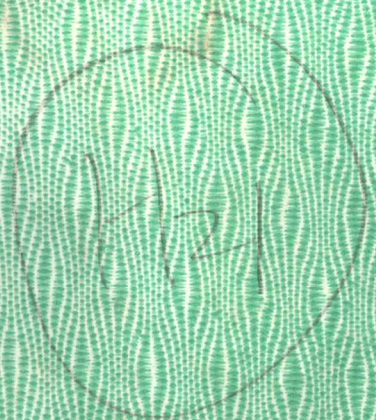
Supercomputing Systems

Architectures, Design, and Performance

Svetlana P. Kartashev and Steven I. Kartashev



POWDERED METALS TECHNOLOGY

John McDermott

NOYES DATA CORPORATION
Park Ridge, New Jersey London, England
1974

Library of Congress Catalog Card Number: 74-82354
ISBN: 0-8155-0550-7
Printed in the United States

Published in the United States of America by
Noyes Data Corporation
Noyes Building, Park Ridge, New Jersey 07656

FOREWORD

The detailed, descriptive information in this book is based on U.S. patents since the late 1960s relating to powdered metals technology.

This book serves a double purpose in that it supplies detailed technical information and can be used as a guide to the U.S. patent literature in this field. By indicating all the information that is significant, and eliminating legal jargon and juristic phraseology, this book presents an advanced, technically oriented review of modern methods of manufacture and use of metals in powder form.

The U.S. patent literature is the largest and most comprehensive collection of technical information in the world. There is more practical, commercial, timely process information assembled here than is available from any other source. The technical information obtained from a patent is extremely reliable and comprehensive; sufficient information must be included to avoid rejection for "insufficient disclosure." These patents include practically all of those issued on the subject in the USA during the period under review, there has been no bias in the selection of patents for inclusion.

The patent literature covers a substantial amount of information not available in the journal literature. The patent literature is a prime source of basic commercially useful information. This information is overlooked by those who rely primarily on the periodical journal literature. It is realized that there is a lag between a patent application on a new process development and the granting of a patent, but it is felt that this may roughly parallel or even anticipate the lag in putting that development into commercial practice.

Many of these patents are being utilized commercially. Whether used or not, they offer opportunities for technological transfer. Also, a major purpose of this book is to describe the number of technical possibilities available, which may open up profitable areas of research and development. The information contained in this book will allow you to establish a sound background before launching into research in this field.

Advanced composition and production methods developed by Noyes Data are employed to bring our new durably bound books to you in a minimum of time. Special techniques are used to close the gap between "manuscript" and "completed book." Industrial technology is progressing so rapidly that time-honored, conventional typesetting, binding and shipping methods are no longer suitable. We have bypassed the delays in the conventional book publishing cycle and provide the user with an effective and convenient means of reviewing up-to-date information in depth.

The Table of Contents is organized in such a way as to serve as a subject index. Other indexes by company, inventor and patent number help in providing easy access to the information contained in this book.

15 Reasons Why the U.S. Patent Office Literature Is Important to You —

1. The U.S. patent literature is the largest and most comprehensive collection of technical information in the world. There is more practical commercial process information assembled here than is available from any other source.

2. The technical information obtained from the patent literature is extremely comprehensive; sufficient information must be included to avoid rejection for "insufficient disclosure."

3. The patent literature is a prime source of basic commercially utilizable information. This information is overlooked by those who rely primarily on the periodical journal literature.

4. An important feature of the patent literature is that it can serve to avoid duplication of research and development.

5. Patents, unlike periodical literature, are bound by definition to contain new information, data and ideas.

6. It can serve as a source of new ideas in a different but related field, and may be outside the patent protection offered the original invention.

7. Since claims are narrowly defined, much valuable information is included that may be outside the legal protection afforded by the claims.

8. Patents discuss the difficulties associated with previous research, development or production techniques, and offer a specific method of overcoming problems. This gives clues to current process information that has not been published in periodicals or books.

9. Can aid in process design by providing a selection of alternate techniques. A powerful research and engineering tool.

10. Obtain licenses — many U.S. chemical patents have not been developed commercially.

11. Patents provide an excellent starting point for the next investigator.

12. Frequently, innovations derived from research are first disclosed in the patent literature, prior to coverage in the periodical literature.

13. Patents offer a most valuable method of keeping abreast of latest technologies, serving an individual's own "current awareness" program.

14. Copies of U.S. patents are easily obtained from the U.S. Patent Office at 50¢ a copy.

15. It is a creative source of ideas for those with imagination.

CONCURRENCY/COMPLEXITY TRADE-OFFS

Trade-off of concurrency with complexity is associated with proliferation of the following outlooks in designing current and future architectures.

1. The use of fewer structural units, each of which is supposed to be quite complex.[3,4] This approach has been implemented in the majority of industrial supercomputers and multiprocessors.

2. The use of a great number of structural units each of which is supposed to be sufficiently simple. This approach was implemented in massively parallel architectures.[5–8]

The interconnections between structural units can be *direct* as in message-passing architectures[7,8] or *reconfigurable*[9,10] The degree of implemented reconfiguration is also varied, from very modest by reconfiguring devices and registers, to very dramatic as is done in dynamic architectures that perform a total restructuring of hardware resources under software control.[9–12]

Industrial (CRAY-type) Supercomputers and Multiprocessors

More detailed description of their evolutionary trends as well as current statuses of implementation of supercomputer architectures can be found in Chapters 1–3 of this book. Here, we will deal only with major highlights and their economic outline justification.

The evolutionary development of industrial architectures is influenced by the following factors.

F1. Gradual and mostly quantitative changes in basic architectural characteristics of a current industrial architecture are caused by economic demands on the reuse of the entire software development for its predecessor.

F2. To be economically feasible and to satisfy user performance needs, an industrial architecture of the current generation must take maximum cost and speed advantage of the component technology that is used.

We can now trace the following evolution in Control Data/ETA architectures in terms of F1.

First generation	→	**Second generation**	→	**Third generation**
STAR-100 (1974)		**CYBER 205 (1981)**		**ETA-10 (1986)**

Similar evolution is true for the CRAY supercomputer family starting with the CRAY-1 and ending with CRAY X-MP/4 and CRAY-2:

First generation	→	**Second generation**	→	**Third generation**
CRAY-1 (1976)		**CRAY X-MP/2, X-MP/4 (1982)**		**CRAY-2, CRAY-3 (1984)(1988–?)**

As for the numerous minisupercomputers, their architectural evolution cannot be traced yet, since we are witnessing proliferation of their first generations manufactured mostly since 1984.

We can identify further evolutionary influences in terms of F2.

Cost optimization. The major source of the cost optimization is associated with minimization in the number of module types used for supercomputer design.

For instance, the STAR-100, built from SSI parts, required 300 different module types each of which was mapped onto a separate board. For the CYBER 205, this approach became infeasible economically because of a peculiarity of LSI cost considerations. To realize cost factors of LSI technology, the total number of different chip types for CYBER-205 was reduced to 14 for scalar unit and 11 for vector and I/O modules.[4]

Speed optimization is achieved by:

- reducing the clock period;
- expanding device parallelism inside each structural unit;
- expanding the process concurrency by gradual increase (not very dramatic) in the number of structural units for each new generation of supercomputers.

Massively Parallel Architectures (MPA)

Massively parallel architectures enjoy popularity in both industrial and academic communities. The most representative architectures are implemented in the following machines.

- *Massively Parallel Processor (MPP)*, developed jointly by Goodyear Aerospace Corporation and NASA-Goddard Space Flight Center, was delivered in 1982.[13,14] The MPP processor has 16 896 one-bit PEs arranged in a 128 row × 132 column rectangular array.[5]
- *Connection Machine*, developed at Thinking Machines Corporation is a parallel computing machine having between 16K and 64K 1-bit processors operating under the control of a single instruction stream broadcast to all processors.
- *Hypercube Supercomputer*, developed at Caltech with the first hardware version, Cosmic Cube, having 64 nodes.[7] Subsequent versions were Mark II and Mark III. The final 128-node version Mark III fp is to be completed in the Summer of 1988.[17]

Note. Strictly speaking, Hypercube lies between the CRAY-type and MPA, since it has a much smaller number of nodes than a classical MPA and each node is a quite complex computer, although not as complex as a typical node (CRAY-1 or CYBER-205) in the latest generations of industrial supercomputers made from 8 to 16 such nodes. However, architecturally, the Hypercube provides the advantages and faces the problems of an MPA in view of the total privacy of each node memory and limited communication capabilities among nodes achieved only with the use of *message-passing*.

CONTROL FLOW VERSUS DATA FLOW

As indicated above, a second factor that has influenced the evolution in supercomputer architectures was the use of two alternative models of computation. The first one, called *control flow*, represents program as a sequence of

instructions in which sequentiality is maintained with the use of step operations and conditional and unconditional branches.

The second model, called *data flow*, also assumes that the program is a sequence of operations. However, the order of execution is maintained by the availability of operands, that is, by data flow. For this model, two operations are sequential only if they are data-dependent, that is, the second one *consumes* the data *produced* by the first one.

Otherwise, they can be executed in parallel. Thus, ideally, as many operations can be executed in parallel as there are available operand pairs.

Control-flow supercomputers implement both *operation* (fine-grain) and *task* (coarse-grain) parallelism. Fine-grain parallelism is implemented in:

- *array systems* in which the same operation is performed over a set of different data pairs;
- *pipelined systems*, in which different phases of the same operation (process) are performed in parallel over a set of different data pairs; and
- *long instruction word architectures*, which implement operation parallelism with the use of very long instruction.[19]

Coarse-grain or task parallelism is implemented in multicomputers/multiprocessors that run independent and asynchronous tasks.

By definition, *data-flow supercomputers implement mostly fine-grain parallelism.* As for task parallelism, it is restricted to data-dependent tasks only within a single data-flow program so that each task producer knows of all task consumers that need its data. If two data-dependent tasks belong respectively to two independent programs, the task-producer may not know in advance of all independent task-consumers that may need the produced data. Therefore, any such task-consumer must access the needed data array itself rather than wait until it become available on its node's arcs. Handling all such cases requires introduction of control flow of operation when the task-consumer initiates the task-rendezvous process with an independent task-producer. Apparently, to implement unrestricted task parallelism requires a hybrid supercomputer that implements both data and control flow orders of execution.

Presently, no consensus exists as to which computational mode gives better performance.

Proponents of control flow argue that it allows improved performance in comparison to data-flow computation because of better utilization of resources.[21] Also, for realistic programs, actual resources involved in data-flow computations are much smaller than the entire complexity of a data-flow supercomputer leading to a considerable resource waste.[20]

The time-overhead created with data-flow supercomputing is associated with (a) the necessity to control the movement of highly distributed data items rather than program instructions; (b) the complexity of allocation algorithms because of the requirement to take into account the locality of both instructions and data; (c) the excessive manipulation with tags during each data entry into a program loop, and so on.

On the other hand, in spite of these drawbacks, data-flow concept remains an attractive architectural idea because of its promise to maximize the potential instruction parallelism, which by definition is beyond the reach of control-flow architecture.

Apparently, a future high-performance supercomputer should be a hybrid that can take advantage of both data-flow and control-flow modes of operation in order to achieve maximal operation parallelism and process concurrency uninhibited by the problems encountered in both types of computation.

State-of-the-art research using prototypes of suffieient power will provide excellent opportunities to alleviate all these problems and build such a supercomputer.

Software Developments

The advent of supercomputing has triggered the following developments in software structures.

- Massive efforts to parallelize execution and simplify development of complex *application programs* through parallel algorithm design and component reuse
- Developing *high-level languages* suitable for parallel computation
- Developing *parallel programming environments* for supercomputers that allow parallel monitoring and servicing of concurrent application programs.

APPLICATION PROGRAMS

In the area of application programs, two major research explorations are taking place aimed respectively at:

1. improving the performance of complex application program, and
2. reducing the amount of time and effort spent on program development processes.

Program Performance

Program performance is optimized provided the following conditions apply.

- It is implemented with fast parallel processing algorithm.
- It is optimally coded and efficiently mapped onto supercomputer architecture.
- Supercomputer architecture and its servicing software introduce minimal delays in program computation and serving data needed by the program.

The major objective of parallel algorithm design is obtaining fast parallel-processing algorithms for application processes. This is achieved by performing equivalent transformations wthin the algorithm in order to make it suitable for computation by synchronous and asynchronous parallel architectures used in supercomputers. *Synchronous* parallel architectures are *arrays* and *pipelines*. *Asynchronous* parallel architectures are *multicomputers/multiprocessors*.

Extraction of maximal performance from the array or pipeline requires efficient parallelization for arrays and vectorization for pipelines of the respective

data structures used in computations since, essentially, each array and pipeline execute one instruction stream at a time.

Computation of an application algorithm efficiently by multicomputer/multiprocessor requires its parallel decomposition into various segments that can run in parallel and asynchronously. Therefore, current work in parallel algorithm design is directed at:

- parallelization and vectorization of data structures to make algorithms suitable for array and pipeline computations;
- parallel decomposition of the application algorithms to make them suitable for computation by multicomputers/multiprocessors.

Also, parallel algorithm design remains the area of interest for two scientific disciplines dependent on the form of algorithm presentation.

If an algorithm is abstracted from specific software implementation, its parallel design methodology was and continues to be the domain of *computational sciences*. On the other hand, when it is presented as a high-level program, its parallel design becomes an object of interest for *software design* directed at developing program and data decompositions that not only perform program and data parallelization and data vectorization, but efficiently map the program and data onto supercomputer architectures. These software design techniques are sometimes attributed to compiler methodologies.

Parallelization and Vectorization of Data Structures

Parallelization and vectorization of data structures is understood as follows. Given an n-element data array:

1. Assign one data element per each PE for array (parallel) data structure or k data elements per PE for pipeline (vectorized) data structure.
2. Perform such equivalent transformations within an application algorithm that it can concurrently control the array of n PEs for array processing or pipeline of p PEs for pipeline processing, where $p = n/k$.

The difference between array and pipeline processing is as follows. Each engaged PE in an array made of n PEs at any moment of time executes the same operation over a different pair of operands; in a pipeline of p PEs, organized into p stages, each PE conceived as the ith stage of the pipeline ($i = 1, \ldots, p$) and, assigned a separate pair of operands, executes the ith phase of the process (operation) assigned to the entire pipeline.

Consequently, both modes of operation imply data parallelism, which is applied at each step of the same operation for arrays or different phases of the same operation (process) for pipelines. Thus, array and pipeline modes of operation require development of data-parallel algorithms that at each step deal with an array of different operand pairs. To become efficient, a data-parallel algorithm must be capable of processing an n-element array in less than n steps, otherwise there is no justification for introducing n PEs for array processing or p PEs for pipeline processing. The typical speed improvement for array processing

is $\log_2 n$; that is, instead of n steps required by serial processing of n data elements, the same algorithm is executed in $\log_2 n$ steps. For a fully engaged pipeline, an n-element array is processed in $p + k$ steps, where p is the number of pipeline stages, and k is the number of operands (or operand pairs) assigned to each stage.

The major and most obvious drawbacks of data-parallel computations as a class are as follows.

D1. By nature, they are not universal but *dedicated*. Thus, for all non-data-parallel applications, array and pipeline supercomputers show inferior performance, although a significant body of research work is now aimed at broadening the class of data-parallel applications, that is, finding the parallel method of execution for algorithms that used to be regarded as entirely serial.[16,23]

D2. *Limited communication capabilities* among different PEs. For arrays, the most typical connection among PEs is a grid, which may prove to be insufficient for nongrid type data exchanges. In all such cases, data words need to be routed among PEs in order to reach their destination. This introduces significant communication overhead and diminishes the benefits of speed-up due to array processing as a concept. For pipelines, the communication capabilities among PEs are even more limited, totally precluding data exchanges among arbitrarily selected PEs while the pipeline is working.

Parallel Decomposition of Application Algorithms

Here we will deal with this problem in the context of software design, assuming that the algorithm has already been brought to a suitable parallel computational form by available numerical methods.

In a supercomputer organized as multiprocessor/multicomputer if two concurrent tasks are data-independent, their parallel computation presents no problem. The problem arises if tasks are data-dependent, that is, if the first one uses the computational data produced by the second one. Organization of parallel computation among such tasks requires proper organization of such tasks, rendezvous process, which features minimal synchronization delays caused by interrupting the task-consumer if the requested data have not yet been produced by the task-producer.

The major thrust of the literature on software design in parallelization/vectorization of data-dependent tasks, however, is directed at concurrentization and vectorization of program loops which feature various types of data dependencies during successive loop iterations (true dependence, anti-dependence and output-dependence[22]).

Loop concurrentization means correct parallel computation of consecutive loop iterations with minimal delays introduced by synchronization processes aimed at passing data-dependent variables from one loop iteration to the next.

Loop vectorization means correct pipeline computation of consecutive loop iterations with the use of vector instructions. Since many supercomputers have vector instruction sets, and a limited number of processors,[3,4] loop vectorization becomes an important factor of their speed-up.

Reducing Program Development Costs

A supercomputing program is usually very complex, requiring many man-hours of program development efforts. Therefore, considerable effort is now expended to reduce the amount of time needed by a program development process. Major approaches here are (a) component reuse, and (b) software design.

Component Reuse

This concerns construction of programs from reusable components (modules) with or without the use of composition principles. The composition approach provides for some abstract initial representation of reused program modules, then the use of a composition technique such as parameterized programming,[25] the Unix pipe mechanism,[38] or special specification techniques,[26] to construct a complex program from reusable modules with the interfaces specified by the composition technique.

The second approach involves the creation of libraries of reused components that do not need preliminary specifications to be connected to each other.[27]

Software Design

This concerns the development of synthesis techniques for automatic generation of an application program. To be most efficient, these generation techniques should apply to all of the software of the supercomputer. Accordingly, any target program is first represented in an abstract and very high-level notation related to the problem domain.[28,29] The executable program is then generated from the initial simplified representation using automatic synthesis techniques.

PARALLEL PROGRAMMING LANGUAGES

The effect of supercomputing on high-level languages is via two approaches.

- *Theoretical approach* aimed at developing an ideal high-level language concept for future supercomputers
- *Practical approach* aimed at improving existing programming languages and creation of new programming languages for existing industrial supercomputers

Theoretical Approach

Theoretical work in high-level languages is focused on the following.

1. Raising the level of abstraction by developing *very high-level languages* that allow:

(a) programming an application with fewer and more powerful language constructs;

(b) efficient translation of the resulting program into a variety of supercomputer architectures.

2. Higher utilization of concurrency present in application with the use of the *declarative* versus the *imperative* style of programming.[30]

Very High-Level Languages

The basic ideas behind *very high-level languages* is to use declarative and concurrent semantics based upon equational logic.[30] This implies that the programs written in such languages are presented as sets of logical axioms and computation is conceived as a set of rules derived from the equations by a compilation process.[32] The current state of computation is defined by substitution of the "pattern" (usually left-hand side of the expression) in place of a replacement template that is a portion of the right-hand side of the expression and which satisfies the logical conditions attached for such a replacement. When the pattern replaces the matching replacement template, the new result is obtained by instantiating (activating) specific values of variable(s).

These substitutions can be applied concurrently to all the available replacement spots, requiring no specific concurrency constructs at the language level.

Another feature of such languages is that by definition they are *declarative* and not imperative. In the beginning, before the compilation process begins, available sets of equations declare the problem that has to be solved. The task of the compiler is to detect all the available parallelism by concurrent application of substitution rules for all available replacement templates that instantiate computations to specific values. Therefore, the only sequentiality that is introduced by such languages is through data dependence, defined by the rules of precedence present in equations. By contrast, the practical supercomputer languages of today are *imperative* in the sense that they explicitly specify the sequence of commands; that is, not only do they say what to do but also when and where to do it.[30] This is the restriction that reduces the concurrency that can be extracted from high-level application programs.

Practical Approach

The practical approach (a) improves existing high-level languages by adding parallel constructs that implement typical supercomputer operations, (b) develops specific machine-parallelism languages that are well suited for available and popular supercomputers,[31] and (c) attempts to express problem-parallelism directly without reference to the hardware of the underlying machine.[33]

Improving Existing High-Level Languages

Most high-level languages currently used to program supercomputers are conventional scientific languages with added concurrent programming constructs.[34,35] Of these, FORTRAN is highly popular because FORTRAN computations are numeric in nature and supercomputers are now mostly used for numeric computations. Current work on FORTRAN modification is aimed at making FORTRAN programs more suitable for array and pipeline computations.[34,37]

Machine Parallelism Languages

A basic characteristic of such languages is their ability for direct expression of *machine parallelism*.[36,37] This allows generation of efficient code during the

compilation stage. These languages provide suitable data representation as well as algorithm construction features that encourage explicit expression of parallelism within the algorithms.

Their major drawback is that the high-level programs generated with their use are overly architecture dependent and are not portable from one architecture to another.

Direct Expression of Problem Parallelsim

The major objective of the third approach in supercomputer languages is to provide for the direct and natural expression of process parallelism (a) unhindered by improving or fixing an existing sequential language, and (b) avoiding dependence of the language on the hardware with all its negative consequences.[31]

The independence of such languages of the architecture is achieved via their ability to give the user the right to define the maximal size of parallelism at compile stage and to adjust the program to a smaller than maximal size at execution time. Also, these languages are portable and equally suitable for vector and array types of computation.[33]

However, their wide adoption by the user community is hindered by the fact that existing industrial supercomputers, for economic reasons, tend to preserve the software developed for earlier generations and thus mostly rely on the languages (say, FORTRAN), in which this software has been written in the past, rather than use new languages that overcome the bottlenecks of sequentiality and dependence on the hardware for improving program performance.[3,4]

PARALLEL PROGRAMMING ENVIRONMENT

The creation of parallel programming environments is aimed at achieving the following goals:

- parallel monitoring of computations,
- parallel servicing of computations.

These functions are performed by distributing operating systems, parallel compilers, and debuggers.

Operating Systems

Current research in operating systems is going in the following directions.

1. Standardization of the OS functions for improving the user's access to supercomputers with the use of industry standard user interface and file systems networked with standard workstations.[38,39]
2. Equipping the OS with the techniques aimed at automatic scheduling and synchronization of multiple processes run on multiple and distributed resources.
3. Concurrent implementation of the OS functions with those of computations in order to exclude or minimize the OS overhead introduced into computations.[40,41]

Standardization of the OS Functions

Currently, the most widely used standard OS in supercomputers is the UNIX OS, which provides facilities for running programs and a file system for managing information.[42] A *file system* is organized as a tree in which each leaf is a *file* or a *directory*. When a user turns on UNIX, a command interpreter accepts commands from the terminal as requests to run program. Any such request includes the program name, which accesses the required file in a file system. If this file exists and is executable, it is loaded as a program.

One of the most productive aspects of UNIX environment is a rich set of software tools for serving programs. The UNIX software is written in the C language. Since C is available on a variety of machines, C programs are portable from one machine to another. Also, the UNIX system contains a variety of facilities that encourage software reuse ranging from the library of standard functions to basic architectural mechanisms called *UNIX pipe* which, acting as a standard buffer interface, allows one user program-producer to generate data for another user program-consumer.

Applied iteratively, the UNIX pipe allows practical and effortless organization of very complex *hierarchical* software systems that would be very difficult if not impossible to organize otherwise.

Distributed Hardware Operating System

The major drawback of standard OS software of the Unix type is that it introduces a considerable time overhead into program computation because almost all program requests for service are accompanied by program interrupts caused by interference of the OS. In a supercomputing environment, to reduce or minimize the OS overhead into computations requires hardware implementation on distributed resources of scheduling and synchronization algorithms. This leads to hardware implementation of basic OS functions, since this is an efficient way of achieving concurrentization of the OS functions with those of computations.[40,41,43] The intention of the hardware-implemented OS is in absolute minimization of program interrupts.

In a typical computational situation, most program interrupts originate from the necessity for the OS to interfere in the *task rendezvous* process between two programs. To perform process synchronization, the OS stops both the program-consumer when it issues a request for rendezvous and the program-producer when it accepts the rendezvous request.

Only following this synchronization stage, does the OS allow the rendezvous to proceed, that is, to send data array to task-consumer memory domain. However, interrupt of both task-producer and task-consumer can be avoided if the task-rendezvous algorithm is implemented via hardware and can be activated locally inside each PE. It allows avoidance of the following interrupt cases.

Case 1. Neither task-producer (TP) and task-consumer (TC) is interrupted if the data are produced before they are requested; that is, if the *accept*

statement issued by the TP precedes the *request* for rendezvous issued by the TC. The hardware OS organizes storage of the requested data in a particular destination and informs TC through the mailbox system about the storage location, requiring no interrupt for either task.

Case 2. TP is not interrupted; TC is interrupted if the data are produced after they are requested, that is, if the *accept* statement issued by the TP follows in time the *request* for rendezvous issued by TC. The hardware OS again uses the same mailbox system with storage location to allow the interrupted TC to resume computations when the data produced by TP are ready.

Hence, for task-rendezvous, implementation of a hardware OS abolishes all interrupts of the task-producer and minimizes interrupts of task-consumer.

Parallel Servicing Software

On-going research in the area of parallel servicing software pursues the following objectives.

- Automatic generation of compilation and debugging systems, given a description of supercomputer architecture
- Implementation of a comprehensive software portability concept that allows creation of retargetable software systems and greatly improves software productivity.

Both objectives will lead towards creation of comprehensive software environments for supercomputers through the use of:

- automatic synthesis applied to software generation, and
- reuse of basic software tools, which becomes possible through implementation of the portable software concept.

However, in the supercomputer environment, both concepts are applicable only for *stored* application programs, since the time overhead they introduce becomes that of program preprocessing aimed at its compilation and debugging, which is not additive to program computation time. Thus, they may become ingredients of standard OS, greatly enriching its repertoire. On the other hand, *dynamic* or *arriving* programs require hardware implementation of compilation systems as a part of its OS hardware aimed at automatic scheduling and resource allocation in real-time.

Applications

Users are the major beneficiaries of these new developments in supercomputing. There is a broadly formed consensus that supercomputing has already established itself as the third mode of scientific research, that is, that it has significantly broadened the traditional base of scientific pursuits formerly restricted to (1) experimentation and (2) theoretical analysis.[44]

The advent of supercomputing in the area of applications was marked by the following milestones.

Milestone 1 Improved match between more precise modeling techniques and high-performance scientific and industrial applications.

Milestone 2 Cost-efficient computerization of new applications with very demanding performance requirements that were unthinkable targets for attempted computerization in the past owing to their complexity and timing demands. (For instance, many mission-critical computations in aerospace are characterized by continuing and dramatic reduction in the time of computer responses to real-time input data acquired from sensors because of the rapid progress in the systems aimed at detecting, intercepting, and homing each real-time target. Similar timing limits are imposed by advanced nuclear and chemical processes controlled by supercomputers; and so on.)

Milestone 3 Comprehensive automation applied to complex application processes which allows a much greater degree of exclusion of human intervention than it was possible to achieve in the past through the use of:

- *intelligent expert systems* with automatic decision making process, and
- *automatic software systems* made of complex application programs and the parallel-programming environment in which they are run.

Milestone 4 Creation of comprehensive numerical laboratories for complex supercomputer applications with all the supporting mechanisms that facilitate understanding of computational results via the use of:

- *visual presentation* of time-dependent multidimensional computational results and their effect on the behavior of modeled application processes;
- *time-dependent multidimensional simulations* of complex application processes; and
- ability to perform *automatic comparison of numerical data with those obtained through the use of laboratory experiments* with far-reaching consequences for improving the human understanding of complex scientific application processes and aiding scientific discovery.[45]

The following categories of users become principal beneficiaries of these advances.

- Users of important industrial applications such as high-speed aerodynamic design, robotics, biotechnology, structural materials, electronic and optical technologies, and so on