e., sections causing much communication. He can try to optimize the communication by applying transformations like scalar forward substitution, induction variable substitution or special MIMD transformations. Beside these optimizations he may change the partition specification.

In the example program, array elements which have to be exchanged between processes are described by an overlap area around the segments (see Figure 3). All array elements read by process *p* and not local to *p* have to be in this overlap area. Here, each process has an overlap area of width one in every direction.

Now the user is able to improve the vectorization of the code interactively. The analysis component offers him the possibility to examine dependence information in loops. Thus cycles in the dependence graph preventing vectorization can be detected. To remove such cycles, the user may apply transformations such as scalar expansion and loop distribution to selected loops or whole units. In this phase no vector code is generated, but loops are marked to be vectorizable.

In our example both loops can be vectorized.

In the last phase the user applies special MIMD transformations to further optimize the communication. These transformations extract communication from loops and combine small messages into larger ones.

Figure 4 shows the result of the transformation process applied to the example. The communication between the processes is organized by the EXCH statements according to the overlap specification. All array elements in the overlap area of the segments are actualized. L1, R1, L2, R2 are variables containing the bounds of the segment assigned to the executing process.

```
  CALL EXCH (UOLD, 0, N+1, 0, N+1,
&     [ 1,1,1,1 ], ...)
  UNEW (L1:R1,L2:R2) = 0.25 (F (L1:R1,L2:R2) +
&     UOLD(L1-1:R1-1,L2:R2) +
&     UOLD(L1+1:R1+1,L2:R2) +
&     UOLD(L1:R1,L2-1:R2-1) +
&     UOLD(L1:R1,L2+1:R2+1))
```

Figure 4. Transformed code segment.

A more detailed description of MIMD parallelization in SUPERB can be found in [9].

A prototype of the parallelizer is completed and can be demonstrated. Currently, some additional transformations are being implemented and parts of the parallelizing process improved so that they work without direct user assistance. The user will be able to define abbreviations for frequently used sequences of transformations. The application of these transformations will be done automatically to selected parts of the program if the corresponding macro is envoked.

References

1. Callahan, D., J. Dongarra and D. Levine, *Vectorizing compilers: a test suite and results,* Argonne National Laboratory Report, ANL-MCS-TM-109, March 1988.

2. Alliant Computer Systems Corp., *Alliant FX/Fortran language manual,* Part number 302-00007, Littleton MA, 1988.

3. Mercer, R., *The Convex Fortran 5.0 compiler,* Convex Computer Corp., 1988.

4. Cray Research, Inc., *Cray X-MP multitasking programmer's reference manual,* SN-0222 Rev. C, 1986.

5. Solchenbach, K., *Application software for Suprenum,* Supercomputer, this issue.

6. Trottenberg, U., *Suprenum—the concept,* Supercomputer, this issue.

7. Zima, H., H.-J. Bast and M. Gerndt, *SUPERB: a tool for semi-automatic MIMD/SIMD parallelization,* Parallel Computing 6, 1-18, 1988.

8. Kremer, U., H.-J. Bast, M. Gerndt and H. Zima, *Advanced tools and techniques for automatic parallelization,* Parallel Computing 7, 387-394, 1988.

9. Gerndt, M. and H. Zima, *MIMD parallelization for Suprenum,* in: Houstis, E., T. Papatheodorou and C. Polychronopoulos (eds.), Supercomputing, 1st International Conference, Lecture Notes in Computer Science 297, Springer Verlag, New York, 278-293, 1988.

10

22161
31
NTIS
ATTN: PROCESS 103
5285 PORT ROYAL RD
SPRINGFIELD, VA 22161

SUBSCRIPTION/PROCUREMENT INFORMATION

The FBIS DAILY REPORT contains current news and information and is published Monday through Friday in eight volumes: China, East Europe, Soviet Union, East Asia, Near East & South Asia, Sub-Saharan Africa, Latin America, and West Europe. Supplements to the DAILY REPORTs may also be available periodically and will be distributed to regular DAILY REPORT subscribers. JPRS publications, which include approximately 50 regional, worldwide, and topical reports, generally contain less time-sensitive information and are published periodically.

Current DAILY REPORTs and JPRS publications are listed in *Government Reports Announcements* issued semimonthly by the National Technical Information Service (NTIS), 5285 Port Royal Road, Springfield, Virginia 22161 and the *Monthly Catalog of U.S. Government Publications* issued by the Superintendent of Documents, U.S. Government Printing Office, Washington, D.C. 20402.

The public may subscribe to either hardcover or microfiche versions of the DAILY REPORTs and JPRS publications through NTIS at the above address or by calling (703) 487-4630. Subscription rates will be provided by NTIS upon request. Subscriptions are available outside the United States from NTIS or appointed foreign dealers. New subscribers should expect a 30-day delay in receipt of the first issue.

U.S. Government offices may obtain subscriptions to the DAILY REPORTs or JPRS publications (hardcover or microfiche) at no charge through their sponsoring organizations. For additional information or assistance, call FBIS, (202) 338-6735, or write to P.O. Box 2604, Washington, D.C. 20013. Department of Defense consumers are required to submit requests through appropriate command validation channels to DIA, RTS-2C, Washington, D.C. 20301. (Telephone: (202) 373-3771, Autovon: 243-3771.)

Back issues or single copies of the DAILY REPORTs and JPRS publications are not available. Both the DAILY REPORTs and the JPRS publications are on file for public reference at the Library of Congress and at many Federal Depository Libraries. Reference copies may also be seen at many public and university libraries throughout the United States.