A Standardized Methodology for the Linkage of Computer Codes. Application to RELAP5 / Mod3.2

CSN

Colección
Otros Documentos
32.2004

CSN CONSEJO DE SEGURIDAD NUCLEAR

A Standardized Methodology for the Linkage of Computer Codes. Application to RELAP5/Mod3.2

Roberto Herrero Santos
Consejo de Seguridad Nuclear

CSN CONSEJO DE SEGURIDAD NUCLEAR

Colección
Otros documentos CSN
Referencia: ODE-04.21

Publicado y distribuido por:
Consejo de Seguridad Nuclear
Justo Dorado, 11. 28040 - Madrid
http://www.csn.es
peticiones@csn.es

Maquetación: RGB Comunicación, S.L.
Imprime:
Depósito legal:

Contents

Presentación 5

I. Introduction 9

II. Code linkage 13
- II.1. Introduction 15
- II.2. Boundary conditions vs. initial conditions linkage 16
- II.3. Non iterative vs. iterative linkage 16
- II.4. Serial vs. parallel linkage 18
- II.5. Time step control 18

III. Code parallelisation 21
- III.1. Introduction 23
- III.2. Parallelisation techniques 23

IV. A proposal of standard 27
- IV.1. Introduction 29
- IV.2. Main characteristics of BABIECA 30
- IV.3. Standard specifications relating the modular driver 31
- IV.4. Standard specifications relating the codes to be coupled 32
- IV.5. Linkage of the RELAP5 code 34

V. Application example 37
- V.1. Problem setup 39
- V.2. Results 41
- V.3. Run statistics 41

VI. Conclusions 45

Bibliography 49

Appendices 55
- A. File `sndcode` 57
- B. File `rcvcode` 83
- C. Remote code standard specifications 91
- D. BABIECA input file 104

Presentación

Presentación

La Escuela de Ingenieros Industriales de la Universidad Politécnica de Madrid ha desarrollado, en colaboración con el Consejo de Seguridad Nuclear, una metodología estándar para conexión y paralelización de códigos de cálculo científico, utilizando el lenguaje de simulación modular de propósito general BABIECA y el paradigma de paso de mensajes implantado en las librerías PVM. El estándar de conexión se implanta a través de una plantilla que sugiere cuál debe ser la estructura de un código de cálculo de manera que se facilite su conexión con otros códigos a través de BABIECA.

Mediante este esquema la conexión de códigos se define de manera similar a una topología de diagramas de bloques, esto es especificando qué salidas de cada código actúan como condiciones de contorno de otros.

Las particularidades de cada código son irrelevantes a los efectos de la conexión, de manera que el esquema no se restringe a ningún tipo concreto de códigos como pueden ser los neutrónico-termohidráulicos.

El esquema de conexión permite también al usuario paralelizar los códigos implicados, mediante técnicas de granularidad gruesa, si el problema definido a través del fichero de entrada así lo permite.

El esquema de conexión propuesto ha sido aplicado satisfactoriamente al código termohidráulico RELAP5/Mod3.2. Se ha utilizado un pequeño problema ejemplo para demostrar tanto las capacidades de paralelización como de conexión con otros códigos de la versión modificada de RELAP5. Bajo ciertas condiciones, se obtiene una aceleración efectiva de la velocidad de ejecución del cálculo.

A la vista de los resultados, se puede concluir que la estructura de cualquier código de cálculo puede ser estandarizada par facilitar su conexión con otros códigos. Se adjunta una propuesta de dicha estructura.

Presentation

The School of Industrial Engineering of the Universidad Polit´ecnica de Madrid (ETSII-UPM), in cooperation with the Spahish Nuclear Safety Council, has developed a standardised methodology to couple and parallelise scientific codes, by means of the modular general purpose simulation language BABIECA and the Message Passing Paradigm, currently using the PVM library routines. A template suggests how a code must be written to ease the connectivity with other codes through BABIECA.

With this approach, the connection is performed by defining a block topology, i.e., what output signals of each code acts as boundary or initial conditions for other codes.

The nature of the individual codes is irrelevant to the connection process. Thus, connectivity is not restricted to the usual TH-neutronic codes.

The linkage scheme also allows the user to undertake coarse grain parallelization of the codes, if the particular problem defined through the input deck so allows.

The proposed linkage scheme has been successfully applied to the RELAP5/Mod3.2 code. A simple example problem has been run to demonstrate the capabilities of the modified version of RELAP5 to be both parallelized and connected to other codes. Effective execution speed up is obtained under certain circumstances.

The trend of the results suggests that the structure of any scientific code can be standardised to ease the coupling with others, which is the main conclusion of the report. A proposal of such a standard is enclosed.

I. Introduction

I. Introduction

Simulationists that use particular purpose scientific codes for certain disciplines can reach their limits of scope, mainly because of the end of applicability of the models capable of handling the scenarios they try to simulate. The scope of application of the codes can be extended by building a full scope code including detailed models for all the disciplines involved. Nevertheless, the effort necessary to write such a code would exceed by far the benefits to be yielded [28,29].

A more feasible solution is to take advantage of already existing codes and couple them, in such a way that the output variables of a given code act as boundary or initial conditions for other codes.

Many examples of code coupling, most of them taken from the nuclear industry, can be found in the literature. For instance, the EUMOD interface [28], which allows the connection of the RELAP5 code [30] to external models defined by the user. Reference [29] describes the connections RELAP5–CONTAIN, RELAP5–COSBWR, RELAP5–PANBOX2, RELAP5–HECHAN2, RELAP5–COCO, TRAC-BF1–NEM-3D, CATHENA–PACE, CATHENA–ELOCA and CANSIM. Reference [24] describes the connection of RELAP5 to the TACCUM model. Another remarkable application is ESTER [19], a code that links a set of codes, with the aim of simulating severe accident scenarios in nuclear power plants.

References [8,7,3,20] illustrate some of the last works on code coupling applications in the Nuclear Safety field.

Special attention to the code connection issue has been paid in works such as [5,31,21,26,1] within the frame of the OECD/CSNI Workshop on Transient Thermal-Hydraulic and Neutronic Requirements held at Annapolis in 1996.

An important issue concerning the aforementioned connections is that they have been implemented taking into account the peculiarities of each individual code; no general rules are provided for code connection. Besides, most of the connections, with the exceptions of the code systems CANSIM and ESTER, link only two codes. Thus the scope of applicability is increased in only one direction.

A number of drawbacks can be found in this linkage strategy:

- the new code that results from this type of linkage is closed, in the sense that the executable file is always the same once the connection has been established. If we want to enhance the available capabilities a new code must be added and the system must be recompiled to generate a new executable file;
- if the transients to be simulated require only the models implemented in one of the codes, or a few of them, for some time, the system described above forces to use the global system formed by all the codes that have been linked.

Parallelisation of computer codes allows shorter execution times and makes real time simulation more feasible. In most cases simulation codes are parallelised taking into account

only the source code, but not the input file defined by the user, which may reflect in many cases an inherent parallel structure of the physical problem to be solved. This last strategy of parallelisation, that will be called *problem level parallelisation*, can lead to coarser granularities and is seldom exploited in scientific computing.

In this report we describe a methodology that suggests how to write a computer code for ease of both connectivity and problem level parallelisation, using the same tools.

II. Code linkage

II. Code linkage

II.1. Introduction

In order to anticipate the behaviour of a certain physical problem, a computer code solves a set of, in general, partial differential equations with given initial and boundary conditions, which can be denoted by $\underline{\underline{z}}$ and $\underline{\underline{u}}$ respectively. The components of vector $\underline{\underline{z}}$ are independent from each other. Closely related to the initial state vector, we can define the *extended state vector* as the vector $\underline{\underline{x}}$ that contains all the information in $\underline{\underline{z}}$ plus additional non-independent components, interrelated through algebraic equations solved by the code, i.e. $\underline{\underline{x}}(t_0) = H\left(\underline{\underline{z}}(t_0)\right)$. The set of partial differential equations is discretized in space and time and solved by means of a numerical algorithm. Loosely speaking, the algorithm consists in obtaining the extended state vector $\underline{\underline{x}}$ at a discrete time $t + \Delta t$, given both its value at time t and the vector of boundary conditions $\underline{\underline{u}}$. With respect to the boundary conditions, if the numerical method makes use of the values at time $t + \Delta t$ it is said to be implicit; if all conditions are taken at time t or prior it is said to be explicit. In between there can exist methods with different degrees of implicitness. Those two schemes can be formally expressed as follows:

$$\vec{x}(t + \Delta t) = F\left(\vec{x}(t),\ \vec{u}(t + \Delta t)\right) \tag{2.1}$$

for an implicit method and

$$\vec{x}(t + \Delta t) = F\left(\vec{x}(t),\ \vec{u}(t)\right) \tag{2.2}$$

for a totally explicit method.

Once the extended state vector is known, output variables are computed by means of some ancillary function:

$$\vec{y}(t + \Delta t) = G\left(\vec{x}(t + \Delta t),\ \vec{u}\right) \tag{2.3}$$

where the vector of boundary conditions $\underline{\underline{u}}$ can be defined at time t or $t + \Delta t$. With this last calculation the time step is considered as finished.

The code must be fed with the vector of initial conditions $\underline{\underline{z}}$, and thus the code must find the extended state vector $\underline{\underline{x}}(t_0)$ at the first time step from $\underline{\underline{z}}$ and the algebraic equations to be solved in the code. Vector $\underline{\underline{x}}$ avoids recomputation of some variables in different time steps, that would be necessary if just $\underline{\underline{z}}$ were used. The boundary conditions are specified by the user by means of tables, with time as abscissae and the values to be input as ordinates. These tables are usually linearly interpolated.

The main objective of code linkage is to provide a code with information derived from the outputs of other codes, i.e. for the case of boundary conditions,

$$\vec{u}_i(t) = \vec{u}_i\left(\vec{y}_1(t), \ldots, \vec{y}_i(t), \ldots, \vec{y}_n(t)\right) \tag{2.4}$$

while for initial conditions,

$$\vec{z}_i(t_0) = \vec{z}_i\left(\vec{y}_1(t_0), \ldots, \vec{y}_i(t_0), \ldots, \vec{y}_m(t_0)\right) \tag{2.5}$$

where the subindices denote different codes.

The linkage between codes can be established following different principles. The main issues concerning code connection are described hereafter.

II.2. Boundary conditions vs. initial conditions linkage

The main purpose of code linkage is to calculate the boundary conditions of a given code as a function of the output variables of other codes, as shown in equation (2.4).

Nevertheless, the coupling of codes via initial conditions, as set in equation (2.5) may be highly interesting in the solution of certain problems. For instance, many times a physical process is simulated with the help of a certain model. Some particular transients may drive the system into a region out of the scope of applicability of the model (for instance, a code for the simulation of transients and accidents in nuclear power plants enters the severe accident region), making it necessary the existence of a second model, usually within a separate code, capable of handling this new situation. Without code linkage, an input deck for this particular initial conditions must be made for running the special code. However, code linkage makes it possible to deactivate the incorrect models when they are just going to leave its domain of applicability and transfer these initial conditions 'on the fly' to the special code. It is very convenient to stop the calculation of the unwanted models both to save CPU time and to prevent loss of stability of the models that are out of their domain of applicability. An undetermined number of transitions between both models may happen during the whole simulation run. A good example of coupling via initial conditions is the linkage TIZONA–MAAP [27]. TIZONA [17] is a thermal-hydraulic code for BWR Nuclear power plants transient analysis. When certain variables reach predefined values, which indicate the beginning of severe accident conditions, the simulation is transferred to the MAAP code [10]. In this case, the transition is made only once, since it is not feasible that the plant recovers the 'non severe' conditions.

II.3. Non iterative vs. iterative linkage

The connection of codes often yields feedback loops. If the codes involved in a feedback loop use an explicit algorithm the linkage may be established without iterations, since all the information needed to advance the solution one time step is available from the previous one. This fact can be realized in the following equations, which represent the coupling of two explicit codes:

$$\square{x}_1(t+\Delta t) = F_1\left(\square{x}_1(t), \square{u}_1\left(\square{y}_2(t)\right)\right) \tag{2.6}$$

$$\square{x}_2(t+\Delta t) = F_2\left(\square{x}_2(t), \square{u}_2\left(\square{y}_1(t)\right)\right) \tag{2.7}$$

If at least one of the codes in the feedback loop is explicit the coupling can also be solved without iterations. We must ensure that the explicit codes are solved in first place, to propagate forward the information available from the previous time step. If code 1 is explicit and code 2 is implicit:

$$\square{x}_1(t+\Delta t) = F_1\left(\square{x}_1(t), \square{u}_1\left(\square{y}_2(t)\right)\right) \tag{2.8}$$

$$\square{x}_2(t+\Delta t) = F_2\left(\square{x}_2(t), \square{u}_2\left(\square{y}_1(t+\Delta t)\right)\right) \tag{2.9}$$

We can see that the first code only needs variables from the previous time step. When the second code demands variables defined in the current time step, i.e. $\sqcup{y}_1(t+\Delta t)$, this information is already available.

Most of the connections mentioned in chapter 1 have overlooked this feature. The implicit code, which is usually the thermal-hydraulic one, is solved in first place. The effect is that the output vector passed to equation (2.9) is $\sqcup{y}_1(t)$ instead of $\sqcup{y}_1(t+\Delta t)$. This misuse of the output vector may distort the numerical algorithm.

If all the codes in a feedback loop are solved by implicit methods the information necessary to advance a time step is not available, as shown in the following equations:

$$\square{x}_1(t+\Delta t) = F_1\left(\square{x}_1(t), \square{u}_1\left(\square{y}_2(t+\Delta t)\right)\right) \tag{2.10}$$

$$\square{x}_2(t+\Delta t) = F_2\left(\square{x}_2(t), \square{u}_2\left(\square{y}_1(t+\Delta t)\right)\right) \tag{2.11}$$

The output vector $\sqcup{y}_2(t+\Delta t)$ needed to calculate $\sqcup{x}_1(t+\Delta t)$ has not been computed yet. The same comment applies to the calculation of $\sqcup{x}_2(t+\Delta t)$. The natural way to solve this deadlock is to use iterative methods. An initial guess of $\sqcup{y}_2(t+\Delta t)$ is taken to evaluate $\sqcup{x}_1(t+\Delta t)$ and subsequently $\sqcup{y}_1(t+\Delta t)$. Now we can compute $\sqcup{x}_2(t+\Delta t)$ and $\sqcup{y}_2(t+\Delta t)$. If this last value is coincident with the initial guess within a tolerance margin the solution is considered valid. Otherwise, another iteration is executed. This way of acting follows the Picard fixed point theorem, by which a sufficient condition for convergence is that the vector function defining $\sqcup{x}_1(t+\Delta t)$ and $\sqcup{x}_2(t+\Delta t)$ in terms of itself be contractive.

The convergence of the loop can be accelerated by using traditional methods for finding roots of functions, such as the secant method, bisection method, Aitken's, Δ^2, etc.

II.4. Serial vs. parallel linkage

We say that a set of codes is coupled serially if the linkage gives raise to a single executable file such that all the codes belong to the same computer process when executed. This type of linkage must be accomplished by turning into sequentially called subroutines all the main programs of the codes to be linked, with the exception of the code that will act as main program of the integrated system and will manage the time step. The information exchange between the codes takes place through the arguments of the subroutines or through a common block of memory. The exchange is performed once the execution of each individual code has finished. These demands compel the developer to maintain two versions of each source code: the original one and that ready to be linked to the code system. The original executable file must also be kept if we want to run simulations that only need models included in that code. The EUMOD interface [28] for the RELAP5 code [30] and the program SIMTRAN [23] are good examples of serial linkage.

On the other hand, the codes are said to be coupled in parallel if they behave as separated computer processes. Each code has an associated executable file. There are as many main programs as codes linked. The connection is established via low level utilities such as the UNIX sockets or memory sharing or using the Message Passing paradigm, by means of higher level interfaces such as PVM [12,13] or MPI [15]. These tools provide flexible, standardised methods for transferring information between processes. The simulation starts in the code that manages the time step. Each code is spawned when needed as an individual process and may be run on a different processor of a parallel machine and even on a different machine (of the same or of a different kind), depending on the hardware resources available and the communication software. Some processes can run simultaneously if the problem conditions so allow. However, we must remark that parallel linkage does not imply always an actual parallelisation of the problem, which will not be possible if:

1. only one single processor machine is available, and thus only one computer process can be executed at the same time,
2. the synchronisation imposed by the simulation time step does not allow a process to advance on its own. Hence, only one code is executed at a given time.

With parallel coupling only one version of the source code and one executable file must be maintained per individual code, since the extra sections of code needed to implement the linkage will be inserted as additions to the original version and will only be entered when the code detects that it has been spawned by other process of the global system. All these features make parallel coupling more versatile and recommendable than serial coupling, even when the different processes do not run simultaneously. The connections RELAP5–TACCUM [24] and RELAP5-PARCS [8], among many others, illustrate this kind of linkage.

II.5. Time step control by code vs. time step control by a simulation driver

In the examples mentioned in chapter I the simulation time step is controlled by one of the codes, generally the thermal-hydraulic one, embedded in the code system. This scheme forces to use that

code in every simulation, even though it may not be necessary for certain transients. Moreover, it must be solved in first place, even if it is implicit. As it has been shown in section II.3, concerning iterative and non-iterative coupling, this is not a desirable practice.

New code coupling applications following this approach are arising. For instance, the general interface used in the RELAP5/PARCS coupling [8], the TALINK interface [7] to couple RELAP5 to the 3D neutronics code PANTHER and ISAS 1 [3], used to couple the TH code CATHARE to the 3D neutronics code CRONOS.

A different approach is to control the time step with a modular general purpose simulation driver, which calls the codes to be linked as if they were the blocks of a block diagram. The time step of the simulation driver must be passed to the individual codes being part of the global system. The simulation driver acts then as a synchronizer of the codes. With this scheme the simulation is not centered in any particular code, as is the case in the previous scheme. The user can choose only the codes needed for the particular transient and make provisions for code replacement upon exhaustion of the models. The modular approach permits the substitution of one of the codes by other code of the same type in order to make comparisons or to achieve different degrees of detail. Moreover, growth of the full model by addition of new codes may be accomplished more easily.

III. Code parallelisation

III. Code parallelisation

III.1. Introduction

Parallelisation consists basically in the concurrent execution of a set of processes that do not interchange information among them, and are part of a larger computer task or program, on different microprocessors (belonging to the same or to different machines). The information computed in each concurrent process is then transferred to the main task, which runs on another microprocessor, at a certain point of the flow diagram usually called *synchronisation barrier*. Once this point has been reached, the main task can proceed further with the program running it serially or spawning new concurrent tasks if the flow diagram so allows, until a new synchronisation barrier is reached. If the concurrent tasks do not interchange information during the execution of the whole program, i.e., there are no synchronisation barriers, the term *distributed computing*, instead of parallelisation, is preferred.

In both cases, the concurrent execution of tasks tends to decrease the real execution time of the whole program. On the other hand, the communication between the different processes increases the real execution time, which is the main shortcoming of parallelisation. Optimal performance is achieved when the three following issues are taken into account:

- the number of concurrent processes equals the number of available processors. If the former is greater than the latter no decrease of the real execution time is attained with respect to the optimal situation, since the extra tasks must be queued until a microprocessor is free. Moreover, the execution time will be charged due to the extra communication between processes,
- the heaviest tasks run on the most powerful microprocessors, i.e. the parallel tasks are balanced,
- the number of synchronisation barriers is minimum, hence decreasing the time spent in the communication between tasks.

With respect to this last item, we say that a parallel program with few synchronisation barriers and large tasks running on each microprocessor has a *coarse granularity*. On the other hand, a parallel program with many barriers, and small tasks running on each microprocessor, is said to have *fine granularity*. As a general rule for the programmer, the larger the granularity, the better the program performance, i.e. the less the real computing time.

III.2. Parallelisation techniques

III.2.1. Introduction

Systematic classification of the parallelisation techniques is not an easy job. It depends, of course, on the features we pay attention to. Many of the categories that would result frome each feature overlap. Nevertheless, we propose here a classification based on two features: the agent that performs the parallelization and the level at which parallelization is applied, either the code or the problem defined through the input level.

III.2.2. Code driven - programmer driven - user driven parallelization

The agent that undertakes the parallelization process may be a first item to set a classification. The degree of parallelism that can be achieved is different for each agent.

The most single parallelisation techniques just try to identify parallel sections in a program originally written to be run on a serial machine. Parallelization is performed by some ancillary tools usually called *autoparallelisers* [11]. These tools look for loop based and functional parallelism, as defined in [22]. With this strategy, that we shall call *code driven*, any chance of parallelisation is achieved at compilation time.

The use of parallel environments when developing a computer program is another way to achieve parallelization [11]. The developer is responsible for exploring the chances of parallelism when writing the program. This is the reason why we call this strategy *programmer driven* parallelization. Another form of programmer driven parallelization is the *message passing paradigm*, in which the developer inserts in the source code library functions to spawn other processes, and to send and receive data from them. PVM [2,13,12] and MPI [15] are the most popular libraries for message passing.

An intermediate cathegory, just between the two previous ones, is the use of parallelized library functions that performs typical algorithms in scientific computing. The most spread library of this type is BLAS (Basic Linear Algebra Subprograms) [11]. The developer that uses these library functions only inserts them into the source code, but is not responsible for the embedded parallelism.

A final category is the so called *user driven* parallelization, in which the program allows the user to specify through the input file the parallelization strategy. This strategy is strongly dependent on the particular problem to be solved. The parallelisation based on hydraulic loops in the CATHARE code, described in [4], is a good example of user driven parallelisation.

III.2.3. Source code level - problem level parallelization

We define *code level parallelization* as the parallelization applied only to the source code. This kind of parallelism shows some disadvantages:

- It usually leads to a medium-fine granularity.
- It can only identify the parallelism embedded in the source code, but not the parallelism that could be eventually derived from the source code plus the data specified through the input file.
- The capability of this technique to identify the parallel sections strongly depends on the way the serial program has been written. This task is much easier, as it is recognized in [22], if the developer takes into account the different possibilities of parallelisation when writing the source code and designing the computational algorithms.
- The parallel code, and hence the execution flow is highly machine-dependent.

As an alternative to the concept of source code level parallelisation, we would like to introduce the concept of *problem level parallelisation*. Many times the problem to be solved by a computer code,

and defined through the input file, shows intrinsic parallel characteristics. In the particular case of a simulation program, the parallel paths of the block diagram describing the simulated system suggest a way of parallelisation. Those loops derive most times from *physical parallelism*. For instance, the loops in a hydraulic system, the parallel branches in an electric circuit, etc.

This kind of parallelism can very hardly be attained by means of code driven techniques. Only programmer driven and user driven techniques are likely to succeed. As far as we know, only two applications have made use of it, both based on the message passing paradigm. The first one is the programmer driven parallelisation, in the terms defined in section III.2.3, of the thermal-hydraulic code TRAC-B by the Polytechnical University of Valencia [16]. In this version of TRAC-B the code identifies, after reading the input data, the hydraulic loops which are concurrently simulated. The time steps define the natural synchronisation barriers. The second application is the user driven parallelization of the CATHARE code [4], which has already been referenced in the previous section.

It is easy to realise that problem level parallelisation techniques may lead to a very coarse granularity, since it implies the concurrent simulation of large physical systems. However, this way of parallelisation allows in most cases to run only a few concurrent processes, very often less than the number of microprocessors available. A combination of problem level and source code level techniques seems to be a reasonable choice to optimise the available computational resources.

IV. A proposal of standard for connection and parallelisation of computer codes

IV. A proposal of standard for connection and parallelisation of computer codes

IV.1. Introduction

In chapter II we have shown that code linking has to do with the interchange of boundary and initial conditions between codes, but not with the particular method used by each code to advance the solution. The advancement of the solution, performed in equations (2.1) or (2.2), plus equation (2.3) is just what makes one code different from another. This allows us to affirm that the structure of any computer code based on the concept of time step can be standardised for ease of coupling with other codes. Reference [6] points out that standardization of scientific software is one of the most important issues for the systematic growth of the capabilities of the codes. The standard does not affect the advancement of the solution, and hence it does not impose any constraint to models or numerical schemes. It has more to do with the flow diagram of the code in each time step. If the standard is fulfilled, allocation of the library routines for sending and receiving messages is straightforward. Furthermore, the developers can insert the library routines in the appropriate places yielding executable files ready to be coupled with other codes, without releasing the source code.

The library used to implement the message passing between different processes is PVM. This choice is justified because PVM can operate among nets of computers with heterogeneous architectures. MPI shows better performance when operating on computers of the same type, but only some implementations (e.g. [25]) can work with heterogeneous architectures. Since the codes to be coupled may be eventually compiled in very different machines, the highest priority has been given to interoperability. The University of Tennessee and Oak Ridge National Laboratory are carrying out research to merge the advantages of both PVM and MPI in a single product called PVMPI [14,9].

The standard also assumes that the linkage will be accomplished through a general purpose modular simulation language acting as an intermediate driver that manages the time step, as described in section II.5. With the methodology introduced here, parallel iterative or non iterative linking, through boundary and/or initial conditions can be attained.

The methodology also permits user driven, problem level parallelisation of the codes, as explained section III.2. This is possible because code linkage through parallel interfaces and parallelisation both use the message passing paradigm between different computer processes. Linkage and parallelisation become then two sides of the same coin when performed through a modular simulation language.

The BABIECA simulation language [18] will be used to illustrate the standardised methodology, although any other program with similar features, such as MATLAB or MATRIX$_X$, would do the job.

With the approach proposed here users can use as many codes as needed for the particular simulated transient, connecting them as if they were the pieces of a wrecker, even substituting different models depending on the course of the simulation.

IV.2. Main characteristics of BABIECA

Here we describe the characteristics of BABIECA needed to understand its behaviour as a simulation driver for linkage and parallelisation of different codes.

BABIECA is the driver of the continuous, general purpose simulation language integrated in the package CAMPEADOR [18]. The simulation language is modular, which means that the problem to be solved is defined as a block diagram. Each block may be considered as a multi input-multi output relation between time-evolving variables, and represents a physical system or a mathematical procedure. The input-output relation consists of a set of differential equations plus a numerical solution algorithm (implicit). There is no common solution algorithm for all the blocks, as it is the norm in most simulation languages. Blocks are particular instances of computational entities called modules. The set of private data of the block defines it, while it is the connection among the blocks and the overall computation order that constitute the global numerical scheme. Modules cover a broad range of mathematical tools, physical models and special components, and are mainly oriented, but not limited to the solution of large thermal-hydraulic networks like those appearing in Nuclear Power Plants.

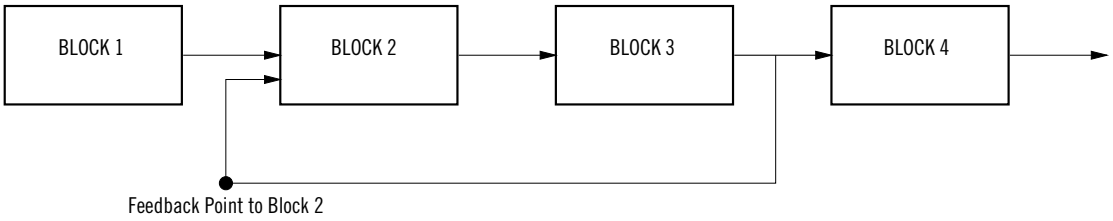

Figure 4.1. Example of block calculation in BABIECA

The driver routine of BABIECA manages the time step control and calls the modules sequentially in a user-defined order. When all the blocks have been successfully computed the time step is considered to be finished, and a new time step calculation begins.

The calculation order may be broken by feedback loops. These arise in situations when an output signal is needed and it is not yet calculated. Feedback loops can be solved iteratively by defining *feedback points*. A feedback point checks if the values of a given output signal in two successive iterations are coincident within a tolerance margin. The convergence may be accelerated by using numerical methods for fixed point or root finding. The sequence of solution used by BABIECA is illustrated with the example shown in figure 4.1.

Blocks 1, 2 and 3 are computed in sequence. If the output of block 3 does not agree with the initial guess value used as input of block 2 within the specified tolerance margin, blocks 2 and 3 are computed again, after resetting the extended state vector to the value at the beginning of the time step. Once the convergence criterion has been met block 4 is called, finishing the current time step.

One remarkable feature of BABIECA that allows the dynamic replacement of blocks is that the execution of a block can be stopped, remaining in a stand-by state, when a certain criterion has been reached. When the block is restarted new initial conditions must be provided. This feature is crucial to the code linkage scheme via initial conditions.

IV.3. Standard specifications relating the modular driver

The linkage of codes via the BABIECA simulation driver, or any other similar, is accomplished by turning them into BABIECA modules. Despite the very different nature of the codes to be linked it is possible to build a single general module that constitutes the framework for the connection of any code, covering the code-independent operations and allowing specific tasks to be inserted appropriately. To be more precise two modules have been written, one to spawn and send data to the code and other to receive data from it. These modules are called `sndcode` and `rcvcode` respectively. They are documented in appendices A and B. This apparently artificial splitting into two modules is justified by parallelisation reasons, as it will be seen later. These two modules work together as follows:

Although the documentation provides full explanation about the functioning of those modules, we will point here some general guidelines. First of all, the emission block spawns the remote code and sends the names of those variables that will act as boundary conditions and those variables that will act as initial conditions by using PVM library functions. The block sends the code the current simulation time and the BABIECA time step. Moreover, the block computes the boundary conditions from the block input signals and sends them to the remote code. Allowance is made for the case where the remote code was not active during the previous time step and needs to be initialised. The initial conditions are computed from the block input variables supplied for this purpose and sent to the remote code. No more tasks are to be done by the emission block.

The reception block takes from its counterpart emission block the process identification number of the remote code, necessary to receive the output variables. In the time step in which the remote code is spawned the reception block sends the remote code the names of the output variables demanded by the user. At any time step, the reception block gets from the remote code the values of those variables.

Since the remote code is still running even though the execution of the emission block has been terminated for the current time step (or iteration), parallelisation of problems can be easily achieved with the use of both modules. As it has already been explained in section III.2, in many cases the problem to be solved has intrinsic parallel characteristics, in the sense of information flow, due to *physical parallelism*. For instance, a thermal-hydraulic circuit consisting of a number of loops and a central vessel where the flows of all the loops are mixed, may be simulated with a single thermal-hydraulic code. The input may be split into the parallel components of the circuit, i.e. the vessel plus the loops. If the loops are identical they may even share the same input file. The vessel will act as the physical synchronisation barrier. Figure 4.2 shows how the parallelisation of this system would be performed with BABIECA.

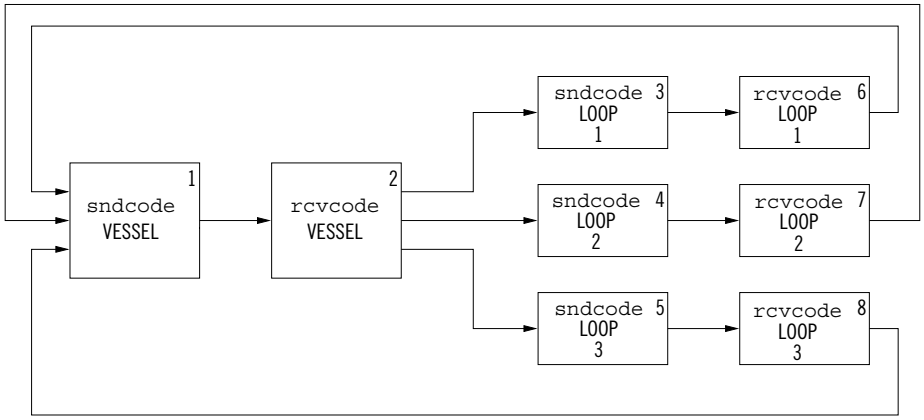

Figure 4.2. Parallelisation of a hydraulic circuit

The 1st block to be called is that which spawns the vessel simulation. The outputs which act as input signals for the loops are received by the 2nd block. The 3rd block is called and the process simulating the 1st loop is spawned. The 4th and 5th blocks are also called, spawning respectively the 2nd and 3rd loops. At this point the three loops are running simultaneously in parallel, saving a considerable amount of time if each process runs on a different processor or in a different machine. Blocks 6, 7 and 8 are then called to receive the output data from the respective processes once they have finished their computations. Feedback points could have been defined between blocks 6, 7, 8 and block 1, if iterative coupling were desired. If the code linkage were implemented in only one BABIECA module for both sending and receiving data, the processes could not run simultaneously and hence true parallelisation of the problem would not be feasible.

The boundary conditions linkage proposed in this paper may be done with any other modular simulation language, for instance, SIMULINK and MATRIX$_X$. The source code of the two modules would be almost the same in any case. However, the coupling of codes via initial conditions cannot be done with these codes, since they are not capable of stopping the execution of a given block. Additionally, this provides a method for overriding the problems that arise when the physical models approach the end of their applicability range. This is often the case when very different situations arise in the course of the simulation, and very different models are needed. One solution is to integrate all models needed in a single code, starting them at its due time; however, this makes coding cumbersome and results in large codes. In BABIECA, the model approaching the limit would deactivate itself and a new code would be triggered with the appropriate model and initial conditions.

IV.4. Standard specifications relating the codes to be coupled

Appendix C shows a documented template of a C-like program fulfilling the proposed standard. The template does not develop the tasks relating the input file reading, solution advancement and

output management. We emphasize that these code dependent tasks do not disturb the code linkage, which is just what makes standardization possible. If a program developer desires to follow this standard, he just needs to add those tasks to the template. The resulting code will be ready to be coupled to the BABIECA program (or a similar one) without additional effort.

The template intends to be self-explanatory. Nevertheless, the main items will be explained here.

After reading the input file, which is a code dependent task, the code tries to enroll into PVM and to get the task identification number of the parent process. If this identification number is zero, it means that the code has been run stand-alone. The PVM functions used to couple the code with BABIECA are enclosed into conditional statements that depend on whether the code has a been spawned by a parent process or not. In the latter case the conditional statements are skipped, and the code behaves as usually.

On the other hand, if the code has been spawned by a parent process, it is assumed that it will be coupled with BABIECA. The code receives from BABIECA, if it is so demanded, the names of the initial conditions that will be overwritten. The names of the boundary conditions will be received as well. The code must obtain then pointers to the memory allocations of the variables represented by those names. The program developer must report in the program documentation the nomenclature used to refer to those variables. The values of the variables are received and put into the proper allocations, as well as the initial time, overwriting the values obtained from the input file. With these new boundary and initial conditions the code computes the extended vector and the outputs in the initial time. The code receives then from the BABIECA module rcvcode the names of the output variables that will be sent back. The memory allocation of those variables is searched, yielding proper pointers. The data stored in those memory allocations is sent to BABIECA.

In successive solution advancements, the code receives from BABIECA the current time step, which will overwrite the time step chosen in principle by the code. A flag indicating whether the advancement of the solution is the first attempt to solve the current time step, or on the other hand it is a new attempt to solve the same time step (i.e. an iteration), is also received. In the former case, the extended state vector is that obtained in the previous successful time step advancement. In the latter case, the extended state vector is the same as in the previous iteration. This feature is crucial if iterative coupling between codes is desired. The new values of the boundary conditions are received as described before, shifting the old values. The solution is advanced and the outputs are sent to the driver program. The results of the remote code are written in the corresponding file only if the iteration flag indicates that the previous attempt was successful. In this case the outputs from the previous time step are written. The outputs can not be written just after computation of the solution, since the remote code has no means to know if the overall advancement will be successful. Only the driver code is able to decide this. Of course, the emission and reception tasks are perfectly synchronized with their counterparts in the `sndcode` and `rcvcode` modules.

IV.5. Linkage of the RELAP5 code

The RELAP5 code does not fulfil, of course, the standard proposed in this report. However, proper inclusion of PVM message passing routines into the original source code can make RELAP5 behave as if the standard were fulfilled. Searching of the places where the PVM functions must be allocated has been a cumbersome task that would had been avoided with standardization. This section gives an overview on the job undertaken to couple RELAP5 to the BABIECA driver.

The standard states that the code must received the names of the boundary and initial conditions after reading the input file. This is accomplished in RELAP5 by inserting the following code at the end of subroutine `inputd`:

```
c     relap5 gets the tid of the parent process through PVM.
      call pvmfparent(partid)
c     If pvmd is not running, the parent tid is set equal
c     to "PvmNoParent".
      if(partid .lt. 0) then
        partid = PvmNoParent
      endif
c     If RELAP5 has been spawned by other process it will be
c     coupled to other codes via PVM. RELAP5 gets the names
c     of the variables that will act as boundary and initial
c     conditions.
      if (partid .ne. PvmNoParent) then
        call pvmnames
      endif
```

The routine `pvmnames` receives the names of the boundary conditions. Coupling through initial conditions has not been implemented yet. By the moment, only boundary conditions specified through time dependent volumes or junctions can be modified by BABIECA. The name of each boundary condition comprises two fields separated by a '-'. The first field refers to a physical variable that can be specified through a time dependent volume or junction, i,e. `p, uf, ug, voidg, boron, tsatt, quale, sattemp, velfj, velgj, mflowfj and mflowgj`. Coupling through other boundary conditions (general tables) has not been implemented yet. The second field specifies the three digit number of the component that holds the boundary conditions to be modified by BABIECA.

Just after calling the RELAP5 subroutine `trnset`, which makes RELAP5 be ready to run the transient, the new subroutine `bndset` is called. This subroutine takes the previous variable names and component numbers and gets the position of the corresponding tables in array `fa`. `bndset` checks that each name of the physical variables is one of the variables

specified in cards *ccc*0200, through digit t in word 1 for a time dependent volume, and through word 1 for a time dependent junction.

Subroutine bndset calls in turn subroutine rcvbound, which receives the values of the boundary conditions used in initiallization phase and puts them into the memory areas obtained by bndset with the help of the auxiliary routine settdcmp. Moreover, rcvbound receives the flags that indicate if the attempted advancement is a new time step or a new iteration of the same time step and the time increment.

Prior to the code lines

```
      dthy = dtmax(i)
      if (timehy + 1.1d0*dthy .ge. tspend(i)) then
         dthy = tspend(i) - timehy
      endif
      dtht = dthy
      dt = dthy
      if (chngno(15)) dtxmdt = dthy
```

in subroutine dtstep, which is responsible of advancing the solution, the following lines have been added:

```
      if(partid.ne.PvmNoParent) then
         call pvmlink
c     if the parent process is going to finish, the variable tspend
c     is modified to force termination of RELAP5 by end of time
c     step cards. RELAP5 leaves PVM and the control is transferred
c     to label 1125, where RELAP5 checks if the transient is going
c     to finish.
         dtmax(i) = timestep
         if(iterflag.eq.3) then
            tspend(filndx(2)) = timehy
            call pvmfexit(info)
            goto 1125
         endif
      endif
```

As usually, the piece of code is only executed if the RELAP5 process has a parent. Routine pvmlink receives from module rcvcode the names of the output variables that will be sent to BABIECA, if this is the first time pvmlink is called. These names also obey the scheme *variable*

name - component number. The component number has the well known structure *cccvv*`0000` for hydrodynamic components and *cccg*0*nn* for the heat structures. In this last case, the previous field may be appended with digits `00` or `01` to obtain the values of variables relating to the radial nodalization. The memory positions of the variables in array `fa` are obtained, and the required outputs sent to the parent process. At the end of routine `pvmlink`, the routine `rcvbound` is called again, in order to receive the time step interval and the boundary conditions for the following time advancement. Just after, the RELAP5 requested time step `dtmax(i)` is overwritten with the time step `timestep` received from the parent process. If the parent process demands termination of the simulation, the simulation time `timespend` is set to the variable `timehy` to force termination of the RELAP5 process by the time step cards.

V. Application example

V. Application example

V.1. Problem setup

This chapter describes an example used to demonstrate the capability of RELAP5 to be coupled to other codes via BABIECA. To be precise, the example connects four copies of the RELAP5 code among them, solving a consistent simulation problem. The example also illustrates, by the way, how the proposed connection methodology may be used to parallelize simulation problems. However, it must be remarked that this is a code connection example rather than a parallelization one. The performance of the parallelised problem may be even worse than that of the original problem. Particularly, the parallelised problem is stabilized at the expense of decreasing the time step.

The example consists of a three loop hydraulic circuit. The flows from the loops are collected in a central pipe. The flow in the system is maintained by three pumps, each one located in a loop. The system has been simulated with RELAP5/Mod3.2. The nodalization is shown in figure 5.1, where only the central pipe and one of the loops have been represented. The central pipe and the loops are attached through cross-flow junctions, indicated with a "x" in the figure. A heat structure with constant temperature as a boundary condition has been added to the central pipe, to remove the heat supplied by the pumps.

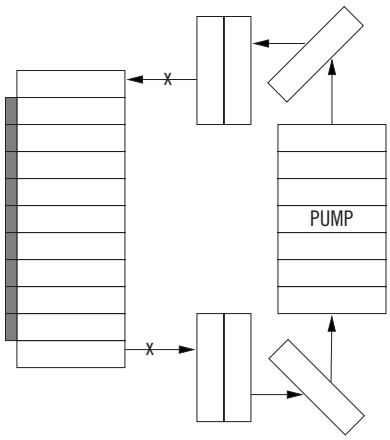

Figure 5.1. Nodalization scheme of the three loop circuit

The following transient has been simulated: after 50 seconds at steady conditions, the pump in loop 1 is triped yielding a coastdown. After 250 seconds of transient a second coastdown takes place as a result of the trip of the pump in loop 2.

In order to check the capability of RELAP5 to be linked to other codes through the BABIECA program, the original input file has been split into four files, each one containing the

central pipe and the loops respectively. In each file, the rest of the circuit is substituted by proper boundary conditions, input through time dependent volumes and junctions.

Two different splitting schemes will be checked. In the first one, depicted in figure 5.2, the central pipe and the loops are dettached through the cross-flow junction. In the second scheme, shown in figure 5.3, the RELAP5 file that contains the central pipe also includes the first node of each loop. As a consequence, the loops are dettached through single junctions.

The four RELAP5 input files will perform the overall simulation of the circuit by linking the corresponding computer processes through the BABIECA program. Appendix D shows the BABIECA input file necessary to link the RELAP5 processes. BABIECA transfers the pressure in the first node of the central pipe/loop to the time dependent volume at the outlet of each loop/central pipe. The internal

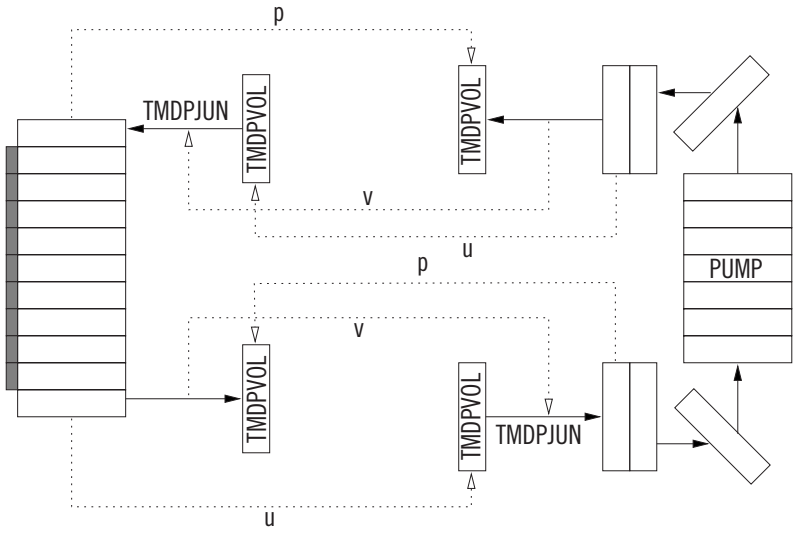

Figure 5.2. Parallelization through cross-flow junctions

energy of the last node of the central pipe/loop is transferred to the time dependent volume at the inlet of each loop/central pipe. The outlet velocity in the central pipe/loop is sent through BABIECA to the time dependent junction at the inlet of each loop/central pipe. This information exchange is represented with dashed lines in figures 5.2 and 5.3. The exercise will demonstrate how RELAP5 can be connected to other codes (in this example they are the RELAP5 code itself, but it does not make any difference) with the scheme proposed in this report. Moreover, it illustrates an example of problem level parallelization like that in figure 4.2, since the processes that simulate the loops will run concurrently.

For each splitting scheme two cases will be run. In the first one the time dependent volumes are very short and with a very large cross-sectional area, as recomended in the RELAP5 guidelines. In the second case, the time dependent volumes have the same dimensions as the control volumes they substitute in the counterpart files.

V.2. Results

Figures 5.4 to 5.7 show the velocity in the main pipe and loops 1, 2 and 3. The symbol • represents the RELAP5 original case. The parallelization denoted as 1, and represented by the symbol ♣, obeys the scheme in figure 5.2, with short and broad time dependent volumes. Parallelization 2, represented by the symbol ◊, obeys the scheme in figure 5.2, with time dependent volumes equal to those they substitute. Parallelization 3, represented by the symbol ♡, obeys the scheme in figure 5.3, with short and broad time dependent volumes. Finally, parallelization 4, represented by ♠, obeys the scheme in figure 5.3, with volumes equal to those they substitute.

It is easy to realize from the trend of the results that parallelization scheme 4 yields the closest results to those obtained in the base case. Parallelization 1 gives raise to particularly poor

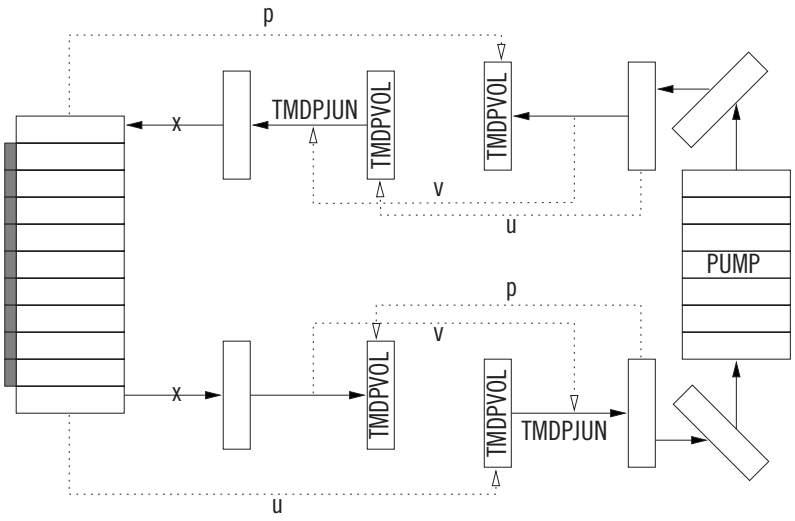

Figure 5.3. Parallelization through single junctions

results. As a conclusion, splitting through cross-flow junctions is not recomended at all. Usage of time dependent volumes with the same dimensions as those they substitute is also recomended. Figures 5.5 and 5.6 also demonstrate that the RELAP5 flow reversal capability is not affected by the linkage and parallelization scheme proposed here.

V.3. Run statistics

The CPU time spent to run the base case has been 4968 seconds. The corresponding time for parallelization 4 has been 4571. Thus, no significative execution acceleration has been attained. Since the coupling between the four hydraulic components is non-iterative, the BABIECA input file can be modified to force execution of all the emission blocks prior to the reception blocks. In this last case, all the four RELAP5 processes are executed simultaneously. The CPU time is then 2859 seconds.

These results can be explained as follows: since the original circuit has been split into four input files, each one with a similar computational load, an acceleration with a factor of 4 is expected. However, the execution of RELAP5 takes at least a certain time, independently from the number of control volumes. This makes that the execution time of each partial input file be one half of the full one, instead of one fourth. Moreover, the loops are executed always after the central pipe. Hence, two serial processes, the central pipe and the loops, are being executed. As a result, the execution time of the parallelized version is comparable to that of the RELAP5 base case. An acceleration with an approximate factor of 2 is attained in the case where the central pipes and the loops are executed concurrently.

The machine used for this work is a CONVEX SPP-2000, with 16 HP-Ultrix processors. The operating system is HP-UX, release B.10.01, version U. The PVM version is 3.3.11. The RELAP5 version is 3.2.

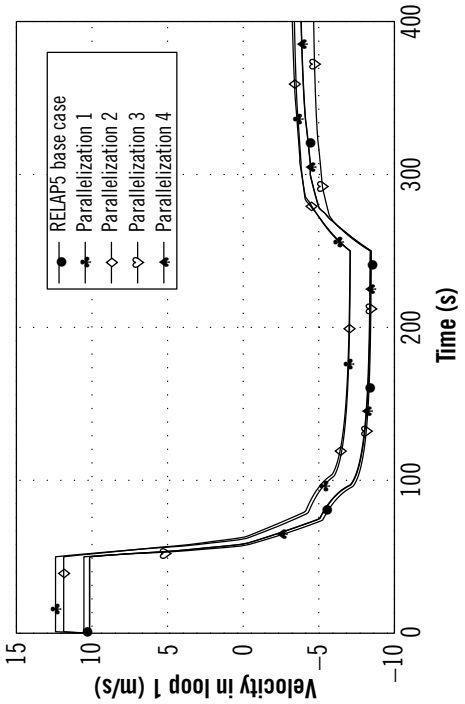

Figure 5.5. Velocity in loop 1

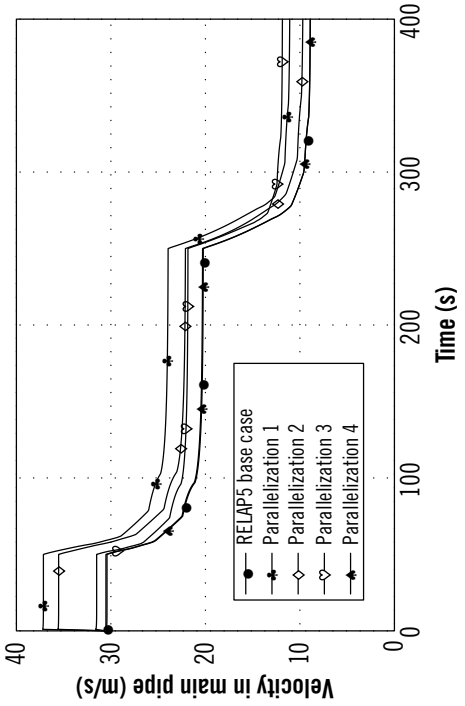

Figure 5.4. Velocity in main pipe

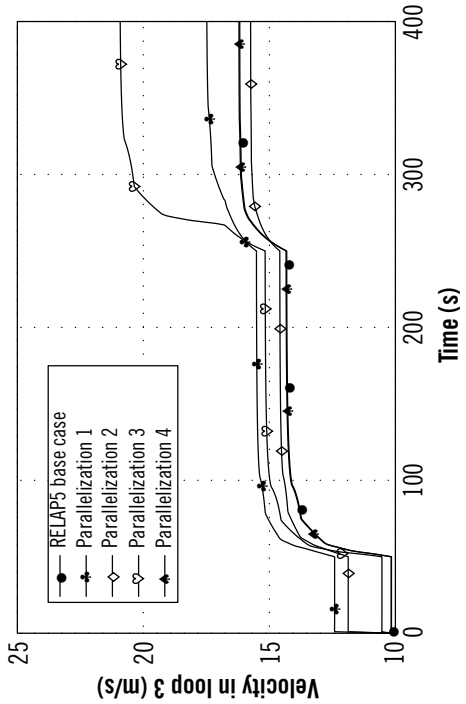

Figure 5.7. Velocity in loop 3

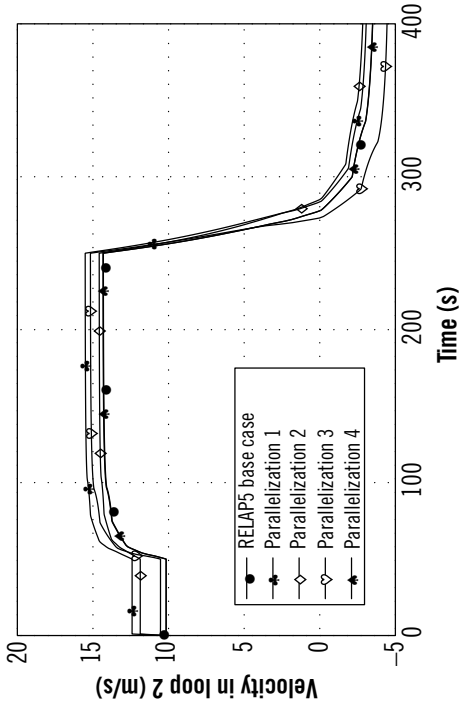

Figure 5.6. Velocity in loop 2

VI. Conclusions

VI. Conclusions

Writing of scientific computer codes for simulation of transients, based on the concept of time step, can be standardised for ease of coupling with other codes (which also fulfill the standard). The coupling is achieved by means of a general purpose simulation program such as BABIECA, which manages the time step and transfers the relevant information among the different codes, according to a given topology defined in the BABIECA input file. The information transfer between the codes is performed by means of the de facto standard for distributed computing, which is currently PVM. Migration to other message passing routines can be easily done, if the standard for message passing changes in the future. The same executable file of each code can be used to be coupled to BABIECA or for stand-alone running, without using any special option in the input file. The scheme proposed here leads to open simulation models, in which new codes can be added at any time to enhance the system capabilities and some codes already linked may not be used in some cases.

The scheme can be also used to attain problem level, user driven parallelisation of problems which show a sort of *physical parallelism.*

A single application example has been used to demonstrate how teh RELAP5 code can be coupled to other codes and how RELAP5 problems with hydraulic loops can be parallelized.

Bibliography

Bibliography

[1] A. Baratta. Interface requirements for coupling a containment code to a reactor system thermal hydraulic code. In *OECD/CSNI Workshop on Transient Thermal-Hydraulic and Neutronic Requirements. Annapolis.* November 1996.

[2] A. L. Beguelin et al. *PVM–3.3: Parallel Virtual Machine System.* Univ. of Tennessee-Knoxville; Oak Ridge National Laboratory; Amory University, Atlanta, Georgia, 1992.

[3] A. Bengaouer and G. Geffraye. Three dimensional kinetics coupling to thermal hydraulics. In *OECD/CSNI Seminar on 'Best Estimate Methods in Thermal-Hydraulics Safety Analysis', Ankara.* June 1998.

[4] B. Brun. Current implementation and future plans on new code architecture, programming language and user interface. In *OECD/CSNI Workshop on Transient Thermal-Hydraulic and Neutronic Requirements. Annapolis.* November 1996.

[5] F. Camous, F. Jacq, P. Chatelard, and J. Flores. Interface requirementsto couple thermal hydraulics codes to severe accident codes: ICARE/CATHARE. In *OECD/CSNI Workshop on Transient Thermal-Hydraulic and Neutronic Requirements. Annapolis.* November 1996.

[6] J. F. Cremer, R. S. Palmer, and R. E. Zippel. Creating scientific software. *Transactions of the Society for Computer Simulation,* 14(1):37–39, 1997.

[7] X. Delhaye, C. M., C. Schneidesch, P. Damas, and L. Vanhoenacker. Coupling of the neutronics code PANTHER with RELAP5. The BELGATOM approach to simulate the core response of non-symmetric reactivity transients. In *Spring 98 CAMP meeting. Ankara*. 1998.

[8] T. Downar, B. Doug, V. Mousseau, D. Ebert, and J. M. Kelly. RELAP5/PARCS. Generalized thermal-hydraulics/neutronic interface. In *Spring 98 CAMP meeting. Ankara*. 1998.

[9] G. E. Fagg, J. J. Dongarra, and A. Geist. Heterogeneous MPI application interoperation and process management under PVMPI. In *Euro PVM-MPI conference, Cracow, Poland.* November 1997.
`http://www.netlib.org/utk/papers/pvmmpi97.ps.`

[10] Fauske & Associates Inc. *MAAP 3.0B, User's Manual,* 1995.

[11] T. Freeman and C. Phillips. *Parallel Numerical Algorithms.* International Series in Computer Science. Prentice Hall, 1992.

[12] A. Geist et al. *PVM, Parallel Virtual Machine. A User's Guide and Tutorial for Networked Parallel Computing.* The MIT press; Scientific and Engineering Computation Series, 1994.
`ftp://netlib2.cs.utk.edu/pvm3/book/pvm-book.ps.`

[13] A. Geist et al. *PVM3 Users Guide and Reference Manual.* Oak Ridge National Laboratory, September 1994.
`ftp://netlib2.cs.utk.edu/pvm3/ug.ps.`

[14] G. Geist, J. Kohl, and P. Papadopoulos. PVM and MPI: a comparison of features. *Calculateurs Parallels,* 8(2), May 1996.
`http://www.epm.ornl.gov/pvm/PVMvsMPI.ps.`

[15] B. Gropp, R. Lusk, and A. Skjellum. *Using MPI.* MIT press, 1994.

[16] V. Hernández, A. Vidal, I. Blanquer, J. Román, F. S., G. Verdú, J. L. Muñoz Cobo, A. Escrivá, X. Sancho, J. Serra, and A. Gómez. Reducción del tiempo de respuesta del código de análisis de transitorios TRACBF1 mediante computación de altas prestaciones. In *Proc. of the XXII meeting of the Spanish Nuclear Society, Santander.* October 1996.

[17] J. M. Izquierdo et al. Tizona: A computer code with an advanced two phase thermal hydraulic package. *In preparation,* 1998.

[18] J. M. Izquierdo and M. Sánchez Perea. CAMPEADOR: An implicit simulation language for continuous sytems combined with discrete events able to apply protection theory. In *1994 European Simulation Multiconference (ESM'94). Barcelona, Spain.* June 1994.

[19] A. Jones et al. Experience from the development of the severe accident code system ESTER. In *Proc of the ANS Winter Meeting, Washington.* November 1994.

[20] Kyrki-Rajamäki. The need of coupled 3D neutronics in DBA and BDBA analyses using conservative or best-estimate approach. In *OECD/CSNI Seminar on 'Best Estimate Methods in Thermal-Hydraulics Safety Analysis'. Ankara.* June 1998.

[21] S. Langenbuch, H. Austregesilo, P. Fomitchenko, U. Rohde, and K. Velkov. Interface requirements to couple thermal-hydraulic codes to 3D neutronic codes. In *OECD/CSNI Workshop on Transient Thermal-Hydraulic and Neutronic Requirements. Annapolis*. November 1996.

[22] L. M. Liebrock. Parallelisation and automatic data distribution for nuclear reactor simulations. In *Proc of the OECD/CSNI Workshop on transient thermal-hydraulic and neutronic code requirements; Annapolis.* 5–8 November 1996.

[23] F. Merino, C. Ahnert, and J. M. Aragonés. Development and validation of the 3-D PWR core dynamics SIMTRAN code. In *Joint Intl Conf on math methods and supercomputing in nuclear applications, Karlsruhe.* April 1993.

[24] R. L. Moore, C. S. Miller, and C. D. Fletcher. Linking external models with RELAP5 using PVM. Technical report, Idaho National Engineering Laboratory.

[25] Ohio Supercomputer Center. The Ohio State University. *MPI Primer / Developing with LAM,* November 1996.
`http://www.itl.nist.gov/div895/sasg/LAM/lam61.doc.ps.`

[26] R. Page and J. Jones. Development of an integrated thermal-hydraulics capability incorporating RELAP5 and PANTHER neutronics code. In *OECD/CSNI Workshop on Transient Thermal-Hydraulic and Neutronic Requirements. Annapolis.* November 1996.

[27] C. Queral and M. A. García Zamorano. Final report on the specific agreement CSN-UPM for the development of accident management guide evaluation tools. Technical report, Departamento de Sistemas Energéticos, UPM, December 1997.

[28] T. Rothe. Linking external models to RELAP5: A new era of RELAP5 applications. In *Proc. of the Intl. Conf. on New trends in Nuclear System Thermohydraulics, Pisa.* 1994.

[29] Z. Stosic. Enhancing the scope of applications of standard thermal hydraulic codes by linking with others. In *Proc of the ASME International Conference on Nuclear Engineering, New Orleans,* volume 1, part. B. 1996.

[30] The RELAP5 Code Development Team. RELAP5/MOD3 code manual, volume I: Code structure, system models and solution methods. Technical report NUREG/CR-5535, INEL-95/0174, Idaho National Engineering Laboratory, June 1995.

[31] K. Trambauer. Interface requirements to couple thermal-hydraulic codes to severe accident codes: ATHLET-CD. In *OECD/CSNI Workshop on Transient Thermal-Hydraulic and Neutronic Requirements. Annapolis.* November 1996.

Appendices

Appendix A. File `sncode`

Appendix B. File `rvcode`

Appendix C. Remote code standard specifications

Appendix D. BABIECA input file

Appendix A. File `sndcode`

A.1. Description of the module

This module belongs to BABIECA, and is the responsible for sending its inputs to the remote code. Initial and/or boundary conditions can be sent depending on the input file spceifications. The file complies with the mandatory structure for a BABIECA module, namely

```
1⟨ Include files 0 ⟩
    ⟨ Functions prototypes 89 ⟩
    BAB_MODULE(sndcode_)
    {
        ⟨ Module variables 2 ⟩;
        switch (control [0]) {
        case BabRead:
            {
                ⟨ Input file reading 8 ⟩;
                break;
            }
        case BabCalc:
            case BabSSCalc:
            case BabFbck:
            case BabSsFbck:
            {
                ⟨ Calculation mode 5 ⟩;
                break;
            }
        case BabSave:
            {
                ⟨ Send save message to remote 15 ⟩;
                break;
            }
        case BabRestore:
            {
                ⟨ Send restore message to remote 16 ⟩;
                break;
            }
        case BabLastp:
            {
                ⟨ Termination mode 13 ⟩;
```

```
            break;
        }
    default: ;
    }
    return errcode;
}
⟨ Auxiliary functions 87 ⟩:
```

¶ Variable *errcode* will carry the error code to be returned to the calling routine. A zero value indicates no error, other values are set according to the include `baberr.h`. *ninputs* and *n* are dummy index variables for later use in **for** constructs.

2⟨ Module variables 2 ⟩ ≡

```
int errcode;
int ninputs, n;
```

See also chunks 7, 22, 36, 37, 39, 46, 52, 67 and 77.

This code is used in chunk 1.

A.1.1. Code sections

A.1.1.1. Calculation mode

¶ In the calculation mode, the first input to the block is checked. If it is equal to one, the block will be active during the current time step. The remote code is started if this is the first time this occurs. Then, the code is executed *nsteps* times, and the appropriate values are stored for later retrieval. If the code has been executed, the driver is informed (by means of *control* [2]) of the change in the execution mode, so that it can skip the appropriate blocks. A copmplementary behaviour is done if the code is not executed. Later, the ouput of the block is set.

5⟨ Calculation mode 5 ⟩ ≡

```
if (1.0 ≡ ent [0]) {    /* Active during this time step */
    ⟨ Load variables for this time step 6 ⟩;
    if (0 ≡ rcspawn) {
            /* If this is the first time the remote code has been active. */
        ⟨ Start the remote code 34 ⟩;
    }
    ⟨ Execute the remote code nsteps times 45 ⟩;
    vinvar [3] = (double) 1.0;    /* This is prevexec */
    control [2] = (int) vinfij [0];
        /* The code requests a change in the executing mode */
    }
else {    /* Not active in this time step */
```

```
        vinvar [3] = (double) 0.0;    /* prevexec */
    }
    sal [0] = (double) rcodetid;
    errcode = BabWAtt;    /* No error occured */
```

This code is used in chunk 1.

¶ *Sndcode_* retrieves stored variables:

1. Executable variables
2. Boundary conditions variables
3. INitial conditions variables
4. The tid of the remote code is stored in *rcodetid.*
5. The value of the execution flag in the previous time step is stored in *prevexec*.
6. The flag that indicates if the remote code has already been spawned is stored in *rcspawn*.

```
6⟨ Load variables for this time step 6 ⟩ ≡
    ⟨ Load executable variables 21 ⟩;
    ⟨ Load boundary conditions variables 65 ⟩;
    ⟨ Load initial conditions variables 50 ⟩;
    rcodetid = (int) vinvar [0];
    if (control [0] > 0) {
        vinvar [1] = vinvar [3];
    }
    prevexec = (int) vinvar [1];
    rcspawn = (int) vinvar [2];
```

See also chunk 44.

This code is cited in chunk 10.

This code is used in chunk 5.

¶ The following vaiables need to be declared.

```
7⟨ Module variables 2 ⟩ +≡
    int prevexec = 0, rcspawn = 0;
    int rcodetid;
```

A.1.1.2. Reading of the input file

```
8⟨ Input file reading 8 ⟩ ≡
    {
```

```
    ⟨Variables declarations for the input file reading section 9 ⟩;
    nument = control [3];
    ⟨ Read execmode 31 ⟩;
    ⟨ Read the host name 25 ⟩;
    ⟨ Read the executable name and arguments 26 ⟩;
    ⟨ Read nulltrsteps 32 ⟩;
    ⟨ Read boundary conditions 69 ⟩;
    ⟨ Read initial conditions 54 ⟩;
    ⟨ Save values for the driver 10 ⟩;
}
```

This code is used in chunk 1.

¶ Some variables need not be shared by the whole of *Sndcode,* because they are only used within the reading mode. These are declared here as they appear.

```
9⟨ Variables declarations for the input file reading section 9 ⟩ ≡
    int nument;
```

See also chunks 29 and 33.

This code is used in chunk 8.

¶ The driver structure needs several parameters to be stored:

- *vinfij* values that will be restored in ⟨ Load variables for this time step 6 ⟩;
- The *control* array

*Sndcode_* also performs some basic error checking.

```
10⟨ Save values for the driver 10 ⟩ ≡
    ⟨ Save executable information 20 ⟩;
    ⟨ Save boundary conditions information 64 ⟩;
    ⟨ Save initial conditions information 49 ⟩;
    ⟨ Check the number of inputs 11 ⟩;
    ⟨ Fill the control array 12 ⟩;
```

This code is used in chunk 8.

¶ The number of inputs must be equal to the number of boundary conditions plus the number of initial conditions plus the activation signal.

```
11⟨ Check the number of inputs 11 ⟩ ≡
    if (nument ≠ (nbound + ninit + 1)) {
        errcode = −24;
```

```
    return (errcode );
}
```

This code is used in chunk 10.

¶ *Sndcode_* updates "control" array:
- Return code, *errcode,* is set to 0, i.e., function has been executed correctly.
- Number of outputs, i.e., *control* [3], is set to 1.
- Size of *vinfij* array is saved in *control* [4], i.e, 15.
- Size of *vinvar* array is saved in *control* [5], i.e, 4.
- Size of *stat* array, i.e., 0, is saved in *control* [6]

```
12⟨ Fill the control array 12 ⟩ ≡
    errcode = 0;
    control [3] = 1;
    control [4] = 15;
    control [5] = 4;
    control [6] = 0;
```

This code is used in chunk 10.

A.1.1.3. Termination mode

The remote code is informed that it must exit.

```
13⟨Termination mode 13 ⟩ ≡
    rcodetid = (int) vinvar [0];
    rcspawn = (int) vinvar [2];
    if (rcspawn ≡ 1) {    /* The code was once started */
        initflag = 0;
        prevexec = 0;    /* They will not be used in the remote code */
        ⟨ Obtain a buffer 79 ⟩;
        ⟨ Pack control [0] 80 ⟩;
        ⟨ Pack initflag 84 ⟩;
        ⟨ Pack prevexec 83 ⟩;
        info = pvm_send (rcodetid, TSTEPVAR);
        if (info < 0) return (babpvmerror_(info, "pvm_send"));
        pvm_free (bufid );
    }
```

This code is used in chunk 1.

A.1.1.4. Save and restart

15¶ ⟨ Send save message to remote 15 ⟩ ≡ /∗ Not yet done ∗/

This code is used in chunk 1.

16¶ ⟨ Send restore message to remote 16 ⟩ ≡ /∗ Not yet done ∗/

This code is used in chunk 1.

A.1.2. Particular tasks

A.1.2.1. Driving the remote code

A.1.2.1.1. Save/restore variables

¶ *Sndcode_* saves

- The execution mode in *vinfij* [0].
- The host name in *vinfij* [1].
- The pointer to the executable name in *vinfij* [2].
- The pointer to the remote code arguments in *vinfij* [3].
- The number of time steps of the null transient to be run in steady-state mode in *vinfij* [12].

20⟨ Save executable information 20 ⟩ ≡

```
vinfij [0] = (double) execmode;
vinfij [1] = (double) ((int) hostname);
vinfij [2] = (double) ((int) execname);
vinfij [3] = (double) ((int) argspt );
vinfij [12] = (double) nulltrsteps;
```

This code is used in chunk 10.

¶ The name of the remote code stored in *execname* is retrieved.

21⟨ Load executable variables 21 ⟩ ≡

```
execname = (char ∗) ((int) vinfij [2]);
```

This code is used in chunk 6.

¶ *execname* is a string,

22⟨ Module variables 2 ⟩ +≡

```
char ∗execname;
```

¶ *Sndcode_* reads the hostname, the executable name, the command arguments, the pointer to the names of the boundary conditions and the pointer to the names of initial conditions.

23⟨ Get the parameters needed for starting the remote code 23 ⟩ ≡

```
hostname = (char *) ((int) (vinfij [1]));
argspt = (char **) ((int) vinfij [3]);
⟨ Get the boundary conditions names 66 ⟩;
⟨ Get the initial conditions names 51 ⟩;
```

This code is used in chunk 34.

A.1.2.1.2. Read host, name and arguments of the remote code

¶ *Sndcode_* calls nextlin to read the string describing the host where the remote code is installed. Memory is allocated and the name is read. String *msg1* is written if an end-of-file is detected, and *msg6* if the memory allocation fails.

25⟨ Read the host name 25 ⟩ ≡

```
nextlin_ (linea, 80, msg1, 80);
linecount = 1;
saltesp_ (linea, &linecount, 80);
⟨ Obtain the pointers to the first and last non-blank characters delimiting the string 86 ⟩;
hostname = (char *) malloc((strlen (wordpt ) + 1) * sizeof (char));
if (Λ ≡ hostname) {
    errcode = −24;
    strcpy (msg6, "Fail␣to␣allocate␣memory.");
    errmsg_ (&errcode, msg6, msg7, 80, 80);
    return (errcode);
}
strcpy (hostname, wordpt);
```

This code is used in chunk 8.

¶ The executable name and its arguments must also be read from the input file.

26⟨ Read the executable name and arguments 26 ⟩ ≡

```
nextlin_ (linea, 80, msg1, 80);
linecount = 1;
saltesp_ (linea,&linecount, 80);
⟨ Obtain the pointers to the first and last non-blank characters delimiting the string 86 ⟩;
execname = (char *) malloc((strlen (wordpt) + 1) * sizeof (char));
if (Λ ≡ execname) {
```

```
        errcode = −24;
        strcpy (msg6, "Fail␣to␣allocate␣memory.");
        errmsg_ (&errcode, msg6, msg7, 80, 80);
        return (errcode);
    }
    strcpy (execname, wordpt);
    linecount += 2;
        /* one to convert FORTRAN to C indices and another to skip the null character that ends
        the previous word in string linea. */
    info = linecount;
    saltesp_ (linea,&linecount, 80);
    argspt = (char **) malloc(sizeof (char *));
    for (nargs = 0; linecount > 0; nargs ++) {
        ⟨ Realloc argspt 28 ⟩;
        ⟨ Read an argument to the remote code 27 ⟩;
    }
    ⟨ Realloc argspt 28 ⟩;
    argspt [nargs] = Λ;     /* Standard to end the array argspt */
```

This code is used in chunk 8.

¶ Only ine argument at a time.

27⟨ Read an argument to the remote code 27 ⟩ ≡

```
    ⟨ Obtain the pointers to the first and last non-blank characters delimiting the string 86 ⟩;
    argspt [nargs] = (char *) malloc((strlen (wordpt) + 1) * (sizeof (char)));
    if (Λ ≡ argspt [nargs]) {
        errcode = −24;
        strcpy (msg6, "Fail␣to␣allocate␣memory.");
        errmsg_ (&errcode, msg6, msg7, 80, 80);
        return (errcode);
    }
    strcpy (argspt [nargs], wordpt);
    linecount += 2;
        /* one to convert C to FORTRAN indices and another to skip the null character that ends
        the previous word in string linea. */
    info = linecount;
    saltesp_ (linea,&linecount, 80);
```

This code is used in chunk 26.

¶ Variable *argspt* ia a list of strings carying the arguments. It has to be enlarged each time a new argument apperas, and once more for the last Λ pointer.

28⟨ Realloc *argspt* 28 ⟩ ≡

```
argspt = (char **) realloc(argspt, (nargs + 1) * (sizeof (char *)));
if (Λ ≡ argspt) {
    errcode = −24;
    strcpy (msg6, "Fail␣to␣allocate␣memory.");
    errmsg_(&errcode, msg6, msg7, 80, 80);
    return (errcode);
}
```

This code is used in chunk 26.

¶ We need to declare

- *linecount* is an integer to count the position read within a line
- *wordpt* is the actual pointer to the string **line**
- **line** is the line read by *nextlin*
- *msgx* are error messages

29⟨ Variables declarations for the input file reading section 9 ⟩ +≡

```
int linecount;
char *wordpt, linea[80];
char msg1 [80] = "SNDCODE.␣ERROR␣READING␣INPUT␣LINE.";
char msg6 [80];
char msg7 [80] = "␣";
```

A.1.2.1.3. Line by line integer parameters reading

¶ *Sndcode_* calls nextlin, and reads the value for *execmode.*

31⟨ Read *execmode* 31 ⟩ ≡

```
nextlin_ (linea, 80, msg1, 80);
linecount = 1;
execmode = leerint_ (linea,&linecount, 80);
```

This code is used in chunk 8.

¶ *Sndcode_* calls nextlin to read the number of time steps of the null transient that will be run in steady-state mode.

32⟨ Read *nulltrsteps* 32 ⟩ ≡

*nextlin_* (*linea,* 80*, msg1,* 80);
linecount = 1;
nulltrsteps = *leerint* (*linea,*&*linecount,* 80);

This code is used in chunk 8.

¶ The two variables read are **int**.
33⟨ Variables declarations for the input file reading section 9 ⟩ +≡
int *nulltrsteps, execmode;*

A.1.2.1.4. Procedure to start the remote code
34⟨ Start the remote code 34 ⟩ ≡
⟨ Enroll PVM 35 ⟩;
⟨ Get the parameters needed for starting the remote code 23 ⟩;
⟨ Connect to the specified host 38 ⟩;
⟨ Spawn the remote code 40 ⟩;
⟨ Send the data for the remote code 41 ⟩;

This code is used in chunk 5.

¶ *Sndcode_* enrolls in PVM.
35⟨ Enroll PVM 35 ⟩ ≡
mytid = *pvm_mytid* ( );
if (*mytid* < 0) **return** (*babpvmerror_*(*mytid,* "pvm_mytid"));

This code is used in chunk 34.

¶ *mytid* takes the value of the task identification assigned to this code by the pvm daemon.
36⟨ Module variables 2 ⟩ +≡
int *mytid;*

¶ The following are the name of the host where the code will run and the arguments array.
37⟨ Module variables 2 ⟩ +≡
char ∗*hostname,* ∗∗*argspt:*

¶ *Sndcode_* tries to connect to the host where the remote code will run. If we want to run the remote code in the current host the connection is not needed. In the latter case the hostname is changed from *thishost* to void string, which is the word understood by PVM.

38⟨ Connect to the specified host 38 ⟩ ≡

```
if (0 ≠ strcmp(hostname, "thishost")) {
    info = pvm_addhosts(&hostname, 1,&infos);
    if (info < 0) return (Babpvmerror_(info, "pvm_addhosts"));
}
else {
    free (hostname);
    *hostname = calloc(1, sizeof (char));
}
```

This code is used in chunk 34.

¶ For the first time, the variable *info* is used. It will be an **int** that takes the error codes returned by PVM routines. Likewise, *infos* is also a reproting variable. Below, *bufid* will also be used in this manner.

39⟨ Module variables 2 ⟩ +≡

```
int info, infos, bufid;
```

¶ *Sndcode_* spawns the remote code if this is the first time the block is active. The remote process identification number is saved in *vinvar* [0].

40⟨ Spawn the remote code 40 ⟩ ≡

```
pvm_spawn(execname, argspt, 1, hostname, 1,&rcodetid);
if (rcodetid < 0) return (babpvmerror_(rcodetid, "pvm_spawn"));
vinvar [0] = (double) rcodetid;
```

This code is used in chunk 34.

¶ *Sndcode_* obtains a buffer to pack the data to be transferred to the remote code, and packs

1. The boundary conditions.
2. The initial conditions.
3. The initial and target time.

Then, the packed data are sent, and the temporary memory used is freed. For later use, since the block has been activated once at least, the flag *rcspawn* is set to 1.

41⟨ Send the data for the remote code 41 ⟩ ≡

```
⟨ Obtain a buffer 79 ⟩;
⟨ Pack the boundary conditions names 73 ⟩;
⟨ Pack the intial conditions names 58 ⟩;
info = pvm_send (rcodetid, INBNDNAM);
```

```
if (info < 0) return (babpvmerror_(info, "pvm_send"));
⟨ Free used memory 42 ⟩;
vinvar [2] = (double) 1.0;    /* This is actually rcsspawn */
```

This code is used in chunk 34.

¶ The memory used by the host name, by the arguments, by the names of the initial and boundary conditions. Finally, the buffer provided by PVM is freed.

42⟨ Free used memory 42 ⟩ ≡

```
free (hostname);
for (nargs = 0; argspt [nargs] ≠ Λ; nargs ++) free (argspt [nargs]);
free (argspt [nargs]);
free (argspt);
for (ninputs = 0; ninputs < nbound; ninputs ++) {
    for (nvars = 0; nvars < boundnumber [ninputs]; nvars ++)
        free (boundpt [ninputs][nvars]);
    free (boundpt [ninputs]);
}
free (boundpt);
for (ninputs = 0; ninputs < ninit; ninputs ++) {
    for (nvars = 0; nvars < initnumber [ninputs]; nvars ++)
        free (initpt [ninputs][nvars]);
    free (initpt [ninputs]);
}
free (initpt);
pvm_freebuf (bufid);
```

This code is used in chunk 41.

A.1.2.1.5. Executing a transient in the remote code

¶ In case of steady-state calculation, BABIECA will demand the execution of a number of time steps read from the input file and saved in *vinfij* [12]. It is given the name *nsteps.*

44⟨ Load variables for this time step 6 ⟩ +≡

```
if (abs (control [0]) ≡ BabSSCalc) {
    nsteps = (int) vinfij [12];
}
else {
    nsteps = 1;
}
```

¶ The execution of the remote code is demanded *nsteps* times.

```
45⟨ Execute the remote code nsteps times 45 ⟩ ≡
    for (n = 0; n < nsteps; n++) {
    ⟨ Obtain a buffer 79 ⟩;
    ⟨ Pack control [0] 80 ⟩;
    ⟨Determine if initial conditions will be transferred and pack the flag 60 ⟩;
    ⟨ Pack prevexec 83 ⟩;
    if (initflag) {    /* If initial conditions given to the remote code */
        ⟨ Compute initial conditions and pack them 61 ⟩;
    }
        ⟨ Compute boundary conditions and pack them 75 ⟩;
        ⟨ Pack current time 82 ⟩;
        ⟨ Pack the flags for output reception 76 ⟩;
        info = pvm_send (rcodetid, TSTEPVAR);
        if (info < 0) return (babpvmerror_(info, "pvm_send"));
        pvm_freebuf (bufid);
    }
```

This code is used in chunk 5.

¶ Variables needed here.

```
46⟨ Module variables 2 ⟩ +≡
    int nsteps, initflag;
    int nargs, nvars;
```

A.1.2.2. Initial coditions management

A.1.2.2.1. Save/restore initial conditions information

¶ *Sndcode_* saves

- The pointer to the names of the input variables for initial conditions in *vinfij* [8].
- In *vinfij* [9] the address of the array initnumber.
- The number of input signals from which the initial conditions will be derived in *vinfij* [10].
- The total number of initial conditions that will be transferred to the remote code in *vinfij* [11].

Memory is reclaimed for the array initto and the pointer to the area requested saved in *vinfij* [14].

```
49⟨ Save initial conditions information 49 ⟩ ≡
    vinfij [8] = (double) ((int) initpt);
    vinfij [9] = (double) ((int) initnumber);
```

```
vinfij [10] = (double) ninit;
for (ninputs = 0, totinit = 0; ninputs < ninit; ninputs ++) {
    totinit += initnumber [ninputs];
}
vinfij [11] = (double) totinit;
initto = (double *) malloc(totbound * sizeof (double));
vinfij [14] = (double) ((int) initto);
```

This code is used in chunk 10.

¶ The variables previously saved are restored.

```
50⟨ Load initial conditions variables 50 ⟩ ≡
    initnumber = (int *) ((int) vinfij [9]);
    ninit = (int) vinfij [10];
    totinit = (int) vinfij [11];
    initto = (double *) ((int) vinfij [14]);
```

This code is used in chunk 6.

¶ *initpt* takes the pointer to the memory assigned to the names of the initial conditions.

```
51⟨ Get the initial conditions names 51 ⟩ ≡
    initpt = (char ***) ((int) vinfij [8]);
```

This code is used in chunk 23.

¶ Initial conditions variables needed

```
52⟨ Module variables 2 ⟩ +≡
    int *initnumber;
    int ninit, totinit;
    double *initto;
    char ***initpt;
```

A.1.2.2.2. Initial conditions reading

¶ *Sndcode_* calls nextlin to read the number of input signals that will be used to compute the remote code initial conditions. String *msg1* is written if an end-of-file is detected. The number of input signals is stored in *ninit*. Two arrays are needed. One, *initnumber,* that stores the number of initial conditions. Other, *initpt,* that stores an array of arrays of strings. These string are the names of the variables that will act as initial conditions.

```
54⟨ Read initial conditions 54 ⟩ ≡
```

```
nextlin_(linea, 80, msg1, 80);
linecount = 1;
ninit = leerint_(linea,&linecount, 80);
initnumber = (int *) malloc(ninit * sizeof (int));
if (Λ ≡ initnumber) {
    errcode = −24;
    strcpy (msg6, "Fail␣to␣allocate␣memory.");
    errmsg_(&errcode, msg6, msg7, 80, 80);
    return (errcode);
}
initpt = (char ***) malloc((ninit + 1) * sizeof (char **));
if (Λ ≡ boundpt) {
    errcode = −24;
    strcpy (msg6, "Fail␣to␣allocate␣memory.");
    errmsg_(&errcode, msg6, msg7, 80, 80);
    return (errcode);
}
initpt [ninit] = Λ;
for (ninputs = 0; ninputs < ninit; ninputs ++) {
        /* Fill the array initpt */
    nextlin_(linea, 80, msg1, 80);
    ⟨ Read a line of variables for initial conditions 55 ⟩;
}
```

This code is used in chunk 8.

¶ In each line, a set of varibles for initial conditions is read. This needs allocating memory for the variables.

55⟨ Read a line of variables for initial conditions 55 ⟩ ≡

```
initpt [ninputs] = (char **) malloc(sizeof (char *));
if (Λ ≡ initpt [ninputs]) {
    errcode = −24;
    strcpy (msg6, "Fail␣to␣allocate␣memory.");
    errmsg_(&errcode, msg6, msg7, 80, 80);
    return (errcode);
}
linecount = 1;    /* First word in the line */
saltesp_(linea,&linecount, 80);    /* Skip blanks */
```

```
for (nvars = 0; linecount > 0; nvars ++) {
    /* Look for the names in this line */
    ⟨ Read an initial conditions variable name 56 ⟩;
}
initnumber [ninputs] = nvars;
initpt [ninputs] = (char **) realloc(initpt [ninputs],
        (nvars + 1) * (sizeof (char *)));
if (Λ ≡ initpt [ninputs]) {
    errcode = −24;
    strcpy (msg6, "Fail␣to␣allocate␣memory.");
    errmsg_(&errcode, msg6, msg7, 80, 80);
    return (errcode);
}
initpt [ninputs][nvars] = Λ;
```

This code is used in chunk 54.

¶ Reading avariable implies assigning more memory to the existing array *boundpt* [*ninputs*], computing the pointer to the place in the line where the variable is and reading and storing it.

56⟨ Read an initial conditions variable name 56 ⟩ ≡

```
initpt [ninputs] = (char **) realloc(initpt [ninputs],
        (nvars + 1) * (sizeof (char *)));
if (Λ ≡ initpt [ninputs]) {
    errcode = −24;
    strcpy (msg6, "Fail␣to␣allocate␣memory.");
    errmsg_(&errcode, msg6, msg7, 80, 80);
    return (errcode);
}
btain thepointer...;
initpt [ninputs][nvars] = (char *)
        malloc((strlen (wordpt) + 1) * (sizeof (char)));
if (Λ ≡ initpt [ninputs][nvars]) {
    errcode = −24;
    strcpy (msg6, "Fail␣to␣allocate␣memory.");
    errmsg_(&errcode, msg6, msg7, 80, 80);
    return (errcode);
}
strcpy (initpt [ninputsj][nvarsj], wordpt);
```

```
    linecount += 2;
    info = linecount;
    saltesp_(linea,&linecount, 80);    /* Prepare the next read */
```

This code is used in chunk 55.

A.1.2.2.3. Transfer the names of the initial conditions

58¶ ⟨ Pack the intial conditions names 58 ⟩ ≡

```
    info = pvm_pkint (&totinit, 1, 1);
    if (info < 0) return (babpvmerror_(info, "pvm_pkint"));
    for (ninputs = 0; ninputs < ninit; ninputs ++) {
        for (nvars = 0; nvars < initnumber [ninputs]; nvars ++) {
#if 0    /* Deleted code */
        wordlen = strlen (boundpt [ninputs][nvars]);
        info = pvm_pkint (&wordlen, 1, 1);
        if (info < 0) return (babpvmerror (info, "pvm_pkint"));
#endif
        info = pvm_pkstr (initpt [ninputs][nvars]);
        if (info < 0) return (babpvmerror_(info, "pvm_pkstr"));
        }
    }
```

This code is used in chunk 41.

A.1.2.2.4. Transfer the initial conditions

¶ *Sndcode_* computes a flag that indicates if initial conditions will be transferred to the remote code.

60⟨ Determine if initial conditions will be transferred and pack the flag 60 ⟩ ≡

```
    initflag = (0 ≡ prevexec) ∧ (control [0] ≠ BabSSCalc) ∧ (0 ≡ n);
    ⟨ Pack initflag 84 ⟩;
```

This code is used in chunk 45.

¶ The initial conditions are obtained from the input signals to the block. The assignment of values is deferred to an ancillary routine.

61⟨ Compute initial conditions and pack them 61 ⟩ ≡

```
    if (Λ ≠ strstr (execname, "relap5.x"))
        relap5_init (&(ent [nbound + 1]), ninit, initnumber, initto);
    if (Λ ≠ strstr (execname, "pvmbis"))
```

```
        relap5_init (&(ent [nbound + 1]), ninit, initnumber, initto);
    info = pvm_pkdouble (initto, totinit, 1);
    if (info < 0) return (babpvmerror_(info, "pvm_pkdouble"));
```

This code is used in chunk 45.

A.1.2.3. Boundary conditions management

A.1.2.3.1. Save and restore variables

¶ *Sndcode_* saves

- The pointer to the names of the input variables in *vinfij* [4].
- In *vinfij* [5] the address of the array boundnumber.
- The number of input signals from which the boundary conditions will be derived in *vinfij* [6].
- The total number of boundary conditions that will be transferred to the remote code in *vinfij* [7].

Memory is reclaimed for the array boundto and the pointer to the area requested saved in *vinfij* [13].

64⟨ Save boundary conditions information 64 ⟩ ≡

```
    vinfij [4] = (double) ((int) boundpt);
    vinfij [5] = (double) ((int) boundnumber);
    vinfij [6] = (double) nbound;
    for (ninputs = 0, totbound = 0; ninputs < nbound; ninputs ++) {
        totbound += boundnumber [ninputs];
    }
    vinfij [7] = (double) totbound;
    boundto = (double *) malloc(totbound * sizeof (double));
    vinfij [13] = (double) ((int) boundto);
```

This code is used in chunk 10.

¶

1. The pointer to the number of boundary conditions per input signal is saved in *boundnumber.*
2. The number of input signals from which boundary conditions will be derived is stored in *nbound.*
3. The total number of bondary conditions to the remote code is saved in *totbound.*

65⟨ Load boundary conditions variables 65 ⟩ ≡

```
    boundnumber = (int *) ((int) vinfij [5]);
    nbound = (int) vinfij [6];
    totbound = (int) vinfij [7];
```

boundto = (**double** ∗) ((**int**) *vinfij* [13]);

This code is used in chunk 6.

¶ *boundpt* takes the pointer to the memory assigned to the names of the initial conditions.

66⟨ Get the boundary conditions names 66 ⟩ ≡

boundpt = (**char** ∗∗∗) ((**int**) *vinfij* [4]);

This code is used in chunk 23.

¶ Boundary conditions variables needed

67⟨ Module variables 2 ⟩ +≡

int ∗*boundnumber;*
int *nbound, totbound;*
double ∗*boundto*;
char ∗∗∗*boundpt;*

A.1.2.3.2. Boundary conditions reading

¶ *Sndcode_* calls nextlin to read the number of input signals that will be used to compute the remote code boundary conditions. String *msg1* is written if an end-of-file is detected. The number of input signals is stored in *nbound.* Two arrays are needed. One, *boundnumber,* that stores the number of boundary conditions. Other, *boundpt,* that stores an array of arrays of strings. These strings are the names of the variables that will act as boundary conditions.

69⟨ Read boundary conditions 69 ⟩ ≡

```
nextlin_(linea, 80, msg1, 80);
linecount = 1;
nbound = leerint (linea,&linecount, 80);
boundnumber = (int *) malloc(nbound * sizeof (int));
if (Λ ≡ boundnumber) {
    errcode = −24;
    strcpy (msg6, "Fail␣to␣allocate␣memory.");
    errmsg_(&errcode, msg6, msg7, 80, 80);
    return (errcode);
}
boundpt = (char ***) malloc((nbound + 1) * sizeof (char **));
if (Λ ≡ boundpt) {
    errcode = −24;
    strcpy (msg6, "Fail␣to␣allocate␣memory.");
```

```
        errmsg_(&errcode, msg6, msg7, 80, 80);
        return (errcode);
    }
    boundpt [nbound] = Λ;
    for (ninputs = 0; ninputs < nbound; ninputs ++) {
            /* Fill the array boundpt */
        nextlin_(linea, 80, msg1, 80);
        ⟨ Read a line of variables for boundary conditions 70 ⟩;
    }
```

This code is used in chunk 8.

¶ In each line, a set of varibles for boundary conditions is read. This needs allocating memory for the variables.

```
70⟨ Read a line of variables for boundary conditions 70 ⟩ ≡
    boundpt [ninputs] = (char **) malloc(sizeof (char *));
    if (Λ ≡ boundpt [ninputs]) {
        errcode = −24;
        strcpy (msg6, "Fail␣to␣allocate␣memory.");
        errmsg_(&errcodemsg6, msg7, 80, 80);
        return (errcode);
    }
    linecount = 1;    /* First word in the line */
    saltesp_(linea,&linecount, 80);    /* Skip blanks */
    for (nvars = 0; linecount > 0; nvars ++) {
            /* Look for the names in this line */
        ⟨ Read a boundary conditions variable name 71 ⟩;
    }
    boundnumber [ninputs] = nvars;
    boundpt [ninputs] = (char **) realloc(boundpt [ninputs],
        (nvars + 1) * (sizeof (char *)));
    if (Λ ≡ boundpt [ninputs]) {
        errcode = −24;
        strcpy (msg6, "Fail␣to␣allocate␣memory.");
        errmsg_(&errcode, msg6, msg7, 80, 80);
        return (errcode);
    }
    boundpt [ninputs][nvars] = Λ;
```

This code is used in chunk 69.

¶ Reading avariable implies assigning more memory to the existing array *boundpt* [*ninputs*], computing the pointer to the place in the line where the variable is and reading and storing it.

```
71⟨ Read a boundary conditions variable name 71 ⟩ ≡
    boundpt [ninputs] = (char **) realloc(boundpt [ninputs],
            (nvars + 1) * (sizeof (char *)));
    if (Λ ≡ boundpt [ninputs]) {
        errcode = −24;
        strcpy (msg6, "Fail␣to␣allocate␣memory.");
        errmsg_(&errcode, msg6, msg7, 80, 80);
        return (errcode);
    }
    btain thepointer...;
    boundpt [ninputs][nvars] = (char *)
        malloc((strlen (wordpt) + 1) * (sizeof (char)));
    if (Λ ≡ boundpt [ninputs][nvars]) {
        errcode = −24;
        strcpy (msg6, "Fail␣to␣allocate␣memory.");
        errmsg_(&errcode, msg6, msg7, 80, 80);
        return (errcode);
    }
    strcpy (boundpt [ninputs][nvars], wordpt);
    linecount += 2;
    info = linecount;
    saltesp_(linea,&linecount, 80);    /* Prepare the next read */
```

This code is used in chunk 70.

A.1.2.3.3. Transfer the boundary conditions names

```
73¶  ⟨ Pack the boundary conditions names 73 ⟩ ≡
    info = pvm_pkint (&totbound, 1, 1);
    if (info < 0) return (babpvmerror_(info, "pvm_pkint"));
    for (ninputs = 0; ninputs < nbound; ninputs ++) {
        for (nvars = 0; nvars < boundnumber [ninputs]; nvars ++) {
#if 0    /* Deleted code. Why keep it? */
        wordlen = strlen (boundpt [ninputs][nvars]);
        info = pvm_pkint (&wordlen, 1, 1);
        if (info < 0) return (babpvmerror_(info, "pvm_pkint"));
#endif
```

```
        info = pvm_pkstr (boundpt [ninputs][nvars]);
        if (info < 0) return (babpvmerror_(info, "pvm_pkstr"));
        }
    }
```

This code is used in chunk 41.

A.1.2.3.4. Transfer of boundary conditions in execution

¶ *Sndcode_* identifies the remote code and applies a different treatment of the boundary conditions for each code. The boundary conditions are obtained from the input signals to the block. The assignment of values is deferred to an ancillary routine.

75⟨ Compute boundary conditions and pack them 75 ⟩ ≡

```
    if (Λ ≠ strstr (execname, "relap5.x"))
        relap5_bound (&(ent [1]), nbound, boundnumber, boundto);
    if (Λ ≠ strstr (execname, "pvmbis"))
        relap5_bound (&(ent [1]), nbound, boundnumber, boundto);
    info = pvm_pkdouble (boundto, totbound, 1);
    if (info < 0) return (babpvmerror_(info, "pvm_pkdouble"));
```

This code is used in chunk 45.

A.1.2.3.5. Outputs

*Sndcode_* sends the remote code a flag to warn it about the reception of the names of the output variables. This flag is set to 1 if this is the last time the "nsteps" loop is executed. A flag to indicate if the outputs must be sent by the remote code is also packed.

76⟨ Pack the flags for output reception 76 ⟩ ≡

```
    rcvouts = (n ≡ nsteps − 1);
    info = pvm_pkint (&rcvouts, 1, 1);
    if (info < 0) return (babpvmerror_(info, "pvm_pkint"));
#if 0    /* Deleted code */
    sndoutnm = (¬rcspawn) ∧ rcvouts;
    info = pvm_pkint (&sndoutnm, 1, 1);
    if (info < 0) return (babpvmerror_(info, "pvm_pkint"));
#endif
```

This code is used in chunk 45.

77¶⟨ Module variables 2 ⟩ +≡

```
    int rcvouts;
```

A.1.3. Standard message passage procedures

¶ *Sndcode_* obtains a buffer to pack the variables, issuing a message and returning an error code in case no buffer is avaliable.

79⟨ Obtain a buffer 79 ⟩ ≡

```
bufid = pvm_initsend (PvmDataDefault);
if (bufid < 0) return (babpvmerror_(info, "pvm_initsend"));
```

This code is used in chunks 13, 41 and 45.

¶ Here you will find packaging of some variables that are sent to the remote code. Since they are essentially equal to one another, they are presented in a separate section. *Sndcode_* packs control[0].

80⟨ Pack *control* [0] 80 ⟩ ≡

```
info = pvm_pkint (control, 1, 1);
if (info < 0) return (babpvmerror_(info, "pvm_pkint"));
```

This code is used in chunks 13 and 45.

¶ *Sndcode_* packs the time step.

81⟨ Pack the time step 81 ⟩ ≡

```
info = pvm_pkdouble(&incrtie, 1, 1);
if (info < 0) return (babpvmerror_(info, "pvm_pkdouble"));
```

¶ The remote code is informed of the current simulation time.

82⟨ Pack current time 82 ⟩ ≡

```
info = pvm_pkdoule(&tiempo, 1, 1);
if (info < 0) return (babpvmerror_(info, "pvm_pkdouble"));
```

This code is used in chunk 45.

¶ *Sndcode_* packs *prevexec*, the flag indicating wether the code has been executed during the last time step.

83⟨ Pack *prevexec* 83 ⟩ ≡

```
info = pvm_pkint (&prevexec, 1, 1);
if (info < 0) return (babpvmerror_(info, "pvm_pkint"));
```

This code is used in chunks 13 and 45.

¶ *Sndcode_* packs *initflag,* the flag indicating wether initial conditions are to be read

84⟨ Pack *initflag* 84 ⟩ ≡

```
info = pvm_pkint (&initflag, 1, 1);
if (info < 0) return (babpvmerror_(info, "pvm_pkint"));
```

This code is used in chunks 13 and 60.

A.1.4. Ancillary algorithms

¶ A provisional pointer *wordpt* points to the same address as pointer linea + linecount - 1. Linecount is decreased to convert FORTRAN to C indices. Then, linea is increased until a non-blank character is found, and a terminating `'\0'` is appended to the name, so that the name can be read.

86⟨ Obtain the pointers to the first and last non-blank characters delimiting the string 86 ⟩ ≡

```
wordpt = linea + (—linecount);
for ( ; (¬isspace (linea[linecount])) ∧ (linecount < 80); linecount ++);
linea[linecount] = '\0';
```

This code is used in chunks 25, 26 and 27.

A.1.5. Functions for initial and boundary conditions

87⟨ Auxiliary functions 87 ⟩ ≡

```
⟨ Initial conditions functions for RELAP5 90 ⟩;
⟨ Boundary conditions functions for RELAP5 88 ⟩;
```

This code is used in chunk 1.

¶ The boundary conditions for RELAP5 need no preprocessing; the values are directly passed to RELAP5

88⟨ Boundary conditions functions for RELAP5 88 ⟩ ≡

```
#ifdef _ _STDC_ _
    void relap5_bound (double *entpt, int nbound, int boundnumber [ ], double *pt)
    {
        int i, j, count;
            /* – simply passes to relap5 the module input signals.  */
        count = 0;
        for (i = 0; i < nbound; i++) {
            for (j = 0; j < boundnumber [i]; j++) {
                pt [count] = entpt [count];
                count ++;
```

```
            }
        }
    }
#else
    void relap5_bound (entpt, nbound, boundnumber, pt)
        double **entpt;
        int nbound;
        int boundnumber [ ];
        double **pt;
    {   /* – simply passes to relap5 the module input signals.    */
        *pt = *entpt;
    }
#endif
```

This code is used in chunk 87.

89¶ ⟨ Functions prototypes 89 ⟩ ≡

```
#ifdef _ _STDC_ _
    void relap5_bound (double **entpt, int nbound, int *boundnumber,
        double **pt);
#else
    void relap5_bound ( );
#endif
```

See also chunk 91.

This code is used in chunk 1.

¶ And the same for initial conditions

90⟨ Initial conditions functions for RELAP5 90 ⟩ ≡

```
#ifdef _ _STDC_ _
    void relap5_init (double *entpt, int ninit, int initnumber [ ], double *pt)
    {
        int i, j, count;
            /* – simply passes to relap5 the module input signals.    */
        count = 0;
        for (i = 0; i < ninit; i++) {
            for (j = 0; j < initnumber [i]; j++) {
                pt [count] = entpt [count];
```

```
                count ++;
            }
        }
    }
#else   /* – Otherwise, functions must be defined as Kernigan-Ritchie functions. */
    void relap5_init (entpt, ninit, initnumber, pt)
        double **entpt;
        int ninit;
        int initnumber [ ];
        double **pt;
    {       /* – simply passes to relap5 the module input signals. */
        *pt = * entpt;
    }
#endif
```

This code is used in chunk 87.

91¶ ⟨ Functions prototypes 89 ⟩ +≡

```
#ifdef _ _STDC_ _
    void relap5_init (double **entpt, int nbound, int *boundnumber, double **pt);
#else
    void relap5_init ( );
#endif
```

¶

Appendix B. File `rcvcode`

B.1. Description of the module

This module belongs to BABIECA, and is the responsible for receiving a set of set of variables from a remote code. The file complies with the mandatory structure for a BABIECA module, namely

```
1⟨ Include files 37 ⟩
    BAB_MODULE(rcvcode)
    {
        ⟨ Module variables 6 ⟩;
        switch (control [0]) {
        case BabRead:
            {
                ⟨ Input file reading 9 ⟩;
                break;
            }
        case BabCalc:
            case BabSSCalc:
            case BabFbck:
            case BabSsFbck:
            {
                ⟨ Calculation mode 3 ⟩;
                break;
            }
        default:;
        }
        return BabNoError;
        }
```

B.1.1. Code sections

B.1.1.1. Calculation mode: Remote code active

If *ent* [0] is 1 the remote code is active during the current time step. *rcvcode* will receive the output signals.

```
3⟨ Calculation mode 3 ⟩ ≡
    ⟨ Recall the number of outputs 16 ⟩;
    if (1.0 ≡ ent [0]) {
        ⟨ Recall variables for active code 21 ⟩;
        if (0 ≡ rcspawn) {
            ⟨ Transfer the names of the variables 5 ⟩;
```

```
        vinvar [0] = 1;    /* Set rcspawn */
    }
    ⟨ Receive and unpack the variables from the remote code 7 ⟩;
}
else {
    ⟨ Assign dummy outputs 8 ⟩;
}
```

This code is used in chunk 1.

B.1.1.1.1. The first contact

¶ If this is the first time the remote code is contacted, the names of the variables needed must be sent. packs and sends the names.

```
5⟨ Transfer the names of the variables 5 ⟩ ≡
    ⟨Recall the names of the variables 18 ⟩;
    ⟨ Obtain a buffer 24 ⟩;
    ⟨ Pack the number of output signals 27 ⟩;
    ⟨ Pack the names of the output signals 28 ⟩;
    ⟨ Send the names of the output signals 29 ⟩;
    ⟨ Free the PVM buffer 25 ⟩;
    ⟨ Free used memory 35 ⟩;
```

This code is used in chunk 3.

¶ *noutputs* is a dummy variable for a loop

```
6⟨ Module variables 6 ⟩ ≡
    int noutputs;
```

See also chunks 17, 19 and 22.

This code is used in chunk 1.

B.1.1.1.2. Receive the outputs

In any case, the outputs are sent to the block outputs.

```
7⟨ Receive and unpack the variables from the remote code 7 ⟩ ≡
    ⟨ Receive the message 30 ⟩;
    ⟨ Unpack the data into sal 31 ⟩;
```

This code is used in chunk 3.

B.1.1.1.3. Inactive remote code

If the block has not been activated, the outputs are those of the previous time step.

8⟨ Assign dummy outputs 8 ⟩ ≡

```
for (noutputs = 0; noutputs < numsal; noutputs ++)
    sal [noutputs] = sal [noutputs + numsal];
```

This code is used in chunk 3.

B.1.1.2. Reading the input file

The first line is the number of variables to be received from the remote code, *numsal* and then the variables themselves are read. The protocol of communication with the driver comes next.

9⟨ Input file reading 9 ⟩ ≡

```
⟨ Particular declarations for the input file reading section 13 ⟩;
⟨ Read numsal 10 ⟩;
⟨ Read the names of the variables 11 ⟩;
⟨ Save values for the driver 20 ⟩;
```

This code is used in chunk 1.

10¶ ⟨ Read *numsal* 10 ⟩ ≡

```
nextlin_(linea, 80, msg1, 80);
linecount = 1;
numsal = leerint_(linea,&linecount, 80);
```

This code is used in chunk 9.

11¶ ⟨ Read the names of the variables 11 ⟩ ≡

```
⟨ Reclaim memory for the names and setup the array 33 ⟩;
for (noutputs = 0; noutputs < numsal; noutputs ++) {
    ⟨ Read one line containing the variable name 12 ⟩;
}
```

This code is used in chunk 9.

12¶ ⟨ Read one line containing the variable name 12 ⟩ ≡

```
nextlin_(linea, 80, msg1, 80);
linecount = 1;
saltesp_(linea,&linecount, 80);    /* Find the first non-blank */
```

⟨ Obtain the pointers to the first and last non-blank characters delimiting the string 36 ⟩;
⟨ Allocate memory for a variable name 34 ⟩;
strcpy (*outputpt* [*noutputs*], *wordpt*);

This code is used in chunk 11.

13¶ ⟨ Particular declarations for the input file reading section 13 ⟩ ≡
int *wordlen, linecount;*
char *linea*[80], ∗*wordpt;*
char *msg1* [80] = "RCVCODE.␣ERROR␣READING␣INPUT␣LINE.";
char *msg6* [80];
char *msg7* [80] = "␣";

This code is used in chunk 9.

B.1.2. Ancillary algorithms

B.1.2.1. Load and restore

¶ The number of outputs is stored in *numsal;*
16⟨ Recall the number of outputs 16 ⟩ ≡
numsal = (**int**) *vinfij* [0];

This code is used in chunk 3.

¶ Declaration of *numsal*
17⟨ Module variables 6 ⟩ +≡
int *numsal;*

¶ *rcvcode_* obtains the pointer to the names of the output variables,
18⟨ Recall the names of the variables 18 ⟩ ≡
outputpt = (**char** ∗∗) ((**int**) *vinfij* [1]);

This code is used in chunk 5.

19¶ ⟨Module variables 6 ⟩ +≡
char ∗∗*outputpt;*

¶ *rcvcode_* saves the number of output variables in *vinfij* [0], saves the pointer to the names of output variables in *vinfij* [1]. The *control* array is filled:

- The number of outputs, i.e., *control* [3], is set to *numsal.*
- The size of the *vinfij* array is saved in *control* [4], i.e, 2.
- The size of the *vinvar* array is saved in *control* [5], i.e, 1.
- The size of the *stat* array, i.e., 0, is saved in *control* [6].

20⟨ Save values for the driver 20 ⟩ ≡

```
vinfij [0] = (double) numsal;
vinfij [1] = (double) ((int) outputpt);
control [3] = numsal;
control [4] = 2;
control [5] = 1;
control [6] = 0;
```

This code is used in chunk 9.

¶ *rcspawn* tells if the code was already started and *rcodetid* is the task identification of the code in the PVM environment.

21⟨ Recall variables for active code 21 ⟩ ≡

```
rcspawn = (int) vinvar [0];
rcodetid = (int) ent [1];
```

This code is used in chunk 3.

¶ Variables

22⟨ Module variables 6 ⟩ +≡

```
int rcspawn, rcodetid = 0;
```

B.1.2.2. Standard message passing procedures

¶ *rcvcode_* obtains a buffer to pack the variables, issuing a message and returning an error code in case no buffer is avaliable.

24⟨ Obtain a buffer 24 ⟩ ≡

```
bufid = pvm_initsend (PvmDataDefault);
if (bufid < 0) return (babpvmerror_(info, "pvm_initsend"));
```

This code is used in chunk 5.

25¶ ⟨ Free the PVM buffer 25 ⟩ ≡

```
info = pvm_free (bufid);
```

if (*info* < 0) **return** *babpvmerror_*(*info,* "pvm_freebuf");

This code is used in chunk 5.

¶ *bufid* is the identifier of the buffer. It also is the error code. *info* is the error code returned by the PVM routines. <Module variables= int bufid, info;

¶ *rcvcode_* packs the number of output signals to receive from the remote code.

27⟨ Pack the number of output signals 27 ⟩ ≡

info = *pvm_pkint* (&*numsal,* 1, 1);
if (*info* < 0) **return** (*babpvmerror_*(*info,* "pvm_pkint"));

This code is used in chunk 5.

¶ *rcvcode_* packs the names of the output variables.

28⟨ Pack the names of the output signals 28 ⟩ ≡

for (*noutputs* = 0; *noutputs* < *numsal; noutputs* ++) {
info = *pvm_pkstr* (*outputpt* [*noutputs*]);
if (*info* < 0) **return** (*babpvmerror_*(*info,* "pvm_pkstr"));
}

This code is used in chunk 5.

¶ *rcvcode_* sends the packed data to the remote code.

29⟨ Send the names of the output signals 29 ⟩ ≡

info = *pvm_send* (*rcodetid,* OUTNAMES);
if (*info* < 0) **return** (*babpvmerror_*(*info,* "pvm_send"));

This code is used in chunk 5.

¶ *rcvcode_* receives the output variables from the remote code.

30⟨ Receive the message 30 ⟩ ≡

info = *pvm_recv* (*rcodetid,* OUTPUTS);
if (*info* < 0) **return** (*babpvmerror_*(*info,* "pvm_recv"));

This code is used in chunk 7.

¶ *rcvcode_* unpacks the data in the *sal* array.

31⟨ Unpack the data into *sal* 31 ⟩ ≡

```
info = pvm_upkdouble (sal, numsal, 1);
if (info < 0) return (babpvmerror_(info, "pvm_initsend"));
```

This code is used in chunk 7.

B.1.2.3. Memory management

¶ The array of names of output variables must end with a NULL pointer.

33⟨ Reclaim memory for the names and setup the array 33 ⟩ ≡

```
outputpt = (char **) malloc((numsal + 1) * sizeof (char *));
if (Λ ≡ outputpt) {
    m = −24;
    strcpy (msg6, "Fail␣to␣allocate␣memory.");
    errmsg_(&m, msg6, msg7, 80, 80);
    errcode = −24;
    return (errcode);
}
outputpt [numsal] = Λ;
```

This code is used in chunk 11.

34¶ ⟨ Allocate memory for a variable name 34 ⟩ ≡

```
outputpt [noutputs] = (char *) malloc((strlen (wordpt) + 1) * (sizeof (char)));
if (Λ ≡ outputpt [noutputs]) {
    m = −24;
    strcpy (msg6, "Fail␣to␣allocate␣memory.");
    errmsg_(&m, msg6, msg7, 80, 80);
    errcode = −24;
    return (errcode);
}
```

This code is used in chunk 12.

¶ *rcvcode_* frees up the memory occupied by the names of the output variables.

35⟨ Free used memory 35 ⟩ ≡

```
for (noutputs = 0; noutputs ≤ numsal; noutputs ++) {
    free (outputpt [noutputs]);
}
free (outputpt);
```

This code is used in chunk 5.

B.1.2.4. Strings reading

A provisional pointer *wordpt* points to the same address as pointer linea + linecount - 1. Linecount is decreased to convert FORTRAN to C indices. Then, linea is increased until a non-blank character is found, and a terminating `'\0'` is appended to the name, so that the name can be read.

36⟨ Obtain the pointers to the first and last non-blank characters delimiting the string 36 ⟩ ≡

```
wordpt = linea + (—linecount);
for ( ; (¬isspace (linea[linecount])) ∧ (linecount < 80); linecount ++);
linea[linecount] = '\0';
```

This code is used in chunk 12.

B.1.3. Header files

First the system header files. We need to use the memory allocation routines, so `stdlib.h` is needed. `string.h` for the string manipulation routines together with the standard character handling from `ctype.h`. The `pvm` routines definition are taken from

37⟨ Include files 37 ⟩ ≡

```
#include <stdlib.h>
#include <string.h>
#include <ctype.h>
#include <pvm3.h>
```

See also chunks 38 and 39.

This code is used in chunk 1.

¶ Then, the BABIECA variables definition. Reading routines from `lectu.h`, error handling from `baberr.h`, the time variables from `tiempo.h` and the module declarations and macros from `modul.h`

38⟨ Include files 37 ⟩ +≡

```
#include "utilidad/lectu.h"
#include "utilidad/baberr.h"
#include "common/tiempo.h"
#include "modulos/modul.h"
```

¶ Last, specific header files: the set of tags used for `vm` message passing.

39⟨ Include files 37 ⟩ +≡

```
#include "./pvmtags.h"
```

¶

Appendix C. Remote code standard specifications

C.1. Code linkage

Existing simulation tools may be combined if their structure complies with a common standard. The pseudocode presented here attempts to provide the essentials of such a standard for code linkage. The code provided is suitable for simple linkage as well as for tree-structured simulation. The additions needed for the latter case will be presented in section C.1.2.2.2. The routines referenced will have three kind of names

get_something when the internal structures are to be searched for the information,

rcv_something when the information is to be received from some process

send_something when the information is to be sent to some process

This code is often referred to as the descendant code, and the process that created it the parent process. This is a pseudo-code, and no care whatsoever has been taken in properly declaring variables. Rather, the functions' arguments are solely a hint on the key parameters they will take.

2¶ The standard is structured in two routines, *main*( ) and *calculo* ( ), and the main characteristics of both are presented. The later is the responsible of driving the time advancement of the simulation.

2 ⟨ Main program 3 ⟩
⟨ *calculo* ( ) routine 14 ⟩

C.1.1. Main program

The main program, as coded here, is responsible of the top level structure of the program, calling the routines for reading the input file(s), setting up the tables for the management of the read data and call the *calculo* ( ) routine. Then, two loops are set up. The outer loop keeps the process waiting for a message from the parent process telling it to exit. The inner loop (used for tree-structured simulation) restarts a *caclulo* ( ) session when the pile of restart points is still occupied. During the simulation, the parent process will have requested some points to be saved in this private stack. In this case, *stack occupied* ( ) will return a non-zero value. Note that in standalone runs requests for restart saving will never arrive, and *stack occupied* ( ) will always return a 0. ηReceive finish message 12 ⟩ will deal with the other loop in case of no connections.

3⟨ Main program 3 ⟩ ≡

```
int main( )
{
    ⟨ Parse command line 5 ⟩;
    ⟨ Setup the internal tables 6 ⟩;
    do {
        do {
```

```
            ⟨ Calculate transient mode 7 ⟩;
        } while (stack_occupied ( ));
        get_finish_message ( );
        ⟨ Receive finish message 12 ⟩;
    } while (¬finish_message);
    ⟨ Results writing 8 ⟩;
    close_files ( );
    exit (0);
}
```

This code is used in chunk 2.

C.1.1.1. Standalone code

C.1.1.1.1. Starting

The command line parsing processes the command line and sets up the files needed. It may also set the variable *parent process,* that will be the flag identifying an externally driven run.

5⟨ Parse command line 5 ⟩ ≡

```
parse_command_line (input_file, flags, parent_process);
if (parserr) report_and_exit (parserr);
```

This code is used in chunk 3.

6¶ The input information has to be read from the named file. This may direct the code to reading further files (e.g. if this is a restart case). The absence of an input file means that the internal tables data are to be read from some other process. This will occur when within a tree simulation.

6⟨ Setup the internal tables 6 ⟩ ≡

```
if (input_file) {
    open_file (input_file);
    read_input_file (calc_type, physical_system, initial_conditions,
            boundary_conditions, initial_trips);
    switch (calc_type) {
    case 'new':
        {
            setup_internal_tables ( );
            get_boundary_conditions ( );
            get_state_vector (initial_conditions);
            break;
        }
    case 'restart':
        {
```

```
            read_internal_tables_and_state_vector (restart_file);
            get boundary conditions ( );
            break;
        }
    }
}
else {
    ⟨ Initialisation for tree simulation 13 ⟩;
}
    ⟨ Receive startup messages 10 ⟩;
```

This code is cited in chunk 17.

This code is used in chunk 3.

C.1.1.1.2. Transient calculation

The transient calculation is done in the *calculo* ( ) routine. It will return an error code in case of unsuccessful computation. Exit from the program is not needed, since there may be more simulations to be done. See section C.1.2 for the details of the routine.

```
7⟨ Calculate transient mode 7 ⟩ ≡
    calcerr = calculo ( );
    if (calcerr) report ("CALCULO␣FUNCTION␣ERROR.");
```

This code is cited in chunk 12.

This code is used in chunk 3.

C.1.1.1.3. Results writing

For the case of running stand alone, the results file is written.

```
8⟨ Results writing 8 ⟩ ≡
    graphical_outputs ( );
statistics ( );
```

This code is used in chunk 3.

C.1.1.2. Code for the linkage

C.1.1.2.1. Linkage to an external driver

In case the code is linked to an external driver, it will receive the same kind of information that was got from the input files, overwriting that. To enter this section, the code must have been started by other process. Then, *parent process* will be *true.*

The first messages to be received relate to the initialisation of the code and of the transmission variables between the two tasks involved.

```
10⟨ Receive startup messages 10 ⟩ ≡
    if (parent_process) {
        rcv_in_trips_names(parent_process);
        ⟨ Return if error 33 ⟩;
        rcv_boundary_conditions_names(parent_process);
        ⟨ Return if error 33 ⟩;
        rcv_states_names(parent_process);
        ⟨ Return if error 33 ⟩;
        rcv_initial_conditions_names(parent_process);
        ⟨ Return if error 33 ⟩;
    }
```

This code is used in chunk 6.

11¶ The trips received here are a sort of manual actuations over the models of the code. They represent discrete transitions induced from the parent code, and directly set the actual variables in the descendant code. The code in section C.1.2.1 clearly reflects this in that one of the requirements for the advancement routine is that a trip triggering stops the advancement of the continuous simulation to allow proper initialisation of the models in the new situation (see ⟨ Advance until the next requested time 18 ⟩). The scheme for the boundary conditions is somewhat different. The values received then are introduced replacing the values in the interpolation tables and are used as needed in the advancement scheme. A corollary from this discussion is that the remote code must be aware of the trips occurring in the descendant code and treat them properly and that when the results of the descendant code imply a trip in the parent process, a procedure that parallels the code in < *dvance...* has to be implemented. this is done via an *iteration flag* (See < *eceive additional...*). These trips we call *out trips,* and their names are received in ⟨ Receive the names of the output variables 24 ⟩

12¶ When all the stored points are computed, a finish message is received from the parent process. If this finish message is not zero the processing will return to ⟨ Calculate transient mode 7 ⟩. Recall that *finish message* may be set to 1 before entering this section (in_*get_finish_message* ( )) so that the code properly ends when it is not connected.

```
12⟨ Receive finish message 12 ⟩ ≡
    if (parent_process) {
        transerr = rcv_finish_message (parent_process);
        if (transerr) report_and_exit (transerr);
    }
```

This code is cited in chunk 3.

This code is used in chunk 3.

C.1.1.2.2. Code for tree simulation

When the tree simulation is in progress, the code may be initialised from an intermediate step of the simulation that has been computed by some other process. The, it has to receive the identification of the task that will provide the restart informations, after which the restart information is received from

this task. if the task is this very process, the restart information will be read from the internal tables. A unique label identifying the run is also received for bookkeeping purposes. The database identifier is received and, if this proces is not a restart process (*i.e.* it is the first process), the outputs names are sent.

```
13⟨ Initialisation for tree simulation 13 ⟩ ≡
    if (parent_process) {
        database_id = rcv_database_id (parent_process);
        ⟨ Return if error 33 ⟩;
        restart_process = rcv_restart_process (parent_process);
        ⟨ Return if error 33 ⟩;
        if (restart_process) {
            rcv_restart_info (parent_process, remote_process);
            ⟨ Return if error 33 ⟩;
            rcv_identifier_of_branch(parent_process);
            ⟨ Return if error 33 ⟩;
            ⟨ Free used restart point 31 ⟩;
        }
        else {
            send_out_names(database_id);
        }
    }
```

This code is used in chunk 6.

C.1.2. *calculo( )* routine

Is the responsible for the advance of the simulation until the final requested time. The numerical algorithm used is sent to another procedure, so that this one only holds the overall time management of the solution and of the links with the parent code. The flag *continue_simulation* is used for controlling the time advancement. It is set in ⟨ Get execution information from read files 16 ⟩ and may be modified by the external program. Since the user (or the remote code) may want to stop the simulation after only the initialisation stage (the first ⟨ Get trips and outputs 20 ⟩), a condition on this variable is set. Then the code advances until the *target_time.* The reason for plotting the outputs and managing the restart information at the

beginning of the loop is that, because of requirements of the parent process (*i.e.* iterations), the solutions computed may be invalid even if the code has successfully computed the advancement. It is only when the remote process validates the advancement that the data are ready for use and *plot_flag* and *save_restart_flag* will be set appropriately. This forces additional plots and restart saves at the end of the function for the last point. With this mechanism, the first (initialisation) point is also plotted.

14⟨ *calculo* ( ) routine 14 ⟩ ≡

```
int calculo ( )
{
    do {
        ⟨ Receive values for initial conditions 23 ⟩;
        ⟨ Get execution information from read files 16 ⟩;
        ⟨Receive execution information from parent 25 ⟩;
        ⟨ Get trips and outputs 20 ⟩;
        ⟨Receive the names of the output variables 24 ⟩;
        ⟨ Send results 28 ⟩;
        if (continue_simulation) { if (plot_flag) plot_outputs (previous_time);
            ⟨ Save restart 21 ⟩;
            previous_time = current_time;
            ⟨ Advance until the next requested time 18 ⟩;
            ⟨ Get trips and outputs 20 ⟩;
            ⟨ Send results 28 ⟩;
        }       * if (continue_simulation) */
    } while (continue_simulation);
    if (plot_flag) plot_outputs (target_time);
    ⟨ Save restart 21 ⟩;
}
```

This code is used in chunk 2.

C.1.2.1. Standalone running

The code presented here is what would be encountered in a generic simulation code. The structure ensures that the code can be running by itself without a driving process.

C.1.2.1.1. Start the loop

The information concerning this run is usually retrieved from the files that were read in the *main* procedure. This information sets the initial requested time and initialise the time variables.

16⟨ Get execution information from read files 16 ⟩ ≡

```
previous_time = initial_time;
requested_time = get_requested_time ( );
```

See also chunk 17.

This code is cited in chunk 14.

This code is used in chunk 14.

17¶ With this, both the requested time and the starting time are set. In case the target time is 0, the run will continue until it is stopped by some external procedure, such as a request from an internal module or from the parent process; otherwise, the code will stop in function of the *continue_simulation* flag (see below). The initial and final times are set in ⟨ Setup the internal tables 6 ⟩, and may be overridden by the remote code. To set the model for steady or transient simulation, an *steady_state* flag is also determined from the input files information, and can be modified by the external program in ⟨ Receive execution information from parent 25 ⟩. The same applies for the other flags. *continue simulation* controls the end of the simulation.

17⟨ Get execution information from read files 16 ⟩ + ≡

```
steady_state_flag = get_steady_state_flag ( );
save_restart_flag = get_save_restart_flag ( );
plot_flag = get_plot_flag ( );
continue_simulation = get_continue_simulation (current_time, target_time);
```

C.1.2.1.2. Time advancement

The simulation will now be advanced. This is done by means of some numerical scheme whose structure need not be discussed. The only requirement to the advancement procedure is that it stops either because *requested_time* has been reached or because a trip has been triggered. In the later case, the trip triggering time will be considered a plotting time and the simulation resumed thereafter.

18⟨ Advance until the next requested time 18 ⟩ ≡

```
timestep_state_vector = state_vector;
state_vector = advance_solution (steady_state_flag, timestep_state_vector);
⟨ Handle unrecoverable error 19 ⟩;
current_time = advancement_time;
    /* = min(trip_time, requested_time) */
```

This code is cited in chunk 11.

This code is used in chunk 14.

19¶ In case the advancement procedure returned with an unrecoverable error, some actions must be made to inform the to routine and to assure a clean exit. Such error can be checked, for instance,

if the last time reached by *advance_solution* ( ) is not the requested time and a trip has not occurred.

19⟨ Handle unrecoverable error 19 ⟩ ≡

```
if (advancement_time < requested_time /\¬trip_flag) {
    check_error ( );
    clean_up ( );
    report_and_return(req_timestep_error);
}
```

This code is used in chunk 18.

20¶ Once the state vector is obtained (either from the input files or from the calculation of a time step), the trips and outputs vector can be obtained.

20⟨ Get trips and outputs 20 ⟩ ≡

```
get_trips ( );
get_outputs (state_vector, boundary_conditions, trips);
```

This code is cited in chunk 14.

This code is used in chunk 14.

C.1.2.1.3. Restarts save

The restart saving consists in two parts; one is saving the current point in the stack if so requested by the external driver, deferred to ⟨ Manage restart stack 29 ⟩; the other is saving the point to a restart file.

21⟨ Save restart 21 ⟩ ≡

```
⟨ Manage restart stack 29 ⟩
if (save_restart_flag) write_restart_to_file ( );
```

This code is used in chunk 14.

C.1.2.2. Code for the linkage

The following sections describe the additions needed to enable the communication capacity of the code. They are inserted in the program stream so that the information received form the remote driving process will overwrite what would be normally done by the code. This section is broken up in two, one for module-like (this code is part of a larger simulation system) and for tree-like simulation (this code is driving the execution of a branch for a dynamically generated tree of simulations). The code remains consistent if the second part is not added.

C.1.2.2.1. Linkage to an external driver

If the code is to be linked for adding new models to an existing simulation, it may be required that certain initialisation be performed from data supplied by the parent process.

In this case some values will be received from the external program. These will have to be processed to obtain the same information than in the nonlinked case. For tree simulation, the whole tables have to be received prior to any other information. This section leaves the code in the same conditions as if initialised from the input file.

23⟨ Receive values for initial conditions 23 ⟩ ≡

```
initial_conditions_flag = rcv_initial_conditions_flag ( );
if (initial_conditions_flag) {
    rcv_initial_time (parent_process);
    ⟨ Return if error 33 ⟩;
    rcv_initial_conditions (parent_process);
    ⟨ Return if error 33 ⟩;
    rcv_initial_trips (parent_process);
    ⟨ Return if error 33 ⟩;
    rcv_initial_boundary_conditions ( );
    ⟨ Return if error 33 ⟩;
    get_boundary_conditions ( );
    get_state_vector (initial_conditions);
}
```

This code is used in chunk 14.

24¶ The names of the outputs to be sent to the parent process are received after the outputs themselves have been computed. This allows the splitting of the parent code linkage routines in two: one for sending variables and one for receiving outputs, allowing useful work to be done in the parent process in parallel to the initialisation tasks of the descendant code.

24⟨ Receive the names of the output variables 24 ⟩ ≡

```
rcv_output_names (parent_process);
⟨ Return if error 33 ⟩;
rcv_out_trips_names ( );
⟨ Return if error 33 ⟩;
```

This code is cited in chunk 11.

This code is used in chunk 14.

25¶ After the initialisation tasks, the code enters in the time loop. Within it, the parent process will send the information that will overwrite that of the code itself. Each blocking signal is a barrier point for the descendant code, so those must be avoided as much as possible. The linkage has to be compatible for both module-like and tree-simulation linkage. The suitable common flag to both

modes of linkage is the restart flag, and it will be received in blocking mode. It is thus essential that the parent process sends this flag as soon as possible. The rest of the flags will be needed only in some cases. We have two kind of flags:

1. Flags that do not affect the following time step.
2. Flags that do.

For the first type, the reception can occur anywhere. The message to send restart information to some other process within the tree simulation is of this kind. The second is the specific information for the next time request that is sent in mode-like connections. The messages are previously checked for arrival and only when actually present are they received.

25⟨ Receive execution information from parent 25 ⟩ ≡

```
if (parent_process) {
    ⟨ Probe send-restart message 30 ⟩;
    ⟨ Receive restart flag 26 ⟩;
    arrived_message = probe (parent_process);
    ⟨ Return if error 33 ⟩;
    if (arrived_message) {
        ⟨ Receive additional information 27 ⟩;
    }
}
```

This code is cited in chunk 17.

This code is used in chunk 14.

26¶ The restart flag is the only flag that is received in blocking mode independently of the driving code.

26⟨ Receive restart flag 26 ⟩ ≡

```
save_restart_flag = rcv_save_restart_flag (parent_process);
⟨ Return if error 33 ⟩;
```

This code is used in chunk 25.

27¶ The information to be received is:

- flags driving the subsequent execution,
- the next requested time,
- the boundary conditions and trips values for the next simulation steps, and
- *iteration_flag*.

The last flag tells the code to throw away the computation of the last advancement and restore the simulation to the last request form the parent process. Because of the way it is coded, this is done restoring the old *state_vector* which was saved in *timestep_state_vector*[1].

27⟨ Receive additional information 27 ⟩ ≡

```
continue_simulation = rcv_continue_simulation ( );
if (continue_simulation) {
    rcv_requested_time (parent_process);
    ⟨ Return if error 33 ⟩;
    rcv_boundary_conditions (parent_process);
    ⟨ Return if error 33 ⟩;
    rcv_trips (parent_process);
    ⟨ Return if error 33 ⟩;
    rcv_iteration_flag (parent_process);
    ⟨ Return if error 33 ⟩;
    if (iteration_ flag) state vector = timestep state vector;
}
```

This code is used in chunk 25.

28¶ The results requested by the parent process are sent as obtained in the last time step. For the tree simulation, the code standard output procedure cannot be used, since the tree may be built in a distributed environment. All the outputs from all branches of the tree will be sent to a central database that will reconstruct the tree results.

28⟨ Send results 28 ⟩ ≡

```
if (parent_process) {
    send_time (parent_process);
    ⟨ Return if error 33 ⟩;
    send_outputs (parent_process);
    ⟨ Return if error 33 ⟩;
    send_out_trips (parent_process);
    ⟨ Return if error 33 ⟩;
    ⟨ Database messages 32 ⟩;
}
```

This code is used in chunk 14.

[1]Note here that it is possible that more information has to be restored, especially the time step value. Note also that a similar 'restart form the previous correct value' takes place when a step calculation is rejected and a new one is tried with a smaller timestep.

C.1.2.2.2. Code for Tree Simulation

Restart information is to be saved into the private stack for later use. This information may be sent to initialise other processes or this very process, if the parent process sends the appropriate flag for entering < *nitialisation* **for** *tree simulation.*

```
29⟨ Manage restart stack 29 ⟩ ≡
    if (save_restart_flag ≡ UpdateRestart) {
        int rcverror, restart_index;
        restart_index = rcv_restart_index (parent_process);
        ⟨ Return if error 33 ⟩;
        update_restart (restart_index);
        ⟨ Return if error 33 ⟩;
    }
```

This code is cited in chunk 21.

This code is used in chunk 21.

30¶ Restart information is to be sent to a remote process. The send routine will return an error if sending the tables to the process itself, so this section must be skipped by the sender process if the restart is from a point saved in < *Manage restart....*

```
30⟨ Probe send-restart message 30 ⟩ ≡
    arrived_message = pvm_probe (parent_process);
    ⟨ Return if error 33 ⟩;
    if (arrived_message) {
        branch_info = rcv_branch_info (parent_process);
        ⟨ Return if error 33 ⟩;
        remote_process = rcv_remote_process (parent_process);
        ⟨ Return if error 33 ⟩;
        if (remote_process) {
            send_tables (branch_info, remote_process);
            ⟨ Return if error 33 ⟩;
        }
        ⟨ Free used restart point 31 ⟩;
    }
```

This code is used in chunk 25.

31¶ If the parent process so demands, the point just used is freed from the private stack. The restart information has to be kept until all branches stemming form a single point are started. The parent process manages this via *free_point.*

31⟨ Free used restart point 31 ⟩ ≡

```
free_point = rcv_free_point (parent_process); ⟨ Return if error 33 ⟩
if (free_point) {
    remove_point_from_stack (free_point);
    ⟨ Return if error 33 ⟩;
}
```

This code is used in chunks 13 and 30.

32¶ When running a tree simulation, the outputs must be sent to a centralised database that collects the results of each individual simulation step in the tree.

32⟨ Database messages 32 ⟩ ≡

```
if (database_id) {
    send_results (database_id);
    ⟨ Return if error 33 ⟩;
}
```

This code is used in chunk 28.

33¶ 2 Error handling. In case the code returned by the receiving routines is less than zero it means that an error occurred in the transmission. The error code is returned to the *main* program.

33⟨ Return if error 33 ⟩ ≡

```
if (somerror < 0) report_and_return (somerror);
```

This code is used in chunks 10, 13, 23, 24, 25, 26, 27, 28, 29, 30, 31 and 32.

34¶

Appendix D. BABIECA input file

```
Test of modules SNDCODE and RCVCODE
400000 0.001 0.0
1
*****************************************
Activation flag
   1
   0
   15
*- - - - - - - -
   2
   -10.0  1000.0
   1.0       1.0
*****************************************
*****************************************
Mainpipe emission
   100
   1  511  611  711  512  612  712  513  613  713
*     ^- - - - - - - -- - - - -     boundary conditions
*  ^- - - - - - - -- - - - - -    activation flag
   60
*- - - - - - - --
   1
*  ^- - - - - - - -- - - - - execution mode
   spp2k
*     ^- - - - - - - - - - remote host
relap5.x -i mainpipe.i -o mainpipe.o -r mainpipe.r
*  ^- - - -  executable name
   1
*  ^- - - - - - - -- - - - - null transient time steps
   9
*  ^- - - - - - - -- - - - - number of boundary conditions
   velfj-206
   velfj-306
   velfj-406
   tsatt-205
   tsatt-305
```

```
   tsatt-405
   p-200
   p-300
   p-400
*     ^-- -- - - - - - - name of the boundary conditions
   0
*  ^- - - - - - - -- - - number of initial conditions
************************************************
************************************************
Mainpipe reception
   111
   1  100
*     ^- - - - - - - -- - - - - - - - remote code tid
*  ^- - - - - - - -- - - - - - - -  activation flag
   61
*- - - - - - - -- - - - -
   6
*  ^- - - - - - - -- - - - - - - -  number of output signals
*                                   received from the remote code
   velfj-600020000
   velfj-600030000
   velfj-600040000
   tempf-600010000
   p-500010000
   velfj-100050000
*     ^- - - - - - - -- - - - - - -  names of output variables
************************************************
************************************************
Loop1 emission
   200
   1  111  114  115
*     ^- - - - - - - -- - - - - - - - - boundary conditions
*  ^- - - - - - - -- - - - - - - - - - activation flag
   60
*- - - - - - - -
   1
*  ^- - - - - - - -- - - - - - - -  execution mode
   spp2k
```

```
*      ^- - - - - - - - -- - - - - remote host
relap5.x -i loop1.i -o loop1.o -r loop1.r
*      ^- - - executable name
   1
*   ^- - - - - - - - -- - - - - - -  null transient time steps
   3
*   ^- - - - - - - - -- - - - - - -  number of boundary conditions
   velfj-601
   tsatt-600
   p-500
*      ^- - - - - - - - -- - - - - - - name of the boundary conditions
   0
*   ^- - - - - - - - -- - - - - - -  number of initial conditions
************************************************
************************************************
Loop1 reception
   511
   1  200
*      ^- - - - - - - - -- - - - - -  remote code tid
*   ^- - - - - - - - -- - - - - - - - activation flag
   61  0  10.1409  600.639  1.52511e+07
*- - - - - - - - -- - - - - - -
   3
*   ^- - - - - - - - -- - - - - - number of output signals
*                   received from the remote code
   velfj-501000000
   tempf-205060000
   p-200010000
*      ^- - - - - - - - -- - - -   names of output variables
************************************************
************************************************
Loop2 emission
   300
   1  112  114  115
*      ^- - - - - - - - -- - - - - -  boundary conditions
*   ^- - - - - - - - -- - - - - - - - activation flag
   60
*- - - - - - - - -
```

```
   1
*  ^- - - - - - - - -- - - - - - - -  execution mode
   spp2k
*     ^- - - - - - - - -- - - -   remote host
relap5.x -i loop2.i -o loop2.o -r loop2.r
*         ^- - - - executable name
   1
*  ^- - - - - - - - -- - - - - - - -  null transient time steps
   3
*  ^- - - - - - - - -- - - - - - - -  number of boundary conditions
   velfj-601
   tsatt-600
   p-500
*     ^- - - - - - - - -- - - - - -  name of the boundary conditions
   0
*  ^- - - - - - - - -- - - - - - - - number of initial conditions
************************************************
************************************************
Loop2 reception
   611
   1  300
*     ^- - - - - - - - -- - - - - -  remote code tid
*  ^- - - - - - - - - - - - - - - -  activation flag
   61  0  10.1409  600.639  1.52511e+07
*- - - - - - - - -
   3
*  ^- - - - - - - - - - - - - - - -   number of output signals
*                    received from the remote code
   velfj-501000000
   tempf-205060000
   p-200010000
*     ^- - - - - - - - -- - - -   names of output variables
************************************************
************************************************
Loop3 emission
   400
   1  113  114  115
*     ^- - - - - - - - -- - - -   boundary conditions
```

```
*   ^- - - - - - - - -- - - - - - - -  activation flag
    60
*- - - - - - - - -
    1
*   ^- - - - - - - - -- - - - - - - -  execution mode
    spp2k
*       ^- - - - - - - - -- - - -   remote host
relap5.x -i loop3.i -o loop3.o -r loop3.r
*       ^- - - executable name
    1
*   ^- - - - - - - - -- - - - - - - -  null transient time steps
    3
*   ^- - - - - - - - -- - - - - - - -  number of boundary conditions
    velfj-601
    tsatt-600
    p-500
*       ^- - - - - - - - -- - - - - -  name of the boundary conditions
    0
*   ^- - - - - - - - -- - - - - - - - number of initial conditions
************************************************
************************************************
Loop3 reception
    711
    1  400
*       ^- - - - - - - - -- - - - - -  remote code tid
*   ^- - - - - - - - - - - - - - - - -  activation flag
    61  0  10.1409  600.639  1.52511e+07
*- - - - - - - - -
    3
*   ^- - - - - - - - - - - - - - - -   number of output signals
*                                      received from the remote code
    velfj-501000000
    tempf-205060000
    p-200010000
*       ^- - - - - - - - -- - - -   names of output variables
************************************************
************************************************
```

A Standardized Methodology for the Linkage of Computer Codes. Application to RELAP5 / Mod3.2

Colección
Otros Documentos CSN

CSN CONSEJO DE SEGURIDAD NUCLEAR