The Supercomputers from Siemens Nixdorf and their Parallelization Strategy

Dr. H. Gietl, W. Kratzer, R. Rohrbach
Siemens Nixdorf Informationssysteme AG
Munich, Germany

1. INTRODUCTION

With the announcement of the new models S/40, Siemens has broadened its spectrum of Supercomputers by two parallel processors. The architecture of these systems is based on a memory hierarchy with primary (main storage unit) and secondary storage (system storage unit). Up to four scalar units and two vector units may be connected to the main storage unit forming a cluster. The system storage unit is used for the communication betweeen different clusters.

To modify existing FORTRAN program codes easily and in addition enable the programmer to write new programs for parallel systems, the following summary of the strategy for the parallelization of FORTRAN programs should be applied:

a) The parallelization of DO loops should be carried out automatically by the compiler in order to allow the easy use of existing programs.
b) To force parallelization and implement synchronization between subroutines, use Optimization Control Lines (OCL) as long as possible. They have the advantage that program portability between different processors is retained and sequentialization for debugging purposes can easily be applied, because ignoring the OCLs does not change the semantics of the program.
c) Keep compatibility with existing FORTRAN programs by relating the data allocation for parallel processing to the FORTRAN meaning of common blocks.

This paper gives an overview of architecture and hardware of the Siemens multiprocessors and a detailed description of the parallel processing concept from the software point of view. It is organized in the following way:

2. SHORT OVERVIEW OF THE S SERIES

2.1 General Description

The recently announced S Series from the Siemens Vector Processing System is based on four models (S100, S200, S400, S600) with different peak performances between 500 MFLOPS and 5 GFLOPS. The difference in performance is due to the fact that the execution unit for vector instructions (the so called vector unit) varies between the models. The vector unit executes arithmetic and logical operations by hardware implemented multifunctional pipelines. These multifunctional pipes are able to carry out add, multiply, or combined add and multiply instructions (SAXPY–operations) as one vector instruction. Thus a maximum of two floating–point operations within one vector–unit cycle can be achieved by one multifunctional pipe.

With a cycle time of 4 ns, the S100 reaches a peak performance of 2 operations / 4 ns = 500 MFLOPS, since one arithmetic pipe may operate at a time. The increase in performance for the S200 is obtained by adding a second multifunctional pipe (both working in parallel), thus doubling the peak performance of the S200 to 1GFLOPS. In the case of the S400 and the S600, the two multifunctional pipes are implemented twofold resp. fourfold by hardware. It means that vector instructions in the S400 partition the vector operands into two (four in the case of the S600) equally sized portions and execute these portions in parallel. As a consequence, the execution time for vector instructions of the same vector length is reduced in the S400 by a factor of two (four in the case of the S600).

From the hardware point of view, the models S400 and S600 may already be regarded as parallel processors based on internal parallelization. With the reduced vector–unit cycle–time of 3.2 ns, the resulting peak performance is 2.5 GFLOPS for the S400 and 5 GFLOPS for the S600. (Some of the S400 models are delivered with 4.0 ns, resulting in 2 GFLOPS peak performance.) Figure 2.1 summarizes the relation between model, pipeline parallelism and performance.

The various vector units of the S series are combined with at least one scalar unit (named S100/10, S200/10, S400/10, resp. S600/10, or collectively S/10). The scalar unit controls the vector unit and executes the scalar parts of a vectorized program. It is a well–known fact, that even with a fast vector unit the scalar costs for a highly vectorized program cannot be neglected and may consume a considerable amount of the total CPU time. Therefore, the S Series is provided with the fastest scalar processors available today, having a sustained performance between 24 and 37.5 MFLOPS for mere scalar code (depending on the model).

Model	S100	S200	S400	S600
Cycle Time (ns)	4	4	3.2	3.2
Number of Parallel Operations	2	4	8	16
Peak Performance (MFLOPS)	500	1000	2500	5000
Sustained Scalar Performance (MFLOPS)	24.0	30.0	37.5	37.5

Some of the models with a S400 vector unit are delivered with a cycle time of 4.0 ns, resulting in 2000 MFLOPS.

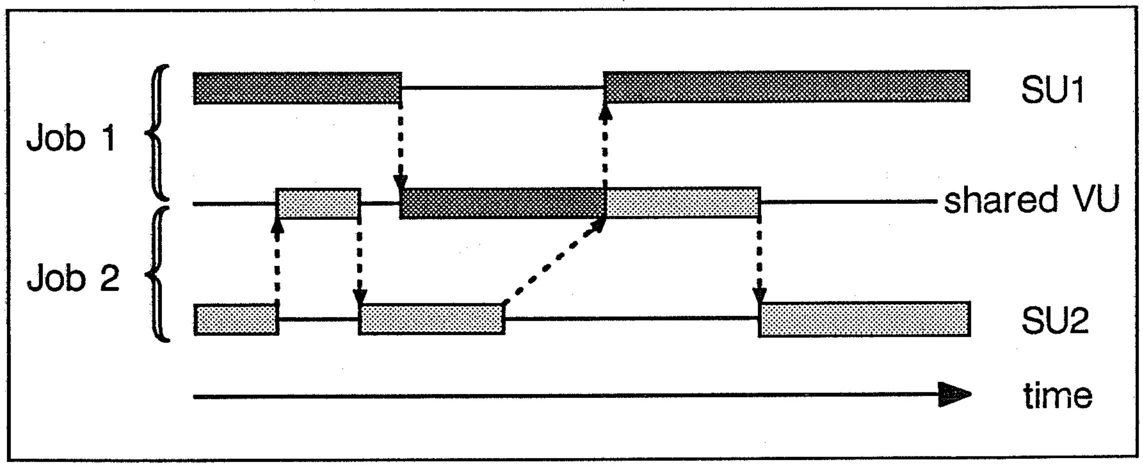
Figure 2.1

The significance of a fast scalar unit is demonstrated in the following example, where the execution behaviour of a highly vectorized job is analyzed. Assume a job running 100 seconds in scalar mode. Assume also that it has a vectorization ratio of 98%, i.e. 98 seconds can be reduced through vectorization. If there is a considerable speed–up factor of, say, 50 for the vectorizable part of the code compared to scalar execution, the 98 seconds are reduced to 1.96 seconds when executed in the vector unit. The execution time of 2 seconds for the scalar part of the code remains unchanged. Hence more than 50% of the total CPU time is spent in the scalar unit, and consequently the vector unit is not busy during this time. Moreover, in a computer center usually more than one job is executed at a time and the utilization of the vector unit is even more reduced due to the fact that jobs with different vectorization ratios and profiles (e.g. I/O bound) enter the vector processing system in a mixed way. (Not every job running on a vector processor has a vectorization ratio of 98%)

Having analyzed this insufficient utilization of the vector unit, the designers of the S Series had the following revolutionary idea: use the remaining vector unit time by a second scalar unit. This leads to the **Dual Scalar Architecture** which is the unique feature of the new Siemens vector processors. This architecture allows two scalar units to share one common vector unit. Since every scalar unit has access to one set of vector registers, effectively only the pipelines are shared: logically two vector units can be found, although physically there is only one. Therefore, in the best case a Siemens system with the **Dual Scalar Architecture** has the throughput of the much more expensive configuration where two complete processors (each with a scalar unit and a vector unit) are available. By means of the **Dual Scalar Architecture**, the S Series allows a much better utilization of the vector pipelines (see the example from figure 2.2) and thus gives a more attractive price/performance ratio.

The models with **Dual Scalar Architecture** are named S/20. The architecture is shown in figure 2.3.

To meet the requirements of applications in science and industry, the programmer should be able to write programs which can use more than one vector unit at the same time, thus speeding up the program through multitasking. On the basis of the **Dual Scalar Architecture**, the multiprocessor model **S/40** is formed by combining two S/20 models, resulting in a processor complex with two vector units and four scalar units. The communication between all cooperating processors is done via the common main storage unit (MSU).



If Job 1 is using only the scalar unit (SU1), Job 2 may use the shared vector unit (VU) and vice versa.

Figure 2.2

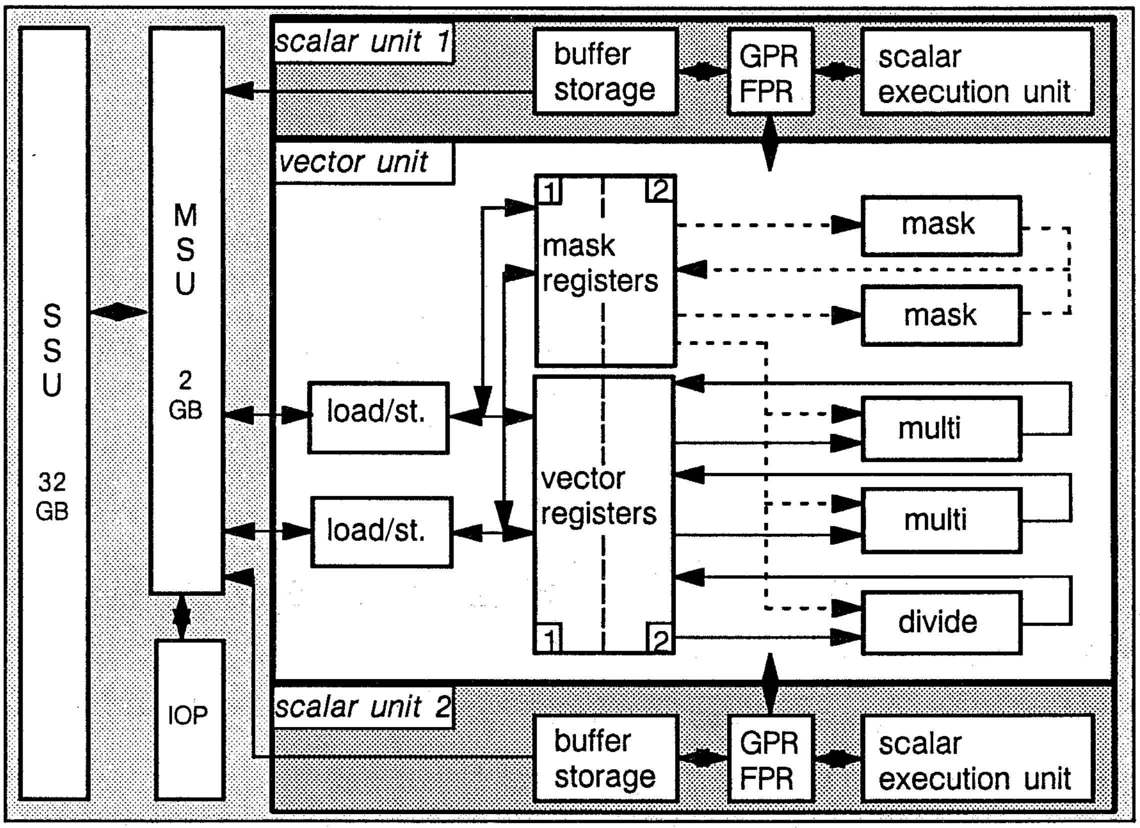
This structure opens up some promising possibilities. On the one hand, the programmer may parallelize his program for execution on two processors. On the other hand, when two parallelized jobs are running concurrently, the scalar units can be allocated to the jobs in such a way, that each job can use both vector units. Together with the common usage of the MSU, this results in a more efficient exploitation of the expensive resources. In a multiprocessor system with N traditional vector processors, a user program (parallelized for N processors) normally cannot keep busy all the resources (CPU, main storage, etc.) over the whole period of its execution time. But within the S/40 model based on the **Dual Scalar Architecture**, the system may be exploited in an optimal way. In this way, the S Series unifies the interests of the programmer (speed-up of the program through multitasking) and the computer center (optimizing the throughput of the system).

Figure 2.4 summarizes the architecture of the Siemens multiprocessors. It should be noted that the upgrade within this line of models can be done in place, simply by add on.

2.2 Hardware Summary

Logic VLSI

The processor logic uses highly integrated, emitter coupled logic (ECL) VLSIs with up to 15000



Architecture of the S/20 model: The multiprocessor consists of vector unit, scalar unit, main storage unit, system storage unit (optional) and I/O processor. The transfer between the main storage unit and the vector registers is realized by the two load/store pipelines. The multifunctional pipelines (multi) are working on the vector registers. Operations on the mask registers are executed by the mask pipelines. Six of the seven pipelines may work in parallel.

SSU :	System storage unit (32 Gbyte)
MSU :	Main storage unit (2 Gbyte)
IOP :	Input/output processor (up to 1 Gbyte/second)

Figure 2.3

gates per chip and a signal propagation delay of 80 picoseconds. Super high speed ECL–RAM chips with a 64 Kbit memory capacity, a chip access time of 1.6 nanoseconds and 3500 gates per chip are used for vector registers, control storages and buffer storage. As a result, the vector unit runs with a cycle time of 4 ns resp. 3.2 ns depending on the model.

Multi–layer glass–ceramic board (MLG)

For the first time ever, glass ceramics has been used as the material for the motherboard to improve propagation speed. Up to 144 VLSIs can be mounted on a 24.5 * 24.5 square centimeter board.

Large–capacity static RAM

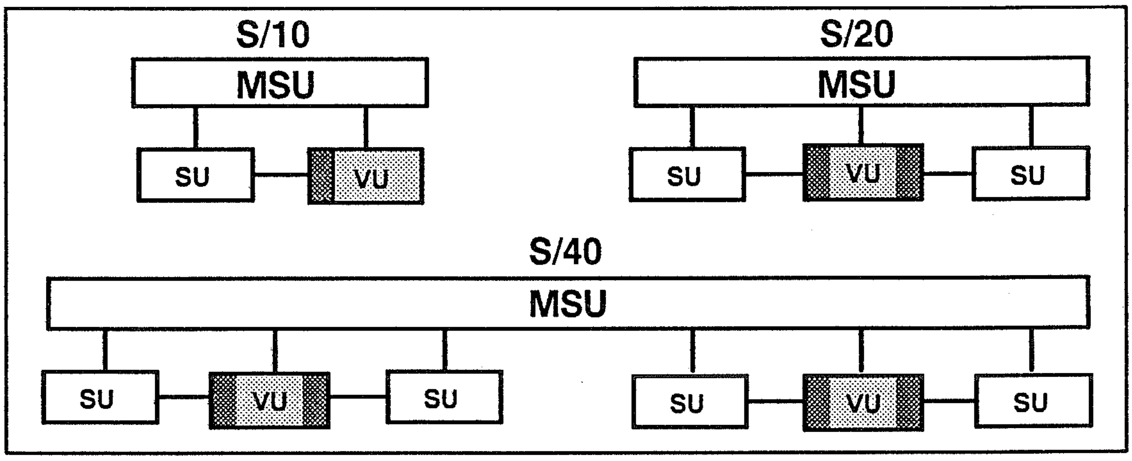
The main storage unit (first level of the two level memory hierarchy) features high-speed large-capacity 1 Mbit static RAM chips with an access time of 35 ns. It can be extended up to a maximum of 2 Gbytes, greatly improving the system performance when large-scale numeric tasks are executed. The main storage array card is composed of high–density printed circuit boards. Up to eight main storage array cards are mounted three dimensionally on the MLG, thus achieving a maximum capacity of 512 Mbytes per MLG.

System Storage Unit (SSU)

The system storage unit (second level of the two level memory hierarchy) features high–speed large–capacity 1 Mbit (4 Mbit) dynamic RAM chips with an access time of 100 ns. The high–speed large–volume SSU of 1 to 8 (32) Gbytes allows flexible system configuration thus improving system throughput and enhancing input/output performance. In particular, with the SSU serving as a swap area, major increases are possible in the number of concurrent users of vector functions during interactive sessions without loss of performance. Moreover, the SSU allows an in–core treatment of temporary files through the Virtual Input Output / FORTRAN (VIO/F) facility.

High throughput channel (I/O processor)

The block multiplexer channels have a maximum transfer rate of 4.5 Mbytes per second, and the optical channel has 9 Mbytes per second. With the maximum of 128 channels, a throughput of 1 Gbyte per second can be achieved. Optical channels allow peripheral devices to be connected over long distances via optical fiber cable – up to a maximum of 1 kilometer.



SU: Scalar unit
VU: Vector unit
MSU: Main storage unit
▩ : Vector register set, exists twice (four times) in the /20 (/40) models
▨ : Execution pipelines of the VU. Exists only once and may be shared by the scalar units

Figure 2.4

3. PARALLEL PROCESSING STRATEGY

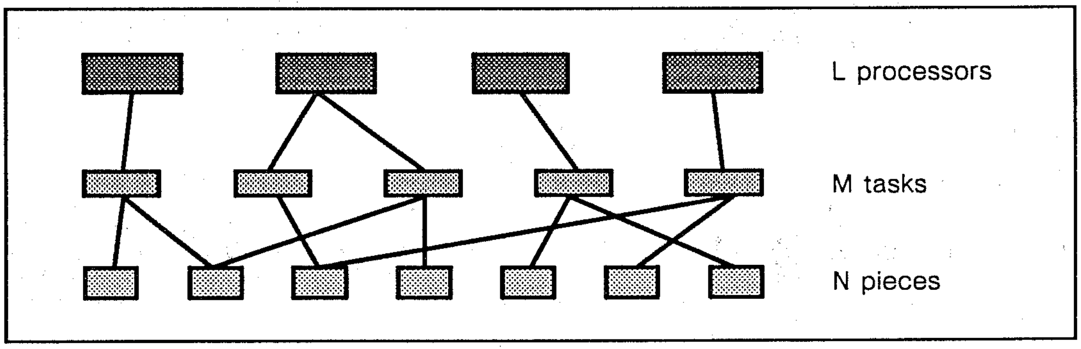
The parallel processing concept has been designed to make the development of parallel FORTRAN programs easy and semantically correct. The parallelization of DO–loops within a Fortran program is always carried out automatically by the compiler in order to facilitate for the programmer the capabilities of a multiprocessor system. Beyond this automatic parallelization, the programmer can furthermore express parallelization by using processing elements like Optimization Control Lines (OCLs) and calls to service subroutines for parallel processing.

Whenever a correct resequentialization of a parallelized program can be achieved by simply ignoring the parallel processing elements, OCLs are preferable: the program can exploit the power of the machine without sacrificing portability. It is easily ported to sequential systems (e.g. for debugging purposes) or to other parallel systems. An example of a situation where this strategy is very suitable is the treatment of critical sections. However, when the resequentialization is more complicated (i.e. POST/WAIT–mechanism, barrier synchronization), the use of service subroutines for parallel processing is the better method. In this context, OCLs would be a dangerous source of error. Portability is not relevant here, since restriction to the FORTRAN standard would completely exclude these mechanisms.

The parallel processing concept within the S Series is based on three features: processor, task and piece.

Processor:	Hardware resource as described in chapter 2. Scalar unit for a mere scalar program (no vector instructions). Scalar unit and vector unit together for a vectorized program which includes scalar and vector instructions.
Task:	A task represents the logical execution unit which is defined and dispatched by the operating system. Dispatching means that the operating system decides, which processor will be assigned to which task.
Piece:	The concept of pieces is significant to understand the semantics of parallelized programs. Parallel processing of a FORTRAN program means that several pieces execute the program concurrently. A piece represents the minimum control unit which can operate in parallel to other pieces. Pieces can be generated and deleted firstly by automatic parallelization (see chapter 4) and secondly by parallel calls of subroutines (see chapter 5).

Tasks and pieces are connected via the FORTRAN library, whereas tasks are mapped to processors by the operating system. As a result, we have the structure shown in figure 3.1 for parallel



Processor:	Hardware resource
Task:	Logical execution unit which will be assigned to a processor by the operating system
Piece:	Pieces concurrently execute a parallelized FORTRAN program

Figure 3.1

processing.

The generation of pieces and their deletion always follows the same FORK–JOIN structure. This will be explained in detail in the example, which is graphically represented in figure 3.2.

Let us assume a specific program point of an existing piece (P1) where several new pieces (P2,P3) will be generated (FORK). During this forking process, the initiating piece (P1) will be set to waiting state. All the newly generated pieces (P2,P3) may now start execution depending on the availability of processors. The waiting state of piece (P1) will be released when all generated pieces have finished their execution (JOIN). In this way, the parallel processing concept is always based on a FORK–JOIN structure. It should be noted, that the FORK–JOIN structure may be nested, which means that newly generated pieces may generate pieces and so on, allowing the programmer to build up tree structures of pieces.

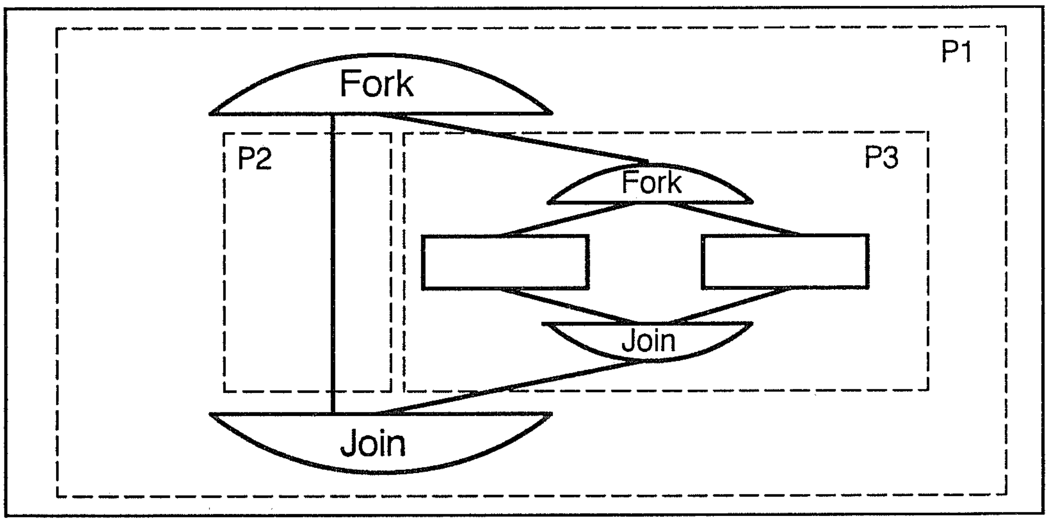
To execute a program in parallel, pieces (parts of the program) have to synchronize and to exchange data and must be able to work on their private data. For the synchronization of pieces, critical sections, POST/WAIT mechanisms, semaphores, and barriers have to be implemented in the FORTRAN system. Concerning the access to data from pieces there are two possibilities. Firstly, to allow only private access to data, data area may be allocated exclusively to some pieces. Secondly, to allow common data (or to exchange data) between different pieces, the address space may be shared by various pieces.

The OCLs and the service routines for parallel processing may be classified according to their functionality into 4 groups:

a) Auxiliary OCLs for automatic parallelization.
   The inline parallelization like DO–loop slicing is carried out automatically by the compiler. Therefore, no OCLs are required. But in special cases it may be appropriate to assist the compiler with OCLs like

   ```
   !OCL NOPREC(var)
   ```

   This directive indicates, that there is no data dependency for variable var which could disturb parallelization.



The Fork structure in piece P1 generates new pieces P2 and P3, setting the initiating piece P1 into a waiting state
The Join structure combines pieces and releases the initiating piece from the waiting state

Figure 3.2

b) OCLs to force parallel execution of subroutines.
Parallel execution of subroutines can be initiated by the following OCLs.

```
!OCL PARCALL
!OCL END PARCALL
```

All CALL statements between PARCALL and END PARCALL generate parallel pieces.

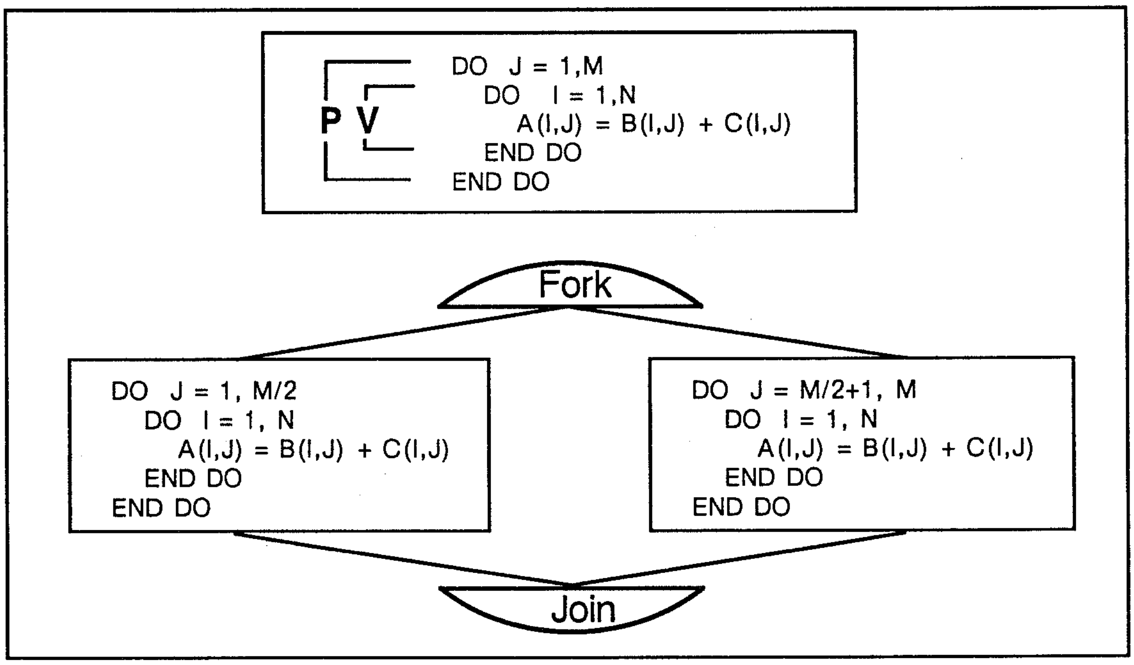
c) OCLs and service routines for synchronization.
Programming tools are available to define critical sections, POST/WAIT structures, semaphores, and barriers.

d) OCLs for Data Allocation.
Data belong to one of the three classes AUTOMATIC, SAVE, and GLOBAL. The selected class determines, firstly, whether the data should be stored in the main storage unit (MSU) or in the system storage unit (SSU) and, secondly, the life range of this data. The life range furthermore determines which pieces have access to the data.

4. AUTOMATIC PARALLELIZATION

4.1 Basics of automatic parallelization

Automatic parallelization is applied to DO loops only. If the data-dependency analysis of the DO loops carried out by the compiler allows for vectorization and parallelization, the compiler has two techniques depending on the loop structure:



```
DO J = 1,M
  DO I = 1,N
    A(I,J) = B(I,J) + C(I,J)
  END DO
END DO
```

```
DO J = 1, M/2
  DO I = 1, N
    A(I,J) = B(I,J) + C(I,J)
  END DO
END DO
```

```
DO J = M/2+1, M
  DO I = 1, N
    A(I,J) = B(I,J) + C(I,J)
  END DO
END DO
```

P means that the loop will be parallelized
V indicates that the loop will be vectorized
The loop J=1,M will be divided into two disjoint sets J=1,M/2 and J=M/2+1,M which can be executed in parallel, because there is no data dependency between the two generated pieces.

Figure 4.1

a) For nested DO loops, the compiler selects the best index for vectorization and the best one for parallelization, permuting the loops (if necessary) in order to make the vectorizing index the innermost and the parallelizing index the outermost one.
b) Concerning a single loop, both vectorization and parallelization may be applied together.

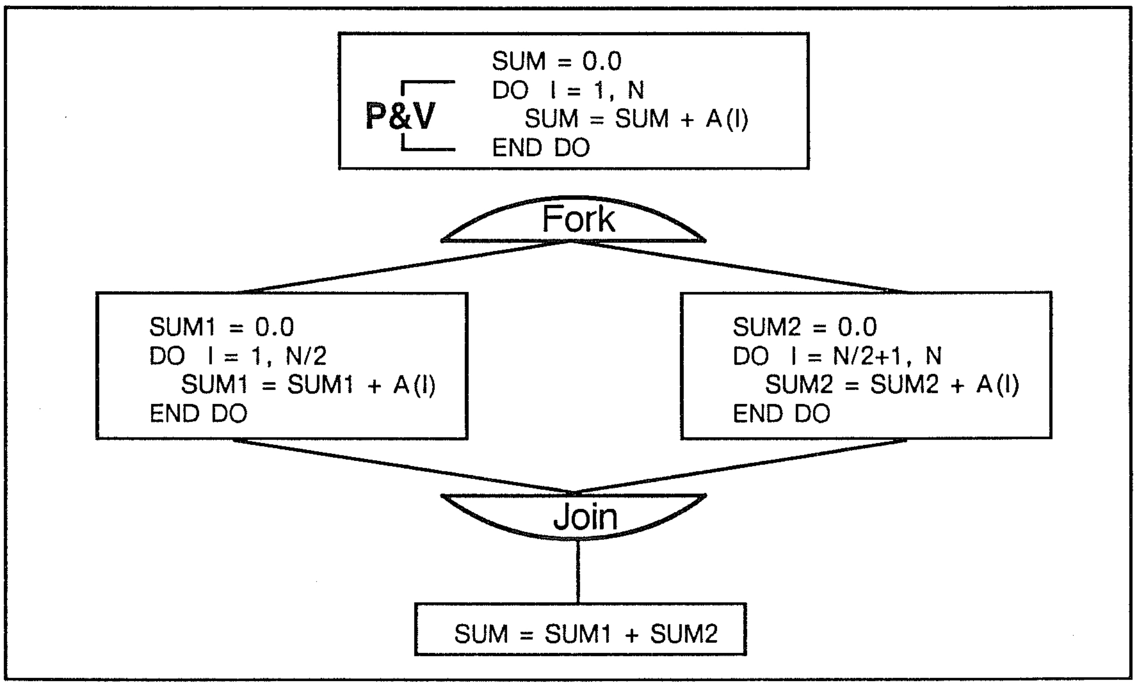
The first technique is a natural way to distribute nested loops onto more than one processor. The set of indices for the outer loop is divided into disjoint parts and each of these parts is used to generate pieces, which may be executed in parallel. The implementation image produced by the compiler for this kind of parallelism is shown in figure 4.1.

The second technique (both vectorization and parallelization for a single loop) may be applied by the compiler for DO loops, if the vector length for the calculation is large enough to be divided into two or more disjoint parts. This technique may also be useful for DO loops containing so many operands that the vector register set of two processors may help to speed up the total execution time of the program. By means of this technique, the overall performance of a program may be easily increased. An example is shown in figure 4.2.

It should be noted that the automatic parallelization of programs is carried out in a semantically correct way. Compiler and runtime library control synchronization and data access. Therefore, the programmer does not need to take care of the pieces generated by automatic parallelization.

4.2 Auxiliary OCLs for automatic parallelization

The user can advance parallelization specifying the same OCLs as during automatic vectorization. With regard to parallelization there are some additional OCLs.



P&V means that the loop will be parallelized and vectorized

Figure 4.2

```
!OCL PARDO(2)
  DO   J = IS, IE
    DO   I = 1, N
      A(I,J) = B(I,J) + C(I,J)
    END DO
  END DO
```

Figure 4.3

4.2.1 OCL for data dependency specification

```
!OCL NOPREC(var)
```

The NOPREC OCL tells the compiler that there is no data dependency which disturbs parallelization of a DO loop or by which some synchronization is necessary. This OCL can be written before a DO statement which should be sliced. The parameter 'var' is the name of a variable or an array which has no data dependency and therefore does not prevent parallelization. The syntax and semantics of this OCL are very similar to the NOVREC OCL for vectorization. But the conditions of data dependency may be different for vectorization and parallelization.

4.2.2 OCL to avoid parallelization

```
!OCL SERIAL
```

This OCL suppresses automatic parallelization of the succeeding DO loop.

4.2.3 OCL for DO loop parallelization

There is only one OCL for DO loop parallelization. It is

```
!OCL PARDO(n)
```

The constant n specifies the number of parallel pieces if the loop can be parallelized. See figure 4.3 for an example.

The line !OCL PARDO(2) does not force parallelization, but shows that the DO loop is a candidate for parallel execution. The compiler analyzes the DO loop and generates parallel object code automatically if this is possible. Without this OCL, the compiler would parallelize the DO loop, generating 4 or 8 pieces (depending on the default value of the compiler). In the example, the OCL specifies two pieces.

```
!OCL PARCALL
  CALL SUB1
    ...
  CALL SUBn
!OCL END PARCALL
```

Figure 5.1

5. PARALLEL PROCESSING BETWEEN SUBROUTINES

To force the parallelization of subroutines, the programmer has to inform the compiler that the subroutines are semantically independent. This is done via OCLs which specify the range for the parallel activation of subroutines. The syntax for these OCLs is shown in figure 5.1.

The CALL statements between !OCL PARCALL and !OCL END PARCALL initiate subroutines which can be executed in parallel. Each CALL statement generates a child piece which is controlled by the corresponding subroutine. After generation of all child pieces, the piece executing !OCL PARCALL and !OCL END PARCALL (i.e. the parent piece) is set to waiting state. After all the called subroutines have completed their execution (which means that all child pieces have completed their execution), the parent piece is released from the waiting state and resumes its execution with the statement following the line !OCL END PARCALL.

The !OCL PARCALL may be extended by WITH BARRIER. See chapter 7 for the meaning of this additional construct.

Between !OCL PARCALL and !OCL END PARCALL, only CALL statements are allowed. So it is not permitted to nest these !OCL in a sequence of statements. Clearly, it is allowed to have a parallel call in a procedure which is called by another parallel call (as is shown in figure 5.2). With this relationship between parent and child pieces, a multilayer tree structure may be built. Figure 5.3 illustrates the structure of example 5.2.

```
!OCL PARCALL              ]
  CALL SUB1               ]   Generates two child pieces
  CALL SUB2               ]
!OCL END PARCALL          ]
    ...

  SUBROUTINE SUB2
!OCL PARCALL              ]
  CALL SUB21              ]   The piece which is generated above
  CALL SUB22              ]   generates two more child pieces
!OCL END PARCALL          ]
  END

  SUBROUTINE SUB1
    ...
  END
```

Figure 5.2

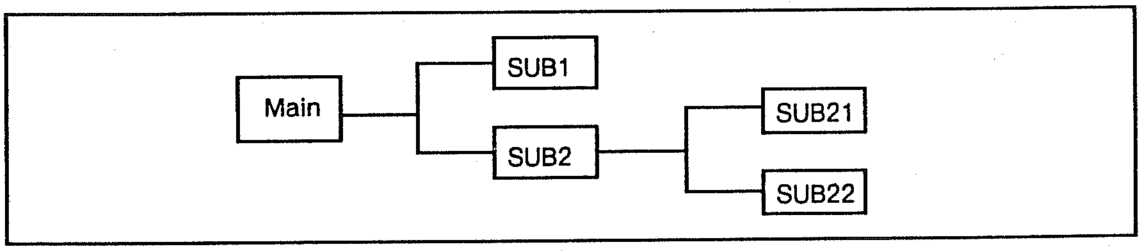


Figure 5.3

6. DATA ALLOCATION

Concerning parallel processing, it is necessary to manipulate address space that can be shared by different pieces. In addition, it must be possible to access address space that is private to a piece. Furthermore, the user can choose the primary (MSU) or secondary storage medium (SSU). For this reason there are two kinds of data attributes.

The first kind is the "Life Range". It may be static or dynamic. The life range determines when data is allocated in the storage and when it is released from the storage. The life range determines also, which pieces have access to the data.

The second kind is the "Allocated Area". In the S Series the allocated area is either MSU or SSU. For common blocks both areas are possible. Variables which are not in a common block must be allocated in the MSU.

Since allocation in the SSU is always static, there remain three combinations:

attribute	life range	area
AUTOMATIC	dynamic	MSU
SAVE	static	MSU
GLOBAL	static	SSU

The three attributes AUTOMATIC, SAVE, and GLOBAL exclude one another, so data cannot have two of these attributes. For common blocks, the default attribute is SAVE. The other two attributes AUTOMATIC and GLOBAL must be set explicitly. For blank common, the attribute SAVE is fixed. For named common blocks all three attributes are possible. The declaration of common blocks must be consistent. This means: "A common block declared with one attribute in some procedure must not be declared with a different attribute in another procedure."

Data which is not in a common block can have the attributes AUTOMATIC or SAVE. Variables specified in a SAVE statement and variables initialized in a DATA statement have the SAVE attribute. All other local variables have the attribute AUTOMATIC.

Variables with the SAVE attribute are allocated on the MSU when the program starts, and released when the program ends. To retain compatibility with existing programs, the default attribute for common blocks is SAVE. (Up to now it has been a common practise to assume that the data in a common block "live" from program start to program end, although the FORTRAN standard does not guarantee this property.)

Data with the GLOBAL attribute is also allocated when the program starts and released when the program ends. But in contrast to the SAVE variables, GLOBAL data is allocated on the SSU.

To provide private data to different pieces, data with the AUTOMATIC attribute is used. This data is dynamically allocated on the MSU, when a routine which uses them starts, and it is released

!OCL GLOBAL COMMON	Allocates statically on the SSU (COMMON statement from FORTRAN)
COMMON SAVE	Static allocation on the MSU (COMMON statement from FORTRAN) (SAVE statement from FORTRAN)
!OCL AUTOMATIC COMMON	Allocates dynamically on the MSU (COMMON statement from FORTRAN)

Figure 6.1

when this routine ends. So far, the life range of our AUTOMATIC attribute is independent from parallelization.

It follows from these considerations, that for data in a main program the life ranges of AUTOMATIC and of SAVE are identical.

In the current S Series, the System Storage Unit (SSU) will be used as a secondary storage for allocation of common blocks. Therefore, the life range of common blocks with the attributes GLOBAL or SAVE is identical. The first purpose of the SSU is to enable the programmer to write programs with a memory size larger than the size of the existing Main Storage Unit. But because of the memory hierarchy, the models of the S Series can easily be upgraded to multiprocessor systems, where the communication between processors is carried out via the SSU; and for these future cases the life range of common blocks with the attributes GLOBAL or SAVE may be different (depending on the program structure). This will be described in detail in a forthcoming publication.

The handling of data which is not in a common block is straightforward. Data with the SAVE attribute is shared. Data with the AUTOMATIC attribute are dynamically allocated and released. If a subroutine is executed twice in parallel, the two instances of the routine use the same SAVE variables but different sets of AUTOMATIC variables. With serial calls, the AUTOMATIC variables may be different or may be the same. This ambiguity does not affect a semantically correct program.

The handling of common blocks is more flexible and more complex. It is outlined in the following pages. For common blocks, the attributes AUTOMATIC and GLOBAL are defined through OCLs. For the SAVE attribute we do not need a OCL, because this attribute can be defined via the SAVE statement within FORTRAN. The definition of the attributes for common blocks is shown in figure 6.1.

6.1 Common blocks with AUTOMATIC attribute

The area for data with the AUTOMATIC attribute is allocated and released dynamically on the MSU. The life range is restricted to the subroutine calling the common: "It will be allocated when the subroutine which uses the common block starts, and deleted when the subroutine ends." Please notice, that the life range of an AUTOMATIC common block is independent from parallel process-

```
      SUBROUTINE DRIVER               Parent piece
!OCL AUTOMATIC(CM)
      COMMON /CM/ X,Y,Z               /CM/ has AUTOMATIC attribute.
      CALL SUB1                       /CM/ is allocated when DRIVER starts
      CALL SUB2                       and released at its termination.
      END  MAIN

      SUBROUTINE SUB1                 Called by DRIVER
!OCL AUTOMATIC(CM)
      COMMON /CM/ A,B,C               /CM/ denotes the same common block
        ...                           as declared in DRIVER.
        ...                           /CM/A,B,C is equal to /CM/X,Y,Z
      END  SUB1

      SUBROUTINE SUB2                 Called by DRIVER
!OCL AUTOMATIC(AA)
      COMMON /AA/ T,U,V               /AA/ is allocated when subroutine SUB2
        ...                           starts and released at its termination.
      END
```

Figure 6.2

ing. If a program is executed sequentially, the !OCL AUTOMATIC will not cause any problem.

```
     !OCL AUTOMATIC(CM)
        COMMON /CM/ varlist
```

When /CM/ is already declared as an AUTOMATIC common block in a procedure of the calling history, then /CM/ will be the same as the instance of that /CM/ which has been declared in the latest procedure of the calling history. On the other hand, when there is no other /CM/ in the calling history, then a new instance is allocated for /CM/. This instance will be released when the procedure terminates. The different cases are shown in Figure 6.2.

To improve program reliability, another variant of the AUTOMATIC OCL can be specified:

```
!OCL OLD AUTOMATIC(CM)
   COMMON /CM/ varlist
```

When /CM/ has been declared as an AUTOMATIC common in a procedure of the calling history, the meaning of !OCL OLD AUTOMATIC does not differ from !OCL AUTOMATIC. The same (already allocated) common block is used. However, when /CM/ is not declared by any procedure in the calling history, the occurrence of this !OCL is a semantic error and the program is abended. In other words, /CM/ must already have been allocated, when a procedure is called which contains !OCL OLD AUTOMATIC.

```
   SUBROUTINE DRIVER           Outer piece
   COMMON /STATIC/ A           /STATIC/ has SAVE attribute; it is allocated when the
     ...                       main program starts and released at its termination
     ...
!OCL PARCALL                   Generates two child pieces
   CALL SUB1
   CALL SUB1
!OCL END PARCALL
   END  DRIVER

   SUBROUTINE SUB1             Called by DRIVER
!OCL AUTOMATIC(AUTO)           /AUTO/ has AUTOMATIC attribute. /AUTO/ is
   COMMON /STATIC/ A           allocated when SUB1 starts and released at its
   COMMON /AUTO/ B             termination.
     ...
!OCL PARCALL                   Generates two child pieces
   CALL SUB2
   CALL SUB2
!OCL END PARCALL
     ...
   END  SUB1

   SUBROUTINE SUB2             Called by SUB1
!OCL AUTOMATIC(AUTO)           /AUTO/ has AUTOMATIC attribute. The version
   COMMON /STATIC/ A           allocated by SUB1 is used.
   COMMON /AUTO/ B
     ...
   END  SUB2
```

Figure 6.3

6.2 Common blocks with SAVE attribute

The instance for common blocks with SAVE attribute is allocated at the beginning of the program and released at its termination. Therefore, the life range is static, which means that it exists as long as the program is in execution and that there is only one instance of this common block. See the program in figure 6.3 for details.

Subroutine SUB1 is called in parallel from the DRIVER subroutine, and subroutine SUB2 is called twice during each execution of SUB1. Because there is no !OCL for common block /STATIC/, the default value SAVE is assumed, and the two instances of SUB1 share common block /STATIC/.

In contrast, there is !OCL AUTOMATIC for common block /AUTO/ in SUB1 and SUB2. So its variables are allocated when SUB1 starts and released when SUB1 ends. The second execution of SUB1 allocates another instance of /AUTO/. Therefore, the two instances of SUB1 do not share common block /AUTO/. Subroutine SUB2 does not allocate /AUTO/, because common block /AUTO/ exists already, when the subroutine SUB2 is invoked; and SUB2 does not delete /AUTO/, because it has been allocated by another routine in the calling history. Each of the four instances of SUB2 uses the /AUTO/ from that instance of SUB1, which called it. Two different versions of common block /AUTO/ exist concurrently in the program.

Figure 6.4 shows the relations between the pieces of this example. /STATIC/ is the same in all pieces. /AUTO/ is different: Pieces P1, P3, and P4 use one instance of /AUTO/, pieces P2, P5, and P6 use a second instance of it.

6.3 Common blocks with GLOBAL attribute

The instance of common blocks with the GLOBAL attribute is allocated on the SSU at the beginning of the program an released at the termination. The difference to the SAVE attribute is the allocated area (MSU for SAVE).

For the moment, the GLOBAL attribute allows the programmer to write programs with a memory size larger than the size of the existing Main Storage Unit by using the SSU via the GLOBAL OCL. In the future, the GLOBAL attribute will become increasingly important, when parallel programs can be written by fully using the hierarchic storage architecture of the S Series.

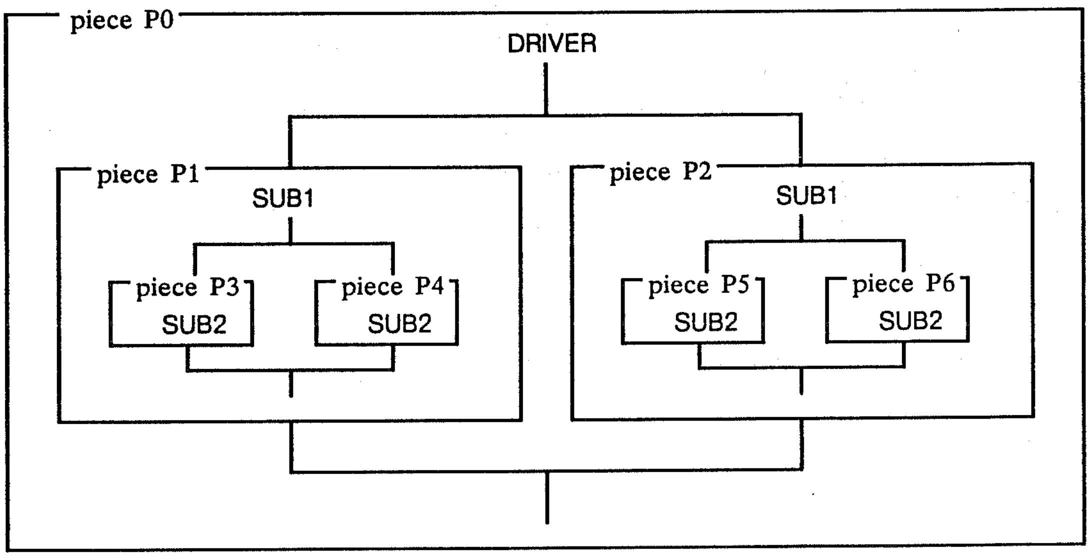


Figure 6.4

7. SYNCHRONIZATION MECHANISMS

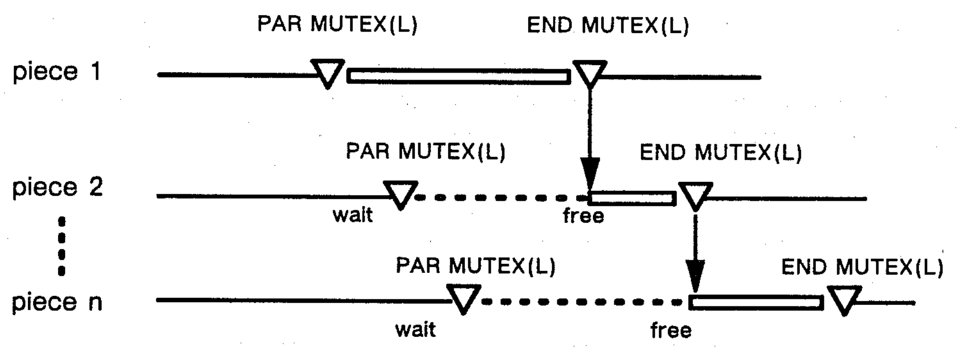
The synchronization between pieces during parallel execution is implemented by means of OCLs and service subroutines for parallel processing. The mechanisms used for synchronization are critical sections, POST/WAIT constructs, semaphores, and barriers.

7.1 Critical Section

Critical sections are parts within pieces which at any time must be executed exclusively by only one piece. Other pieces which would like to enter the same critical section have to wait until the critical section has been released. Imagine an address space where access to and update of the variables should be done exclusively. Because sequential execution of a parallelized program containing critical sections on only **one** processor does not change the semantics of the program, OCLs are suitable for critical sections. For one processor there is always only one piece executing a critical section.

To define critical sections, the OCLs from figure 7.1 are used. The lines !OCL PAR MUTEX(var) and !OCL END MUTEX(var) establish the exclusive control. The variable "var" specified in !OCL PAR MUTEX(var) is called "mutex variable". Critical sections which refer to the same mutex variable are "mutually exclusive". Therefore, during the execution of one critical section, other critical sections which refer to the same mutex variable cannot be executed. If a piece tries to execute such a critical section, it will be put to waiting state at the entry of the critical section.

!OCL MUTEXID(var)	Declares mutex variable
!OCL PAR MUTEX(var)	To get mutex
!OCL END MUTEX(var)	To release mutex



Piece 1 gets the mutex and is executing the critical section exclusively.
Piece 2 tries to get the mutex but will be put to a waiting state until Piece 1 releases the mutex. After that Piece 2 may enter the critical section, etc.
▭ : indicates the critical section

Figure 7.1

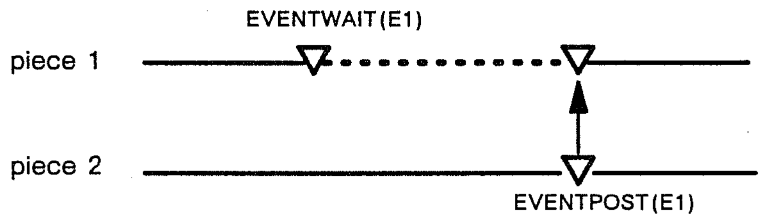
```
!OCL PAR MUTEX(L1)
     CALL SUB              Critical section can reside in SUB
!OCL END MUTEX
```

Figure 7.2

The mutex variable can either be a scalar or an array element. It cannot be referred to as an operand of an operator. No values can be assigned to this variable.

Critical sections can be nested following the sequence of description and execution. If a deadlock occurs by incorrect nesting, the responsibility is always on the user's side.

!OCL EVENTID(var)	Declares an event variable
CALL EVENTPOST (var)	To post the event
CALL EVENTWAIT (var)	To wait for the event
CALL EVENTRESET (var)	To clear the event variable



Piece 1 has to wait for an event and cannot continue execution
Piece 2 posts the event releasing Piece 1 from the waiting status

Figure 7.3

```
   PROGRAM MAIN
!OCL EVENTID(EV1,EV2)                  Declaration of EV1 and EV2 as event
                                                  variables
   REAL A(10), B(10), C(10)
   ...
   CALL EVENTRESET (EV1)               Initialization of event variables
   CALL EVENTRESET (EV2)
   ...
!OCL PARCALL                           Generation of two child pieces.  Event
   CALL SUB1 (EV1, EV2, A, B, C)                  variables may be transferred
   CALL SUB2 (EV1, EV2, A, B, C)                  as arguments
!OCL END PARCALL
   ...
   END  MAIN

   SUBROUTINE SUB1 (EV1, EV2, A, B, C)
!OCL EVENTID(EV1,EV2)                  Declaration of dummy arguments as
   REAL A(*), B(*), C(*)                          event variables
   ...
   CALL EVENTWAIT (EV1)                Wait until A(5) is initialized
   B(5) = A(5) + 1.0
   CALL EVENTPOST (EV2)                Post that B(5) is initilaized
   ...
   END  SUB1

   SUBROUTINE SUB2 (EV1, EV2, A, B, C)
!OCL EVENTID (EV1, EV2)                Declaration of dummy arguments as
   REAL A(*), B(*), C(*)                          event variables
   ...
   A(5) = 17.0
   CALL EVENTPOST (EV1)                Post that A(5) is initialized
   CALL EVENTWAIT (EV2)                Wait until B(5) is initialized
   C(5) = B(5) + 1.0
   ...
   END  SUB2
```

Figure 7.4

7.2 POST/WAIT mechanism

The POST/WAIT mechanism allows the programmer to define certain program points within pieces, where pieces have to wait for an event (i.e. until this event has been posted). Let's assume a piece reaching a statement which asks for the instance of an event. The piece will be put to waiting state, if the event has not yet occurred. This waiting state will be released and the piece will come to ready state, when the event will be posted. The declaration of the event variable is done via OCLs, the POST/WAIT mechanism itself needs service subroutines for parallel processing. Figure 7.3 shows the necessary OCLs and service subroutines and demonstrates the synchronization between pieces via the POST/WAIT mechanism.

The service routines EVENTPOST, EVENTWAIT, and EVENTRESET realize a synchronization mechanism of the POST/WAIT type. The variable for the synchronization of a POST/WAIT type synchronization is declared with !OCL EVENTID. This variable is called event variable or event type variable. Scalar variables or array elements are allowed for event variables.

Figure 7.4 shows an example with two event variables. The POST/WAIT construct with variable EV1 guarantees that SUB1 uses A(5) only after SUB2 did assign a value to this array element. Similarly, through EVENTPOST and EVENTWAIT with varible EV2, it is guaranteed, that the use of B(5) in SUB2 is postponed until SUB1 has assigned a value to B(5).

7.3 Semaphore

To create a synchronization mechanism via semaphores, the OCLs and service routines from figure 7.5 are used.

The service routines SEMAGET, SEMASET, SEMASIGNAL, and SEMAWAIT realize a multi-valued synchronization mechanism. The variable for this synchronization mechanism (here denoted by svar) is declared by !OCL SEMAPHORE. This variable is called semaphore variable or semaphore type variable. Scalar variables or array elements are allowed for semaphore variables.

SEMAGET(svar,ivar): The value of the semaphore svar is stored into integer variable ivar. The value and the state of the semaphore are not changed by this operation.

SEMASET(svar,iexp): The value of the integer expression iexp is stored into the semaphore svar. The value of the expression must be greater than or equal to 0 (iexp >= 0). If the original value of svar was less than 0, all pieces waiting for svar are set to ready state.

SEMASIGNAL(svar): If the value of the semaphore svar is less than 0, one of the pieces waiting on svar is selected and set to ready state. If the value of svar is greater than or equal to 0, no waiting piece is set to ready state. In both cases, svar is incremented by 1.

SEMAWAIT(svar): If the value of the semaphore svar is less than or equal to 0, the piece which is executing this call to SEMAWAIT is set to waiting state on semaphore svar. If the value of the semaphore svar is greater than 0, the execution of the piece can continue. In both cases, the value of svar is decreased by 1.

Figure 7.6 shows an example, where semaphores are used to simulate a shop where only one

```
!OCL SEMAPHORE(svar)
     CALL SEMAGET (svar,ivar)
     CALL SEMASET (svar,iexp)
     CALL SEMASIGNAL (svar)
     CALL SEMAWAIT (svar)
```

Figure 7.5

```
      SUBROUTINE Initialize_empty_shop
     !OCL SEMAPHORE(STOCK,EMPT)
      COMMON /SEMPH/ STOCK, EMPT
      CALL SEMASET (STOCK, 0)
      CALL SEMASET (EMPT, <capacity of shop>)
      END Initialize_empty_shop

      SUBROUTINE CLIENT
     !OCL SEMAPHORE(STOCK,EMPT)
      COMMON /SEMPH/ STOCK, EMPT
      CALL SEMAWAIT (STOCK)
      <take one item>
      CALL SEMASIGNAL (EMPT)
      END CLIENT

      SUBROUTINE SERVER (N)
     !OCL SEMAPHORE(STOCK,EMPT)
      COMMON /SEMPH/ STOCK, EMPT
      CALL SEMAGET (EMPT, NPOSS)
      IF ( NPOSS .GE. N ) THEN
         <put N items into the store>
         CALL SEMASET (EMPT, NPOSS-N)
         CALL SEMAGET (STOCK, NAVAIL)
         CALL SEMASET (STOCK, NAVAIL+N)
      END IF
      END SERVER
```

Figure 7.6

product is sold. The semaphore variable STOCK keeps account of the number of items in the shop. The value of a second semaphore variable EMPT is the current number of items for which there is space available in the shop. Whenever a client buys an item, STOCK is decreased by 1, and EMPT is increased by 1. When the server delivers N items, EMPT is decreased by N, and STOCK is increased by N. Through calls to SEMAGET within the subroutine SERVER, NPOSS receives the current value of EMPT (maximum number of items which can be delivered), and NAVAIL receives the current value of STOCK (number of items in the store). Critical sections should be defined to protect STOCK respectively EMPT from being modified between CALL SEMAGET and the corresponding CALL SEMASET. This was omitted for simplicity.

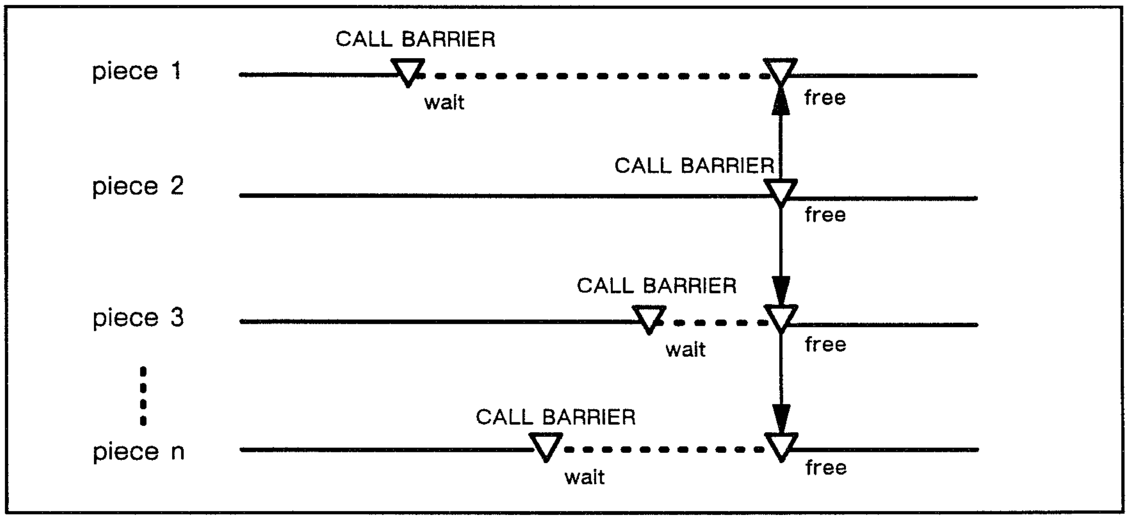
7.4 Barrier

The barrier synchronization follows the principle that pieces generated at a given FORK construct are allowed to pass a specified barrier point only when all of them have reached their barrier point so that all pieces can simultaneously cross the barrier. Because of the different runtime behaviour of pieces, it will usually happen that the barrier will be reached by the pieces at different times. Therefore, incoming pieces will be put to waiting state. After all pieces have reached the barrier, all of them may resume their execution. Our barrier synchronization does not need any barrier variable, because a barrier is connected to a fork of pieces and the service subroutines for parallel processing are aware of the number of parallel pieces. The OCL and service routine for barriers are shown in figure 7.7. The mechanism is shown in figure 7.8, and an example is given in figure 7.9.

The lines from !OCL PARCALL WITH BARRIER until !OCL END PARCALL will be called "pieces–

```
!OCL PARCALL WITH BARRIER
CALL BARRIER
```

Figure 7.7



When a piece reaches the barrier, it has to wait until all pieces arrived at the barrier. When the last piece has arrived, all pieces are simultaneously freed.
- - - - - - - - : indicates wait time

Figure 7.8

```
   PROGRAM main

!OCL PARCALL WITH BARRIER
   CALL G1
   CALL G2
!OCL END PARCALL
   END  main

   SUBROUTINE G1
!OCL PARCALL
   CALL XY
   CALL XY
!OCL END PARCALL
   END  G1

   SUBROUTINE G2
   CALL BARRIER
   END  G2

   SUBROUTINE XY
   CALL BARRIER
   END  XY
```

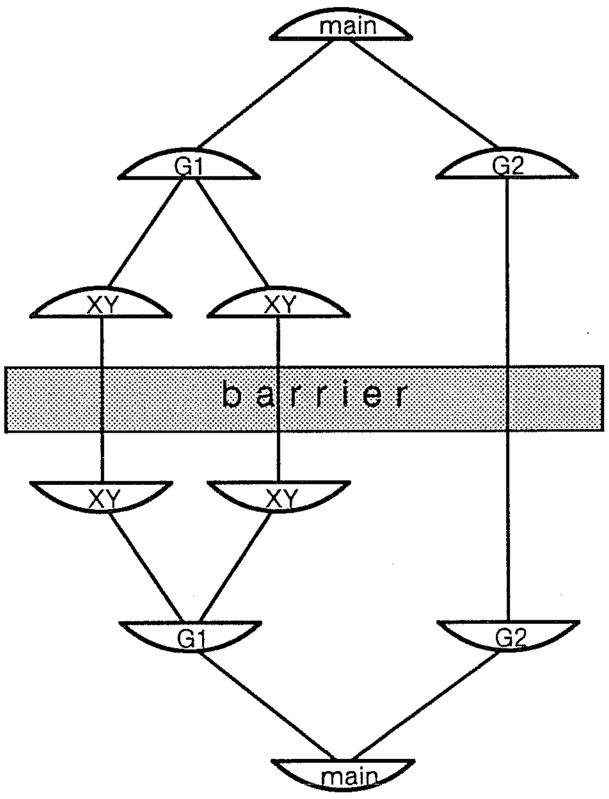


Figure 7.9

generating sequence with barrier". The statement CALL BARRIER can only be executed within a piece which has been generated in such a "pieces-generating sequence with barrier" or by a piece which descends from a piece generated in such a sequence. If a piece executes CALL BARRIER, the nearest fork with a barrier is searched and the calling piece is set to waiting state until all pieces generated at this fork have reached a barrier. When all pieces have reached the barrier, all pieces are set to ready state and are allowed to proceed.

8. CONCLUSION

As parallel programming becomes more and more important, the Siemens multiprocessors from the S Series are the only system in which the requirements of the programmer (writing parallel code so as to have the shortest elapsed time for his own program) are fulfilled without any conflict to the supercomputer center's philosophy (with maximum utilization of the resources and therefore best throughput of the whole system). The S/40 systems formed by a two-fold combination of the **Dual Scalar Architecture** meet both requirements. The corresponding programming concept is easy and powerful and has been oriented to the standard FORTRAN syntax. It has been designed in such a way that the programmer will be removed as far as possible from the burden of parallelization.

REFERENCES

Siemens, 1990: S Series General Description

Siemens, 1989: S Series Summary Description

Siemens, 1986: FORTRAN77 Reference Summary

Siemens, 1988: FORTRAN77 User's Guide

Siemens, 1988: FORTRAN77/VP User's Guide

Siemens, 1990: FORTRAN77/VP Vectorization Handbook