Object-Oriented Parallel Programming: Library Design and Development for SPMD Programming

Jean_Marc Adamo
International Computer Science Institute
Suite 600, Berkeley, CA 94704-1105

Technical Report

Abstract.

In the process of writing portable parallel applications, one particular way of viewing the parallel programming activity is as an application-centered one. This paper reports on the object-oriented design of a library supporting such an approach. The library has been developed within C++ and implemented on the CM5. The code has been carefully written so that the library could easily be ported to any MIMD parallel machine supporting C++. The library allows parallel program development in the SPMD style. It has been designed so that the compiler can perform a complete type checking of user programs. This was a major requirement: we wanted the library to provide facilities close to those one normally expects from a programming language (i.e., with compiled programming primitives). we was actually interested in checking how far it would be possible to go toward achieving a programming tool, that provides facilities, close to those offered by a language, via the natural object-oriented extension mechanisms available in C++. This report brings evidence that such a goal is quite achievable. The library consists of a set of four layers providing: threads, processes, process synchronization and synchronous message passing, remote read/write facilities, and spread arrays and pointers

Introduction.

In the process of writing portable parallel applications, one particular way of viewing the parallel programming activity is as an application-centered one. Once defined, an application domain allows you to define the set of basic operators and functions related to what the application domain basically is and to what the basic processes in the application domain do. The subsequent development of specific applications in the domain can be carried out upon this set of operators and functions regardless of how they are implemented. The linear algebra domain [6], for instance, provides a good example of what an application domain is, and what the set of basic operators and functions can be (other domains involve more irregular distributed data structures). In the application domain oriented view, the parallel programming activity can be seen as being split into two parts: that of the development programmer and that of the application programmer. Given a particular parallel machine and a particular application domain, the job of the former is to provide an optimal and scalable implementation of the basic domain operators and functions up to the point from which the machine architecture is no longer visible. In order to achieve optimality, the development programmer should take advantage of the low level facilities provided by the hardware and the basic software as much as possible (low level features of the processing units, network features, threading possibilities, etc.). Thereafter, the application programmer can develop particular applications upon the set of basic functions provided to him that speak the language of the application domain and ensure machine transparency.

In such a view, the application programmer is no longer concerned with application portability. The issue is shifted to the development programmer level. For an application to be ported from a machine to another, the development programmer needs to rewrite all the basics of the concerned application domain from which the low level features of the machine are visible. Here, we assume that the basics have been cleanly developed as a hierarchy, so that only the levels of the hierarchy from which the machine is visible have to be rewritten. In fact, this view is not new and just follows what is currently done when porting sequential applications from a (sequential) machine to another. For a program written in a given language to be ported, the compiler available on the first machine must also be available (must have been rewritten) for the second. Rewriting machine-dependent basic functions for each specific parallel machine just corresponds to rewriting the machine-dependent part of a compiler from a sequential machine to another.

The object-oriented programming technology exactly fits the needs of the above presented methodology for parallel application development and porting. The purpose of object-oriented languages is to provide the suitable set of built-in facilities that will enable you to extend the set of basic operators available in the underlying algorithmic language. Such facilities can then be utilized to fill in the gap separating them to those related to the application domain at hand. We suggest that, in the parallel programming case, the basic algorithmic language should be augmented with additional threading facilities. This is the most natural and flexible means that should be provided for processes (pieces of local code) to be appropriately scheduled so that computation could be optimally overlapped with communication. This is even true for machines based on processors for which context switching is an expensive operation, as it is the case for CM5 on which the present library has been implemented. In this case it is up to the programmer to choose a sensible light-weight process grain so that the cost of context switching is kept acceptable.

This paper reports on the design of a library that can be used to develop parallel applications according to the methodology described above. The library has been developed within C++ and implemented on the CM5. The code has been carefully written so that the library could easily be ported to any MIMD parallel machine supporting C++. The small set of machine dependent functions (context-switching, process-stack initialization, communication-handling routines) has been carefully separated from the rest of the code so that possible porting could easily be carried out by only rewriting them quite independently. The library allows parallel program development in the SPMD style (i.e., a program consists of a single code copy that is executed with different data on the processors of a parallel machine). It has been designed so that the compiler can perform a complete type checking of programs written with it. This was a major requirement: we actually wanted the library to provide facilities, close to those one normally expects from a programming language (i.e., with compiled programming primitives). We were actually interested in checking how far it would be possible to go toward achieving a programming tool, that provides facilities close to those offered by a language, via the natural object-oriented extension mechanisms available in C++. This report brings evidence that such a goal is quite achievable.

The library consists of a set of layers as shown in Fig. 1. The most basic objects in the library are threads. Threads are rather unstructured parallel programming constructs. A thread executes in an independent stack and can be subject to a set of operators that perform: creation, initialization, scheduling, suspension with or without rescheduling. However threads are rather unstructured, hence unsound, constructs although they offer high flexibility that can prove useful in various applications. Threads are used in the construction of more disciplined objects: processes. A process is a thread that can only be handled within a special "par" function . This function automatically handles process creation, initialization and synchronized termination. Processes meet the semantics of OCCAM processes [9] with one exception: processes are allowed to read from or write to shared variables.

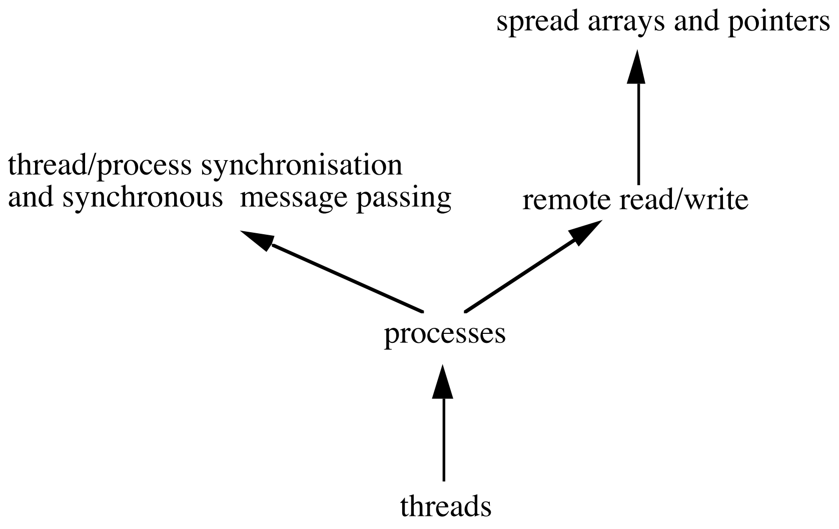
spread arrays and pointers
thread/process synchronisation and synchronous message passing
remote read/write
processes
threads

Fig.1 Architecture of the library

The third layer of the library aims at providing tools for synchronization and communication. Processes that are run on the same or remote processors can synchronize by applying pairs of matching functions (send, recv) on a common data object called a token. Synchronization may give rise to message transfer from sender to receiver. Processes can also read from or write to a remote memory by using remote read/write functions specially designed to this purpose. Using such functions could prove preferable in some situations, for instance, when most processors need simultaneously to read from or write to a remote memory. This can be much more efficiently performed by using remote read and write functions on a synchronization barrier than point-to-point synchronized communication (as far as fast synchronization barrier mechanisms happen to be available on the machine at hand). Once again, remote read/write functions appear as rather unstructured programming primitives and it was appealing to go beyond and propose more structured constructs. This is what is achieved via spread arrays and pointers. A spread array [4, 5] is a multidimensional array that has some of its dimensions spread over the set of local memories. Spread pointers generalize usual pointers and allow the processes to access the data items in spread arrays transparently.

The present paper aims at providing both accurate ideas about how the library has been designed and sufficient information for the library to be effectively used. Each section is completed with a sample program that exemplifies the programming material described in the section. The code of the .h files is provided in appendices which give all relevant details about the design and about how each programming construct can effectively be used as well.

1. Threads, Processes and basic processing facilities.

1.1 Threads.

Threads are the most basic objects provided by the library (see Appendix 3). A thread may be a user thread or a predefined special one, called *main_thread*. Threads are instances of the *Thread_base* class which provides the minimal data structure required for a thread to be descheduled and resumed, namely, the current values of the stack pointer and program counter (the registers are saved on top of the stack) . Thread_base additionally provides a basic function *next_thread()* that performs context switching. Actually, two instances of this function, overloading the same name, are provided that perform context switching under protection against the interrupts or not. Most function names in the library are overloaded this way which provides a very convenient technique for the control of protection. However, the final user is not expected to know about the existence of the two instances. Only the protected one (the simplest: no additional integer parameter) is to be used at his level.

Thread_base has two derived classes: *Main_thread* and *Thread*. The first one is a predefined class intended to be instantiated only once. It provides two functions: *Main_thread::start_keep_up_and_terminate()* and *Main_thread::exit()*. The start_keep_up_and_terminate function is executed by the C++ main function to start the scheduling machinery. It is next handled as a special thread that is run each time there is no user thread in the scheduling queue. This occurs on the nodes when the resident processes need an external event to occur (termination of a communication) to resume execution. start_keep_up_and_terminate() is then kept executing until the queue becomes non empty: the queue is filled in under interrupt by a handler routine. The start_keep_up_and_terminate thread is used to keep the scheduling machinery up. It does not need any particular additional data to perform as it makes use of the main stack (the stack implicitly associated with the C++ program). The Main_thread::exit() function is intended to cause the scheduling machinery termination. It is executed in the root thread (starting thread of a user program) to cause return to the main function.

Neither Main_thread::start_keep_up_and_terminate() nor Main_thread::exit() needs to be explicitly handled by the final user as long as he uses the predefined programming framework defined in the *environment* file (see appendix 2). This file contains a macro that defines the standard main() function. The macro is parametrized by the identifiers of the root user_process and the class it belongs to. This class must be derived from the predefined class Process (described later on). The predefined main() function allocates all the data structures needed for the programming system to work (scheduler data, etc.), instantiates the user root-process from the parameters provided to the macro, schedules this process (which is intended to spawn subsequently), and executes the start_keep_up_and_terminate() function. As a Process can handle return to their father process on their own (or the start_keep_up_and_terminate thread if the object is the user root-process), termination of the user root-process will cause the automatic execution of the

Main_thread::exit() function. In turn, that will cause switching to the start_keep_up_and_terminate thread with a special status which will finally make the thread return from the main function, causing the node termination.

The Thread class is the base class for all user-defined threads. Threads need additional data to execute in relation to their execution stack and their entry-point, namely: stack space, stack size, stack pointer, and entry-point pointer. As opposed to tasks provided by the standard C++ library [1], a thread is not the class constructor. A thread can be any function in a class derived from the Thread base. This allows separating thread creation from thread initialization and execution. This also allows more flexibility in the sense that different Thread objects can be created from the same class with different entry points and the Thread class can be derived more than once. The Thread class provides a small set of functions:

Thread::Thread(membfunc_ptr entry_pt, int stack_size) is the constructor that essentially performs: stack allocation, stack size storage, entry point storage and from-sons-to-father thread linking (for the sons to get access to the data members of their father).

void Thread::initialise() loads the initial stack frames and stores the current stack pointer and initial program counter (entry point stored in the constructor). This function calls machine dependent routines.

void Thread::schedule() ,
void Thread::schedule(int) append the pointer to the thread it is executed for to the scheduling queue (cooperative multitasking).

*void Thread::schedule(Thread **proc_table)* ,
*void Thread::schedule(Thread **proc_table, int)* apply a similar function to thread lists.

void Thread::reschedule() ,
void Thread::reschedule(int) stop the currently running thread (see the next function) and append its pointer to the scheduling queue.

void Thread::stop() ,
void Thread::stop(int) simply stop the currently running process and cause switching to the first one available in the scheduling list or to Main_thread::start_keep_up_and_terminate() when the scheduling list happens to be empty.

All these scheduling functions perform on a data structure *Sched_data* (a queue with two pointers and two integers that allows queue manipulation) that is defined as a separate class instantiated once (see Appendix 2, 3).

1.2 Processes.

Threads are very low level constructs. Processes are higher level, but also much more constrained programming objects (see Appendix 3). Processes are restricted threads, hence the private derivation from Thread to Process. A process is a thread whose scheduling is totally handled via the use of the *par* function. The termination of the set of processes launched by the par function is also automatically handled by the implicit execution of a special function: *exit()*. Used in conjunction with the communication-synchronization primitives described in section 2, Processes allow one to write programs in the OCCAM programming style [9]. The process class provides the following functions:

Process::Process(membfunc_ptr entry_pt, int stack_size=PROC_STACK_SIZE) is the constructor that simply invokes the Thread constructor.

void Process::initialise() redefines the Thread::initialise() function so that, the initialized process can automatically call Process::exit() upon termination. This function handles automatic termination of the par function and automatic return to father or to Main_thread::start_keep_up_and_terminate() via a call to Main_thread::exit(). The Process::initialise() function calls machine dependent routines.

void Process::reschedule() ,
void Process:reschedule(int) are exactly those defined in the private base Thread whose access level through Process is maintained by explicit declaration.

*void Process::par(Process **proc_table)* launches the sub-processes, whose pointers are held in proc_table, and stops the current process or thread. The sub-process of the list, that terminates last, reschedules this process/thread making it return from the par function.

void Process::exit() is automatically (i.e. need not be called by the final user) executed at the termination point of processes. It allows the synchronization of sub-process termination. It performs the scheduling of the father process or switches to Main_process::start_keep_up_and_terminate whenever the exited process is the root process.

1.3. Sample program.

The program below does not perform anything really interesting. It is only intended to provide possible use patterns to potential users of the library. The code can be run on the CM5. Each node executes root_proc with creates and runs two Proc instances that respectively run two Proc instances in turn. The *environment* file (see appendix 2) includes the library and a standard macro: *MAIN* that defines the standard main() function. This function creates the scheduling data structures, starts the scheduler and schedules root_proc, of the Root_proc process class, as initial process. SelectPrintf is a macro that allows you to select the nodes that are permitted to print out .

```
#include "environment"

//user program

class Proc: public Process{

  public:
        int pid;
        int step;

        Proc(int p, int s)
                : Process((membfunc_ptr)&Proc::body){pid = p; step = s;}
```

```
          void body(){
                    SelectPrintf("on node %d, in proc %d before loop at step %d\n",
                              self_address, pid, step);
                    int i;
                    for(i = 0; i < 20; i++){
                      SelectPrintf("on node %d, proc %d in for loop, i = %d\n",
                                self_address, pid, i);
                      reschedule();
                    }
                    SelectPrintf("on node %d, in proc %d after for loop at step %d\n",
                              self_address, pid, step);
                    if(step<1){
                      Proc *proc1 = new Proc(pid*10 + 1, step+1);
                      Proc *proc2 = new Proc(pid*10 + 2, step+1);
                      Process *proc_tabl[3] = {proc1, proc2, 0};
                      par(proc_tabl);
                      delete proc1;
                      delete proc2;
                    }
                    SelectPrintf("on node %d, proc %d the end at step %d\n",
                              self_address, pid, step);
          }
};

class Root_proc: public Process{

  public:

          Root_proc()
                    : Process((membfunc_ptr)&Root_proc::body){}

          void body(){
                    SelectPrintf("on node %d, in Root_proc::body1\n", self_address);
                    Proc *proc1 = new Proc(1, 0);
                    Proc *proc2 = new Proc(2, 0);
                    SelectPrintf("on node %d, in Root_proc::body2\n", self_address);
                    Process *proc_tabl[3] = {proc1, proc2, 0};
                    par(proc_tabl);
                    SelectPrintf("on node %d, in Root_proc::body4\n", self_address);
                    delete proc1;
                    delete proc2;
          }
};
//end of user program

MAIN(root_proc, Root_proc)
```

2. Threads/processes synchronization and communication.

So far, we have described the set of classes and functions that allow thread/process creation, scheduling and termination. The current section is devoted to describing the set of additional classes and functions required for thread/process synchronization and communication (Appendix 4). Thread/process synchronization and communication is performed via the execution of matching pairs of send-recv functions that are applied to common tokens. Tokens are similar to OCCAM channels [9] but they apply both to intra- and inter-process communication. As in this language, communication is point-to-point and synchronous. The tokens and the send and recv functions are defined in such a way that the C++ compiler is able to check the type consistency. Point-to-point communication is ensured by the atomic execution of Synchronization/communication functions. Unmatching pairs of send-send or recv-recv statement are detected dynamically. This seems, at first, to be rather inefficient since additional code has to be run for checking unmatching pairs of communication statements, but can be easily handled by providing two versions of the library. One that checks possible inconsistencies or errors at run time (such as unmatching communication statements, heap overflow, etc.) and a second one, that does not check anything, hence runs faster, to be used with already debugged programs.

2.1 Synchronization-communication token.

There are four possible token classes derived from one another according to Fig. 2.

***Token* class.**

This class corresponds to the minimal data structure required for process synchronization. Tokens of this class can only be used for local, pure, point-to-point synchronization. The Token data structure contains three members. The first one is a process pointer loaded by the process that requested a synchronization and is waiting for it. When the token is involved in an *alt* operation (see section 2.3.2), the second one holds the pointer to the control data-structure used by the function executing the alt function, and the third is used to keep track of the rank of the token among those involved in the alt operation (see section 2.3). Pointers to the tokens are gathered in a table whose pointer is member of the control data-structure.

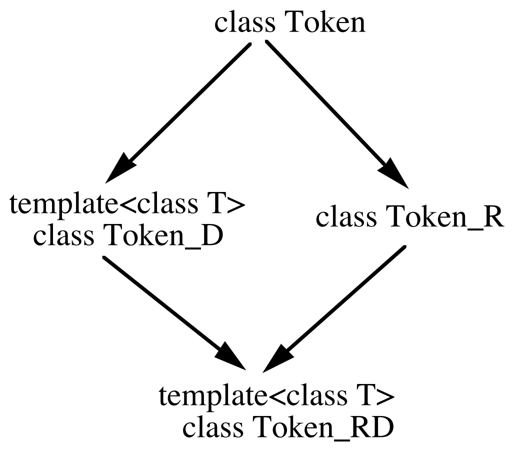

Fig. 2. Synchronization tokens

***Token_D* class.**

These tokens should be used for local point-to-point synchronization plus data transfer from sender to receiver. A Token_D holds three additional data items: the size of the data item transferred and two pointers respectively to the source and the destination of data transfer. This is a generic class that is parametrized by the type of data items transferred from sender to receiver.

Token_R **class.**

Processes that are run on remote processors can synchronize as well. This is achieved via the use of a Token_R token. As we are concerned with SPMD programming (same code image is run on each processor), each Token_R, once defined, appears on each processor (but contains different data values). Information about the location of Token_R objects is global. Given a Token_R t, each node knows the location of t on the other processors from the local address of t as it is exactly the same on all the processors. Same location everywhere is achieved on the CM5 implementation by making allocation and disallocation of Token_R's synchronous over the nodes (CMMD_sync_with_nodes() is used in the Token_R constructor and destructor) and by performing Token_R allocations in a special heap (see *token_R_heap* and the *TOKEN_R_HEAP_SIZE* - which has to be properly set- in the environment file). Such a solution is a simple but also a very efficient one that should be sufficient for our purposes that deals with algorithm implementation. It avoids the need of executing an expensive protocol to ensure that all the required Token_R's are created before the remote synchronization functions that use them are executed. Token_R's should be placed on global synchronization barriers (no request in progress on the node or via the network) to avoid any deadlock problem. A Token_R contains new data items for remote synchronization handling. Given a triple of nodes n1, n2 and n3, the same Token_R is used on n2 both by a send statement (to send a synchronization request to n3) and by a recv statement (to reply to a synchronization request received from n1). So the enabled_ptr field must be replicated and the Token_R needs to hold both addresses of processors from/to which it can recv/send synchronization signals.

The protocol for remote synchronization is based on "master-slave" behaviors followed by the processes that want to synchronize via a Token_R. Only that executing send is entitled to send the synchronization signal to the remote processor. A process run on this processor is expected to execute the matching recv eventually. The signal reception on the remote node is handled by a procedure (interrupt handler) that locally takes care of the meeting request on the behalf of the remote sender-process. An acknowledgment is eventually sent back when the matching local recv is performed. In the CM5 implementation, the protocol has been developed via the use of the CMAML active message passing library[13, 14]. All phases of the protocol are processed by interrupt handlers and send and recv are suspensive functions. Resumption (i.e. scheduling) of the synchronized processes is performed by the handlers processing the termination of the protocol on both sides (more details are given in the description of the recv(Token_R *) and send(Token_R *) functions below).

Token_RD **class.**

This token generic class allows the synchronization of processes located on remote processors and the transfer of a data item of some type from sender to receiver. The synchronization-communication protocol follows the same "master-slave" protocol as that described above. As Token_R's, these new tokens are also constructed synchronously over the nodes of the multiprocessor.

Token_D and Token_RD are template classes parametrized by the type of data transferred when synchronization occurs. This allows the C++ compiler to check the type consistency of pairs of recv-send instructions that are applied to tokens of these classes.

2.2 Synchronization-communication functions.

There is a pair of matching send-recv functions corresponding to each of the token types described above.

*void send(Token *token_ptr),*
*void send(Token *token_ptr, int).* These functions check the token passed as an argument. If there is a process waiting on the token (the process executed a recv statement and stopped just after enabling the token, i.e., placing its own pointer into the enabled_ptr member), then the process is scheduled, and the function continues execution. If not, the function enables the token and stops. The unprotected *send* instance is identified by the presence of an additional *int* dummy parameter.

*void recv(Token *token_ptr),*
*void recv(Token *token_ptr, int).* This function performs symmetrically.

template<class T> void send(Token_D token_ptr, T *from_ptr), t*
*emplate<class T> void send(Token _D*token_ptr, T *from_ptr, int).* As far as synchronization is concerned, this function performs as the previous one. Additionally, if the token is enabled and the to_ptr pointer is set, a T-type data item is transferred from *from_ptr to *to_ptr. If the token is not enabled, from_ptr is loaded in the from_ptr member of the token. If the token is enabled but the to_ptr member is not set, this causes a matching error.

template<class T> void recv(Token_D token_ptr, T *from_ptr),*
*template<class T> void recv(Token _D*token_ptr, T *from_ptr, int).* These functions perform symmetrically.

*void send(Token_R *token_ptr), void send(Token _R*token_ptr, int).* These functions enable the token to which it is applied. The token becomes next unavailable for any new send execution till it is disabled. Then, a request is sent to the remote node (whose address has been set at token creation time) before the process is stopped. The process executing this function will be resumed (scheduled) by the interrupt handler that processes termination of the whole send-recv transaction on the sender side. The send request is handled on the receiver side by an interrupt handler which checks the corresponding token on this side (same memory location as already explained). If the token happened to be enabled, then the receiver process is already currently waiting on the token. So, it is scheduled for further execution and a signal is sent to the sender node. The signal is processed by an interrupt handler that schedules the sender process for further execution and disables the token that can be used for a subsequent transaction. If the token happened not to be enabled, the handler enables it and loads in the token all information needed for the subsequent steps of the protocol already described to be processed later by the receiver process.

*void recv(Token_R *token_ptr),*
*void recv(Token _R*token_ptr, int)*. Performs symmetrically.

template<class T> void send(Token_RD token_ptr, T *from_ptr),*
*template<class T> void send(Token _RD*token_ptr, T *from_ptr, int)*. This function works as the RD-type one regarding the protocol philosophy. Of course, the protocol has now two more steps required by the data transfer. The function also works as the D-type one regarding data transfer.

2.3 Non deterministic synchronization-communication functions.

The send instructions can also be performed in conjunction with a non-deterministic receive function (see Appendix 4). This function, denoted as *alt* from now on, can possibly receive a synchronization (+ communication) request via each the tokens (t_i) held in a list which can be accessed by alt via its calling parameter. The list can be a mix of tokens of any types described previously. A particular function (f_i) is attached to each token. Basically, the alt function follows the OCCAM semantics and works as follows. When alt is executed, if none of the token happened to be enabled, the alt enables all the token in the list before stopping the process which invoked it. The process will be resumed (scheduled) by the first process executing a send on one of the tokens in the list. When this happens, all the tokens are disabled. If one or several tokens are enabled, alt chooses the first one (say t_i) in the list, schedules the process waiting on the token, disables the token (only this one), and calls the corresponding function f_i just before returning.

The behavior of the previously described alt function can be unfair in the sense that the same token is always selected and excludes indefinitely others' selection (e.g., the first token in the list happens to be enabled each time alt is executed). The library provides another function: *alt_fair* which has a fair behavior at the expense of some additional code (hence performance loss). The alt functions need the definition of a few data structures that are described below.

2.3.1 Alt branches.

There are four types of alt branch that derive from one another according to the scheme pictured in Fig. 3 that is in exact correspondence with the Token one (see Appendix 4).

An Altbranch data object is intended to gather the set of objects that defines an alt branch. Namely: a token (with similar generic type, if any), a thread member-function (which is the function to be executed when this alt branch is selected), the parameter list of this function (if any) and the destination of the data transfer in the case of AltBranch_recv or AltBranch_Rrecv.

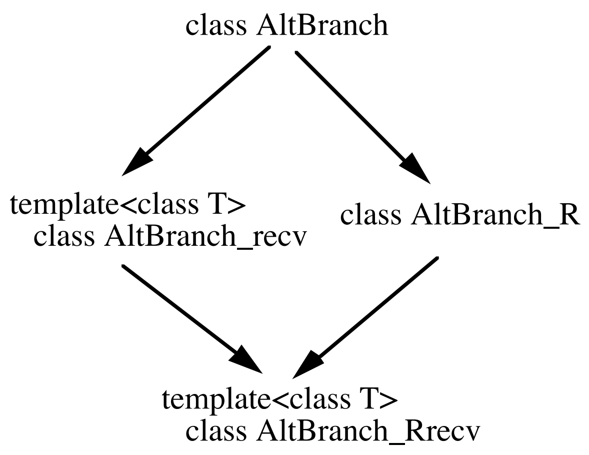

Fig. 3 possible alt branches

2.3.2 Alt control data structures.

Each alt is executed on a data stucture which is an instance of the *AltCtrl_struct* class. Each AltCtrl_struct holds a pointer to an AltBranch table and a small set of integer control variables: table size, index from which the table is to be scanned in the case of alt_fair function, and rank of the selected branch which is set at selection time and used as first calling parameter of member functions above mentioned. Note that the rank parameter is assumed to be explicitly declared, by the user, as first parameter of any function intended to be used in an alt branch.

2.3.3. Alt-functions' profile.

*void alt(AltCtrl_struct *altCtrl_struct);*

*void alt(AltCtrl_struct *altCtrl_struct , int);*

*void alt_fair(AltCtrl_struct *altCtrl_struct);*

*void alt_fair(AltCtrl_struct *altCtrl_struct , int);*

2.4. Sample program.

As the one given in the previous section, the program below does not perform anything really interesting. The intention is to provide possible use patterns to potential users of the library. The code can be run on the CM5. The process creation structure is similar to the previous one. There is an additional process Sync which the Proc processes synchronize with. The first two level processes synchronize with and send a data aggregate to the Sync process respectively run on the left or the right processor. The Sync process replies via the alt statement. The second four level processes also synchronize with the Sync process but locally.

```
#include "environment"
```

```
//user program

Token_D<int> *token[4];

class Id{
   public:
            int processor_Id;
            int process_Id;
};

Token_RD<Id> *proc_to_left;
Token_RD<Id> *proc_to_right;


class Proc: public Process{

   public:
            int pid;
            int step;
            Id id;

            Proc(int p, int s): Process((membfunc_ptr)&Proc::body){
                        pid = p;
                        step = s;
                        id.processor_Id = self_address;
                        id.process_Id   = pid;
            }

            void body(){

                        SelectPrintf("on node %d, in proc %d before loop at step %d\n",
                                          self_address, pid, step);
                        int i;
                        for(i = 0; i < 20; i++){
                           SelectPrintf("on node %d, proc %d in for loop, i = %d\n",
                                          self_address, pid, i);
                           reschedule();
                        }
                        SelectPrintf("on node %d, in proc %d after for loop at step %d\n",
                                          self_address, pid, step);
                        if(step==0 && pid==1 && self_address>0){
                               send(proc_to_left, &id);
                               SelectPrintf("on node %d, in proc %d, after send left, at step %d\n",
                                                 self_address, pid, step);
                        }
                        if(step==0 && pid==2 && self_address<31){
                            send(proc_to_right, &id);
                            SelectPrintf("on node %d, in proc %d after send right at step %d\n",
                                             self_address, pid, step);
                        }
                        int index;
                        if(step==1){
                           if( ((Proc *)father_ptr)->pid == 1 )
                              index = pid%10 - 1;
                           else
                        index = pid%20 + 1;
                    send(token[index], &pid);
                    SelectPrintf("on node %d, in proc %d after send at step %d, index = %d,
                                    data sent: %d\n", self_address, pid, step, index, pid);
                    }
                    if(step<1){
                      Proc *proc1 = new Proc(pid*10 + 1, step+1);
                      Proc *proc2 = new Proc(pid*10 + 2, step+1);
                      Process *proc_tabl[3] = {proc1, proc2, 0};
                      par(proc_tabl);
                      delete proc1, delete proc2;
                    }
                    SelectPrintf("on node %d, proc %d the end at step %d\n",
                                    self_address, pid, step);
          }
};

class Sync: public Process{

  public:
          int pid[4];
          Id id_left, id_right;

          Sync()
            : Process((membfunc_ptr)&Sync::body){}

          void body(){
                    SelectPrintf("on node %d, in Sync::body1\n", self_address);
                    if(self_address > 0 && self_address < 31){
                       AltBranch
                       *alt_table2[2] =
                              {new AltBranch_Rrecv<Id>(proc_to_left,
                                                              (membfunc_ptr)&Sync::alt_body2,
                                                              0,
                                                              &id_left),
                               new AltBranch_Rrecv<Id>(proc_to_right,
                                                              (membfunc_ptr)&Sync::alt_body2,
                                                              0,
                                                              &id_right)
                              };
                      AltCtrl_struct *alt_ctrl2= new AltCtrl_struct(2, alt_table2);
                      for(int i = 0; i< 2; i++){
                        alt(alt_ctrl2);
                      }
                      SelectPrintf("on node %d, sync after for 2\n", self_address);
                    }
                    if (self_address == 0){
                      recv(proc_to_left, &id_left);
                      SelectPrintf("on node %d, sync after recv from proc_to_left,
                                       id_left.processor_Id = %d, id_left.process_Id = %d\n",
                                       self_address, id_left.processor_Id, id_left.process_Id);

                    }
                    if (self_address == 31){
                      recv(proc_to_right, &id_right);
                      SelectPrintf("on node %d, sync after recv from proc_to_right,
                                  id_right.processor_Id = %d, id_right.process_Id = %d\n",
                                  self_address, id_right.processor_Id, id_right.process_Id);
                }
                AltBranch
                *alt_table4[4] =
                        {new AltBranch_recv<int>(token[0],
                                                    (membfunc_ptr)&Sync::alt_body4,
                                                    0,
                                                    &pid[0]),
                         new AltBranch_recv<int>(token[1],
                                                    (membfunc_ptr)&Sync::alt_body4,
                                                    0,
                                                    &pid[1]),
                         new AltBranch_recv<int>(token[2],
                                                    (membfunc_ptr)&Sync::alt_body4,
                                                    0,
                                                    &pid[2]),
                         new AltBranch_recv<int>(token[3],
                                                    (membfunc_ptr)&Sync::alt_body4,
                                                    0,
                                                   &pid[3])
                   };
                AltCtrl_struct *alt_ctrl4 = new AltCtrl_struct(4, alt_table4);
                for(int i = 0; i< 4; i++){
                  alt(alt_ctrl4);
                }
                SelectPrintf("on node %d, sync after alt\n", self_address);

                delete alt_table2[0],  delete alt_table2[1];
                delete alt_table4[0],  delete alt_table4[1], delete alt_table4[3],  delete alt_table4[4];
                delete alt_ctrl2, delete alt_ctrl4;


        }

        void alt_body2(int rank, void *arg_ptr){//notice the first implicit parameter
            if(rank==0)
               SelectPrintf("on node %d, alt_body2.%d, id_left.processor_Id = %d,
                            id_left.process_Id = %d\n", self_address, rank,
                            id_left.processor_Id, id_left.process_Id);
            if(rank==1)
             SelectPrintf("on node %, alt_body2.%d, id_right.processor_Id = %d,
                          id_right.process_Id = %d\n", self_address, rank,
                          id_right.processor_Id, id_right.process_Id);
        }

        void alt_body4(int rank, void *arg_ptr){//notice the first implicit parameter
             SelectPrintf("on node %d, alt_proc%d, pid[%d] = %d, &pid[%d] = %d\n",
                          self_address, rank, rank, pid[rank], rank, &pid[rank]);
        }
};

class Root_proc: public Process{

  public:
        Root_proc()
```

```
                : Process((membfunc_ptr)&Root_proc::body){}

        void body(){

                SelectPrintf("on node %d, in Root_proc::body1\n", self_address);
                proc_to_left   = new Token_RD<Id>(self_address-1);  //synchronous
                proc_to_right = new Token_RD<Id>(self_address+1); //synchronous
                token[0]        = new Token_D<int>;
                token[1]        = new Token_D<int>;
                token[2]        = new Token_D<int>;
                token[3]        = new Token_D<int>;
                Proc *proc1 = new Proc(1, 0);
                Proc *proc2 = new Proc(2, 0);
                Sync *sync  = new Sync;
                SelectPrintf("on node %d, in Root_proc::body2\n", self_address);
                Process *proc_tabl[4] = {proc1, proc2, sync, 0};
                par(proc_tabl);
                SelectPrintf("on node %d, in Root_proc::body4\n", self_address);
                delete proc1, delete proc2;
                delete proc_to_left, delete proc_to_right;
                delete token[0],  delete token[1], delete token[3],  delete token[4];
        }
};

//end of user program


MAIN(root_proc, Root_proc)
```

3. Global memory spread data structures and pointers.

The programming tools developed so far allow a process to get access to its personal data, those belonging to its father (via the standard *father_ptr* pointer), those belonging to other processes by using the message passing mechanisms defined in the previous section, and to global data located on the same node. Direct access to global data can prove very convenient in many situations. For instance, when no synchronization is required for a process to access a given data, or when copying data appears too expensive and can be advantageously replaced by synchronization plus access via pointers. It would be very useful to extend direct access to remote global data (located on other processors). This could prove useful in various situations. For instance, when all the processors, involved in a computation, simultaneously need to read a remote data or to write a data to a remote address. In such a case, the overall data movement would much more efficiently be performed via direct remote read/write executed on a synchronization barrier than via execution of several point-to-point message passing operations. The library offers several sorts of tools to achieve that. The most obvious and low level ones consist in simple general *read/write* statements that can access any memory location over the multiprocessor distributed memory. More sophisticated tools can be achieved via the definition of *spread and remote data structures* that can be accessed or traversed by using *spread or remote pointers* which are direct generalizations of common pointers.

3.1 First step toward global memory.

3.1.1 Remote read/write.

The library provides a set of generic functions that allow processes to perform remote read/write operations (see Appendix 5). These use the CMAML message passing facility and are handled under interrupts.

*void write(int dest_node, char *from, char *to, int size, Thread *thread_ptr);*
Writes a character string and schedules the specified thread upon completion of the data transfer. This is non-blocking function (i.e., the process executing the function is not stopped).

*void write(int dest_node, char *from, char *to, int size);*
Executes as the previous one except that the process executing the function is stopped and automatically resumed (i.e. scheduled) upon completion of the data transfer.

template<class T>
*void write(int dest_node, T *from, T*to, int size, Thread *thread_ptr);*
Executes as the first one but allows any T type data item to be written from a processor to another.

template<class T>
*void write(int dest_node, T *from, T*to, int size);*
Executes as the second one but allows any T type data item to be written from a process to another.

The library also provides four read functions that work symmetrically.

*void read(int dest_node, char *from, char *to, int size, Thread *thread_ptr);*
*void read(int dest_node, char *from, char *to, int size);*
template<class T>
*void read(int dest_node, T *from, T*to, int size, Thread *thread_ptr);*
template<class T>
*void read(int dest_node, T *from, T*to, int size);*

The remote read/write functions require two special data structures to be used. The final user is not required to know about them except that he would have to take care of their size if the default settings happened not to be appropriate (see the *WCB_SIZE* and *REP_PARAM_SIZE* constants in the environment file).

3.1.2. Sample program.

In the code below only one Proc process is run on each node. Each process reads an integer from the right (except 31) and writes an integer to the left (except 0). Each is stopped and next is resumed upon termination of the read or write operation. Global synchronization is only intended to ensure that reading/writing is over when the read/wrote data items are printed.

```
#include "environment"

//user program
```

```
int read_from  = 0;
int read_to       = 0;
int write_from = 0;
int write_to      = 0;

class Proc: public Process{

   public:
           Proc()
                     : Process((membfunc_ptr)&Proc::body){}

           void body(){
                     SelectPrintf("on node %d, in Proc, start\n", self_address);

                     read_from   = self_address;
                     write_from = self_address;
                     CMMD_sync_with_nodes();
                     SelectPrintf("on node %d, read_from = %d, write_from = %d\n",
                                         self_address, read_from, write_from);

                     if(self_address < 31)
                               read(self_address + 1, &read_from, &read_to);
                     if(self_address > 0)
                               write(self_address - 1, &write_from, &write_to);

                     CMMD_sync_with_nodes();
                     SelectPrintf("on node %d, read_to = %d, write_to = %d\n",
                                         self_address, read_to, write_to);

                     SelectPrintf("on node %d, in Proc, exit\n", self_address);
           }
};

//end of user program


MAIN(proc, Proc)
```

3.2 Spread data structures and pointers.

3.2.1. Spread arrays.

A spread data structure is a multidimensional array that has some of its dimensions distributed over the set of local memories. A spread array is defined by the type of data items it contains and the list of its spread and internal dimensions. A spread array is wrapped around the processor local memory set so that: if B denotes any of the sub-arrays generated by the internal dimensions then the B sub-arrays are placed successively, in increasing order of processor numbers starting from 0, according to the natural order of spread index combinations. Fig. 4 displays the placement of a integer spread array with two spread and one internal dimensions of size: 4, 5, 2, respectively. We assumed existence of only 3 processors in order to simplify the presentation.

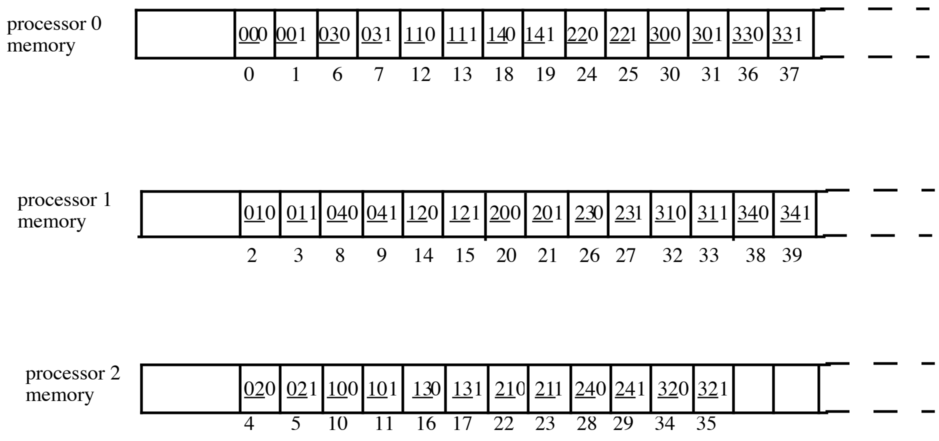

Fig. 4 SpreadArray with
two spread dimensions and one internal dimension.

This is the standard placement. By properly choosing the internal and spread dimensions we can generate block or cyclic placements as those defined in [8], see also [4]. The user can derive new classes which in turn can redefine their own placement by redefining the access functions, directly or via the standard one described above. This is one of the advantages of building via the object-oriented approach: nothing is hard wired as it would be with compiler-inserted tools. Everything can move according to the needs of the application at hand or simply for experimentation purposes. Spread arrays have been borrowed from Split-C [4, 5]. The idea is simple, easy to implement and combining spread an remote pointers (see section 3.2.5) we can achieve the mechanisms provided by more complex systems [12].

Spread arrays are instances of a special generic type (see Appendix 6, 7):

template<classT>
class SpreadArray;

The library Provides two means for spread array declaration (see SpreadArray constructors). The symbols used below are self-explanatory.

-1-

```
int spread_dim_size_array[spread_dim_nb] = {initialization list};
int intern_dim_size_array[intern_dim_nb] = {initialization list};
SpreadArray<actual_type>
SA(spread_dim_size_array, intern_dim_size_array, spread_dim_nb, inter_dim_nb);
```

for instance, in the case of Fig. 4,

```
int spread_dim_size_array[ 2] = {4, 5};
int intern_dim_size_array[1] = {2};
SpreadArray<int> SA(spread_dim_size_array, intern_dim_size_array, 2, 1);
```

SpreadArray<actual_type>
SA(dim_size_1, dim_size_2, ..., spread_dim_nb);

In this second representation there are exactly 'spread_dim_nb' spread dimensions whose sizes are respectively given by the first 'spread_dim_nb' 'dim_size_i' arguments.

For instance, in the case of Fig. 4,

SpreadArray<int> SA(4, 5, 2, 2);

Access to a spread array is expressed via operator( ) which is used in place of operator[□]. For instance, in the case of Fig. 4,

int k;
k = SA(2, 0, 1);

k is loaded with the 21st element of the SA spread array held by #1 processor memory.

3.2.2. Spread array construction /destruction.

Instantiating a spread array requires the participation of all nodes. The spread array sections are constructed synchronously by the nodes (in the CM5 implementation, the spread array constructor invokes CMMD_sync_with_nodes() before performing allocation). Space allocated to spread arrays is taken from a local heap especially devoted to this purpose (see *SDS_heap* and the *SDS_HEAP_SIZE* constant -which has to be properly set- in the environment file). This allows the sections of spread arrays to be located at the same place on all the nodes, which in turn ensures security and simple hence fast access functions. This is very constraining but again is claimed as not to be a too hard constraint for the purposes (algorithm implementation) we are developing for. Spread array destruction is also performed synchronously by all the nodes. To avoid any deadlock problems creation and destruction of spread arrays should be placed on global synchronization barriers.

As the spread array type is a direct generalization of the array type, a spread array is represented on each processing node as a memory area that contains the local section of the spread array together with a special data structure (see spread pointers below). This data structure is initialized with the base of the spread array section local to node 0 (actually the base of the whole array) and a pointer to a record that gathers on the node all the attributes of the spread array (pointer to local base, spread and internal dimension description: see the *SA_attributes* structure: see Appendix 6, 7)

3.2.3. Spread pointers.

Access to spread arrays is performed via special pointers denoted as spread pointers which are instances of a special type:

template<enum GPtrTyp s, class T>
class Star;

for instance:

```
Star<S, actual_type> spread_ptr;
```

where S is an explicit declaration that spread_ptr is a spread pointer that references *actual_type* data items. The enumeration type GPtrTyp has two possible values: S for spread pointers and R for another type of global pointers: the remote pointers, that we will describe later on.

A spread pointer actually is a data triple: (proc_id, local_ptr, SA_attributes_ptr). proc_id identifies the processor the referenced data is located on, local_ptr defines the location of the data on the processor and SA_attributes_ptr points to the attribute record of the spread array the Star pointer is attached to. The SA_attributes structure is constructed when the spread array is created. Any spread pointer must reference a SA_attributes structure. SA_attributes_ptr is set by the Star pointer constructor. The SA_attributes members are used for address computation and any operation that requires reference/rank-to-coordinate conversions or the converse, as involved in expressions like:

```
spread_ptr(2, 0, 1);
```

where spread_ptr could be defined as follows, in relation with the spread array defined in Fig. 4:

```
Star<S, int> spread_ptr = SA;
```

The Star class provides a number of constructors that allow converting any data item, any pair of data items in the triple (proc_id, local_ptr, SA_attributes_ptr) or the whole triple into a spread pointer (or a remote one according to the value of the first generic parameter, see section 3.2.5). The class also provides three functions *to_proc_id()*, *to_local_ptr()*, *to_SA_attr_ptr()* that respectively return the specified data member.

3.2.4. Spread pointer arithmetics.

The library provides predicates and arithmetic functions for spread pointers whose operation symbols overload the usual ones.

-Predicates:

==	same proc_id, same local_ptr and same SA_attributes_ptr,
>	(lhs local_ptr > rhs local_ptr or same local_ptr with proc_id > rhs proc_id) and same SA_attributes_ptr,
>=	(lhs local_ptr >= rhs local_ptr or same local_ptr with proc_id >= rhs proc_id) and same SA_attributes_ptr,
<	(lhs local_ptr < rhs local_ptr or same local_ptr with proc_id < rhs proc_id) and same SA_attributes_ptr,

<= (lhs local_ptr <= rhs local_ptr or same local_ptr with proc_id <= rhs proc_id) and same SA_attributes_ptr,

!= different proc_id or different local_ptr or different SA_attributes_ptr.

-Incrementation.

A spread array is wrapped around the processor memory set so that the sub-arrays generated by the internal dimensions are placed successively, in increasing order of processor numbers starting from 0, according to the natural order of spread index combinations (see Fig. 4). The library provides the usual set of Incrementation/ decrementation operators with a semantics corresponding to this placement rule.

3.2.5. Remote pointers.

Let us consider a spread pointer and suppose that we want to get access to some element local (same local spread array section) to that currently pointed to. The spread array arithmetics does not apply any longer and we would need to switch to the usual pointer arithmetics one. This can be achieved by converting the spread array pointer into a remote pointer. The library provides a pair of functions that perform spread-to-remote and remote-to-spread conversions (see Appendix 6):

template<GPtrTyp s, class T>
Star<s, T>& to_remote(Star<s, T>& sp);

template<GPtrTyp s, class T>
Star<s, T>& to_spread(Star<s, T>& sp);

that allow converting a Star<S, actual_type> spread pointer into a Star<R, actual_type> remote pointer and vice-versa. Remote pointers have a representation similar to that of Spread pointers, except they do not need a SA_attributes_ptr member (as the usual and not the spread arithmetics applies to them). Remote pointers are instances of a special type:

template<enum GPtrTyp s, class T>
class Star;

for instance:

Star1<R, actual_type> spread_ptr;

They allow access to remote data but follow a pointer arithmetics similar to that of usual pointers.

3.2.6. Remote pointer arithmetics.

The same operators as presented above are available in the case of spread pointers.

-Predicates:

== same proc_id and same local_ptr,

> same local_ptr and proc_id > rhs proc_id

>= same local_ptr and proc_id >= rhs proc_id,

< same local_ptr and proc_id < rhs proc_id

<= same local_ptr and proc_id <= rhs proc_id

!= different proc_id or different local_ptr

-Incrementation:

The semantics is the usual one but is applied to pointers that can reference data located on a remote processor identified by their proc_id member.

3.2.7. Nested spread/remote pointer.

-Nesting spread/remote pointer.

Spread and remote pointers that have been described so far are 1-level pointers (i.e., referencing the target data, which can possibly be a spread or remote pointer). i-level nested pointers can be defined as well. Here is a purely formal example. we still assume working on the 3-processor system exemplified in Fig. 4):

```
typedef Star<S, int>      s_type1;
s_type1   s_ptr1(self_address, SA.to_local_ptr(), SA.to_SA_attr_ptr());

typedef Star<R, s_type_1>   s_type2;
s_type2   s_ptr2((self_address+1)%3, &s_ptr1);

typedef Star<R, s_type_2>   s_type3;
s_type3   s_ptr3((self_address+1)%3, &s_ptr2);
```

The sequence above allows you to define a remote pointer on each node. The remote pointer references a second remote pointer located on the next node. In turn, this pointer references a spread pointer located on the next node. On each node, the spread pointer references the base of the SA local section.

-Spread pointer dereferencing.

Dereferencing a spread pointer is achieved via the use of a family of special operators: STAR1, STAR2, STAR3, ... The first one deals with dereferencing level 1 spread/remote pointers, the other ones deal with nested ones. These operators are predefined in the environment file till level 7 (Appendix 1). New ones can be defined, at will, in the same way.

In relation with the spread array defined in Fig. 4, the sequence:

```
Star<S, int> spread_ptr = SA;
int k;
k = STAR1(spread_ptr + 21);
```

stores in k the same value as that returned by SA(2, 0, 1).

Executing the following assignment:

```
int k;
k = STAR3 s_ptr3;
```

stores the value ((i+2)%3)*2 in the variable k allocated to processor i, for i = 1, 2 and 3.

The following assignment:

```
s_type2  another_s_ptr2 = STAR1 s_pt3;
```

reads the value (aggregate) of the remote pointer sp_ptr2 allocated to processor (i+1)%3 and stores it into the remote pointer another_s_ptr2 allocated to processor i, for i = 1, 2 and 3.

The composition of dereferencing operators satisfies the following rule:

STAR(i+j) ≅ STARi STARj STAR1 (in any order, composition is commutative).

Hence, the operator STAR(i+j) is not equivalent to the composed operator: STARi STARj. In designing the dereferencing operator, we had to get around the difficulty that the current version of C++ (and probably the final normalized version as well) is not really suited to cope with nested generic types. But this should not really raise a problem as far as we know about the rule.

The application of STARi is suspensive for any remote access (to intermediate spread or remote pointer or to target data) the operator performs.

3.2.8. Sample program.

This program simply performs a left circular shift of data over the set of processors. The program can be run on CM5.

```
#include "environment"

//user program

class Proc: public Process{
  public:

          SpreadArray<int>  spread_table(32, 2, 1);
```

```
        Proc()
                : Process((membfunc_ptr)&Proc::body){}

        void body(){
                SelectPrintf("on node %d, in Proc, start\n", self_address);
                Star<S, int> spread_ptr = spread_table;
                int *local_ptr = spread_table.to_local_ptr();
                local_ptr[0] = self_address;                    //initialization of  spread_table
                CMMD_sync_with_nodes();
                local_ptr[1] = spread_ptr((self_address+1)%32, 0);  //left circular shift
                SelectPrintf("on node %d, in Proc, shift result = %d\n", self_address, local_ptr[1]);
                SelectPrintf("on node %d, in Proc, exit\n", self_address);
        }
};

//end of user program

MAIN(proc, Proc)
```

3.3. About the object-oriented design of the spread array library.

we had two main goals in mind when we started designing this part of the library. we first wanted to reach a system providing complete type checking regarding global pointer declaration and manipulation. we also wanted the whole system exclusively built upon the object-oriented extension mechanisms provided by C++ (no insertion in the compiler). This was a major requirement since achieving such a goal clearly opens the way to extendibility and portability. As far as simple curiosity is concerned, we was also interested in checking how far it would be possible to go with the C++ template facility in coping with nested template types.

3.3.1 A first class-system.

A possible class system could be the one pictured in Fig. 5. The system is structured according to the different object types that naturally arise in the pointer evaluation process, namely: r and l-values[6, 10]. In Fig. 5 the continuous lines stand for the transformations from object to object that are caused by operator application while the discontinuous ones stand for class inheritance. The Rval and Star classes are parametrized with the pointer type (spread or remote) and the data type (type of pointed to data). SpreadArrays are understood as spread pointers.

However, the design described in Fig. 5 is rather awkward in the sense that it causes too much code to be uselessly instantiated in relation to spread/remote pointer declaration.

3.3.2. A second class-system.

Useless instantiation can be easily avoided by introducing a new set of interface classes and by shifting genericity from the former classes to the new ones (see Appendix 6). The interface classes are intended to contain only calls to the member functions defined in the former classes so that multiple instantiation now becomes quite inexpensive. This gives rise to the new set of classes shown in Fig. 6 and detailed in Appendix 6, 7.

The interface classes provide the set of operators already described to the final user. Each member function contains only one instruction that is a call to the corresponding function in the first class-system that has been made internal and actually contains the whole machinery. Genericity is shifted to the interface class system as described above. Each interface class inherits its functional attributes from the first class-system and pointer type attributes (S/R type) from a new class (Star_base) specially introduced in the class system for this purpose. Two functions have been removed from the first class-system and transferred to the interface. These are operator*() and the conversion operator which allow pointer dereferencing.

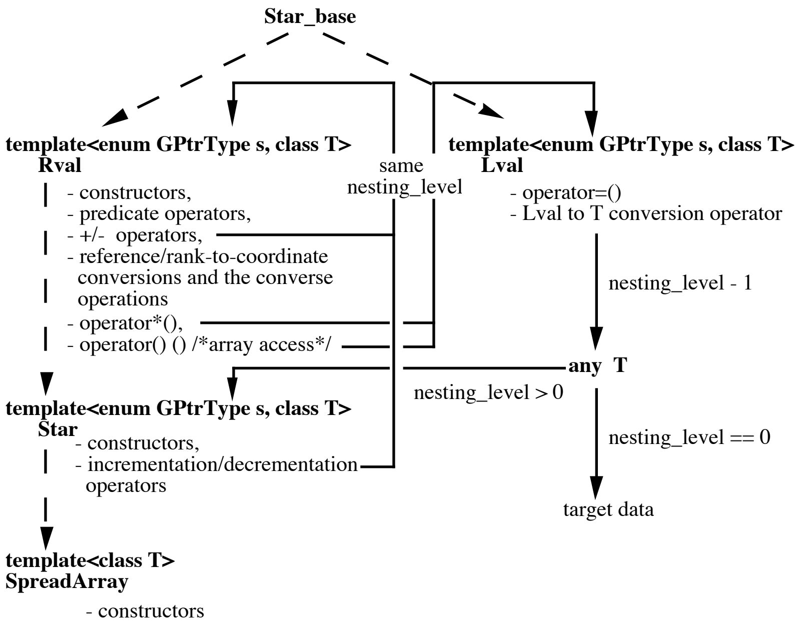
Star_base
template<enum GPtrType s, class T>
Rval
- constructors,
- predicate operators,
- +/- operators,
- reference/rank-to-coordinate conversions and the converse operations
- operator*(),
- operator() () /*array access*/
same nesting_level
template<enum GPtrType s, class T>
Lval
- operator=()
- Lval to T conversion operator
nesting_level - 1
any T
nesting_level > 0
nesting_level == 0
target data
template<enum GPtrType s, class T>
Star
- constructors,
- incrementation/decrementation operators
template<class T>
SpreadArray
- constructors

Fig. 5 Spread arrays and pointers: a first class-system

Conclusion and future works.

This paper reported on the design of a library for SPMD programming. The library is intended to allow the development of portable parallel applications. It is fully portable on any MIMD multiprocessor system at the expense of rewriting a very small set of machine dependent procedures. Starting from scratch, the library was designed as a four layer system providing: threads, processes, synchronization and synchronous communication, remote read/writes and spread arrays and pointers.

The library was written by using the natural object-oriented extension mechanisms available in C++ which opens the way to easy extendibility and portability. we are convinced that the object-oriented approach is particularly well suited to achieving the development of portable applications in the multiprocessor arena. The underlying methodology assumes programming universality be restricted from the whole programming domain to the application one. But, we wonder how much this is a real restriction: programming universality is more myth than reality. Each application domain has got a particular language attached to it from the very beginning (numerical computation, database processing, system development, ...) and the trend massively broadened in the last decade with the use of personal computers.

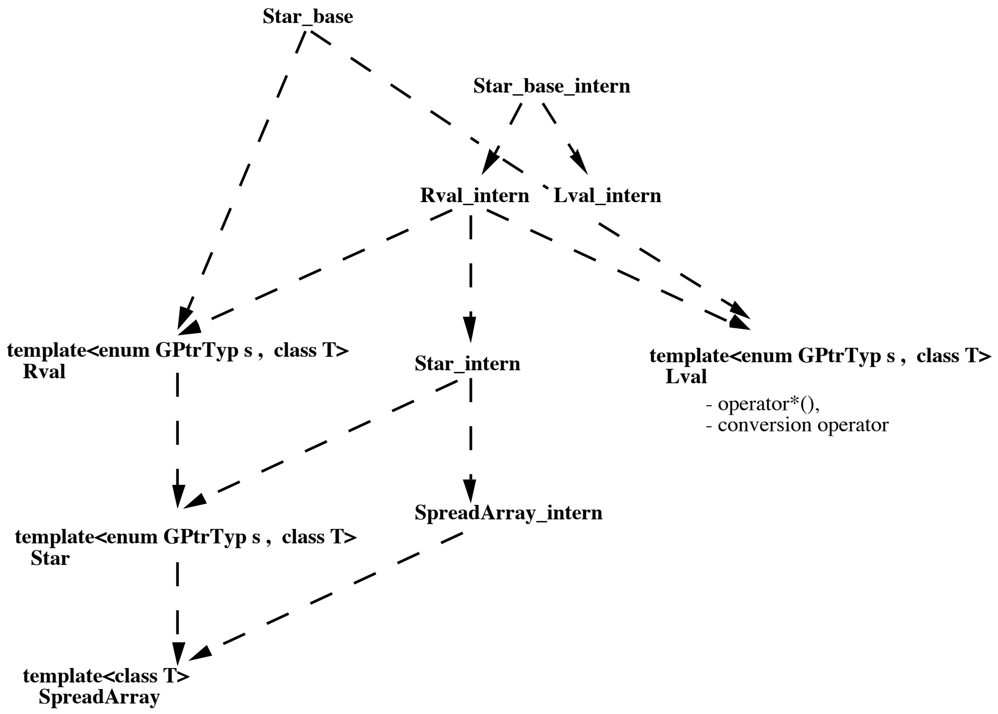

Fig. 6. SpreadArrays and pointers a second class_sytem

The functions of the library have not been assessed, yet. This is the work we intend to perform on the CM5. What is the light-process grain that we can afford on this type of machine that has not been specially designed to switch contexts fast? what costs each of the synchronization, message-passing and read/write functions? Most part of the protocols started by these functions are exclusively handled under interrupts. This gives a very general and regular fashion to handle inter-processor communication, but this can also turn to be expensive as handlers switch contexts too. Although, in the case of the CM5, we guess that this should not really be a problem as the CMAML system polls under interrupt. Some CMMD/CMAML functions should be reexamined in the context of context-switching processes. For instance, the standard input functions (that should cause a context switching), the barrier synchronization function (for similar reasons) and probably other ones.

We are planing to complete the spread array class with additional facilities, similar to those we find in HPF FORTRAN, that provide automatic array alignment and redistribution. This could be derived very simply from the SpreadArray class we described in the paper. It appears also quite easy to develop the library toward data parallelism by deriving new functions that would perform according to the semantics of the HPF FORTRAN INDEPENDENT-DO/FORALL instructions. Furthermore, in addition to spread arrays, it would be interesting to enrich the library with other spread data structures (in the same way as we did for spread arrays). For instance, introducing spread trees, which appear in many applications, is appealing and would be very useful.

References.

[1] AT&T C++ Language System Library Manual, 1991.

[2] Bodin F. et al. Implementing a Parallel C++ Run-time System for Scalable Parallel Systems.

[3] Chapman B. M., Mehrotra P., Zima H. P., Vienna Fortran, A Fortran Language Extension for Distributed Memory Multiprocessors, Compilers and Run-time Environments for Distributed Memory Machines, Elsevier 1992.

[4] Culler D.E. et al., Introduction to Split-C, Computer Science Dept. UC Berkeley, April 1993.

[5] Culler D.E. et al., Parallel Programming in Split-C, Computer Science Dept. UC Berkeley.

[6] Dongarra J., Pozo R., Walker D., An object Oriented Design for High Performance Linear Algebra on Distributed Memory Architectures, Proc. OON-SKI Object oriented numeric conf., pp 268-269, April 1993.

[7] Ellis M.A., Stroustrup B., The Annotated C++ Reference Manual, Addison Wesley, 1990.

[8] High Performance Fortran, Language Specification, Version 1.0, High Performance Fortran Forum, May 3, 1993.

[9] INMOS ltd., OCCAM2, Reference Manual, Prentice, series in comp. Science, 1988.

[10]Lippman S.B. C++ Primer, 1992.

[11]Malki D., Snir M., Nicke, C Extensions for Programming on Distributed-Memory Machines, Compilers and Run-time Environments for Distributed Memory Machines, Elsevier 1992.

[12]Rosing M., Schnabel R. B., Weaver R. P., Scientific Programming Languages for Distributed Memory Multiprocessors: Paradigms and Research issues, Languages, Compilers and Run-time Environments for Distributed Memory Machines, Elsevier 1992.

[13]Thinking Machine Corporation, CMMD Reference Manual, manual, Version 3.0, May 1993.

[14]Thorsten von Eicken, Active Messages: a Mechanism for Integrated Communication and Computation, Report UCB/CSD 92/#675, march 1992.

Appendices.

```
/**********************************************
**                                          **
**               Appendix 1.                **
**                                          **
**            class declaration             **
**                                          **
**********************************************/
/**********************************************
**                                          **
**          Author Jean-Marc Adamo          **
**                                          **
**********************************************/


// Basic scheduling

class Sched_data;

class Thread_base;

class Thread;

class Process;

class Main_thread;

typedef void (Thread::*membfunc_ptr)(...);

typedef void (*func)(...);



//synchronized communication

class Token;

template<class T>
class Token_D;

class Token_R;

template<class T>
class Token_RD;

class AltBranch;

template<class T>
class AltBranch_recv;

class AltBranch_R;

template<class T>
class AltBranch_Rrecv;

class AltCtrl_struct;

class Ctrl_block;


// Remote read/write
```

```
class Rrw;


// Spread data structures and global pointers

enum GPtrTyp {R=0, S=1};

class Star_base;


template<class T>
class Lval;

template<enum GPtrTyp s, class T>
class Rval;

template<enum GPtrTyp s, class T>
class Star;

template< class T>
class SpreadArray;

class Star_base_intern;

class Lval_intern;

class Star_intern;

struct SA_attributes;

class SpreadArray_intern;


//local_heap

class Local_heap;

class CtrlWords;
```

```
/***********************************************
**                                            **
**              Appendix 2.                   **
**                                            **
**              environment :                 **
**                                            **
**     include files and constante setting    **
**                                            **
***********************************************/
/***********************************************
**                                            **
**          Author Jean-Marc Adamo            **
**                                            **
***********************************************/



extern "C" int printf(const char*, ...);
#include<stdio.h>
#include<string.h>

//processing-node number
int self_address;

//include<cm/cmmd.h>
#include "extern"
# include "utilities"

#include "class-declaration"

#define PROC_NB 32

#define TOKEN_R_HEAP_SIZE       1024*100
#define SDS_HEAP_SIZE           1024*5

#define SCHED_QUEUE_SIZE        1024
#define PROC_STACK_SIZE                 1024*50
Sched_data *sched_data;

#define WCB_SIZE 512
#define REP_PARAM_SIZE                  1024
Rrw *rrw;

Local_heap *token_R_heap;
Local_heap *SDS_heap;

# include "local_heap.h"
# include "local_heap.cc"

#define UNPROTECTED             1
# include "sched.h"
# include "sched.cc"

# include "syncSR.h"
# include "syncSR.cc"

# include "remoteRW.h"
# include "remoteRW.cc"

# include "spread.intern.h"
# include "spread.intern.cc"
# include "spread.interface.h"


#define STAR1     *

#define STAR2     ** *

#define STAR3     ** ** *

#define STAR4     ** ** ** *

#define STAR5     ** ** ** ** *

#define STAR6     ** ** ** ** ** *

#define STAR7     ** ** ** ** ** ** *


#define MAIN(root_proc, Root_proc)\
main(){\
\
        self_address = CMMD_self_address();\
\
        CMMD_sync_with_nodes();\
        token_R_heap     = new Local_heap(TOKEN_R_HEAP_SIZE);\
        SDS_heap         = new Local_heap(SDS_HEAP_SIZE);\
        sched_data       = new Sched_data(SCHED_QUEUE_SIZE);\
        rrw              = new Rrw(WCB_SIZE, REP_PARAM_SIZE);\
\
        Root_proc *root_proc = new Root_proc;\
\
        root_proc->initialise();\
\
        ((Thread *)root_proc)->schedule();\
\
        Main_thread *main_thread = new Main_thread;\
\
        CMMD_fset_io_mode(stdout, CMMD_independent);\
\
        CMMD_sync_with_nodes();\
\
        main_thread->start_keep_up_and_terminate();\
\
}
```

```
/***********************************************
**                                            **
**                Appendix 3.                 **
**                                            **
**      Thread, Processes and scheduling: .h  **
**                                            **
***********************************************/
/***********************************************
**                                            **
**            Author Jean-Marc Adamo          **
**                                            **
***********************************************/


class Sched_data{

  public: //protected

          Thread     **queue_put_ptr;
          Thread     **queue_get_ptr;
          int      queue_count;
          int           queue_size;




  public:

          int termination_flag;

          Sched_data(int queue_sz);

};


class Thread_base{

  public: //protected

          int stack_ptr_save;
          int prog_counter_save;

          void next_thread();
          void next_thread(int);
          void next_thread_body();


};


class Thread: public Thread_base{

  public: //protected

          int *stack_base_ptr;
          int proc_stack_size;
          membfunc_ptr entry_point;

          //for stack initialisation, this is machine dependent
          void push_frame();
```

```
            void last_frame();


    public:

            Thread *father_ptr;

            int sub_process_count;
                    //set to 0 at process creation. Set to a non 0 number in the
                    //par function (number of sub-processes run in this function).
                    //decremented by the sub-processes executing in the par
                    //function.


    public:

            Thread(membfunc_ptr entry_pt, int stack_size);

            Thread(){}

            ~Thread(){
               delete [] stack_base_ptr;
            }

            //Thread initialisation
            void initialise();


            //put in the scheduling queue the process it is excuted for
            //(i.e. thread.schedule() or thread_ptr->schedule())

            void schedule();
            void schedule(int);
            void schedule_body();



            //put in the scheduling queue the processes held in proc_table

            void schedule(Thread **proc_table);
            void schedule(Thread **proc_table, int);
            void schedule_body(Thread **proc_table);

            //reschedule simply puts the current process at the end of the
            //scheduling queue. Reference to the current process is present
            //in the global variable: current_thread_ptr

            void reschedule();
            void reschedule(int);
            void reschedule_body();


            //these functions stop the current process and run the next one in the
            //scheduling queue

            void stop();
            void stop(int);
};

class Process: private Thread{

    public: //protected
```

```
};

            //for stack initialisation(redefinition), this is machine dependent
            void push_frame();
            void last_frame();

            //This function deschedules the current process. It causes the
            //scheduling system to exit if executed by the root process. It causes
            //the father process to resume in order to return from par function.

            void exit();
  public:

            Process(membfunc_ptr entry_pt, int stack_size=PROC_STACK_SIZE)
                    : Thread((membfunc_ptr)entry_pt, stack_size){}

            Process()
                    : Thread(){}

            //Process initialisation (redefinition)
            void initialise();

            //This function schedules the processes of the set held in proc_table.
            //It does not return until all the scheduled processes have finished.

            void par(Process **proc_table);

            //maintain public access level through Process of this member
            Thread::father_ptr;

            //redeclaration of rescheduled: maintaining public acccess of
            //overloaded functions is not implemented!
        void reschedule();
        void reschedule(int);
        void reschedule_body();
};

class Main_thread: public Thread_base{

  public: //protected

            // exit() has immediate effect. the processes that could be waiting
            // to run in the scheduling queue are cancelled. This means that
            // the user must synchronize the end of all currently executing
            // threads before executing this function. This is automatically
            // handled when using Processes instead of Threads(see Process::exit().

            void exit();
            void exit(int);

            // the start_keep_up_and_terminate() function is run as a starter process
            // by main() and return to main() at the end the program only (normal
            // termination or error occurrence). In the mean time, it is resumed
            // each time there is no ready process to run in the scheduling queue
            // (the node is waiting for completion of communication operations
            // handled in interruption handlers). It is also resumed upon error
            // occurence to exit from the scheduling process (return from the
            // start_keep_up_and_terminate function to main(). It implicitly uses
            // the main() stack. This process is never put in the scheduling queue.

            void start_keep_up_and_terminate();
};
```

```
/***********************************************
**                                            **
**               Appendix 4.                  **
**                                            **
**         synchronized send/recv: .h         **
**                                            **
***********************************************/
/***********************************************
**                                            **
**          Author Jean-Marc Adamo            **
**                                            **
***********************************************/


class Token {

  public: //protected

        Thread              *enabled_ptr;
        AltCtrl_struct      *alt_flag_ptr;
        int                  alt_rank;

  public:
        Token(){
                enabled_ptr  = (Thread *)0;
                alt_rank = 0;
                alt_flag_ptr = (AltCtrl_struct *) 0;
                  // initialized in AltCtrl_struct_U/F
        }
};

template<class T>
class Token_D : public Token{

  public: //protected

        T *from_ptr;
        T *to_ptr;

  public:

        Token_D(T *f_ptr, T *t_ptr){
                from_ptr = f_ptr;
                to_ptr   = t_ptr;
        }

        Token_D(){
                from_ptr = 0;
                to_ptr   = 0;
                // no data-size is required as data transfer is
                //performed via the use of operator=() provided with T
        }
};


/*
** A copy of any Token_R or Token_RD is simultaneously created on each
** node. The creation is performed synchronously on all the processors
** in a dedicated local_heap.
**
** The construction/destruction of this new token_R/D is performed via
** the use of overloaded new/delete operators which synchronises all the
** nodes with sync_with_nodes() before executing creation or deletion.
** All Token_R/D tokens are consequently located at the same memory place
** on all the processors.
*/

class Token_R : public Token{

   public: //protected

            Thread                 *enabled_ptr_to_send_to;
            //                     enabled_ptr_to_reply_to is enabled_ptr
            //                                                 inherited from base
            int                    remote_proc_to_send_to;
            int                    remote_proc_to_reply_to;


   public:

            Token_R(int r_p_t_s_t){
                    enabled_ptr_to_send_to  = (Thread *)0;
                    remote_proc_to_send_to = r_p_t_s_t;
            }

            Token_R(){
                    enabled_ptr_to_send_to  = (Thread *)0;
            }


            //Token_R/D are allocated in a special heap token_R_heap
            //allocation and desallocation is synchronous

            void *operator new(size_t size);

            void operator delete(void *ptr){
                    CMMD_sync_with_nodes();
                    token_R_heap->free(ptr);
            }
};


template<class T>
class Token_RD : public Token_R{

   public: //protected

            T *from_ptr;
            T *to_ptr;
            int data_size;

   public:

            Token_RD(int r_p_t_s_t, T *f_ptr, T *t_ptr):Token_R(r_p_t_s_t){
                    from_ptr = f_ptr;
                    to_ptr   = t_ptr;
                    data_size = sizeof(T);
            }

            Token_RD(int r_p_t_s_t, T *f_ptr, T *t_ptr, int sz):Token_R(r_p_t_s_t){
                    from_ptr = f_ptr;
                    to_ptr   = t_ptr;
                    data_size = sz; //sz is the string length if T is a char string
```

```
		}

		Token_RD(int r_p_t_s_t):Token_R(r_p_t_s_t){
			from_ptr = 0;
			to_ptr   = 0;
			data_size = sizeof(T);
		}

		Token_RD(int r_p_t_s_t, int sz):Token_R(r_p_t_s_t){
			from_ptr = 0;
			to_ptr   = 0;
			data_size = sz;		//sz is the string length if T is a char string
		}

		Token_RD(){
			from_ptr = 0;
			to_ptr   = 0;
			data_size = sizeof(T);
		}
};



class AltBranch {

  public: //protected

		Token *token_ptr;
		membfunc_ptr alt_body_ptr;
		void  *arg_ptr;

  public:

		AltBranch(Token *t, membfunc_ptr f, void *a){
			token_ptr    = t;
			alt_body_ptr = f;
			arg_ptr  = a;
		}

		AltBranch(){}

		virtual int alt_variation(int i,
						AltCtrl_struct *altCtrl_struct_ptr);
		virtual int alt_fair_variation1(int i,
						AltCtrl_struct *altCtrl_struct_ptr);
		virtual int alt_fair_variation2(int i,
						AltCtrl_struct *altCtrl_struct_ptr);
		virtual void disable();

};

template<class T>
class AltBranch_recv : public AltBranch {

  public: //protected

		T     *to_ptr;

  public:

		AltBranch_recv(Token_D<T> *t, membfunc_ptr f, void *a, T *ptr)
		  :AltBranch(t, f, a){
```

```
                to_ptr = ptr;
        }

        AltBranch_recv(Token_D<T> *t, membfunc_ptr f, void *a)
          :AltBranch(t, f, a){
        }

        AltBranch_recv(){}

        int alt_variation(int i, AltCtrl_struct *altCtrl_struct_ptr);
        int alt_fair_variation1(int i, AltCtrl_struct *altCtrl_struct_ptr);
        int alt_fair_variation2(int i, AltCtrl_struct *altCtrl_struct_ptr);
        void disable();
};


class AltBranch_R : public AltBranch {

  public:

        AltBranch_R(Token_R *t, membfunc_ptr f, void *a):AltBranch(t, f, a){
        }

        AltBranch_R(){}

        int alt_variation(int i, AltCtrl_struct *altCtrl_struct_ptr);
        int alt_fair_variation1(int i, AltCtrl_struct *altCtrl_struct_ptr);
        int alt_fair_variation2(int i, AltCtrl_struct *altCtrl_struct_ptr);
        void disable();
};

template<class T>
class AltBranch_Rrecv : public AltBranch_R {

   public: //protected

        T     *to_ptr;

   public:

        AltBranch_Rrecv(Token_RD<T> *t, membfunc_ptr f, void *a, T *ptr)
          :AltBranch_R(t,f, a){
                to_ptr = ptr;
        }

        AltBranch_Rrecv(Token_RD<T> *t, membfunc_ptr f, void *a)
          :AltBranch_R(t, f, a){
        }

        AltBranch_Rrecv(){}

        int alt_variation(int i, AltCtrl_struct *altCtrl_struct_ptr);
        int alt_fair_variation1(int i, AltCtrl_struct *altCtrl_struct_ptr);
        int alt_fair_variation2(int i, AltCtrl_struct *altCtrl_struct_ptr);
        void disable();
};



class AltCtrl_struct {

   public: //protected
```

```
            AltBranch    **table;
            int          start;
            int          size;
            int               selected_alt;

   public:

            AltCtrl_struct(int sz, AltBranch **alt_table);
            AltCtrl_struct(){;}


};



/*
** alt funtion.
*/

void alt(AltCtrl_struct *altCtrl_struct);

void alt(AltCtrl_struct *altCtrl_struct, int);

void alt_fair(AltCtrl_struct *altCtrl_struct);

void alt_fair(AltCtrl_struct *altCtrl_struct, int);


/*
** local send/recv without data transfer.
*/

void send(Token *token);

void send(Token *token, int);

inline
void send_body(Token *token);

void recv(Token *token);

void recv(Token *token, int);

inline
void rec_bodyv(Token *token);


/*
** local recv with data transfer.
**
** Cannot be used in conjunction with alt.
*/


template <class T>
inline
void send_body(Token_D<T> *token_ptr, T *from_ptr);

template <class T>
void send(Token_D<T> *token_ptr, T *from_ptr);
```

```
template <class T>
void send(Token_D<T> *token_ptr, T *from_ptr, int);

template <class T>
inline
void recv_body(Token_D<T> *token_ptr, T *to_ptr);

template <class T>
void recv(Token_D<T> *token_ptr, T *to_ptr);

template <class T>
void recv(Token_D<T> *token_ptr, T *to_ptr, int);

template <class T>
int AltBranch_recv<T>::alt_variation(int i, AltCtrl_struct *altCtrl_struct_ptr);

template <class T>
int AltBranch_recv<T>::alt_fair_variation1(int start,
                                            AltCtrl_struct *altCtrl_struct_ptr);
template <class T>
int AltBranch_recv<T>::alt_fair_variation2(int i,
                                            AltCtrl_struct *altCtrl_struct_ptr);
template <class T>
void AltBranch_recv<T>::disable();


/*
** remote send/recv without data transfer.
*/

void send(Token_R *token_ptr);

void send(Token_R *token_ptr, int);

inline
void send_body(Token_R *token_ptr);

void recv(Token_R *token_ptr);

void recv(Token_R *token_ptr, int);

inline
void recv_body(Token_R *token_ptr);

void Rsend_request_handler(int remote_proc_to_reply_to, Token_R *token_ptr);

void Rsend_alt_reply_handler(Token_R *token_ptr);


/*
** Remote send & recv with data transfer.
**
** The meeting protocol is based on "master/slave" behaviors followed by the
** processors meeting on a token. Only one processor is allowed to propose a
** meeting to the remote processor. The other one can only wait. The proposing
** processor is the one executing send.
*/

template <class T>
inline
void send_body(Token_RD<T> *token_ptr, T *from_ptr);
```

```
template <class T>
void send(Token_RD<T> *token_ptr, T *from_ptr);

template <class T>
void send_body(Token_RD<T> *token_ptr, T *from_ptr, int);

template <class T>
inline
void recv_body(Token_RD<T> *token_ptr, T *to_ptr);

template <class T>
void recv(Token_RD<T> *token_ptr, T *to_ptr);

template <class T>
void recv(Token_RD<T> *token_ptr, T *to_ptr, int);

void RDsend_request_handler(int remote_proc_to_reply_to,
                                                    Token_RD<char> *token_ptr);
//executed on the sender side
void RDsend_alt_rpc_handler(int port_id, Token_RD<char> *token_ptr,
                                                                  int err_flag);
//executed on the receiver side
void RDend_of_copy_receiver_side_handler(int port_id,
                                                 Token_RD<char> *token_ptr;
//executed on the sender side
void RDsender_resumption_handler(Token_RD<char> *token_ptr);

template <class T>
int AltBranch_Rrecv<T>::alt_variation(int i,
                                        AltCtrl_struct *altCtrl_struct_ptr);
template <class T>
int AltBranch_Rrecv<T>::alt_fair_variation1(int start,
                                             AltCtrl_struct *altCtrl_struct_ptr);
template <class T>
int AltBranch_Rrecv<T>::alt_fair_variation2(int i,
                                             AltCtrl_struct *altCtrl_struct_ptr);
template <class T>
void AltBranch_Rrecv<T>::disable();

inline
int set_port(Token_RD<char> *token_ptr, void *to_ptr);
```

```
/**********************************************
**                                          **
**              Appendix 5.                 **
**                                          **
**         Remote Read/Write: .h            **
**                                          **
**********************************************/
/**********************************************
**                                          **
**          Author Jean-Marc Adamo          **
**                                          **
**********************************************/


class Ctrl_block{       //used by the write instructions on the writer side
                        //to keep information required to perform a write
                        //operation
  public:

        int dest_node;
        char *from;                 //source address
        char *to;                   //destination address
        /* alignement field in s_copy is implicitly 0 */
        int size;           //size of the transfered data
        Thread *thread_ptr;         //for synchronization upon write completion
        Ctrl_block *next;   //next ctrl block
};


class Reply_param{      //used by the write instructions on the written side
                        //to keep track of writer and Ctrl_block identity

  public:

        int requesting_node;
        Ctrl_block *CB;
        Reply_param *next;
};


class Rrw {

  protected:

        Ctrl_block *Ctrl_block_head_ptr;

        Reply_param *Reply_param_head_ptr;


  public:

        Rrw(int WCB_sz, int Rep_param_sz);


        Ctrl_block *alloc(int d, char *f, char *t, int s, Thread *thread_ptr);

        void free(Ctrl_block *ptr);

        Reply_param *alloc(int rn, Ctrl_block *CB);
        void free(Reply_param *ptr);

};


/*
** write a "size"-length string. thread_ptr defines the thread to be scheduled
** on writer node upon termination.
*/

void write(int dest_node, char *from, char *to, int size, Thread *thread_ptr);


/*
**  write a "size"-length string, with self rescheduling.
*/

void write(int dest_node, char *from, char *to, int size);


/*
** write a type T object. thread_ptr defines the thread to be scheduled
** on writer node upon termination.
*/

template <class T>
void write(int dest_node, T *from, T* to, Thread *thread_ptr);


/*
** write a type T object, with self rescheduling.
*/

template <class T>
void write(int dest_node, T *from, T* to);


/*
** read a "size"-length string.
*/

void read(int dest_node, char *from, char* to, int size, Thread *thread_ptr);


/*
**  read a "size"-length string, with self rescheduling.
*/

void read(int dest_node, char *from, char* to, int size);


/*
** read a type T object.
*/

template <class T>
void read(int dest_node, T *from, T* to, Thread *thread_ptr);


/*
** read a type T object, with self rescheduling
*/
```

```
template <class T>
void read(int dest_node, T *from, T* to);


/*
** Handlers.
*/

void write_request_handler(int requesting_node, char *to,
                                    int size, Ctrl_block *CB);

void write_reply_handler_1(int port_id, Ctrl_block *CB, int err_flag);

inline
void write_recv_side_term_handler(int port_id, Reply_param *reply_param);

void write_reply_handler_2(Ctrl_block *CB);

inline
void read_request_handler (int requesting_node, int port_id,
                                            char *from,  int size);

void read_recv_side_term_handler(int port_id, Thread *thread_ptr);
```

```
/***********************************************
**                                            **
**                Appendix 6.                 **
**                                            **
**   Spread data structures and global pointers **
**                                            **
**             spread.interface.h             **
**                                            **
***********************************************/
/***********************************************
**                                            **
**           Author Jean-Marc Adamo           **
**                                            **
***********************************************/



class Star_base{

  public:
          enum GPtrTyp ptr_type;

          Star_base(){}

 };



/*Lval*/
template<class T>
class Lval: public Star_base, public Lval_intern{

  public:

          Lval( unsigned p_id,
             T *l_ptr,
             SA_attributes *SA_a_ptr)
          :Star_base(), Lval_intern(p_id, l_ptr, SA_a_ptr)
          {}

          Lval(const Lval_intern& lv)
               : Lval_intern(lv)
          {}

          Lval()
          :Star_base()
          {}


          //for use when a Lval<T> data occurs as lhs of operator=

          T operator= (T rhs){

            if(proc_id < 0 || proc_id >= PROC_NB ||!local_ptr)

               error(16);

            else

              if(proc_id != self_address){
                write(proc_id, (char *)&rhs, (char *)local_ptr, sizeof(T));
            }else
                *(T *)local_ptr = rhs;
        return rhs;
        }


        //Lval<T> to T conversion, for use when a Lval data occurs as rhs
        //of operator=() or as operand of any other operator requiring a T

        operator T(){
          T *next_ptr = (T *)local_ptr;
          if(proc_id != self_address){
                  T next;
                  next_ptr =&next;

                  if(proc_id >= 0 && proc_id < PROC_NB){

                          //read with self rescheduling
                          read(proc_id, (char *)local_ptr,
                               (char *)next_ptr,
                               sizeof(T) );
                  }else{
                          error(24); //referenced processor does not exist
                  }
          }
          return *next_ptr;
        }


        //This is to force conversion in the dereferencing process
        //code is the same as in operator T() above

        T operator*(){
          T *next_ptr = (T *)local_ptr;
          if(proc_id != self_address){
                  T next;
                  next_ptr =&next;

                  if(proc_id >= 0 && proc_id < PROC_NB){

                          //read with self rescheduling
                          read(proc_id, (char *)local_ptr,
                               (char *)next_ptr,
                               sizeof(T) );
                  }else{
                          error(24); //referenced processor does not exist
                  }
          }
          return *next_ptr;
        }
};


/*Rval*/
template<enum GPtrTyp s, class T>
class Rval: public Star_base, public virtual Rval_intern{

  public:

        Rval(const Rval<s, T>& rv)
             : Rval_intern(rv)
```

```
{
  ptr_type=s;
}


//The following constructor is  only used by this class or fully
//typed Star and SpreadAray subclasses. It is not available to the
//final user. this would open a hole to type cheking

Rval(const Rval_intern& rv)
        : Rval_intern(rv)
{
  ptr_type=s;
}


Rval(unsigned p_id, T *l_ptr, SA_attributes *SA_a_ptr)
                : Rval_intern(p_id, l_ptr, SA_a_ptr)
{
  ptr_type=s;
}


Rval(unsigned p_id, T *l_ptr)
                : Rval_intern(p_id, l_ptr)
{
  ptr_type=s;
}


Rval(T *l_ptr, SA_attributes *SA_a_ptr)
                : Rval_intern(l_ptr, SA_a_ptr)
{
  ptr_type=s;
}


Rval(T *l_ptr)
                : Rval_intern(l_ptr)
{
  ptr_type=s;
}

Rval(unsigned p_id, SA_attributes *SA_a_ptr)
                : Rval_intern(p_id, SA_a_ptr)
{
  ptr_type=s;
}


Rval(SA_attributes *SA_a_ptr)
                : Rval_intern(SA_a_ptr)
{
  ptr_type=s;
}

Rval()
                : Rval_intern()
{
  ptr_type=s;
}
```

```
        T *to_local_ptr(){
          return (T *)Star_base_intern::to_local_ptr();
        }


        Lval<T>
        operator*(){
                    return Lval<T>(proc_id, (T *)local_ptr, SA_attributes_ptr);
        }

        unsigned operator==(Rval<s, T> rv){
          return equal(rv, s);
        }
        unsigned operator>(Rval<s, T> rv){
          return greater(rv, s);
        }
        unsigned operator>=(Rval<s, T> rv){
          return greater_equal(rv, s);
        }
        unsigned operator<(Rval<s, T> rv){
          return less(rv, s);
        }
        unsigned operator<=(Rval<s, T> rv){
          return less_equal(rv, s);
        }
        unsigned operator!=(Rval<s, T> rv){
          return diff(rv, s);
        }



        Rval<s, T> operator+(unsigned i){

          return Rval<s, T>
                            (plus_i(i, s, sizeof(T)));

  }

        Rval<s, T> operator-(unsigned i){

          return Rval<s, T>
                            (minus_i(i, s, sizeof(T)));

        }



        Lval<T> operator()(unsigned *idx_ptr){

          return Lval<T>
                            (access(idx_ptr, s, sizeof(T)));

  }

Lval<T>
operator()(unsigned i0){
          return Lval<T>
                            (access(i0, s, sizeof(T)));
        }

Lval<T>
```

```
         operator()(unsigned i0, unsigned i1){
                return Lval<T>
                                    (access(i0, i1, s, sizeof(T)));
               }

        Lval<T>
         operator()(unsigned i0, unsigned i1, unsigned i2){
                return Lval<T>
                                    (access(i0, i1, i2, s, sizeof(T)));
               }

        Lval<T>
         operator()(unsigned i0, unsigned i1, unsigned i2, unsigned i3){
                return Lval<T>
                                    (access(i0, i1, i2, i3, s, sizeof(T)));
               }

        Lval<T>
         operator()(unsigned i0, unsigned i1, unsigned i2, unsigned i3,
                                                                        unsigned i4){
                return Lval<T>
                          (access(i0, i1, i2, i3, i4, s, sizeof(T)));


               }

        Lval<T>
         operator()(unsigned i0, unsigned i1, unsigned i2, unsigned i3,
                                                           unsigned i4, unsigned i5){
                return Lval<T>
                          (access(i0, i1, i2, i3, i4, i5, s, sizeof(T)));

               }

        Lval<T>
          operator()(unsigned i0, unsigned i1, unsigned i2, unsigned i3,
                                                  unsigned i4, unsigned i5, unsigned i6){
                return Lval<T>
                          (access(i0, i1, i2, i3, i4, i5, i6, s, sizeof(T)));


               }
        Lval<T>
          operator()(unsigned i0, unsigned i1, unsigned i2, unsigned i3,
                                      unsigned i4, unsigned i5, unsigned i6, unsigned i7){
                return Lval<T>
                     (access(i0, i1, i2, i3, i4, i5, i6,i7, s, sizeof(T)));

               }


};



/*Star*/
template<enum GPtrTyp s, class T>
class Star: public Rval<s, T>, public virtual Star_intern{

   public:

               Star(const Rval<s, T>& rv)
        :Rval<s, T>(rv),
                Star_intern(rv),
                Rval_intern(rv)
{}

Star(unsigned p_id, T *l_ptr, SA_attributes *SA_a_ptr)
               :Rval<s, T>(p_id, l_ptr, SA_a_ptr),
                Star_intern(p_id, l_ptr, SA_a_ptr),
                Rval_intern(p_id, l_ptr, SA_a_ptr)
{}

Star(unsigned p_id, T *l_ptr)
               :Rval<s, T>(p_id, l_ptr),
                Star_intern(p_id, l_ptr),
                Rval_intern(p_id, l_ptr)
{}

Star(unsigned p_id, SA_attributes *SA_a_ptr)
               :Rval<s, T>(p_id, SA_a_ptr),
                Star_intern(p_id, SA_a_ptr),
                Rval_intern(p_id, SA_a_ptr)
{}

Star(T *l_ptr, SA_attributes *SA_a_ptr)
               :Rval<s, T>(l_ptr, SA_a_ptr),
                Star_intern(l_ptr, SA_a_ptr),
                Rval_intern(l_ptr, SA_a_ptr)
{}

Star(unsigned p_id)
               :Rval<s, T>(p_id),
                Star_intern(p_id),
                Rval_intern(p_id)
{}

Star(T *l_ptr)
               :Rval<s, T>(l_ptr),
                Star_intern(l_ptr),
                Rval_intern(l_ptr)
{}

Star(SA_attributes *SA_a_ptr)
               :Rval<s, T>(SA_a_ptr),
                Star_intern(SA_a_ptr),
                Rval_intern(SA_a_ptr)
{}

Star()
               :Rval<s, T>(),
                Star_intern(),
                Rval_intern()
{}

//conversion to Remote and Spread pointers

Star<s, T>& to_Remote(Star<s, T>& sp){
               ptr_type=R;
               return sp;
}

Star<s, T>& to_Spread(Star<s, T>& sp){
               ptr_type=S;
```

```
                        return sp;
                }


                Rval<s, T> operator++(){

                   return Rval<s, T>
                                (incr_pre(s, sizeof(T)));
                }

                Rval<s, T> operator++(int){

                   return Rval<s, T>
                                (incr_post(s, sizeof(T)));
                }

                Rval<s, T> operator--(){

                   return Rval<s, T>
                                (decr_pre(s, sizeof(T)));
                }

                Rval<s, T> operator--(int){

                   return Rval<s, T>
                                (decr_post(s, sizeof(T)));
                }

                Rval<s, T> operator+=(unsigned i){

                   return Rval<s, T>
                                (incr_asgn(i, s, sizeof(T)));
                }

                Rval<s, T> operator-=(unsigned i){

                   return Rval<s, T>
                                (decr_asgn(i, s, sizeof(T)));
                }
};


/*SpreadArray*/
template<class T>
class SpreadArray:
        public Star<S, T>, public SpreadArray_intern{

   public:
                SpreadArray(unsigned *sd_ptr, unsigned *id_ptr, unsigned sdn,
                                unsigned idn)
                        :Star<S, T>(),
                                SpreadArray_intern(sd_ptr, id_ptr, sdn, idn, sizeof(T)),
                                Star_intern(),
                                Rval_intern()
                {}

      SpreadArray(unsigned d0)
                        : Star<S, T>(),
                          SpreadArray_intern(d0, sizeof(T)),
                          Star_intern(),
                          Rval_intern()
                {}
        SpreadArray(unsigned d0, unsigned d1, unsigned sdn)
                        : Star<S, T>(),
                          SpreadArray_intern(d0, d1, sdn, sizeof(T)),
                          Star_intern(),
                          Rval_intern()
                {}

        SpreadArray(unsigned d0, unsigned d1, unsigned d2, unsigned sdn)
                        : Star<S, T>(),
                          SpreadArray_intern(d0, d1, d2, sdn, sizeof(T)),
                          Star_intern(),
                          Rval_intern()
                {}

        SpreadArray(unsigned d0, unsigned d1, unsigned d2, unsigned d3,
                           unsigned sdn)
                        : Star<S, T>(),
                          SpreadArray_intern(d0, d1, d2, d3, sdn, sizeof(T)),
                          Star_intern(),
                          Rval_intern()
                {}

                SpreadArray(unsigned d0, unsigned d1, unsigned d2, unsigned d3,
                          unsigned d4, unsigned sdn)
                        : Star<S, T>(),
                          SpreadArray_intern(d0, d1, d2, d3, d4, sdn, sizeof(T)),
                          Star_intern(),
                          Rval_intern()
                {}

        SpreadArray(unsigned d0, unsigned d1, unsigned d2, unsigned d3,
                          unsigned d4,  unsigned d5, unsigned sdn)
                        : Star<S, T>(),
                          SpreadArray_intern(d0, d1, d2, d3, d4, d5, sdn, sizeof(T)),
                          Star_intern(),
                          Rval_intern()
                {}

        SpreadArray(unsigned d0, unsigned d1, unsigned d2, unsigned d3,
                          unsigned d4,  unsigned d5, unsigned d6, unsigned sdn)
                        : Star<S, T>(),
                          SpreadArray_intern(d0, d1, d2, d3, d4, d5, d6, sdn, sizeof(T)),
                          Star_intern(),
                          Rval_intern()
                {}

        SpreadArray(unsigned d0, unsigned d1, unsigned d2, unsigned d3,
                    unsigned d4,  unsigned d5, unsigned d6,  unsigned d7,
                          unsigned sdn)
                        : Star<S, T>(),
                          SpreadArray_intern(d0, d1, d2, d3, d4, d5, d6, d7, sdn, sizeof(T)),
                          Star_intern(),
                          Rval_intern()
                {}
};
```

```
/**********************************************
**                                          **
**               Appendix 7.                **
**                                          **
**  Spread data structures and global pointers  **
**                                          **
**             spread.intern.h              **
**                                          **
**********************************************/
/**********************************************
**                                          **
**          Author Jean-Marc Adamo          **
**                                          **
**********************************************/


/*
The classes in this file are not intended to be used by the final user. The
entry points to the Spread and Remote pointers are the classes defined in the
Spread.interface.h file.
*/


struct SA_attributes{
		void *array_base_ptr;
		unsigned spread_dim_nb;
		unsigned intern_dim_nb;
		unsigned total_dim_prod;	//to speed up ref computation
		unsigned intern_dim_prod;	//to speed up ref computation
		unsigned *spread_dim_ptr;
		unsigned *intern_dim_ptr;
		unsigned *index_ptr;
};


class Star_base_intern{

  public: //protected

		unsigned proc_id;
		void *local_ptr;
		SA_attributes *SA_attributes_ptr;

  public:

		Star_base_intern(const Star_base_intern& sp){
			proc_id = sp.proc_id;
			local_ptr = sp.local_ptr;
			SA_attributes_ptr = sp.SA_attributes_ptr;
		}

		Star_base_intern(unsigned p_id, void *l_ptr, SA_attributes *SA_a_ptr){
			proc_id = p_id ;
			local_ptr = l_ptr;
			SA_attributes_ptr = SA_a_ptr;
		}

		Star_base_intern(unsigned p_id, void *l_ptr){
			proc_id = p_id ;
			local_ptr = l_ptr;
```

```
                SA_attributes_ptr = 0;
        }

        Star_base_intern(unsigned p_id, SA_attributes *SA_a_ptr){
                proc_id = p_id ;
                local_ptr = 0;
                SA_attributes_ptr = SA_a_ptr;
        }

        Star_base_intern(void *l_ptr, SA_attributes *SA_a_ptr){
                proc_id = PROC_NB;
                local_ptr = l_ptr;
                SA_attributes_ptr = SA_a_ptr;
        }

        // proc_id to Star_base_intern conversion
        Star_base_intern(unsigned p_id){
               proc_id = p_id ;
               local_ptr = 0;
               SA_attributes_ptr = 0;
        }


        // void * to Star_base_intern conversion
        Star_base_intern(void *l_ptr){
               proc_id = PROC_NB;
                       local_ptr = l_ptr;
               SA_attributes_ptr = 0;

        }

        // SA_attributes_ptr to Star_base_intern conversion
        Star_base_intern(SA_attributes *SA_a_ptr){
               proc_id = PROC_NB;
               local_ptr = 0;
                       SA_attributes_ptr = SA_a_ptr;
        }

        Star_base_intern(){
               proc_id = PROC_NB;
               local_ptr = 0;
                       SA_attributes_ptr = 0;
        }


        unsigned to_proc_id (){
                       return proc_id;
        }

        void *to_local_ptr (){
                       return local_ptr;
        }

        SA_attributes *to_SA_attr_ptr (){
                       return SA_attributes_ptr;
        }


};
```

```
class Lval_intern: public Star_base_intern{

  public:

            Lval_intern(unsigned p_id, void *l_ptr, SA_attributes *SA_a_ptr)
                :Star_base_intern(p_id, l_ptr, SA_a_ptr)
            {}

            Lval_intern( const Lval_intern& lv)
                :Star_base_intern(lv.proc_id, lv.local_ptr, lv.SA_attributes_ptr)
            {}

            Lval_intern(){}

};



class Rval_intern : public Star_base_intern{

  public:

            Rval_intern(const Rval_intern& rv)
                   :Star_base_intern(rv){}

            Rval_intern(unsigned p_id, void *l_ptr, SA_attributes *SA_a_ptr)
                         :Star_base_intern(p_id, l_ptr, SA_a_ptr){}

            Rval_intern(unsigned p_id, void *l_ptr)
                         :Star_base_intern(p_id, l_ptr){}

            Rval_intern(unsigned p_id, SA_attributes *SA_a_ptr)
                         :Star_base_intern(p_id, SA_a_ptr){}

            Rval_intern(void *l_ptr, SA_attributes *SA_a_ptr)
                         :Star_base_intern(l_ptr, SA_a_ptr){}

            Rval_intern(unsigned p_id)
                         :Star_base_intern(p_id){}

            Rval_intern(void *l_ptr)
                         :Star_base_intern(l_ptr){}

            Rval_intern(SA_attributes *SA_a_ptr)
                         :Star_base_intern(SA_a_ptr){}

            Rval_intern()
                         :Star_base_intern(){}

            unsigned equal(Rval_intern sp, enum GPtrTyp GPType);
            unsigned greater(Rval_intern sp, enum GPtrTyp GPType);
            unsigned greater_equal(Rval_intern sp, enum GPtrTyp GPType);
            unsigned less(Rval_intern sp, enum GPtrTyp GPType);
            unsigned less_equal(Rval_intern sp, enum GPtrTyp GPType);
            unsigned diff(Rval_intern sp, enum GPtrTyp GPType);

            Rval_intern plus_i(unsigned i, enum GPtrTyp GPType, unsigned size);
            Rval_intern minus_i(unsigned i, enum GPtrTyp GPType, unsigned size);

      Lval_intern
                        access(unsigned *idx_ptr, enum GPtrTyp GPType, unsigned size);
      Lval_intern
```

```
                    access(unsigned i0, enum GPtrTyp GPType, unsigned size);

        Lval_intern
                    access(unsigned i0, unsigned i1,
                                        enum GPtrTyp GPType, unsigned size);
        Lval_intern
                    access(unsigned i0, unsigned i1, unsigned i2,
                                        enum GPtrTyp GPType, unsigned size);
        Lval_intern
                    access(unsigned i0, unsigned i1, unsigned i2, unsigned i3,
                                         enum GPtrTyp GPType, unsigned size);
        Lval_intern
                    access(unsigned i0, unsigned i1, unsigned i2, unsigned i3,
                             unsigned i4, enum GPtrTyp GPType, unsigned size);
        Lval_intern
                    access(unsigned i0, unsigned i1, unsigned i2, unsigned i3,
                            unsigned i4, unsigned i5,
                                         enum GPtrTyp GPType, unsigned size);
        Lval_intern
                    access(unsigned i0, unsigned i1, unsigned i2, unsigned i3,
                            unsigned i4, unsigned i5, unsigned i6,
                                         enum GPtrTyp GPType, unsigned size);
        Lval_intern
                    access(unsigned i0, unsigned i1, unsigned i2, unsigned i3,
                            unsigned i4, unsigned i5, unsigned i6, unsigned i7,
                                         enum GPtrTyp GPType, unsigned size);


            void ref_comp(unsigned *idx_ptr, Rval_intern *rval_ptr, unsigned size);
            void coord_to_ref(unsigned *idx_ptr, unsigned size);
            void rank_to_coord(unsigned rank);
            unsigned coord_to_rank(unsigned *idx_ptr);
            int ref_to_rank(unsigned size);

};


class Star_intern: public virtual Rval_intern{

  public:


            Star_intern(const Rval_intern& rv)
                :Rval_intern(rv){}

            Star_intern(unsigned p_id, void *l_ptr, SA_attributes *SA_a_ptr)
                    :Rval_intern(p_id, l_ptr, SA_a_ptr){}

            Star_intern(unsigned p_id, void *l_ptr)
                    :Rval_intern(p_id, l_ptr){}

            Star_intern(unsigned p_id, SA_attributes *SA_a_ptr)
                    :Rval_intern(p_id, SA_a_ptr){}

            Star_intern(void *l_ptr, SA_attributes *SA_a_ptr)
                    :Rval_intern(l_ptr, SA_a_ptr){}

            Star_intern(unsigned p_id)
                    :Rval_intern(p_id){}

            Star_intern(void *l_ptr)
                    :Rval_intern(l_ptr){}
            Star_intern(SA_attributes *SA_a_ptr)
                         :Rval_intern(SA_a_ptr){}

            Star_intern()
                         :Rval_intern(){}

      Rval_intern incr_pre(enum GPtrTyp GPType, unsigned size);
      Rval_intern incr_post(enum GPtrTyp GPType, unsigned size);
      Rval_intern decr_pre(enum GPtrTyp GPType, unsigned size);
      Rval_intern decr_post(enum GPtrTyp GPType, unsigned size);
      Rval_intern incr_asgn(unsigned i, enum GPtrTyp GPType, unsigned size);
      Rval_intern decr_asgn(unsigned i, enum GPtrTyp GPType, unsigned size);

};


class SpreadArray_intern: public virtual Star_intern{

   public:

      SpreadArray_intern(unsigned *sd_ptr, unsigned *id_ptr, unsigned sdn,
                                                   unsigned idn, unsigned size);
      SpreadArray_intern(unsigned d0, unsigned size);
      SpreadArray_intern(unsigned d0, unsigned d1, unsigned sdn,
                                                                 unsigned size);
      SpreadArray_intern(unsigned d0, unsigned d1, unsigned d2,
                                                       unsigned sdn, unsigned size);
      SpreadArray_intern(unsigned d0, unsigned d1, unsigned d2, unsigned d3,
                                                       unsigned sdn, unsigned size);
      SpreadArray_intern(unsigned d0, unsigned d1, unsigned d2, unsigned d3,
                                           unsigned d4, unsigned sdn, unsigned size);
      SpreadArray_intern(unsigned d0, unsigned d1, unsigned d2, unsigned d3,
                            unsigned d4,  unsigned d5, unsigned sdn, unsigned size);
      SpreadArray_intern(unsigned d0, unsigned d1, unsigned d2, unsigned d3,
              unsigned d4,  unsigned d5, unsigned d6, unsigned sdn, unsigned size);
      SpreadArray_intern(unsigned d0, unsigned d1, unsigned d2, unsigned d3,
              unsigned d4,  unsigned d5, unsigned d6,  unsigned d7, unsigned sdn,
                                                                 unsigned size);

      virtual ~SpreadArray_intern();

};
```