ARCH, An Object-Oriented Library for Asynchronous and Loosely Synchronous System Programming

Jean-Marc Adamo*

Cornell Theory Center
Ithaca, NY 14853-3801

Technical Report CTC95TR228,
Revised

January 14, 1996

abstract

ARCH is a C++-based library for asynchronous and loosely synchronous system programming. The current version offers a set of programming constructs that are outlined below:

- **Threads**. *The construct is presented as a class from which the user can derive his own classes. The class encapsulates a small set of status variables and offers a set of functions for declaration, initialization, scheduling, priority setting, yielding and stopping.*
- **Processes**. *A process is a more regular and structured programming construct whose scheduling and termination obey additional synchronization rules. Together with the synchronous point-to-point communication system offered in the library (see below), processes favor a parallel programming style similar to OCCAM's (actually, an extension of it that removes most static features and allows processes to share data). The semantics of this model is well understood and will undoubtedly facilitate the development of correct large asynchronous code. The library has been designed so that the C++ compiler is able to check the static semantics of programs (complete type checking, send-recv correct matching, ...).*

*Also a Professor of Computer Science at Université Claude-Bernard, Lyon, France.

- **Synchronous communication**. *Threads and processes synchronize and communicate via communication channels. There are four types of communication channels for local or remote synchronization or synchronous point-to-point communication. Inter-processor channels are essentially tools for building virtual topologies. The channel classes offer functions to send to or receive from a channel and get the size of the latest received message. More specialized synchronization-communication tools can be derived from channels.*

- **Global data and pointers**. *Beside threads, the library offers basic tools for developing distributed data abstractions. Global data are data that can be defined at given locations in the distributed memory but are visible from all processors. Global pointers are a generalization of C++ pointers that allow for addressing global data at any place over the distributed memory. As usual pointers, global pointers are subjected to arithmetic and logic manipulations (incrementation, dereferencing, indexing, comparison...). The library provides basic operators for global data and pointer definition.*

- **Global read/write functions**. *Global pointer expressions provide global references over the distributed memory that can subsequently be used as arguments to global read/write functions. These functions allow the processors to get access to all global data regardless of their locations over the distributed memory. In their most complete form, the read/write functions operate as remote procedure calls. At the programmer's level, global read/write functions appear as "one-sided": a read/write operation is executed on the processor that needs to read/write global data but need not be explicitly handled by the processor associated to the memory holding the data.*

- **Spread and remote Arrays**. *Two basic distributed data structures have been built in the library. Spread arrays are arrays that have some of their dimensions spread over the distributed memory according to a given policy. Remote arrays are arrays that are defined at a given place in the distributed memory but can be accessed from any other. The spread and remote array classes (*SpreadArray *and* RemoteArray*) provide functions for global reference calculation. Global references can subsequently be used as arguments to global read/write functions. One can specialize global pointers to operate on spread or remote arrays. The global pointer class (*Star *class) offers distinct arithmetic and logic operator sets for unassigned, spread and remote global pointers.*

The library encourages parallel code writing in a style that relies on the object-oriented approach: first, build the abstractions that the application at hand relies on; next, make an efficient implementation of the abstractions; and finally, develop the application on top of them. The abstractions can be distributed data types derived from those built in the library (spread and remote arrays: see code of the segmentation algorithm provided with the library) or new distributed types built in the same way or types reused from other applications. This approach should favor parallel code production with many desirable properties such as efficiency, portability, reusability, ...

The library uses MPI as a communication interface. The current implementation runs on the IBM-SP2. Two versions of the library have currently been released. The first one is based on the IBM C++ compiler and MPI library. The second one makes use of the GNU g++ compiler and the MPICH public domain version of MPI. Porting the latter to any parallel machine supporting these two software systems should be straightforward.

Contents

1 Introduction

1.1 Parallel code writing

Several types of tools can be used to write parallel codes. One can use a pre-defined language: this is the right thing to do when the requirements of the application at hand fit into the programming model the language relies on. For instance, HPF-fortran[18] can be used to produce efficient codes for algorithms processing highly regular arrays but is not well suited to algorithms processing irregular data structures or capable of producing irregular data structures dynamically. LPARX[5] proposes to extend the validity of the model to algorithms processing irregularly spread arrays and offers tools that partially solve the problem of dynamic load-balancing. Split-C[10, 11] is well suited to coding uni-threaded algorithms but fails to be convenient when it comes to deal with systems presenting complex asynchronous behaviors. OCCAM[19] exactly fits the needs of systems with complex asynchronous behaviors but the language is too closely related to a processor architecture (Transputers).

As an alternative to using fixed languages, I suggest a pragmatic approach that relies on the use of a library offering basic tools for writing asynchronous and loosely synchronous programs. These two classes are large enough to cover most interesting algorithms that do not have special requirements as time constraints for example.

The library should also meet the following requirements:

1. Portability.

   The library should allow one to write portable programs. This means that the library is itself easily portable and that, once the library is ported to a platform, the codes written there can readily be compiled and run on any other.

2. Scalability.

   We mean here scalability at the software level. this means that programs are able to run on machines of any size (any number of processors), so the user can find out the right size to execute on. The complete notion of scalabilty, meaning scalabilty at the performance level, is a much more difficult issue. It is just impossible or limited for some algorithms. For others, assuming very simplistic models of parallel machines, results show [4] that the notion is only asymptotically meaningful and of no practical use for machines of today.

3. Extensibility.

The library should allow the programmer to complement the existing constructs with new programming abstractions that fit closely the needs of the application at hand. In this view, code writing can be seen as an application-centered process which consists first in developing the appropriate abstractions (distributed data types) the application relies on and next in writing the application code.

4. Reusability.

   The programming constructs, as well as the general framework, should enable the production of reusable components as much as possible. This is a key issue. Writing parallel programs often turns out to be a difficult task. It would undoubtedly save much time if standard components could be reused.

5. Code checking.

   The programming constructs should be built so the compiler is able to check their static semantics as much as possible. This should be fully compatible with the extensibility requirement and apply not only to the library code but to the user code as well.

1.2 Identifying the programming constructs

I propose to perform this work on a segmentation algorithm by region growing that presents both asynchronous and loosely synchronous behavior, hence enough variety to serve our purpose. Although the algorithm is borrowed from a specific application domain, the materials drawn from the discussion below in fact only relies on the asynchronous or loosely synchronous nature of processes involved. They consequently apply to any other application domain. The algorithm consists of three phases described below.

1.2.1 Segmentation algorithm by region growing

1. First phase:
   - input: array of pixels,
   - output: set of quadtrees whose leaves are homogeneous regions.
2. Second phase:
   - input: set of quadtrees,
   - output: connectivity graph G.

G is a graph whose vertices are homogeneous regions such that region i is connected to region j if and only if i and j are neighbors and the new (possibly non-square) region formed from the union of i and j satisfies the homogeneity criterion.

3. Third phase:
   - input: connectivity graph G,
   - output: reduced connectivity graph Gr.

Each step of the reduction process applies the following procedure:

- edge(i, j) is assigned a weight wij that measures the homogeneity property of the region formed by the union of i and j.
- each vertex selects the edge with the smallest weight. In case of ambiguity, the edge corresponding to the smallest connected vertex is selected.
- when a pair of vertices selects the same edge, the one with the smaller identifier absorbs the other and the weight on the new edges are re-computed accordingly.

1.2.2 Parallelizing the first phase

We start by conveniently arranging the set of processors into a grid and by dividing the image space into square tiles so that many tiles are mapped onto each processor and all the processors get the same number of tiles. In the first phase, the processors build the same number of quadtrees (one quadtree per image tile) independently from one another. The computing load is the same everywhere since for each image tile each processor needs to reach all pixels to form the seed homogeneous square regions.

1.2.3 Parallelizing the second phase

The first phase is expected to produce a set of quadtree pages whose leaves are the seeds from which the regions will grow. The regions can irregularly be spread over the image space, so some quadtrees can have many leaves and other a very small number. The variation of the quadtree page size may be large, as the number of leaves in a tree varies exponentially as a function of the height. Consequently, at the end of the first phase, the load can be very irregularly spread over the processors. Let us make the following observations:

1. We will need to balance the load in the second phase so no processor is kept idle while others are swamped.
2. In the process of edge building, a processor will sometimes need to read quadtree pages located on another processor (neighboring quadtree pages).

3. Each node is expected to interleave several activities: computing the edges for the leaves of a given quadtree page, reading the required neighboring pages and waiting for their completion, addressing load-balancing requests (whenever load is balanced dynamically), and replying to neighboring page requests, replying to load balancing requests.

Load balancing issues There are two possible options, static and dynamic load balancing. With the first scheme, the image is first pre-analyzed. This is generally performed in interaction with the user and leads to a spreading decision that consists in some irregular placement of the tiles over the nodes. Dynamic load balancing solves the problem on the fly.

Static load balancing has several weaknesses: the processing time can be large, especially if reaching the spreading decision needs user participation. The load balancing problem is not really solved but just moved from time to space. The scheme is no longer applicable to algorithms producing unbalanced data dynamically that need to be balanced. For instance, the graph reduction problem in the third phase belongs to that category.

Dynamic load-balancing provides a much more satisfactory solution but needs more sophisticated software support. Dynamic load balancing also provides an automatic solution to static load balancing in the sense that the place where the load is processed is the very place to which it could have been assigned by a static scheme. As far as efficiency is concerned, dynamic load balancing will undoubtedly do better than any static scheme requiring user participation. Finally, dynamic load balancing is the only possible solution to deal with algorithms that produce unbalanced data dynamically.

Global data Either in the load balancing procedure or in the procedure that computes the edges, any node needs to get access to tiles located on other nodes. Things would be much simpler if the tiles could be accessed symbolically as items of some global structure over the nodes and transparently (that means without any explicit participation of the node holding it (i.e. at the programmer level)).

Interleaved activities issues There are two possible options. The first one consists in making use of a threading system. In this case it will be quite easy to design the program code as a system of concurrent processes that communicate and synchronize as shown in Fig. 1. The second option is to proceed sequentially. This is possible at the expense of expanding the asynchronous behavior of the nodes as a large system of guarded commands. The approach requires the user to carry out a complete analysis of all combinations of events for which some action is to be performed and then writing down the system of guarded commands. This is boring, time consuming, error prone and does not favor producing reusable components. By contrast, writing a system of concurrent processes is much simpler and reusable. For instance, the load balancing server in Figure 1 can be written once and for all, encapsulated and reused with other algorithms requiring the same load balancing scheme.

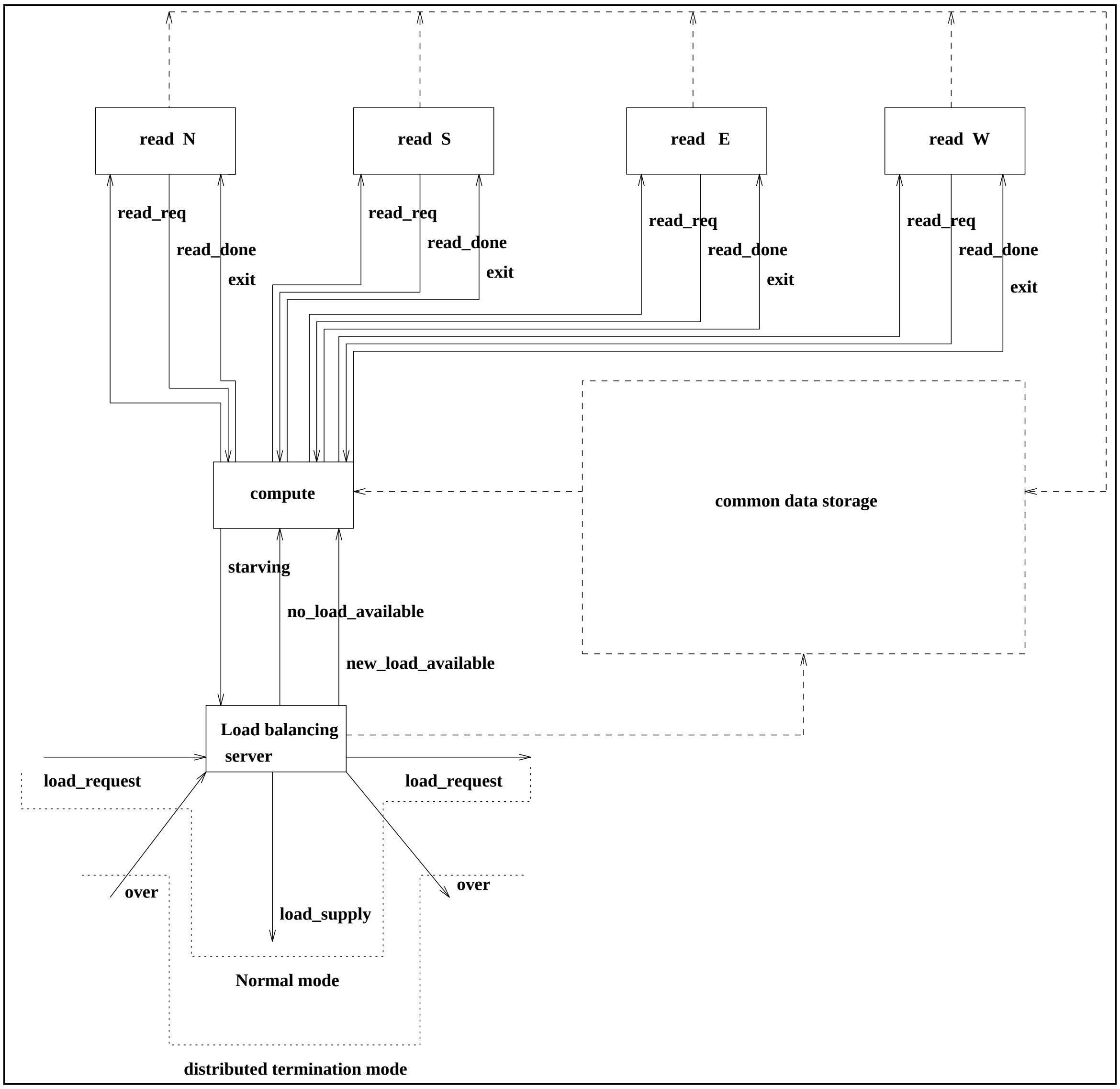

Figure 1: *A set of concurrent processes*

1.2.4 Parallelizing the third phase

The second phase produces a linked list of square regions on each node (vertices of the region connectivity graph) each identified by a global rank. Each vertex points to a local array of edges, each identified by the global rank of the region at the other end. The first step in the third phase consists in linking all pairs of corresponding edges (i.e. (i,j) on node k and (j,i) on node l), via a global array spread over the nodes. Since vertex ranks are global information, each entry of the global array is known in any node as long as each node knows how spreading the array was performed. In the next step, the graph reduction loop is started in which the vertices communicate by (remotely) reading and writing information in the global array. The operations performed in each reduction step are as follows:

- For each vertex (i), the entries of the local edge array are examined, a vertex (i,j) is selected and a mark is placed in the (remote) global array slot (i,j).
- A barrier synchronization is executed to ensure that writing all the marks out is complete before the nodes start reading them again.
- Each vertex (i) that selected edge (i,j) read slot (i,j) of the global array to know whether vertex j made a matching choice.
- The algorithm checks for termination that occurs when no matching was found out in the previous step. (this is performed by using a global reduction operation.)
- If termination did not occur, each vertex knows whether it is 'involved or not' in a match and whether it is an 'absorbing' or 'absorbed' vertex in the positive case.
- Vertex reduction is performed on each node: For each absorbing vertex, the corresponding absorbed one is read, edge arrays are merged, and data are updated both in the vertex records, the edge records and the spread array. This operations make use of global pointers stored in the spread array.
- In each reduction step, the set of vertices is checked for approximate uniform repartition. This is done by computing a dispersion index via global reduction performed over the processors. If the index gets greater than a given threshold, load balancing is executed and some vertices get transferred from the overloaded nodes to under-loaded ones.

Let us make the following observations:

1. The algorithm above outlined typically is a loosely synchronous algorithm. The steps constantly apply the following pattern: computation $->$ remote read/write operations in a volley $->$ barrier synchronization $->$ computation ...

2. Here again we need a global data structure spread in some way over the nodes and one-sided operations capable of reading from and writing to this data structure.

3. The load-balancing scheme applied in this phase is quite different from the one that was applied in the second phase. Requests in the latter are individually triggered by the nodes discovering they are going to get starve and there is a mechanism distributed over the nodes for the load searching process to be carried out. In the former, load-balancing is decided at the global level by examining all local situations. Performing load-balancing is not interleaved on the nodes with any other activity (because that does not make any sense). Finally, as non-uniform spreading of vertices is dynamically generated, load balancing cannot be performed statically.

4. The load-balancing algorithm also performs loosely synchronously (see details in the source code provided with the library). The code shows how convenient it is for a node to get access, symbolically and transparently (one-sided access), to the global data defined on other nodes. These data are not spread over the nodes but just accessed remotely, from nodes to others.

1.2.5 What have we learned?

It comes out from the above discussion that a library for asynchronous and loosely synchronous system programming should provide: Threads, programming constructs for threads to communicate and synchronize, spread data structures, remote data structures, programming constructs for symbolic and transparent access to spread and remote data. We require additionally that the threads and synchronization constructs be both parts of a coherent and well understood model of parallel-system programming. This is roughly what the ARCH library offers.

1.3 Outlining the ARCH library

1.3.1 Asynchronous system programming

Threads and processes The thread construct is presented as a class from which the user can derive his own classes. A thread encapsulates a set of data: stack space, stack size, stack pointer, entry-point pointer, and priority level and two booleans indicating whether floating point registers need to be saved or not on context switching and defining what the current status of the thread is (virgin or already been run). The *Thread* class offers a small set of functions for: thread declaration, initialization, scheduling, priority setting, yielding and stopping.

Threads are rather primitive constructs. The library offers a more structured but also more constrained programming construct encapsulated in the *Process* class. A process is a thread whose activation and end are both handled by a special function: *par*. *par* synchronizes both activation and end of process sets. Used in conjunction with the synchronous message passing primitives outlined below, processes allow parallel code writing in the OCCAM programming style.

Synchronous message passing Synchronous message passing relies on the use of special data structures called channels. These data structures are encapsulated in four generic classes:

- class ChanL;
- template<class T> class ChanD;
- class ChanR;
- template<class T> class ChanRD;

that are respectively devoted to: local pure synchronization, local synchronous communication, remote pure synchronization, remote synchronous communication. The ChanR and ChanRD channels actually are tools for building virtual communication networks over the processor set.

Send and receive functions are supplied in each class for synchronization or synchronous message passing. In addition to the receive function, the library supplies a function for receiving non deterministically on sets of channels.

Global data These data are defined at the places where their declaration as global data occurs and are accessible from any processor. According to the way it is declared, a global data can be accessed either directly or via the execution of a remote procedure that computes its address at the remote location. Communication channels implicitly are global data, declaring a channel is implicitly considered as declaring it as a global data.

Remote read/write functions These functions are generic functions that can be called on a node to get access to a data defined as global. According to the arguments that are passed to it the function can get access to the remote data directly or via the execution of a remote procedure. According to the arguments, the completion of a remote read/write function is monitored either explicitly by the user or implicitly by the local scheduler.

Spread arrays and spread pointers A spread array is a multidimensional array that has some of its dimensions distributed over the set of nodes. 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 node memory set. Spread arrays are instances of a generic *SpreadArray*<*Type*> class that offers a set of functions: construction, destruction, indexing, status checking. Spread arrays implicitly are global data. Declaring a spread array is considered as implicitly declaring the data items it is composed of as global.

One can get access to spread array data directly via the spread array name or via spread pointers. Spread pointers are generalization of C pointers that can address data everywhere on the parallel machine. The library supplies a generic class with functions for spread pointer manipulation: construction, incrementation (+, -, ++, –, +=, -=), indexing, dereferencing, spread-pointer based predicate evaluation, access to object members. Global read/write functions can be applied to spread arrays in the same way as that we described earlier (see ‘Remote read/write functions’ above).

Remote arrays and remote pointers A remote array is a multidimensional array that has been declared on some node but can be remotely accessed from any node. A remote array is defined by the type of data items it contains and the list of its dimensions. Remote arrays implicitly are global data, declaring a remote array is considered as implicitly declaring as global the data it is composed of. Spread arrays are instances of a generic $RemoteArray< Type >$ class that offers a set of functions for: construction, destruction, indexing, status checking.

The library supplies remote pointers that can be used for accessing remote array or global data. The difference between remote and spread arrays lies into the arithmetic of incrementation functions, indexing and predicate evaluation, which is the usual one. Global read/write functions can be applied to remote arrays in the same way as to spread arrays.

1.3.2 Loosely synchronous system programming

The library offers the same programming constructs as those described earlier: point-to-point message passing, spread/remote arrays, spread/remote pointers, remote read/write functions. The difference lies into the following: loosely synchronous system are uni-threaded systems and the user is expected to handle on his own the completion of the communication functions. The library offers tools that makes it possible to do it in a standard way.

Of course it can be argued that loosely synchronous systems are special instances of asynchronous ones. Specialized tools were developed for them essentially for efficiency reasons.

1.3.3 Compatibility with MPI

The library is built upon MPI. All MPI functions can therefore be used in codes written with the library, specially all functions performing collective operations (barrier, broadcasting, gathering, scattering, reducing,...) that are essential to loosely synchronous systems. Some examples can be found in the source code of the region growing algorithm (third phase).

1.4 How to avoid reading this documentation

I suggest you first read the introduction, next you look at the program samples: they cover most useful programming constructs. Then, you use the index to go to the details.

2 Threads and processes

2.1 Threads

A thread may be a user thread or a special pre-defined one denoted as *main_thread*. Threads are all instances of the *Thread_base* class which provides the minimal data structure assigned to all threads namely: the current value of the stack pointer, and a boolean indicating whether floating point registers need to be saved or not on context switching and another boolean defining what the current status of the thread is (virgin or already been run). *Thread_base* additionally provides an internal basic procedure that performs *context switching*. In the current implementation this function comes from the *NewThreads* package. It has been setup to run on RS/6000 systems but could be run as well on many other computer architectures which makes the ARCH library readily portable on any multiprocessor of the market place. The procedure has been modified at several places to suit the needs of the ARCH library (see the *A.cswitch.cc* file where all modification have been marked). *Thread_base* has two derived classes: *Main_thread* (pre-defined ARCH thread) and *Thread* (for user in thread derivation).

The context switching procedure maintains the built-in pointer *current_thread_ptr* that permanently holds the pointer to the thread that currently run the cpu.

2.1.1 Main_thread class

This class is a pre-defined one intended to be instantiated only once. It essentially 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:*main_thread*, that is run each time there is no user thread in the scheduling queues. This occurs on the nodes when the local processes need an external event to occur (communication completion) to resume execution. *main_thread* is then kept polling until one of the queues become non empty. There are two scheduling queues one for low the other for high priority threads. *main_thread* only polls and keeps 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 that started it).

The *Main_thread::exit()* function is intended to cause the scheduling machinery termination. It is silently executed at the end the root thread (starting thread of a user program) to cause a return to the main function.

Neither *Main_thread::start_keep_up_and_terminate()* nor *Main_thread::exit()* needs be explicitly handled by the final user as long as he uses the standard programming framework defined in the *A.environment* file. This file contains a macro that defines the pre-defined *main()* function. The macro is parametrized by the class the root user_process belongs to and the C standard arguments argc and argv. The root class must be derived from the pre-defined Process class (described later on). The pre-defined *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. Completion 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 (i.e. *main_thread*) with a special status that will finally make the thread return from the main function, causing the node termination.

Example This is the main program of the region growing algorithm outlined in the introduction. The user should always follows the order things are written below (one should also take a look at the *A.environment* file to see how this actually is expanded):

```
/*
** inclusion of the A.environment file (brings down all the library stuff)
*/
#include "A.environment"

/*
** user's root process definition
*/
class RGrowing_proc: public Process{

   public:
      RGrowing_proc()
      :Process((Pmf0_ptr)&RGrowing_proc::main_procedure){}

      void main_procedure();
};

/*
** main function defined in the environment file is expanded here
*/
```

```
MAIN(RGrowing_proc, argc, argv)


/*
** rest of user program begins here
*/

#include <math.h>

inline
void preempt(){
        error(1);
}

extern void read_image();
extern void write_result();

#include "mpi-lib.C"

#include "const.def"
#include "loc-class-dec"

//global declarations for the region-growing algorithm

Vertex_rcd *vertex_list_ptr = 0;
int vertex_local_nb = 0;

Vertex_rcd *absorbing_vertex_ptr_first = 0;
Vertex_rcd *absorbing_vertex_ptr_last = 0;

Vertex_rcd *absorbed_vertex_ptr = 0;

Vertex_rcd *dead_vertex_ptr = 0;

RW_count *rw_c;

ChanRD_c<Load_offered> *chan_c;

// end of global declarations
```

```
#include "all.h"

SpreadImage<char, C_VPS, C_HPS, C_VIS, C_HIS, C_IBS> *image;
SpreadQuadtree<C_VPS, C_HPS, C_VIS, C_HIS, C_IBS> *quadtree;
SpreadLinkArray<C_VPS, C_HPS> *link_array;


#include "split.C"
#include "tree_to_graph.C"
#include "merge.C"

/*
** root procedure
*/

void RGrowing_proc::main_procedure(){

        rw_c = new RW_count;

        image = new SpreadImage<char, C_VPS, C_HPS, C_VIS, C_HIS, C_IBS>(1);
        read_image();

        quadtree = new SpreadQuadtree<C_VPS, C_HPS, C_VIS, C_HIS, C_IBS>(2);
        barrier();

  //split phase

        split();

        barrier();
        delete image;

  //from tree-to-graph: changing the data representation

        from_tree_to_graph();

        barrier();
        delete quadtree;
```

```
        int size = compute_link_array_size();
        link_array = new SpreadLinkArray<C_VPS, C_HPS>(3, size);
        barrier();

        set_up_global_links();

 //merge phase

        merge();

        barrier();
        delete link_array;

        write_result();
}
```

2.1.2 Thread Class

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, entry-point pointer and priority level (0: high and 1: low). As opposed to tasks provided by the standard C++ library, a thread is not the class constructor. A thread can be any function in a class derived from Thread_base. This allows separating thread creation from thread initialization and execution. This also brings in more flexibility as 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 functions offered are described below.

Constructor

Thread::Thread(void (Thread::*entry_point)(),
    int stack_size=PROC_STACK_SIZE,
    int usesFP=0);

This function essentially performs: stack allocation, stack size storage, entry point storage and from-sons-to-father linking(see *father_ptr* below). *entry point* is a pointer to any member function of the root class (see example above) with type *Tmf0_ptr* with:

typedef void (Thread::*Tmf0_ptr)();

stack_size is the automatic value or any value provided by the user. The PROC_STACK_SIZE value can also be changed by updating the *A.setconfig* file and recompiling the library. A thread is implicitly considered as a non floating point user, unless the user decides otherwise by explicitly executing the constructor with usesFP=1. A Thread holds a built-in pointer *father_ptr* for any son thread/process to access the data in the father space.

Destructor

Thread:: Thread();

just deletes the stack space.

Initialization

void Thread::setup(int priority=1);

sets the thread priority up and initializes the stack pointer value. Priority is set to low unless the user decides otherwise.

Priority level updating

void Thread::setpri(int priority=1);

sets priority of the thread the function is executed for. Priority is set to low unless the user decides otherwise. There are two priority levels low (0) and high (1).

Thread scheduling

void Thread::schedule();

appends the pointer to the thread it is executed for to one the scheduling queue depending on the thread priority. There two scheduling queues each attached to a priority level.

void Thread::schedule(int size, Thread **proc_table) ,

appends the thread pointers in *proc_table* to the scheduling queue. The numberof appended pointers is *size* starting from rank 0 in proc_table.

Thread yielding

void Thread::reschedule();

This is for cooperative multi-tasking. We assumed no hardware mechanism a priori is available to force fair cpu sharing. This is left to the user who is expected, by using this function, to tune the right size of tasks that are to be interleaved by the scheduler (task here means a piece of thread code) .

reschedule() stops executing the thread holding the cpu and appends it to one of the scheduling queue (0 or 1) depending on the current thread priority level.

Thread stopping

void Thread::stop();

Stops the currently running thread and causes switching to the first one available in the scheduling lists or to *main_thread* when the scheduling lists happens to be empty. As it is appended to no scheduling queue the stopped thread won't be resumed, unless voluntarily.

2.2 Process class

As Compared to threads, processes are higher level but also more constrained programming constructs. Processes can be seen as restricted threads, hence the private derivation from the Thread to Process class. A process is a thread whose both activation and end are handled by a special *par* function. The *par* function synchronizes both activation and end of process sets. Used in conjunction with the communication-synchronization primitives described in section 3, processes allows writing parallel code in the OCCAM programming style (without some of its restrictions and the full power of C++ around). The Process class supplies the functions listed below.

Constructor

```
Process::Process(void (Thread::*entry_point)(),
                 int stack_size=PROC_STACK_SIZE,
                 int usesFP=0);
```

The *entry_point* argument has type *Pmf0_ptr* with:

typedef void (Process::*Pmf0_ptr)();

Destructor

Process:: Process();

Initialization

void Process::setup(int priority=1);

Priority level updating

void Process::setpri(int priority=1);

Process yielding

void Process::reschedule();

Parallel process execution

void Process::par(int size, Process **proc_table, int *priority);

This function schedules for execution the set of sub-processes pointed to in *proc_table*. The function completes only when all the sub-processes are completed.

The *priority* argument is meant to be a pointer to an array of priority numbers. priority[i] (0 or 1) sets the priority of process pointed to by proc_table[i].

Example 1 Here is a procedure borrowed from the image segmentation algorithm that was outlined in the introduction. Given a quadtree, the procedure performs several tasks asynchronously on each node: leaf-to-connected-leaf linking, neighbor fetching over the set of local memories and load-balancing. It could, of course, be written as a sequential code at the expense of writing a large guarded command requiring a detailed analysis of all events and combination of events that need some action to be performed. As already mentioned in the introduction, using asynchronous processes not only makes the algorithm design and code writing much easier (see 1) but also favor the writing of reusable code. This is difficult to achieve with guarded commands in which the code strongly depends on the specificity of the system at hand. The code below uses ARCH point-to-point communication that is described in section 3.

```
void make_vertex_rcd_list(int local_max_nb_of_leaves){

  //local_max_nb_of_leaves: leaf max-nb per block over set of local blocks
  //global_max_nb_of_leaves: leaf max-nb per block over set of all spread blocks

   int global_max_nb_of_leaves;
   allreduce_max(&local_max_nb_of_leaves, &global_max_nb_of_leaves);

   //creating buffers and working areas

   Quadtree_rcd *qt_rcd_buff_ptr[5];
   for(int i = 0; i < 5; i++)
      qt_rcd_buff_ptr[i] = new Quadtree_rcd;
```

```
Qt_heap_rcd *qt_heap_buff_ptr[5];
for(i = 0; i < 5; i++)
   qt_heap_buff_ptr[i] =
   new Qt_heap_rcd[(7*global_max_nb_of_leaves*global_max_nb_of_leaves - 1)/3];
   //see comment in  split() function that explains the size used above

Quadtree_rcd *qt_ptr[5];

Neighbor_heap *neighbor_heap_ptr = new Neighbor_heap(global_max_nb_of_leaves);

Edge_heap *edge_heap_ptr = new Edge_heap(global_max_nb_of_leaves);

Load current_load(self_address, 0, 0, (C_VIS/C_IBS)/C_VPS - 1,
                  (C_HIS/C_IBS)/C_HPS - 1 );
Task current_task;

current_load.next_task(&current_task);
             //can't have empty load the first time next_task is
             //executed.  Do not need to check the boolean value
             //returned by the function


//creating communication and synchronization channels

ChanL *read_req[4];
ChanL *exit[4];
ChanL *done[4];
for(i = 0; i < 4; i++){
   read_req[i] = new ChanL;
   exit[i] = new ChanL;
   done[i] = new ChanL;
}

ChanL *starving = new ChanL;

ChanL *new_load_available = new ChanL;

ChanL *no_load_available  = new ChanL;
```

```
ChanRD<int> *load_request = new ChanRD<int>(4, (self_address + 1)%PROC_NB);

ChanRD<Load> **load_supply = new ChanRD<Load> *[PROC_NB - 1];
for(i = 1; i < PROC_NB; i++)
   load_supply[i - 1] = new ChanRD<Load>(i+4, (self_address + i)%PROC_NB);
   //i-1 is the nb of processors between source and target

ChanR *over = new ChanR(PROC_NB+4, (self_address + 1)%PROC_NB);

barrier();

//creating the processes
Process *proc_table[6];

proc_table[0] =
new Compute_proc( &qt_ptr[0], qt_rcd_buff_ptr[4], qt_heap_buff_ptr[4],
                  &current_task,&current_load, neighbor_heap_ptr,
                  edge_heap_ptr, &read_req[0], &done[0], &exit[0],
                  starving, new_load_available, no_load_available);

for(i = 1; i < 5; i++)
   proc_table[i] =
   new Read_proc( i-1, &qt_ptr[i-1], qt_rcd_buff_ptr[i-1],
                  qt_heap_buff_ptr[i-1], &current_task,
                  read_req[i-1], done[i-1], exit[i-1]);

proc_table[5] =
new Load_balancing_proc( &current_load, &current_task, starving,
                          new_load_available, no_load_available,
                          load_request, &load_supply[0]);
//start processing
int pri[6] = {1, 1, 1, 1, 1, 0};
root_proc->par(6, proc_table, pri);

//end processing

if(!self_address)
   //processor 0 plays a special role
   end_monitoring_0(load_request, over);
```

```
   else
      end_monitoring_i(load_request, over);


   //deleting all stuff used in this function

   for(i = 5; i > -1; i--)
      delete proc_table[i];
   delete over;
   for(i = PROC_NB - 1; i > 0; i--)
      delete load_supply[i];
   delete []load_supply;
   delete load_request;
   delete no_load_available;
   delete new_load_available;
   delete starving;
   for(i = 3; i > -1; i--){
      delete done[i];
      delete exit[i];
      delete read_req[i];
   }
   delete edge_heap_ptr;
   delete neighbor_heap_ptr;
   for(i = 4; i > -1; i++)
      delete qt_heap_buff_ptr[i];
   for(i = 4; i > -1; i++)
      delete qt_rcd_buff_ptr[i];
}
```

Example 2 This is intended to give one the flavor of how a process is defined. Again, the example comes from the segmentation algorithm code. Read_proc is instantiated four times in the code given above.

```
class Read_proc: public Process{

      int pid;
      int keep_running;
      Quadtree_rcd **qt_ptr;
      Quadtree_rcd *qt_rcd_buff_ptr;
      Qt_heap_rcd *qt_heap_buff_ptr;
```

```
    Task *current_task;
    ChanL *read_req;
    ChanL *done;
    ChanL *exit;

  public:

    Read_proc(int id,
              Quadtree_rcd  **qt_rcd_ptr2, Quadtree_rcd *qt_rcd_ptr,
              Qt_heap_rcd *qt_heap_ptr, Task *task,
              ChanL *r, ChanL *d, ChanL *e )
    :Process((Pmf0_ptr)&Read_proc::body)
    {
       pid = id;
       keep_running = 1;
       qt_ptr = qt_rcd_ptr2;
       qt_rcd_buff_ptr = qt_rcd_ptr;
       qt_heap_buff_ptr = qt_heap_ptr;
       current_task = task;
       read_req = r;
       done = d;
       exit = e;
    }

    void body();

    void read_branch(int, void *);

    void exit_branch();
};
```

2.3 Scheduler data

All the scheduling functions above perform on two data structure *Sched_data[0]* and *Sched_data[1]* that are created when the *MAIN()* macro is executed (see *A.environment* file). The size of each scheduling queue has a default value: SCHED_QUEUE_SIZE, set in the *A.setconfig* file. The user can change each size dynamically in the *A.environment* file by giving arguments to the *Sched_data()* constructors there or statically by changing the value in *A.setconfig*. the signature of the Sched_data constructor is:

Sched_data(int queue_sz=SCHED_QUEUE_SIZE);

3 Threads/processes synchronization and communication

In the previous section we described a class system relating to thread and process creation, scheduling and termination. The current section is devoted to describing additional classes and functions we need for threads and processes to synchronize and communicate. Thread/process synchronization and communication is performed by executing matching pairs of send-recv functions that are applied to common channels. Channels are similar to OCCAM's, althought their application has been extented here to support both intra- and inter-process communication. Communication is point-to-point and synchronous. The length of messages sent across a channel can vary within limits: 1 and a bound set at channel declaration time. The channels and the send and recv functions are defined in such a way that the C++ compiler is able to check type consistency. Point-to-point communication is ensured by atomic execution of Synchronization-communication functions. non-matching pairs of send-send or recv-recv statements are detected dynamically.

3.1 Synchronization-communication channels

There are four possible channel classes deriving from one another according to Figure 2.

3.1.1 Chans class

This class corresponds to the minimal data structure shared by both local and remote synchronization/communication functions. It is not intended for direct instantiation (the class does not supply any function).

3.1.2 Local channels

ChanL class

Channels of this class can only be used for local, pure, point-to-point synchronization. ChanL is derived from Chans by public derivation.

Constructor

ChanL();

Declaration form

ChanL name;

Creates a new channel with name *name* and type *ChanL*.

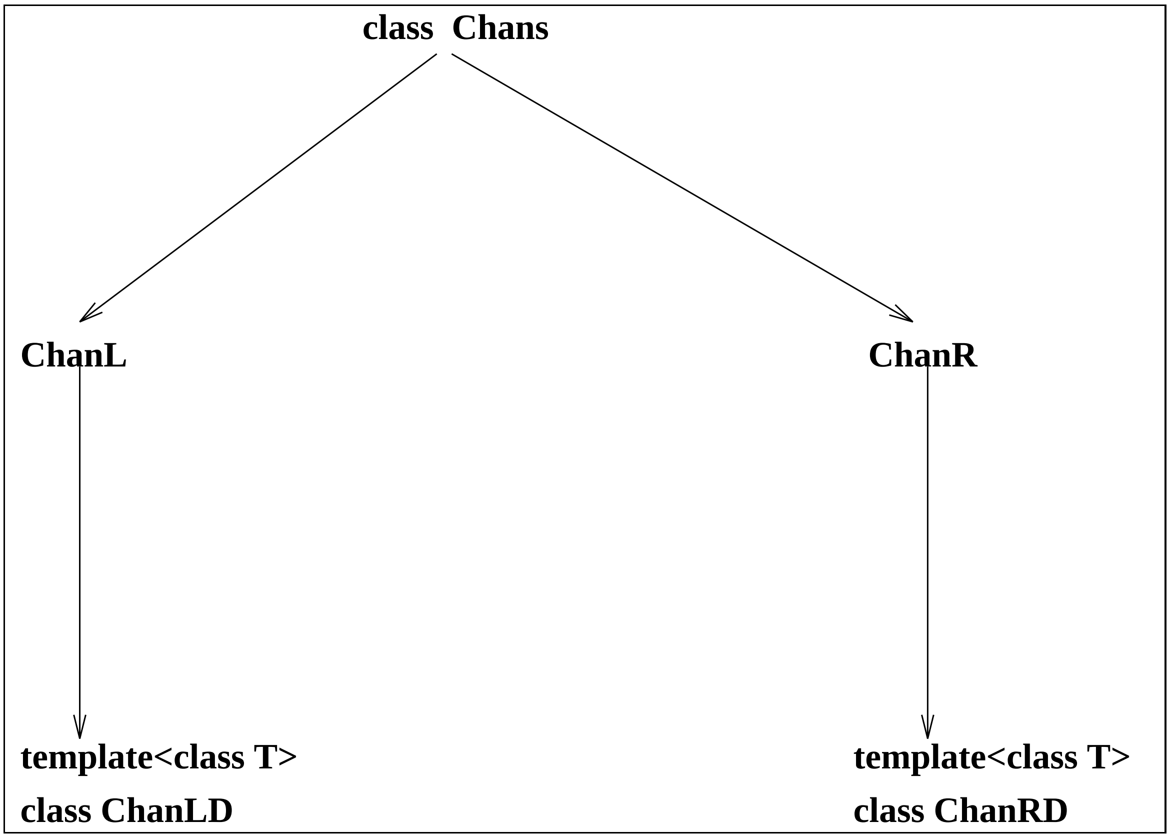

Figure 2: *Communication channels*

Synchronization functions

The class offers a pair of send-recv functions that can only be used for pure synchronization purposes:

void ChanL::send();

void ChanL::recv();

The first process in a pair that executes send/recv on a *chanL* gets descheduled. It is resumed by the other process executing the matching send/recv function that just pursues executing.

ChanLD class

This class allows both local point-to-point synchronization and message passing from sender to receiver (running on the same processor).

Constructor

```
template<class T>
ChanLD::ChanLD(int max_data_length = 1);
```

Declaration forms

```
ChanLD<type> name (length);
ChanLD<type> name;
```

Both declarations create a local channel for synchronous message passing with name *name* and type $ChanLD < type >$. If present, the argument indicates the maximum length of T-typed items this channel can carry. Default is 1.

Communication/synchronization functions

```
template<class T>
void ChanLD::send(T *from_ptr, int l = 1);

template<class T>
void ChanLD::recv(T *to_ptr);
```

The first function sends messages via the channel it is attached to. *from_ptr* is the message source. *l* is the number of T-typed items the sent message can carry. the *l* value is checked against the maximum data length value attached to the channel.

The second function performs receptions through the channel it is attached to. the *to_ptr* parameter is the message destination. One can know about the length of the last received message by executing the *get_data_size()* function below.

The first process in a pair that executes send/recv on a *chanLD* gets descheduled. It is resumed by the other process executing the matching send/recv function. The message from send to recv is passed before the first process is resumed and the second pursues executing.

```
template<class T>
int ChanLD::get_data_size();
```

This function returns the number of T-typed items composing the latest received message.

3.1.3 Remote channels

The remote communication channels are tools for building virtual communication networks over the processor set. One should not think of these channels as for building communication links between individual pairs of processors but virtual communication networks.

For instance, take the image segmentation algorithm. The load balancing procedure executing in its second phase (tree-to-graph transformation) needs a ring for sending and transmitting requests on, and a complete graph for sending information about load assignment. These two communication structures can be defined as shown in the following example.

Example

1. ChanRD<int> *load_request =
   new ChanRD<int>(4, (self_address + 1)%PROC_NB);

2. ChanRD<Load> **load_supply =
   new ChanRD<Load> *[PROC_NB - 1];
   for(i = 1; i < PROC_NB; i++)
      load_supply[i - 1] =
      new ChanRD<Load>(i+4, (self_address + i)%PROC_NB);

In the first example, the *new* function is executed on each node. a new remote channel is then created on node n that declares node (n+1)%PROC_NB as its target (PROC_NB is the total number the parallel program is run on and self_address stands for the rank of the local processor). This actually amounts to a ring construction.

In the second example, a table of (PROC_NB-1) remote channels pointers is created on each node. Then a new remote channel is created for each table slot. for i in the range [1, PROC_NB], remote channel (i-1) on node n declares node (n+i)%PROC_NB as its target. This actually amounts to building a communication network that is a complete graph over the set of nodes.

Both examples relate to constructing regular structures. What about irregular ones. Given some irregular structure, one possible solution consists of building the smallest regular one that contains it. Another is to deal with irregularity by using C++ conditional statements.

ChanR class

Using a *ChanR* channel makes it possible to synchronize processes running on separate processors.

Constructor

ChanR(int tag, int remote_proc);

Declaration forms

ChanR name (tag, proc_nb);

This declares a channel with name *name* and type *ChanR*. A *ChanR* channel is considered as attached to a Global-typed data (see section 4), so it is given a tag. *tag* is a strictly positive integer. The second argument is an integer identifying the target processor. The virtual communication network generated by a *ChanR* corresponds to a directed graph whose any pair of nodes is connected by only one edge at most.

Synchronization functions

template<class T>
void ChanR::send();

template<class T>
void ChanR::recv();

The first function sends synchronization signals via the channel it is attached to. The second performs reception of such signals. There can only be one send on one side and one recv on the other operating on a *ChanR* at a time.

The first process in a pair that executes send/recv on a *chanR* gets descheduled. It is resumed by the remote process executing the matching send/recv function that pursues executing.

ChanRD class

This class allows both point-to-point synchronization and message passing from sender to receiver running on separate processors.

Constructor

template<class T>
ChanRD::ChanRD(int tag, int remote-proc, int max_data_length = 1);

Declaration forms

ChanRD<type> name (tag, remote_proc, length);

ChanRD<type> name(tag, remote_proc);

This creates a remote channel for synchronous message passing with name *name* and type *ChanRD<type>*. The *tag* and *remote_proc* parameters have exactly the same meaning as for ChanR channels. The *length* parameter has the same meaning as for *ChanLD channels*.

Communication/synchronization functions

```
template<class T>
void ChanRD::send(T *from_ptr, int l = 1);

template<class T>
void ChanRD::recv(T *to_ptr);
```

The first function sends messages via the channel it is attached to. *from_ptr* is the message source. *l* is the number of T-typed items the corresponding message can carry. the *l* value is checked against the maximum data length value attached to the channel.

The second function performs receptions through the channel it is attached to. the *to_ptr* parameter is the message destination. One can know about the length of the last received message by executing the *get_data_size()* function below. There can only be one send on one side and one recv on the other operating on a *ChanRD<type>* at a time.

The first process in a pair that executes send/recv on a *chanRD* gets descheduled. It is resumed by the remote process executing the matching send/recv function. The message from send to recv is passed and stored at the remote location before both processes are resumed.

```
template<class T>
int ChanRD::get_data_size();
```

This function returns the number of T-typed items composing the latest received message.

3.1.4 Examples

Here are a few declarations borrowed from the image segmentation algorithm.

```
ChanL *read_req[4];

ChanL *exit[4];

ChanL *done[4];

for(i = 0; i < 4; i++){
   read_req[i] = new ChanL;
   exit[i] = new ChanL;
   done[i] = new ChanL;
}

ChanL *starving = new ChanL;

ChanL *new_load_available = new ChanL;

ChanL *no_load_available  = new ChanL;

ChanRD<int> *load_request = new ChanRD<int>(4, (self_address + 1)%PROC_NB);

ChanRD<Load> **load_supply = new ChanRD<Load> *[PROC_NB - 1];
for(i = 1; i < PROC_NB; i++)
   load_supply[i - 1] = new ChanRD<Load>(i+4, (self_address + i)%PROC_NB);
   //i-1 is the nb of processors between source and target

ChanR *over = new ChanR(PROC_NB+4, (self_address + 1)%PROC_NB);
```

Here the code of processes borrowed from the segmentation algorithm that performs distributed termination at the end of the second phase. *Over_monitor_0* is run by node 0, *Over_monitor_i* by the other nodes. *over* channels form a virtual ring. The distributed termination procedure consists in flushing the ring out so one can be sure all the nodes have completed.

```
class Over_monitor_0 :public Process{

   ChanR *over;
```

```
      ChanL *exit;

   public:

      Over_monitor_0(ChanR *o, ChanL *e)
      :Process((Pmf0_ptr)&Over_monitor_0::body)
      {
         over = o;
         exit = e;
      }

      void body(){
         over->send();
         over->recv();
         over->send();
         over->recv();
         exit->send();
      }
};

class Over_monitor_i :public Process{

      ChanR *over;
      ChanL *exit;

   public:

      Over_monitor_i(ChanR *o, ChanL *e)
      :Process((Pmf0_ptr)&Over_monitor_i::body)
      {
         over = o;
         exit = e;
      }

      void body(){
         over->recv();
         over->send();
         over->recv();
         over->send();
```

```
        exit->send();
    }
};
```

3.2 Non deterministic synchronization-communication function

3.2.1 The *alt* functions

The send functions described earlier can also be performed in conjunction with a non-deterministic receive *alt* function. The *alt* function can possibly receive and synchronize via, not only one channel, but a list of channels held in a table whose pointer is passed to the function. The list can be a mix of channels of all types described previously. A given function, say *fi()*, is attached to each channel. Basically, *alt* operates as follows.

When *alt* is executed:

(1) If none of the channels attached to it happens to be enabled (a *send* has been executed on it), the function enables all the channels in the list before stopping. Execution will be resumed (scheduled) by the first process executing a send on one of the channels in the list. When this happens, all the channels are disabled.

(2) If one or several channels happen to be enabled, alt chooses the first one of them, say *ci*, schedules the process waiting on the ci channel, disables the *ci* channel (only this one), and executes the corresponding function *fi()* before returning.

The behavior previously described can be unfair in the sense that the same channel can always be selected excluding endlessly the others. The library supplies another function: *alt_fair* that has a fair behavior but is slightly slower.

Signature of the *alt* functions:

void alt(AltCtrl *altCtrl_ptr);

void alt_fair(AltCtrl *altCtrl_ptr);

3.2.2 *AltCtrtl* class

Each *alt* function makes use of a data structure that holds a few variables the function needs to operate. Namely, the table of pointers to the ports (see below) the function is applied to, the size of that table and two other variables internally used by the function.

Constructor

AltCtrl(int size, Alts *alt_table[]);

Declaration form

AltCtrl(size, alt_table);

declares an *AltCtrtl* data structure holding *size* ports whose pointers are held in *alt_table* for subsequent use as an *alt* argument.

3.2.3 Alt ports

There are four types of *alt* ports that derive from one another according to the derivation system pictured in Figure 3, that is in exact correspondence with the one we gave for the channels in Figure 2.

An *alt* port gathers the set of objects defining the port. Namely, a channel (with same generic terminal type), a function that is to be executed when this port is selected by the function as explained above, the pointer to the argument list this function needs to execute (if any) and the destination of corresponding incoming messages (if any).

***Alts* class**

This class is an abstract class for the whole alt port class system. It holds the data shared by all *alt* ports. It is not subject to instantiation but is generally used for defining pointers to any *alt* port (see *AltCtrl* constructor above).

***AltL* class**

The ports of this class are intended to be used for local synchronization purposes.

Constructors

AltL::AltL(ChanL *chan, Tmf2_ptr funct, void *arg);

AltL::AltL(ChanL *chan, Pmf2_ptr funct, void *arg);

with *Tmf2_ptr* and *Pmf2_ptr* defined as follows:

typedef void (Thread:: *Tmf2_ptr)(int, void *);

typedef void (Process::*Pmf2_ptr)(int, void *);

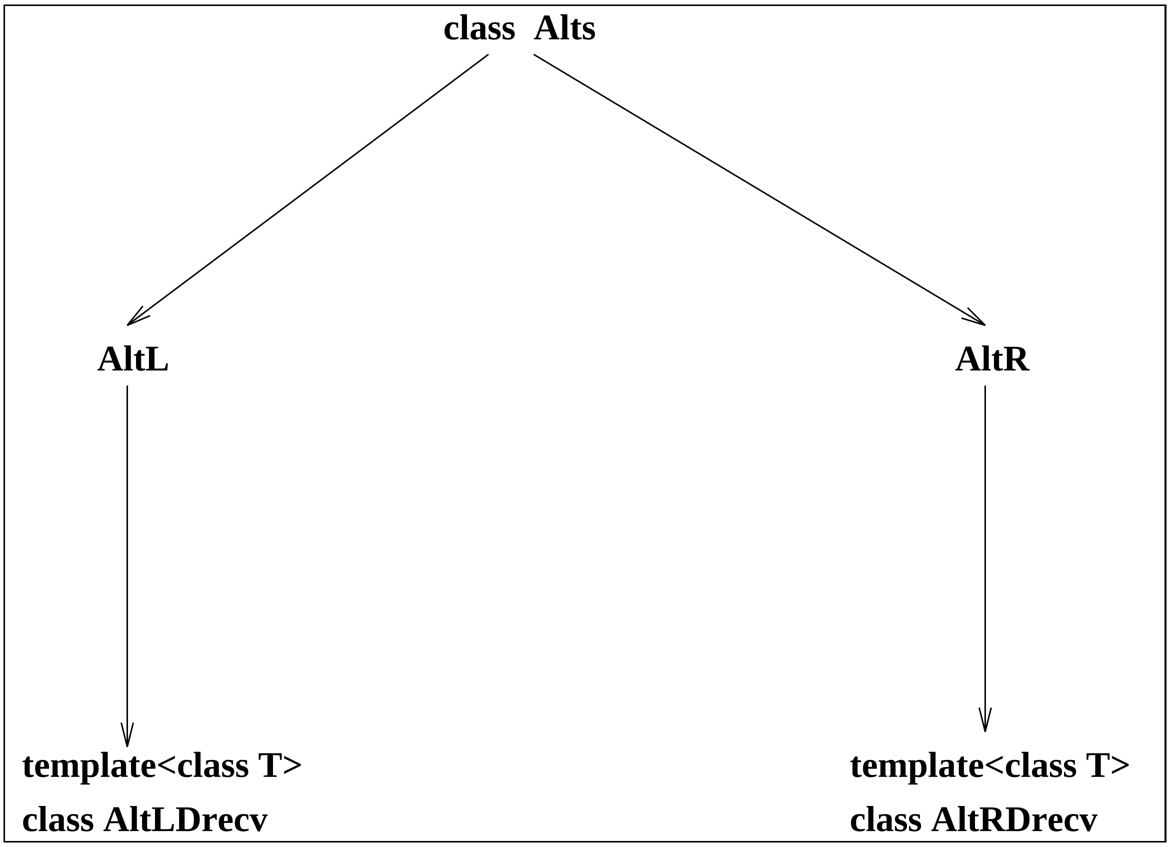

Figure 3: *Alt ports*

Declaration forms

AltL(chan, funct, arg);

The argument given to the constructors are: the channel assigned to the port, the pointer to a *Thread/Process* member function that is to be executed upon this port selection and the pointer to the list of arguments the function applies to. *chan* and the port are expected to be of the same terminal generic type.

AltLDrecv class

This class is attached to the ports supporting local synchronization and communication. The *recv* postfix was appended to stress the fact that a port can only be used for reception purposes exclusively.

Constructors

template<class T>
AltLDrecv::AltLDrecv(ChanLD<T> *chan, Tmf2_ptr funct, void *arg, T *to_ptr);

template<class T>
AltLDrecv::AltLDrecv(ChanLD<T> *chan, Tmf2_ptr funct, void *arg);

template<class T>
AltLDrecv::AltLDrecv(ChanLD<T> *chan, Pmf2_ptr funct, void *arg, T *to_ptr);

template<class T>
AltLDrecv::AltLDrecv(ChanLD<T> *chan, Pmf2_ptr funct, void *arg);

Declaration forms

AltLDrecv<T>(chan, funct, arg, to_ptr);

AltLDrecv<T>(chan, funct, arg);

Tmf2_ptr and *Pmf2_ptr* are defined as above, as well as the arguments: *chan*, *funct* and *arg*. *to_ptr* is the pointer to the message destination (whenever it is a constant one, otherwise it is handled dynamically). Note that here again *funct* is a *Thread/Process* member and that both *chan* and the port are expected to be of the same terminal generic type.

AltR class

This class is attached to the ports intended for remote synchronization.

Constructors

AltR::AltR(ChanR *chan, Tmf2_ptr funct, void *arg);

AltR::AltR(ChanR *chan, Pmf2_ptr funct, void *arg);

Declaration forms

AltR(chan, funct, arg);

Tmf2_ptr and *Pmf2_ptr* are defined as above, as well as the arguments: *chan*, *funct* and *arg*. *funct* is a *Thread/Process* member. *chan* and port are expected to be of the same terminal generic type.

AltRDrecv class

This class Is devoted to ports supporting both remote synchronization and communication. The *recv* postfix was appended to stress the fact that a port can only be used for reception purposes exclusively.

Constructors

```
template<class T >
AltRDrecv::AltRDrecv(ChanRD<T> *chan, Tmf2_ptr funct, void *arg, T *to_ptr);

template<class T>
AltRDrecv::AltRDrecv(ChanRD<T> *chan, Tmf2_ptr funct, void *arg);

template<class T>
AltRDrecv::AltRDrecv(ChanRD<T> *chan, Pmf2_ptr funct, void *arg, T *to_ptr);

template<class T>
AltRDrecv::AltRDrecv(ChanRD<T> *chan, Pmf2_ptr funct, void *arg);
```

Declaration forms

```
AltRDrecv<T>(chan, funct, arg, to_ptr);

AltRDrecv<T>(chan, funct, arg);
```

Tmf2_ptr and *Pmf2_ptr* are defined as above, as well as the arguments: *chan*, *funct* and *arg*. *to_ptr* is a pointer to the message destination (whenever it is a constant one, otherwise it is handled dynamically). Note that *funct* is a *Thread/Process* member and that both channel and port are expected to be of the same terminal generic type.

3.2.4 Example

Below is the code of a reading process borrowed from the image segmentation algorithm. The process constantly waits for either a read-request signal or an exit signal to occur. Depending on which signal is received, the process executes either the *read_branch* or the *exit_branch* member function.

```
void Read_proc::body(){

       AltL branch_read(read_req, (Pmf2_ptr)&Read_proc::read_branch, 0);
       AltL branch_exit(exit, (Pmf2_ptr)&Read_proc::exit_branch, 0);
       Alts *alt_table[2] = {&branch_read, &branch_exit};
       AltCtrl altctrl(2, alt_table);

       while(keep_running)
          alt(&altctrl);
}
```

4 Global Data

A global data is a type of data that can be accessed from any processor, be it remote or not. There are two sorts of global data:

1. *ARCH pre-defined* global data: interprocessor channels and spread/remote arrays (see section 3 and section 7)

2. *User-defined* global data: any data that is intended to be remotely accessed via global pointers (section 6).

only user-defined global data need be explicitly declared as *Global*.

4.1 Declaration

Constructors

1. template<class T> Global<T>::Global(int tg, T *ptr);

2. template<class T> Global<T>::Global(int tg, T* (*f_ptr)(IDH_rcd *));

3. template<class T> Global<T>::Global();

The *IDH_rcd* class is an abstract class that provides a convenient user interface for use in *indirect data handles* (see below and section 5):

```
class IDH_rcd{
  public:
    virtual int get_tag()=0;
    virtual unsigned get_offset()=0;
    virtual int get_size()=0;
    virtual int get_rarg1()=0;
    virtual int get_rarg2()=0;
};
```

Declaration forms

1. Global<actual-type> name(int tag, actual-type *handle);
2. Global<actual-type> name(int tag, actual-type* (*handle)(IDH_rcd *ptr));
3. Global<actual-type> name;

In declaration (1) *name* is declared as a *Global* data. A tag and data handle are attached to it. *tag* and *handle* respectively are a global symbolic reference and a local pointer to the data that will be from now on denoted as a *direct (data) handle*. In declaration (2) the data handle is not a direct pointer to the data but a pointer to the function: *handle*, that computes a pointer to the data given the pointer *ptr* as an argument. The data handle will be denoted as an *indirect (data) handle*. The system dynamically binds *ptr* to an *IDH_rcd* record (actually a record with type derived from *IDH_rcd* (see section 5). The *IDH_rcd* members can be accessed in the *indirect data handle* function via the calls:

```
ptr->get_tag();          // tag returned in coded form
ptr->get_offset();
ptr->get_size();         // size returned as a number of bytes
ptr->get_rarg1();
ptr->get_rarg2();
```

Declaration (3) allows for incomplete non-initialized declarations and assumes the existence of additional operators intended for subsequent completion (see below). Any processor can get access to a remote instance of a Global-defined variable by supplying a tag together with an offset value and, in case (2), a pair of extra integer arguments (see sections 5 , 7). These data are remotely used to compute the address of the variable (see section 5).

The *tag* argument should be given using one of the two following forms:

- DIR(tag) for direct data handle,
- INDIR(tag) for indirect data handle.

The given tag value should be a positive integer. *DIR(tag)* and *INDIR(tag)* actually are recorded in coded form. Note that *get_tag()* returns the value of *tag* as coded by the system. One can apply to this value one of the two decoding functions:

- *int decode_tag(in tag_code);*
- *int decode_tag(int tag_code, int *tag_type);*

The first function just returns the decoded value *tag*. The second one returns both the decoded value *tag* and the tag_type: 0 for *DIR* and 1 for *INDIR*.

The tag values must be supplied throughout a program in increasing order.

4.2 Operators

1. template<class T> Global<T>::Global name(int tag);
2. template<class T> void Global<T>::set_tag(int tag);
3. template<class T> void Global<T>::set_handle(T *handle);
4. template<class T> void Global<T>::set_handle(T* (*handle)(IDH_rcd *));

Constructor (1) is intended for the creation of copies of an already defined *Global* data. the argument is given in one of the two forms: *DIR(tag)* or *INDIR(tag)*, with the same conventions as above.

Operator (2) allows for setting the tag value of an already defined *Global* object. This operator must be applied before one of the next two operators which allow for setting either a direct or an indirect data handle.

4.3 A special tag : 0

Tag 0 is a special pre-defined tag that has a null pointer attached to it as a handle. This tag can be taken as a base for access to remote global variables of any type. It provides a convenient means for getting access to remote data given their current (dynamic) absolute address as an offset. Tag can be declared as direct in this case the offset value is the data address. It can be declared as indirect as well. In this case, the offset value is the address of a procedure whose execution provides the address to data as described earlier.

4.4 Global synchronization of global data definitions

The definition of a global data should involve global synchronization to ensure that definition is performed on all the processors before any of them starts accessing the data. Performing automatic global synchronization in each Global declaration would have been possible but would have also been too much inefficient and was left to the programmer. Ideally, the global declarations should be gathered as much as possible in the code so that a limited number of synchronization barriers only are needed for that purpose.

Notes

1. Getting *Global* declaration with automatic synchronization is straightforward by deriving a new class, say Global_s, from *Global* with a constructor and a destructor that just perform a synchronization barrier.

2. When performing a global (barrier) synchronization, one should use one of the functions: *void flush();* or *void flush(int n);* (see section 5). They perform the same job as a usual barrier but additionally poll so possibly not yet completed *read/write* functions can complete. The *flush* functions perform a distributed completion of *read/write* functions by reducing local completion upon a processor virtual tree and broadcasting global completion down the tree. The degree of reduction/broadcasting tree is passed as an argument to the second function.

3. When the non-synchronized declaration mode is used, one should be aware that the constructors are silently called at the end of blocks so global synchronization should be explicitly performed at the end of each block containing *Global* declarations. A good practice could be to make use of synchronized declarations first and next to switch to non-synchronized ones at the performance optimization stage.

The programmer may need to get information relative to a *Global* defined variable. Three functions are supplied for that purpose:

```
template<class T>
int Global<T>::get_tag();

template<class T>
T *Global<T>::get_direct_handle();

template<class T>
Idh Global<T>::get_indirect_indirect_handle();
```

where Idh is a type declared in Global<T> as:

```
typedef T* (*Idh)(IDH_rcd *);
```

get_tag() returns the coded value of tag attached to the Global data it is applied to. The value can be decoded as explained earlier. *get_direct_handle()* returns the local pointer to a global data.

get_indirect_handle() returns the local pointer to a function that computes the pointer to a global data. An error is issued when function and data handle type do not match.

4.5 Exemple

```
Load_picked_up
*export_data_ptr = new Load_picked_up[underloaded_nb];
Global<Load_picked_up> g_export_data_ptr(PROC_NB+6, export_data_ptr);
```

Declares a global data with name *g_export_data_ptr* that makes the local array pointed to by *export_data_ptr* remotely accessible. Here the tag is set as PROC_NB+6 (this is the way the tag was set in the program the code fragment was extracted from: *PROG_NB* is an ARCH predefined variable that holds the number of processors the code is run on).

This global data can subsequently be used in any remote access using a remote pointer (see section 7 and the code of segmentation algorithm):

```
//define global pointer export_data_remote_ptr
Star<Load_picked_up>
export_data_remote_ptr(load_offered_in.proc_id,
                       g_export_data_ptr, underloaded_rank, NOCP);
.
.
.
//set the control pointer initialized as NOCP (no ctrl ptr)
//ctrl contains the user-function upon_end_of_RW() that
//will be executed with &rw_c as a parameter among others
RW_count rw_c;
RWCtrl ctrl( (func)upon_end_of_RW, &rw_c);
export_data_remote_ptr.set_ctrl(&ctrl);
.
.
.
//write second parameter to remote location
write_c(*export_data_remote_ptr,
Load_picked_up(self_address, import_data_ptr,
                       -over_or_under_load_left));
.
.
.
//busy waiting
```

```
wait_while(RW_not_complete);
.
.
.
```

4.6 Configuration setting

The Global data facility makes use of a data handle table whose size has arbitrarily been set in the *A.setconfig* file. The size corresponds to the number of tags that can be used in an application code. This number can be changed by updating the *A.setconfig* file and then recompiling the library. It can also be changed more dynamically by giving a parameter to the *Global_data_handle()* constructor in the *A.environment* file.

5 Remote read and write functions

5.1 Store and get functions

One should not be using these functions unless developing new basics for the library. get and store are low level functions for storing and getting data to/from a remote location. Higher level functions are offered in the library: remote read and write functions that are described further on in this section and in the global pointers section.

5.1.1 Store

```
void store(int to_node, int tag, unsigned to_offset, int *rarg,
           char *from, int size,
           HRW_ptr handler, void *arg, Thread *thread_ptr);
```

where,

```
typedef void (*HRW_ptr)(HRW_rcd *);
```

```
class HRW_rcd{
  public:
    virtual int get_remote_proc()=0;
    virtual char *get_data_ptr()=0;
    virtual int get_data_size()=0;
    virtual Thread *get_thread_ptr()=0;
    virtual void *get_arg()=0;
};
```

The *HRW_rcd* class is an abstract class that provides a convenient user interface for use in completion handlers (see section 5.1.3).

The store function stores *size* characters from the local address *from* to the memory attached to processor *to_node*. At the programmer's level, the function executes silently on the target processor. The arguments: *tag*, *to_offset* and *rarg* are remotely used to compute the target address. *rarg* is a pointer to an array of two integers. *tag* can be attached to a *direct* or an *indirect handle* (see section 4):

- **If the data handle is direct:** *tag* gets access to the handle, then *to_offset* is added to it to form the complete address.

- **If the data handle is indirect** (see section 4): *tag* gets access to the handle which, in this case, is a pointer to a function that returns an address. *to_offset* is then added to it to form the complete address. For any actual type *type* the function should have the following signature:

  type handle(IDH_rcd *ptr);

  The system dynamically binds *ptr* to a record with type derived from *IDH_rcd*. *IDH_rcd* is an abstract class providing a convenient user interface for use in *indirect handles*.

```
class IDH_rcd{
  public:
    virtual int get_tag()=0;
    virtual unsigned get_offset()=0;
    virtual int get_size()=0;
    virtual int get_rarg1()=0;
    virtual int get_rarg2()=0;
};
```

  One can get access to the members of *IDH_rcd* in the *handle* function using the calls:

```
ptr->get_tag();          // tag returned as a coded value
ptr->get_offset();
ptr->get_size();         // size returned as a number of bytes
ptr->get_rarg1();
ptr->get_rarg2();
```

  rarg1 and *rarg2* are the two extra arguments that were pointed to by *rarg* when *store* was called and were next transmitted. *tag* is returned as a coded value that can be decoded as explained in section 4.

Upon completion of the store function *handler* is executed with a pointer to a record whose type derives from the abstract *HRW_rcd* class. *handler* is expected to be supplied by the user. See section 5.1.3 for details.

5.1.2 Get

```
void get(int from_node, int tag, unsigned from_offset, int *rarg,
         char *to, int size,
         HRW_ptr handler, void *arg, Thread *thread_ptr);
```

This function gets *size* characters in the local destination *to* from the source processor *from_node*. At the programmer's level, the function executes silently on the source processor *tag*, *from_offset* and *rarg* are remotely used to compute the source location as already described for the store function. Upon completion *handler* is executed as explained in section 5.1.3.

5.1.3 Completion

The completion handler *handler* is a user-defined function whose signature is:

```
void handler(HRW_rcd *ptr);
```

The *HRW_rcd* class is an abstract class that provides a convenient user interface to communication objects for use in completion handlers.

The system dynamically binds *ptr* to a data structure that contains the information items listed below:

- for *Store*: to_node, from, size, thread_ptr, arg,

- for *get*: from_node, to, size, thread_ptr, arg.

In the body of *handler*, each item is available via the following calls:

```
ptr- >get_remote_proc();
ptr- >get_data_ptr();
ptr- >get_data_size();          // size returned as a number of bytes
ptr- >get_thread_ptr();
ptr- >get_arg();
```

Again, note that *get_data_size()* returns a number of bytes. Upon completion of the *store* (resp. *get*) function, *handler* is executed on the sender (resp. receiver) side by the poll function in the way explained below:

- **if handler is not the null pointer**: then the execution is meant to occur in the *user-controlled-completion mode*: the user gave *handler* as a handler and expects this function to be executed upon *store* (resp. *get*) completion. Which is executed next is decided by *handler*.
- **if handler is the null pointer**: then the execution is meant to occur in the *scheduler-controlled-completion mode*: the user did not give a handler, upon completion the Thread whose pointer was passed as an argument to *store* (resp. *get)* is automatically scheduled (by the local scheduler, see section 2).

The *store* (resp. *get*) function returns immediatly but does not complete until the data are stored at the remote (resp. local) destination. In the meantime, the user can execute asynchronously.

5.1.4 Setting configuration

The code that performs the *store* and *get* functions uses memory space, actually three tables each. The size of the tables exactly corresponds to the number of *store*'s and *get*'s that can be simultaneously performed in a program code. The size has arbitrarily been set in updating the *A.setconfig* file. That could be too much or too less depending on the application. The user can change the values by updating *A.setconfig*. This requires recompiling the library. He can also proceed more dynamically by supplying arguments to the constructors: Store_ctrl_rcd() and Get_ctrl_rcd() in the *A.environment* file.

The store function makes use a faster protocol for short data remotely stored. The protocol was made faster at the expense of some additional memory space consumed. The protocol makes use of a threshold value to decide how it should behave. This value has arbitrarily been set to 20 bytes (size of short messages) but should normally be tuned for better efficiency. This can be changed by resetting the value in the *A.setconfig* file (needs recompiling) or supplying the value as an argument to the Store_ctrl_rcd() constructor.

5.2 Read/write functions

There are several sorts of *read/write* functions offered in the library. They differ to one another on how completion control is performed (i.e. *user-* or *scheduler*-controlled).

5.2.1 Remote read/write with user-controlled completion

```
template<class T>
void write(int to_node, int tag, unsigned to_offset, int *rarg,
           T* from,
           HRW_ptr handler, void *arg);
```

```
template<class T>
void write(int to_node, int tag, unsigned to_offset, int *rarg,
          T* from, int size,
          HRW_ptr handler, void *arg);

template<class T>
void read(int from_node, int tag, unsigned from_offset, int *rarg,
          T* to,
          HRW_ptr handler, void *arg);

template<class T>
void read(int from_node, int tag, unsigned from_offset, int *rarg,
          T* to, int size,
          HRW_ptr handler, void *arg);
```

These instances read/write *size* T-type (remote) objects. *offset* is given as a number of T-type objects. At the programmer's level, the functions execute silently on the destination/source processor. The functions are implicitly considered as executing in the *user-controlled completion* setting, so *handler* is expected to be a non null pointer. Upon completion *handler* is executed as described earlier (see *store* and *get* sections). *rarg* has the same meaning as in section 5.1.1 (actually, this argument is simply passed to these functions). *tag* can be attached either a direct an indirect handle (see section 5.1.1).

5.2.2 Remote read/write with scheduler-controlled completion

Explicit process continuation

```
template<class T>
void write(int to_node, int tag, unsigned to_offset, int *rarg,
          T *from,
          Thread *thread_ptr);

template<class T>
void write(int to_node, int tag, unsigned to_offset, int *rarg,
          T *from, int size,
          Thread *thread_ptr);
```

```
template<class T>
void read(int from_node, int tag, unsigned from_offset, int *rarg,
          T *to,
          Thread *thread_ptr);

template<class T>
void read(int from_node, int tag, unsigned from_offset, int *rarg,
          T *to, int size,
          Thread *thread_ptr);
```

Here the *read/write* are implicitly considered as executing in the *scheduler-controlled completion* setting. the functions *stop* (see section 2) the current thread just after executing *store()/get()*. These functions are called with a null handler pointer. Upon completion, the thread pointed to by *thread_ptr* will be scheduled for subsequent execution. The *offset* argument is given as a number of T-type objects. *rarg* has the same meaning as in section 5.1.1.

Implicit process continuation

```
template<class T>
void read(int from_node, int tag, unsigned from_offset, int *rarg,
          T *to);

template<class T>
void read(int from_node, int tag, unsigned from_offset, int *rarg,
          T *to, int size);

template<class T>
void write(int to_node, int tag, unsigned to_offset, int *rarg,
          T *from);

template<class T>
void write(int to_node, int tag, unsigned to_offset, int *rarg,
          T *from, int size);
```

Same as above except that *thread_ptr* is implicily *current_thread_ptr*.

Notes

Even when it involves only one user-thread a code written with ARCH is always a multi-threaded one (as far as one decides to execute within the ARCH standard environment: see

A.environment file). Indeed, the user thread is run asynchronously on each node with the ARCH pre-defined *main_thread* (see section 2.1.1). Switching to the latter occurs each time the user-thread is implicitly or explicitly waiting for communication operations to complete and the local scheduling queues are empty. Switching is silently performed upon executing a read/write functions with scheduler-controlled termination. It is explicitly performed upon executing a *stop()* (see section 2) after a series of read/write functions with user-controlled termination. *main_thread* then keeps polling until a handler is eventually executed that reschedules a user thread (i.e. one of the scheduling queues becomes non empty).

Switching to *main_thread* involves context-switching that one might want to avoid in single-thread user codes. The user can decide to avoid functions that involve switching to *main_thread* (by avoiding read/write functions with scheduler-controlled termination or *stop()*). In this case polling has to be explicitly performed in his code.

In all cases, no communication operation can progress unless polling is performed: the "engine" that makes inter-processor communication circulate is polling. This is a direct consequence of using MPI for implementing ARCH. The communication events occur silently in MPI and the user is expected to read statuses to check whether a communication is ready to take place or has completed. Avoiding polling could only be achieved with a communication layer sending interrupt signals that could be caught and processed by handlers, right at the time the signals occur, and as long as interrupts are enabled. It seems that one can expect such a mechanism to be included in the future MPI version-2 (personal communication).

5.3 Poll and flush

The library offers two functions: *void flush();* and *void flush(int d);*, that check for *distributed termination* of all communication operations performed in some program step. We need such a function to keep all the nodes polling while only some are still wanting to store/write or get/read remote data.

Flush functions will most often be used for global phase completion (barrier synchronization) in the user root-process (see section 2.1.1).

For instance, suppose a simple system with two nodes A and B and a phase where A needs to store a data at some location in B, but B does nothing. If node B is not kept waiting for distributed completion, it will locally complete hence fail to poll for A to complete. *flush()* is to prevent such a situation from occurring.

flush() acts as a global synchronization barrier. It performs a reduction of local completions up a virtual tree, then broadcast global completion down the same tree. In the mean time the function keeps polling on each node until global completion is reached. *flush()* can also be executed with a parameter that determines the degree of the reduction tree (default is 2) so the depth can properly be tuned depending on the machine size. The range of possible degrees has arbitrarily been set to the range [2, 16]. the upper bound can be changed by changing *DEGREE_MAX* in the *A.setconfig* file (needs recompiling).

6 Global pointers

Global pointers are a generalization of C++ pointers that allow for addressing global data at any place over the distributed memory. As usual pointers, global pointers are subjected to arithmetic and logic manipulations (incrementation, dereferencing, indexing, comparison...). Global pointer expressions provide global references over the distributed memory that can subsequently be used as arguments to global read/write functions. These functions allow the processors to get access to all global data regardless of their locations over the distributed memory. In their most complete form, the read/write functions operate as remote procedure calls. At the programmer's level global read/write functions appear as "one-sided": a read/write operation is executed on the processor that needs to read/write global data but need not be explicitly handled by the processor associated to the memory holding the data.

6.1 Global pointer data members

A global pointer actually is a record that holds the following data items:

- **unsigned proc_id**: rank of the processor holding the data this global pointer is pointing to.
- **int tag**: tag of the Global data the global pointer is attached to (see setion 4), direct as well as indirect data handle are allowed.
- **unsigned offset** (see section 4).
- **A_attributes *attributes_ptr**: this member's value is NOAP (see section 6.2 below) if the global pointer is unassigned. It contains the pointer to an array descriptor if the global pointer has been assigned to a spread or a remote array (sections 7.1 and 7.4). Information held in the descriptor is used to decode index-based addresses and to retrieve the data held in the spread/remote array.
- **RWctrl *ctrl_ptr**: the record pointed to by ctrl_ptr holds data required to control completion when access to data: assignment of global data (section 6.7), implicit Lval<T> to T conversion (section 6.8), or read/write functions (sections 5, 6.9 and 6.10) is performed using this global pointer (see *user-controlled completion* mode in sections 5.1.3 and 6.13).

6.2 Global pointer type and data member null values

The library currently provides three global pointer type descriptors:

enum **GPType** {U=0, R=1, S=2};

U, *S* and *R* respectively are associated to *unassigned*, *spread* and *remote* global pointers. The pointer type (GPType) is returned by the following function applied to a global pointer:

enum GPType **get_gptr_type**() const;

The null values attached to the members of global pointers respectively are: NOPID, NOTAG, NOFFST, NOAP and NOCP (see A.setconfig file), note that most constants are not 0.

6.3 Global pointer evaluation

ARCH global (**Star**) pointers are designed more or less after the C philosophy, with generalized r-values (**Rval**) and l-values (**Lval**) associated with global pointers:

template<class T> Rval;

template<class T> Lval;

Evaluating global pointers expressions recursively transforms Star objects into Rval or Lval ones as shown in Figure 4. Applying operators ++, –, += and -= to a template<class T>Star provides a template<class T>Rval. Applying operators + and -(addition and substraction to integers) to a template<class T>Rval provides a template<class T>Rval. The Rval and Lval and Star records exactly have the same internal structure. Each stands for a particular state in the evaluation process of global pointer expressions.

Global pointers can possibly be nested. This is what *nesting level* in Figure 4 refers to. for instance, writing the following sequence is perfectly possible:

```
//at the file level
Stardef(int, Star1)
Stardef(Star1, Star2);
Stardef(Star2, Star3);
Stardef(Star3, Star4);

//at the function level
int i = 5;
Global<int> g_i(9, &i);
```

```
Star1  str_ptr1;
Global<Star1> g_str_ptr1(10, &str_ptr1);

Star2 str_ptr2;
Global<Star2> g_str_ptr2(11, &str_ptr2);

Star3  str_ptr3;
Global<Star3> g_str_ptr3(12, &str_ptr3);

Star4 str_ptr4;

str_ptr1  =  Star1(0, g_i, (unsigned)0, NOAP, 0);
str_ptr2  =  Star2(0, g_str_ptr1, (unsigned)0, NOAP, 0);
str_ptr3  =  Star3(0, g_str_ptr2, (unsigned)0, NOAP, 0);
str_ptr4  =  Star4(0, g_str_ptr3, (unsigned)0, NOAP, 0);

int k1;
int k2;
int k3;
int k4;

k1 = *str_ptr1;
k2 = **str_ptr2;
k3 = ***str_ptr3;
k4 = ****str_ptr4;

int l = 1;

Global<int> g_l(13, &l);

StarR p1(0, g_l, (unsigned)0, NOAP, 0);

Global<Star1> g_p1(14, &p1);

Star2 p2(0, g_p1, (unsigned)0, NOAP, 0);

int k5 = **p2;

**p2 =  ****str_ptr4;
```

```
**p2 = 6;

int m = (**p2 + ****str_ptr4)*(**p2 + ****str_ptr4);
```

Stardef declaration

Note that, when defining nested pointers, one have to use a special declaration of the form:

Stardef(data_type, type_name);

Such a declaration must be performed at the file level. This is consistent with standard usage since this kind of declarations should normally occur in some header file.

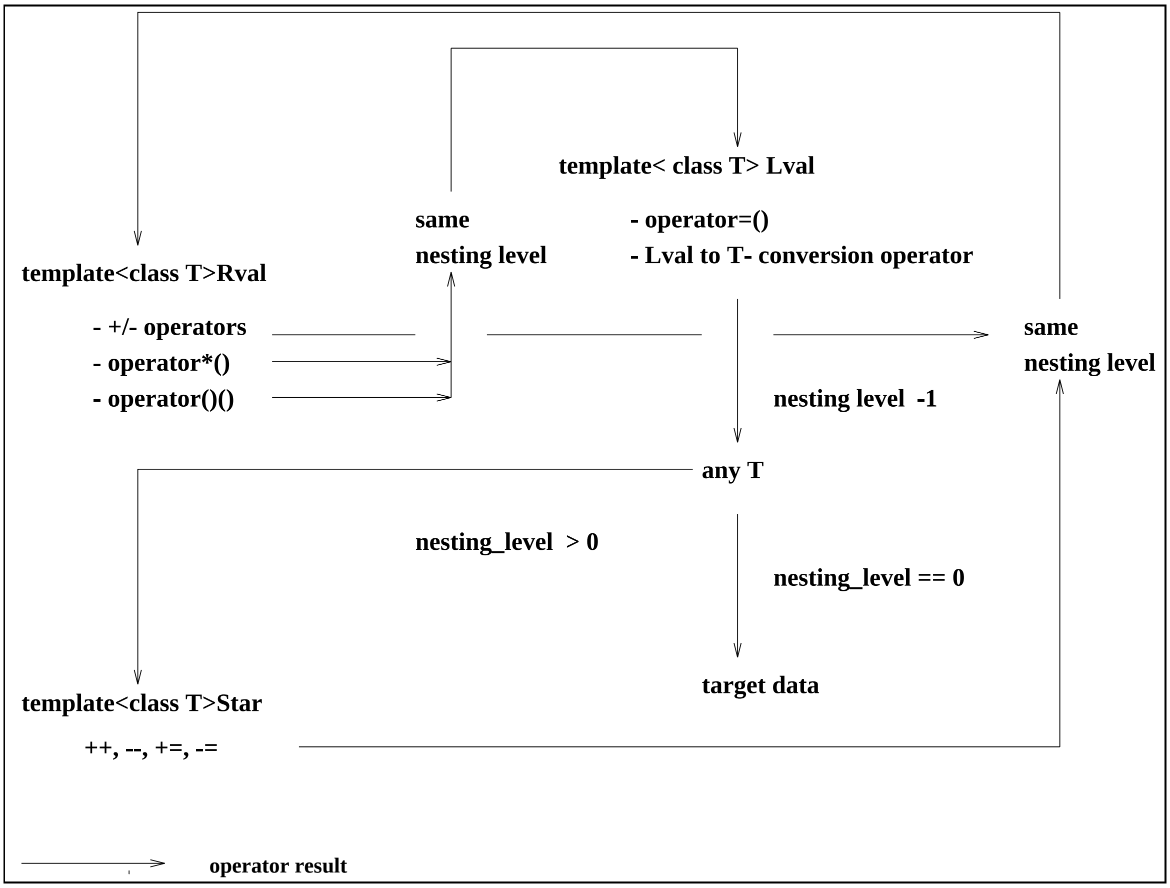

Figure 4: *Global pointer evaluation scheme.*

Note

One should be aware that using nested spread pointers without care will lead to inefficiencies. Most often it will be much better to use read/write functions with indirect handles (see section 6.10) instead of nested spread/remote pointers.

The evaluation process of nested pointers follows the scheme described in Figure 4. Applying *operator*()* (dereferencing) and *operator()()* (indexing) to a template<class T>Star provides a template<class T>Lval. Un-nesting is performed on Lval's as expected, either by the template<class T>Lval to T conversion operator (see section 6.8) or by the assignment operator (see section 6.7) when the Lval appears at the left hand side of the assignment. Conversion and assignment are two operators provided in the template<class T>Lval class.

6.4 Architecture of the global pointer class sytem

The library has been architectured so that no code function is uselessly duplicated when *spread* or *remote* arrays and *global* pointers are instantiated in a program. The class system has been designed with two sorts of classes: intern classes and interface classes. The former are non template classes that contains the essential part of the code (no duplication under class instantiation). the latter contains the set of template classes that forms the typed user-interface. These classes essentially contain entry points to functions belonging to the internal classes. They are shown in the two boxes of Figure 5. Note that we have avoided multiple inheritance and virtual classes to ban any pointer from the implementation of generic objects, as these objects can possibly travel.

6.5 Global pointer declaration

6.5.1 New global pointer issued from a conversion

Constructors

```
template<class T>
Star<T>::Star(const Rval<T>& rv);
```

```
template<class T>
Star<T>::Star(const SpreadArray<T>& sa);
```

```
template<class T>
Star<T>::Star(const RemoteArray<T>& ra);
```

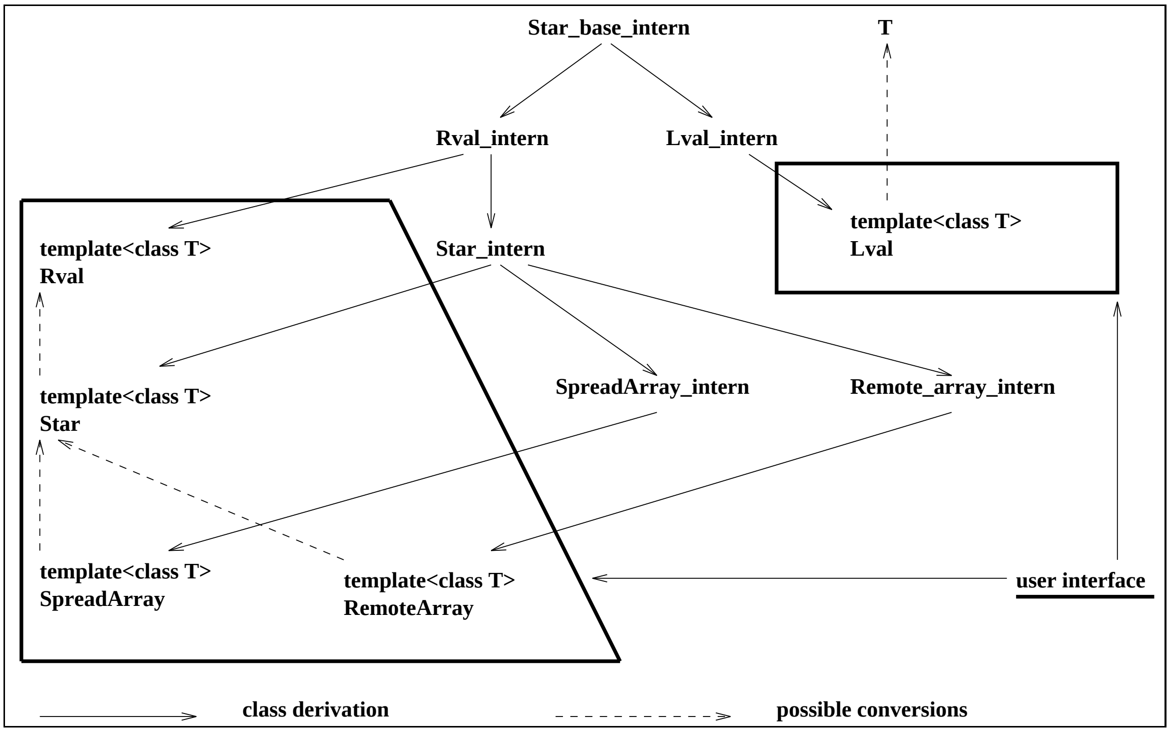

Figure 5: *Object-oriented design of spread/remote arrays & global pointers.*

Declaration forms

Star<type> name(rv); //rv is a Rval identifier

Star<type> name(sa); //sa is a SpreadArray identifier

Star<type> name(ra); //ra is a RemoteArray identifier

Example

SpreadArray<int> SA(1, 4, 5, 2, 2);
Star<int> SA_crt(SA);
Star<int> SA_crt = SA; //also is possible

subsubsectionNew initialized global pointer

Constructors

template<class T>
Star<T>::Star(unsigned proc_id, Global<T> global_data,
unsigned offset, A_attributes *A_a_ptr, RWCtrl *c);

template<class T>
Star<T>::Star(unsigned proc_id, Global<T> global_data,
T *ptr, A_attributes *A_a_ptr, RWCtrl *c);

template<class T>
Star<T>::Star(unsigned proc_id, Global<T> global_data,
T* (*ptr)(IDH_rcd *), A_attributes *A_a_ptr, RWCtrl *c);

where *IDH_rcd* is defined in section 4.

Declaration forms

Star<type> name(proc_id, global_data, offset, SA_a_ptr, c);
Star<type> name(proc_id, (Global<type>)DIR(0), ptr, SA_a_ptr, c);
Star<type> name(proc_id, (Global<type>)INDIR(0), ptr, SA_a_ptr, c);

The first form above is the "regular" form. *global_data* can have a direct or indirect handle attached to it. The second and third form is to be used when dealing with offsets consisting of remote absolute addresses.

(Global<type>)DIR(0) builds a *Global* object with *tag 0* attached to it. The data handle associated to *tag 0* is a direct handle. *offset* is the absolute address of the data on the remote processor.

(Global<type>)INDIR(0) builds a *Global* object with *tag 0* attached to it. The data handle associated to *tag 0* is an indirect handle. *ptr* is the absolute address on the remote processor of a procedure whose execution supplies the address of the data. The procedure is defined and executed as described in section 5.1.1.

6.5.2 New non-initialized global pointer

Constructor

template<class T>
Star<T>::Star();

Declaration form

Star<type> name;

In this case, the members of the global pointer record are implicitly initialized with null values: NOPID, NOTAG, NOFFST, NOAP, NOCP.

6.6 Operations on global pointers

As usual C pointers, global pointers can be part of conditional statements or address expressions.

6.6.1 Global pointer based predicates

== same proc_id, same tag, same offset, same A_attributes_ptr, and same ctrl_ptr.

> ((lhs proc_id > rhs proc_id and same offset) or lhs offset > rhs offset) and same tag, same A_attributes_ptr and same ctrl_ptr.

>= ((lhs proc_id >= rhs proc_id and same offset) or lhs offset >= rhs offset) and same tag, same A_attributes_ptr and same ctrl_ptr.

< ((lhs proc_id < rhs proc_id and same offset) or lhs offset < rhs offset) and same tag, same A_attributes_ptr and same ctrl_ptr.

<= ((lhs proc_id <= rhs proc_id and same offset) or lhs offset <= rhs offset) and same tag, same A_attributes_ptr and same ctrl_ptr.

!= different proc_id or different local_ptr or different SA_attributes_ptr and same tag, same A_attributes_ptr and same ctrl_ptr.

6.6.2 Global pointer arithmetic operators

The usual operators: +, -, ++(pre/post), –(pre/post), +=, -=, can be applied to a global pointer. The semantics follow the rules of the regular arithmetic.

6.6.3 Dereferencing '*'

A global pointer can be dereferenced in the same way as usual pointers. Performing this operation provides an *Lval*. A dereferenced global pointer expression can appear in the right and left hand side of operator=() or as an argument to a global *read/write* function, (see sections 6.7 and 6.8 below).

6.7 Assignment to a global reference

One can apply the following assignment operator to Lval objects that result from dereferencing and indexing:

```
template<class T>
void Lval<T>::operator=(const T& rhs);
```

Operator=() assigns the local T-type *rhs* value to the Global<T> data whose global reference is held in the Lval<T> object to which operator=() is applied. Operator=() in general requires a global writing.

6.8 Lval<T> to T conversion

The template<class T>Lval class provides a T-conversion operator:

```
template<class T>
Lval<T>::operator T();
```

operator T() automatically returns a local T-type object that is a copy of the remote object whose global reference is held in the Lval<T> object to which it is applied. operator T() in general requires a global reading.

This operator silently takes care of global read/writes that need to be performed, for example, in expressions such that:

```
int y = *g_ptr_x;
int y = z + *g_ptr_x;
```

where g_ptr_x is a global pointer to a given global integer x and z is some integer. Control upon completion should be handled as explained in section 6.13.

6.9 Read/Write using global pointers and direct handle

6.9.1 write with user-controlled completion

```
template<class T>
void
write_c(const Lval<T>& global_to, const T& local_from, int l);

template<class T>
void
write_c(const Lval<T>& global_to, const T& local_from);
```

These functions can be used to write 1 or l consecutive T-typed objects. Upon completion a handler is executed whose reference is found in the RWCtrl record attached to *Lval<T>*. The pointer to this record should have been set in the corresponding global pointer before a *write_c* function is executed (*ctrl_ptr* member, see section 6.13).

6.9.2 write with scheduler-contolled completion

```
template<class T>
void
write(const Lval<T>& global_to, const T& local_from, int l);

template<class T>
void
write(const Lval<T>& global_to, const T& local_from);
```

These functions work the same as the two previous ones but completion is automatically monitored by the local scheduler that automatically suspends and resumes the process executing the function (no *RWCtrl* record is required from the user).

6.9.3 read with user-controlled completion

```
template<class T>
void
read_c(T& local_to, const Lval<T>& global_from, int l);

template<class T>
void
read_c(T& local_to, const Lval<T>& global_from);
```

Same as above but for read functions.

6.9.4 read with scheduler-controlled completion

```
template<class T>
void
read(T& local_to, const Lval<T>& global_from, int l);

template<class T>
void
read(T& local_to, const Lval<T>& global_from);
```

Same as above but for read functions.

6.10 Read/Write using global pointers and indirect handle

In the previous section we defined *read/write* remote function using a direct remote data handle. There, the handle is simply used as a base address. An offset value is added to it to get the complete address of the target data. In some cases, it can be convenient to make use of data handles that provide indirect access to data. In this case the handle is not the base address but the pointer to a function whose dynamic evaluation supplies this address.

As compared to the previous ones, the functions below need an additional argument: *rarg* holding a pointer to an array of two integers. These integers are transmitted to the remote destination and can be accessed in *handle*, among other arguments (see section 5.1.1).

6.10.1 write with user-controlled completion

```
template<class T>
void
write_c(const Lval<T>& global_to, int *rarg, const T& local_from, int l);

template<class T>
void
write_c(const Lval<T>& global_to, int *rarg, const T& local_from);
```

6.10.2 write with scheduler-controlled completion

```
template<class T>
void
write(const Lval<T>& global_to, int *rarg, const T& local_from, int l);

template<class T>
void
write(const Lval<T>& global_to, int *rarg, const T& local_from);
```

6.10.3 read with user-controlled completion

```
template<class T>
void
read_c(T& local_to, const Lval<T()>& global_from, int *rarg, int l);

template<class T>
```

```
void
read_c(T& local_to, const Lval<T>& global_from, int *rarg);
```

6.10.4 read with scheduler-controlled completion

```
template<class T>
void
read(T& local_to, const Lval<T>& global_from, int *rarg, int l);

template<class T>
void
read(T& local_to, const Lval<T>& global_from, int *rarg);
```

6.11 Access to members

One can get access to members of a data accessed via a spread pointer using the overloaded operator>>(). It was not possible to overload either operator.() or operator− >() because the first cannot be, while the second can but with such limitations that it was just impossible to get what was needed.

```
template <class T1, class T2>
inline Lval<T2> operator>>( const Lval<T1>& lv, T2 T1:: *mbr);

template <class T1, class T2>
inline Lval<T2> operator>>( const Star<T1>& sp, T2 T1:: *mbr);
```

Notes

1. Before using this operator the pointer on member: *mbr* must be set (see examples below and C++ manual). Setting the second argument of the access to member functions can be make easier by using the *member* macro available in the library and defined as follows (also, see examples below):

   ```
   #define member(T2, T1, s) T2 T1:: *s = &T1::s
   ```

2. read/write functions with indirect data handles can be used to get remotely access to global data structure members. In many cases using *operator>>()* is just more convenient as the name of member to be accessed explicitly appears in the access expression. However, this operator implicitly makes the assumption that the internal organization of data in C++ classes and records is exactly the same on all the processors, which might not be the case for certain heterogeneous systems.

Examples

```
member(Link_rcd, Link_rcd_pair, low);
write_c((*link_array)(id1, id2)>>low, rcd);
```

Here is an other example that shows how can *operator>>()* be applied recursively.

```
member(int, Link_rcd, selected);
member(Link_rcd, Link_rcd_pair, low);
write_c(((*link_array)(id1, id2)>>low)>>selected, id_absorbing_vertex);
```

6.12 Other operators applying to spread pointers

The spread pointer class offers a set of operators for setting/getting the spread pointer member current values.

```
  //Returns processor identifier
template<class T>
unsigned Star<T>::get_proc_id() const;

  //Sets processor identifier
template<class T>
void Star<T>::set_proc_id(unsigned p_id);

  //Returns copy of the Global object attached to a spread pointer
template<class T>
Global<T> Star<T>::get_global();

  //Assigns Global object to a spread pointer
template<class T>
void Star<T>::set_global(Global<T> g);

  //Returns current offset value as a number of T-typed data
template<class T>
unsigned Star<T>::get_offset() const;
```

```
   //Sets current offset value as a number of T-typed data
template<class T>
void Star<T>::set_offset(unsigned os);

   //Sets current offset value with a T-type pointer.
   //Assumes that tag has been already defined
template<class T>
void Star<T>::set_offset(T *ptr);

   //Sets current offset value with a T*(*)(IDH_rcd *) pointer
   //Assumes that tag has been already defined
template<class T>
void Star<T>::set_offset(T* (*ptr)(IDH_rcd *));

   //Gets the pointer to array attributes
template<class T>
A_attributes *Star<T>::get_attr_ptr() const;

   //Gets pointer to SpreadArray attributes.
   //The global pointer must have been assigned to a SpreadArray
template<class T>
SA_attributes *Star<T>::get_SA_attr_ptr() const;

   //Gets pointer to RemoteArray attributes
   //The global pointer must have been assigned to a RemoteArray
template<class T>
RA_attributes *Star<T>::get_RA_attr_ptr() const;

   //Sets pointer to array attributes
template<class T>
void Star<T>::set_attr_ptr(A_attributes *A_a_ptr);
```

```
   //Gets pointer to control-upon-completion record
template<class T>
RWCtrl * Star<T>::get_ctrl_ptr() const;
```

```
   //Sets pointer to control-upon-completion record
template<class T>
void Star<T>::set_ctrl(RWCtrl *c);
```

```
   //Returns local pointer to data with direct handle
template<class T>
T *Star<T>::to_local_ptr();
```

```
   //Returns local pointer to data with indirect handle
template<class T>
T *Star<T>::to_local_ptr(IDH_rcd *ptr);
```

where *IDH_rcd* is defined in section 5.1.1.

The user will have to bind *ptr* to a record whose type is derived from the IDH_rcd class. This can be any class of his own or the class *IDH_record*, built in the library. The latter provides the following functions:

```
IDH_record(int s, int t, unsigned o, int r1, int r2);

int get_tag();                //returns encoded tag value

unsigned get_offset();

int get_size();               //returns a size as a number of bytes

int get_rarg1();

int get_rarg2();
```

6.13 Control upon completion

Most often, global pointers are involved in global *read/write* operations executing asynchronously with local code. Depending on which type of *read/write* operations is used, completion is monitored either automatically by the local scheduler or explicitly by the user. The library supplies a specific class: *RWCtrl*, devoted to user-controlled completion. The class holds the following data items:

- void (*fct)(HRW_rcd *ptr): handler to be executed upon completion of the remote operation,
- void *arg: handler arguments,

fct stands for the pointer to a function, say *f*, whose signature is:

void f(HRW_rcd *ptr);

HRW_rcd is the following abstract class:

```
class HRW_rcd{
  public:
    virtual int get_remote_proc()=0;
    virtual char *get_data_ptr()=0;
    virtual int get_data_size()=0;
    virtual Thread *get_thread_ptr()=0;
    virtual void *get_arg()=0;
};
```

The system dynamically binds the *ptr* argument of *f* to a data record whose type is derived from the HRW_rcd class and contains the information items listed below:

- if the operation performs a *write* to a remote processor:

  remote processor identification, data source, data size (**as a number of bytes**), thread pointer, pointer to extra arguments,

- if the operation performs a *read* from a remote processor:

  remote processor identification, data destination, data size (**as a number of bytes**), thread pointer, pointer to extra arguments.

Each item can be accessed from the *f* body via the following calls:

```
ptr− >get_remote_proc();
ptr− >get_data_ptr();
ptr− >get_data_size();                // size returned as a number of bytes
ptr− >get_thread_ptr();
ptr− >get_arg();
```

Before executing a global *read/write* an assignment or a conversion operator that involves a global pointer the user is expected to setup a *RWCtrl* data object and to attach it to the spread pointer (ctrl_ptr member). Upon completion, handler and arguments are read from the data object for execution.

Setting a RWCtrl record or getting information from it can be performed with the functions below:

```
//constructor
RWCtrl(HRW_ptr, void *);

//empty constructor
RWCtrl();

//set pointer to handler
void set_fct(HRW_ptr handler);

//get pointer to handler
HRW_ptr get_fct() const;

//set pointer to handler arguments
void set_arg(void *p);

//get pointer to handler arguments
void * get_arg() const;
```

with:

```
typedef void (*HRW_ptr)(HRW_rcd *);
```

7 Spread and remote arrays and pointers

7.1 Spread Arrays

A spread Array 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 ascending order of processor numbers starting from 0, according to the natural order of spread index combinations. Figure 6 shows how a spread array of intergers with two spread and one internal dimensions of size: 4, 5, 2 respectively is distributed over a set of 3 processors.

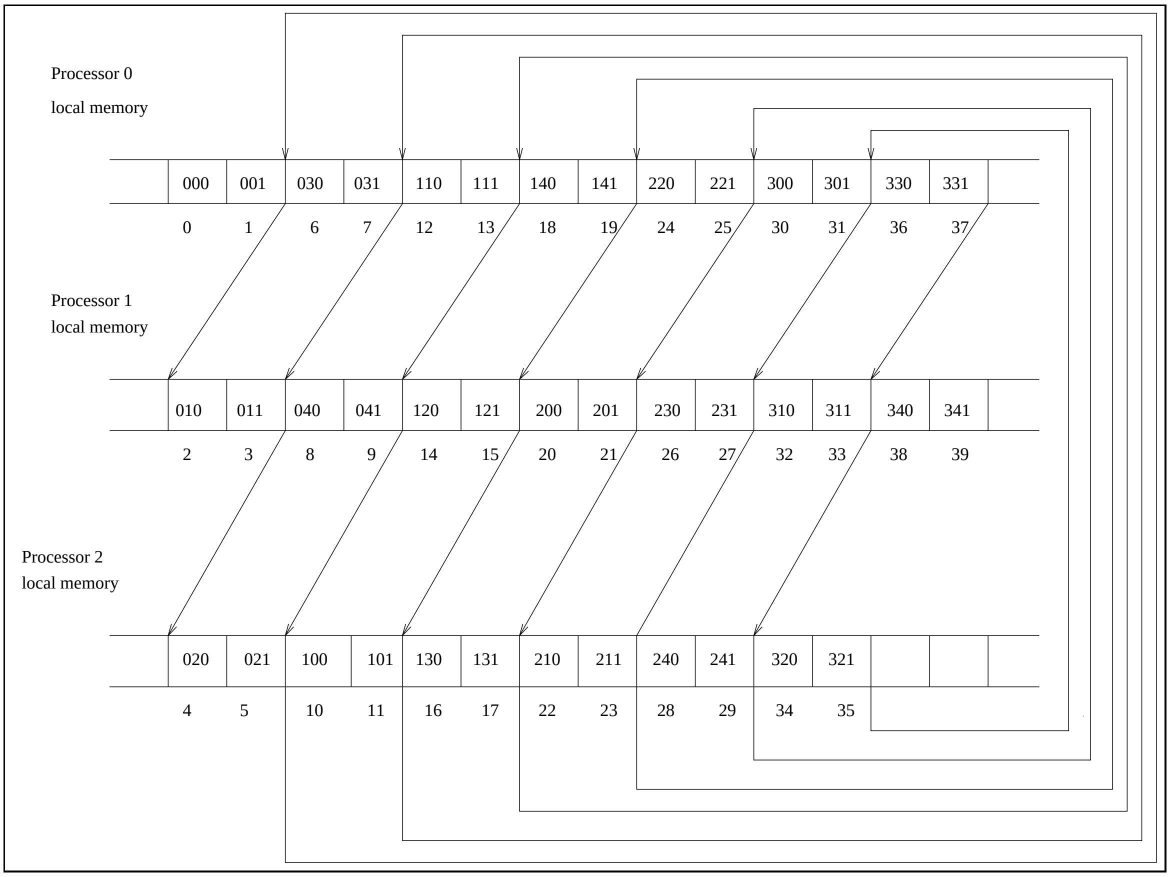

Figure 6: *A spread array.*

The idea has been borrowed from Split_C. With ARCH the user can build his own SpreadArrays by deriving from the basic SpreadArray class found in the library. Below are a couple of illustrations taken from the image segmentation algorithm. Spread arrays are extensively used in all phases.

7.1.1 Example 1

The image is defined as an instance of the *SpreadImage* class derived from the standard *SpreadArray* class. *SpreadImage* is a spread array of square blocks of pixels. the SpreadImage constructor defines how the image blocks are spread over the distributed memory. It calls the *SpreadArray* constructor, that builds all (transparent) data structures required for storing and retrieving the pixels or block of pixels and checks the image structural parameters. The spreading policy is one among many possible. This choice was made just because it appeared convenient. The spreading policy may be changed (for instance, if some other appeared to be better as regard to performances) without changing anything in the code except that of access functions inside the *SpreadImage* class. All access functions in *SpreadImage* are derived from *SpreadArray*'s by simple coordinate mapping.

```
// VPS = Vertical Processor-matrix Size
// HPS = Horizontal Processor-matrix Size
template<class T, int VPS, int HPS>
class SpreadImages: public SpreadArray<T>{

   protected:
      int Vsize;
      int Hsize;
      int IBsize;

// VIS = Vertical Image Size
// HIS = Horizontal Image Size
// IBS = Image Block Size  (block = elementary image tile)
   public:
      SpreadImages(int tag, int VIS, int HIS, int IBS)
      : SpreadArray<T> (tag, VPS, HPS, (VIS/IBS)/VPS, (HIS/IBS)/HPS, IBS, IBS, 2)
      {
          if(double(int(log(IBS)/log(2))) != log(IBS)/log(2)
             || HPS < VPS
             || VIS%IBS || (VIS/IBS)%VPS
             || HIS%IBS || (HIS/IBS)%HPS ){
                printf("on node %d: incorrect SpreadImage size\n",
                              self_address);
```

```
                preempt();
            }
            Vsize  = VIS;
            Hsize  = HIS;
            IBsize = IBS;
        }

        ~SpreadImages(){}

        virtual Lval<T> operator()(int v, int h) = 0;
        virtual Lval<T> operator()(int v1, int h1, int v, int h) = 0;
        virtual Lval<T> block(int v1, int h1) = 0;
        virtual T& local(int v1, int h1, int v, int h) = 0;
        virtual T& local_block(int v1, int h1) = 0;
};


template<class T, int VPS, int HPS, int VIS, int HIS, int IBS>
class SpreadImage: public SpreadImages<T, VPS, HPS>{

   public:
      SpreadImage(int tag)
      : SpreadImages<T, VPS, HPS>(tag, VIS, HIS, IBS)
      {}

      ~SpreadImage(){}

      //access to pixel:
      //(v_pix, h_pix) refers to pixel in the whole image

      Lval<T> operator()(int v_pix, int h_pix){
         return SpreadArray<T>::operator()(v_pix/(VIS/VPS),
                                           h_pix/(HIS/HPS),
                                           (v_pix%(VIS/VPS))/IBS,
                                           (h_pix%(HIS/HPS))/IBS,
                                           (v_pix%(VIS/VPS))%IBS,
                                           (h_pix%(HIS/HPS))%IBS );
      }
```

```
//access to pixel in a square block of the whole image:
//(v_block, h_block) refers to given block in image
//(v_pix, h_pix) refers to pixel in the block wrt bottom left corner

Lval<T> operator()(int v_block, int h_block, int v_pix, int h_pix){

   int hh_pix = h_block*IBS + h_pix;
   int vv_pix = v_block*IBS + v_pix;
   return SpreadArray<T>::operator()(vv_pix/(VIS/VPS),
                                     hh_pix/(HIS/HPS),
                                     (vv_pix%(VIS/VPS))/IBS,
                                     (hh_pix%(HIS/HPS))/IBS,
                                     (vv_pix%(VIS/VPS))%IBS,
                                     (hh_pix%(HIS/HPS))%IBS );
}

//access to square block:
//(v_block, h_block) refers to block in the whole image,
//actually the pixel in its bottom left corner

Lval<T> block(int v_block, int h_block){
   return operator()(v_block, h_block, 0, 0);
}

//local access to pixel in a square block of image:
//(v_block, h_block) refers to a local block of image
//(v_pix, h_pix) refers to pixel in the block wrt bottom left corner

T& local(int v_block, int h_block, int v_pix, int h_pix){
   return
     to_local_ptr()[(v_block*(HIS/HPS) + h_block*IBS + v_pix)*IBS + h_pix];
}

//access to square local block:
//(v_block, h_block) refers to local block,

T& local_block(int v_block, int h_block){
   return local(v_block, h_block, 0, 0);
}
```

```
      //conversion from local coordinates of pixel in a block to
      //the offset of pixel relative to the beginning of block

      int to_linear_addr(int v_pix, int h_pix){
         return v_pix*IBS + h_pix;
      }

      //compute the v_coordinate of a pixel in the whole image
      //from v_cood of processor, v-coord of block in processor
      //and v_coord of pixel in block

      int v_spread_to_bidim(int v_proc, int v_block, int v_pix){
         return (VIS/VPS)*v_proc + IBS*v_block + v_pix;
      }

      //compute the h_coordinate of a pixel in the whole image
      //from h_cood of processor, h_coord of block in processor
      //and h_coord of pixel in block

      int h_spread_to_bidim(int h_proc, int h_block, int h_pix){
         return (HIS/HPS)*h_proc + IBS*h_block + h_pix;
      }
};
```

7.1.2 Example 2

This example describes the class *SpreadQuadtree*. The first phase of the segmentation algorithm builds a spread array of quadtree records. There is one quadtree record generated per image block. The second phase builds a graph of connected homogeneous square regions from this spread array. It extensively makes use of the SpreadQuadtree access functions that allows for symbolic access (indexing) to the records.

```
template<int VPS, int HPS, int VIS, int HIS, int IBS>
class SpreadQuadtree: public SpreadArray<Quadtree_rcd>{

   public:

      SpreadQuadtree(int tag)
      : SpreadArray<Quadtree_rcd>(tag, VPS, HPS, (VIS/IBS)/VPS, (HIS/IBS)/HPS, 2)
      { for(int v = 0; v < (VIS/IBS)/VPS; v++)
```

```
for(int h = 0; h < (HIS/IBS)/HPS; h++){
           (local(v, h)).v_block = v;
           (local(v, h)).h_block = h;
}
   }


   ~SpreadQuadtree(){
       for(int v = 0; v < (VIS/IBS)/VPS; v++)
          for(int h = 0; h < (HIS/IBS)/HPS; h++)
             delete []local(v, h). qt_heap_ptr;
             //these were created in split_one_block()
   }

   //access to Quadtree_rcd slot in the whole SpreadQuadtree array:
   //(v_proc, h_proc) refers to processor
   //(v_slot, h_slot) to slot in processor

   Lval<Quadtree_rcd> operator()(int v_proc, int h_proc,
                                 int v_slot ,int h_slot){
      return SpreadArray<Quadtree_rcd>::operator()(v_proc, h_proc,
                                                    v_slot, h_slot );
   }

   //local access to Quadtree_rcd slot:
   //(v_slot, h_slot) are local slot coordinates

   Quadtree_rcd& local(int v_slot, int h_slot){
      return to_local_ptr()[v_slot*(HIS/IBS)*HPS + h_slot];
   }

   //conversion from local coordinates of a SpreadQuadtree slot into
   //the offset of the slot relative to the beginning of local section

   int to_linear_addr(int v_slot, int h_slot){
      return v_slot*((HIS/IBS)/HPS) + h_slot;
   }

   //check locality of access
```

```
      //(v_proc, h_proc) are processor coordinates

      int is_local(int v_proc, int h_proc){
         if(self_address == v_proc*HPS + h_proc)
            return 1;
         else
            return 0;
     }

};
```

7.1.3 Example 3

In the reduction phase the algorithm needs to use a global structure *SpreadLinkArrayTr* intended to link the edge descriptors to one another. There is a table of edge descriptors per vertex on each node. *SpreadLinkArrayTr* actually is an abstraction re-mapping triangular arrays as spread arrays (symmetry of vertex relationship in this problem).

```
template<int VPS, int HPS>
class SpreadLinkArrayTr: public SpreadArray<Link_rcd_pair>{

      int size;

   public:

      //SpreadLinkArrayTr exactly has the size s required and no more.
      //It actually maps a triangular array to a rectangular one.
      //X is an int parameter that can be used to get the right load balance
      //(fairest repartition) of the triangular array.
      //((s-1)(s+2))/2 +1) is the linear size of the triangular array,
      //where s is the vertical (and horizontal size) of complete array.
      //The horizontal size of the local arrays in  SpreadLinkArrayTr is:
      //(!(((s-1)(s+2))/2 +1)%(VPS*HPS*X)?
      //(((s-1)(s+2))/2 +1)/(VPS*HPS*X):
      //(((s-1)(s+2))/2 +1)/(VPS*HPS*X)+1);
      //the vertical size is X.

      SpreadLinkArrayTr(int tag, int s)
      : SpreadArray<Link_rcd_pair>(tag, X, VPS*HPS,
                                   (!(((s-1)(s+2))/2 +1)%(VPS*HPS*X)?
```

```
                              (((s-1)(s+2))/2 +1)/(VPS*HPS*X):
                              (((s-1)(s+2))/2 +1)/(VPS*HPS*X)+1), 2)
{
   size = s;
}

~SpreadLinkArrayTr(){}

Lval<Link_rcd_pair>operator()(int v, int h){

   //access is considered as performed in the upper triangle (arbitrary).
   //Note that getting access to (v, h) thus is the same as getting access
   //to (h, v). The procedure automatically put the pair in the form (i, j)
   //with i>=j. v and h must be in the range 0, size-1.

   if(v > size-1 || h > size-1){
      printf("on node %d: incorrect access to triangular array\n",
                   self_address);
      preempt();
   }

   if(v > h){
      int h1 = h; h = v; v = h1;
   }

   //(v*(v+3))/2 + h + 1 - s is the linear rank of (v, h) element
   //in the upper triangular array starting from element (0, s-1).
   //For efficiency reasons, the access pointer is constrained to
   //satisfy proc_id == 0 and offset == 0. We do not make use here
   //of the SpreadArray<Link_rcd_pair> access function. That would
   //be too general for our purpose. This one is more efficient.


   if(get_proc_id() || get_offset()){
      printf("on node %d: incorrect access to triangular array\n",
                   self_address);
      preempt();
   }
   rank_to_coord((v*(v+3))/2 + h + 1 - s);
```

```
        coor_to_ref(get_SA_attr_ptr()->index_ptr, sizeof(Link_rcd_pair));
        }
};
```

7.1.4 Spread array declaration

Spread array declarations can be explicitly or implicitly performed.

Explicit declaration

Constructor

template<class T>
SpreadArray<T>::SpreadArray(int t, int s0, int s2,..., int sn, int m);

Declaration form

SpreadArray<type> name(t, s0, s2,..., sn, m);

This declares a spread array of *type* data. The name of the spread array is *name*. It has n+1 dimensions, the first m of which (m<=n+1) are spread dimensions. *t* is the tag associated to the spread array as a SpreadArray is implicitly considered as global data. *t* **must** be a non 0 tag given as *DIR(integer)* (a tag given as *INDIR(integer)*would not make sense).

Currently, the number n is limited to 7 but can be increased at the expense of a minor expansion that needs to be performed on the code of the library.

Global synchronization

A spread array construction or destruction needs be globally synchronized for the data to be defined before any processor starts performing access to them. For efficiency reasons, global synchronization is not automatically performed in the spread array constructor and destructor itself. This is left to the user who is expected to gather these operations as much as possible in order to reduce the cost of barrier synchronizations.

Notes

- Getting a *SpreadArray* class with implicitely synchronized constructors and destructors is straightforward by deriving a new class, say *SpreadArray_s*, from *SpreadArray* with constructors and a destructors that just perform a synchronization barrier.

- When performing a global (barrier) synchronization, one should use one of the functions: *void flush();* or *void flush(int n);* (see section 5). They perform the same job as a usual barrier but additionally perform polling so possibly not yet completed *read/write* functions can complete. The flush functions perform a distributed completion of read/write functions by reducing local completion upon a processor virtual tree and broadcasting global completion down the tree. The degree of reduction/broadcasting virtual tree can be passed as an argument to the second function.
- When the non-synchronized declaration mode is used, one should be aware that the destructors are silently called at the end of blocks so global synchonization should explicitly be performed at the end of each block containing *SpreadArray* declarations. Making use of synchronized declarations and switching to non-synchronized ones at the performance optimization stage could be a good practice.

Examples

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

  declares SA as a SpreadArray of int-typed data. SA has 2 spread dimensions with size 4 and 5 respectively and one internal dimension with size 2.

- In example 1, the class SpreadImages derives from a SpreadArray with 2 spread and 2 internal dimensions of any T-typed data:

  SpreadArray<T> (tag, VPS, HPS, (VIS/IBS)/VPS, (HIS/IBS)/HPS, IBS, IBS, 2)

- In example 2, SpreadQuadtree derives from a SpreadArray with 2 spread and 1 internal dimension of any Quadtree_rcd-typed data:

  SpreadArray<Quadtree_rcd>(tag, VPS, HPS, (VIS/IBS)/VPS, (HIS/IBS)/HPS, 2)

- In example 3, SpreadLinkArrayTr derives from a SpreadArray with 2 spread and 1 internal dimensions of Link_rcd_pair-typed data:

  SpreadArray<Link_rcd_pair>(tag, X, VPS*HPS,
  $(!(((s-1)(s+2))/2+1)\%(VPS*HPS*X)?$

$$(((s-1)(s+2))/2+1)/(VPS*HPS*X):$$
$$(((s-1)(s+2))/2+1)/(VPS*HPS*X)+1),2)$$

Implicit declaration

Constructor

template<class T>
SpreadArray<T>::SpreadArray(int t, int *d_ptr, int *s_ptr, int s_nb, int i_nb);

Declaration form

SpreadArray<type> name(t, d_ptr, s_ptr, s_nb, i_nb);

This declares a SpreadArray of *type* data with name *name*. The spread and internal dimensions are supplied in the array pointed to by *d_ptr* and *i_ptr* respectively. There are *s_nb* and *i_nb* spread and internal dimensions respectively. *t* is the tag attached to the *SpreadArray*. *t* must be a non 0 tag given as *DIR(integer)* (using indirect handle would not make sense).

Example

The explicit declaration given earlier can be turned into this implicit one:

int s_ptr[2] = {4, 5};
int i_ptr[1] = {2};
SpreadArray<int> SA(tag, s_ptr, i_ptr, 2, 1);

Getting the local base of a SpreadArray

template<class T>
T *SpreadArray<T>::to_local_ptr();

The function is to be used for access to the local section of a *SpreadArray*.

Data structures generated for a SpreadArray

The declaration of a SpreadArray gives rise to the generation of data structures for storing and retrieving data in the SpreadArray. Two data records are created for this purpose:

- a S-GPType global pointer record,
- a SA_attributes record.

The internal structure of global pointer records is described in section 6.1. The members of the S-GPType global pointer record generated by a SpreadArray creation are set as follows:

- **proc_id**: 0 (note this is not the null value NOPID),
- **tag**: non 0 and direct tag attached to the SpreadArray,
- **offset**: 0 (note this is not the null value NOFFST),
- **attributes_ptr**: pointer to the SA_attributes record attached to the SpreadArray,
- **ctrl_ptr**: NOCP (i.e. 0).

The *SA_attributes* record holds the set of data that describes the spread and internal dimensions of the SpreadArray. these data are used for index expression decoding.

7.2 Spread pointers

An S-GPType global pointer is automatically generated each time a *SpreadArray* is created which is more simply denoted as a *spread pointer*. A spread pointer is a restricted U-GPType global pointer for which:

- the tag must be associated a direct data handle and be different from 0,
- the attributes_ptr is set with the pointer to a SA_attributes record.

Any U-GPType global pointer can be turned into a spread pointer by setting these two members appropriately. This can be performed by assigning a SpreadArray or spread pointer identifier to the global pointer by using the global pointer constructors described in section 6.5 or by individually setting the global pointer members with the functions presented in section 6.12. The type of a spread pointer can be checked by using the function get_gptr_type() described in section 6.2.

Example

```
SpreadArray<int> SA(1, 4, 5, 2, 2);
Star<int> SA_crt1(SA);
Star<int> SA_crt2 = SA;
```

7.3 Operations on spread pointers

It was earlier noted that a spread pointer is a S-GPType global pointer. All operators that apply to global pointers and are described in section 6 thus apply to spread pointers. Some operators however apply with different semantics and there is an additional indexing operator.

7.3.1 Spread pointer based predicates

== same proc_id, same tag, same offset, same SA_attributes_ptr, and same ctrl_ptr.

> ((lhs proc_id > rhs proc_id and same offset) or lhs offset > rhs offset) and same tag, same SA_attributes_ptr and same ctrl_ptr.

>= ((lhs proc_id >= rhs proc_id and same offset) or lhs offset >= rhs offset) and same tag, same SA_attributes_ptr and same ctrl_ptr.

< ((lhs proc_id < rhs proc_id and same offset) or lhs offset < rhs offset) and same tag, same SA_attributes_ptr and same ctrl_ptr.

<= ((lhs proc_id <= rhs proc_id and same offset) or lhs offset <= rhs offset) and same tag, same SA_attributes_ptr and same ctrl_ptr.

!= different proc_id or different local_ptr or different SA_attributes_ptr and same tag, same SA_attributes_ptr and same ctrl_ptr.

7.3.2 Spread pointer arithmetic operators

The semantics of usual operators: +, -, ++(pre/post), –(pre/post), +=, -= follows the wrapping rule described earlier in section 7.1.

7.3.3 Indexing '()'

A spread pointer attached to a spread array can be used for accessing the data held in it via indexing. Address computing is done according to the wrapping rule described in section 7.1 and makes use of the pointer to the *SA_attributes* record attached to the spread array. Note that the current *offset* value of the spread pointer also takes part to the address computation. Indexing provides a *Lval* that can subsequently be used with *operator=()* (see section 6.7), *operator T()* (see section 6.8), or as an argument to a global *read/write* function (see sections 6.9 and 6.10).

7.4 Remote Arrays

A remote array is a data structure that is created on a processor but is intended to be accessed from remote ones.

7.4.1 Example

The Image segmentation algorithm runs a load-balancing procedure in its third phase that is expected to keep the number of vertices attached to the processors approximately balanced as the reduction process goes on. The load balancing procedure is executed from time to time when the current load appears not to be uniformly spread.

When it is started, the load balancing procedure builds an array on each overloaded node. Each slot is assigned to a given under-loaded processor. As the load-balancing procedure is executed, the array is remotely used by the under-loaded processors for writing the number of vertices they decide to get from this node and the location in their local memories into which these vertices should be imported. Each under-loaded processor uses the slot it has been assigned. Writing is remotely done by the under-loaded processors in a subsequent step and in a way transparent to the overloaded one.

Symetrically, each under-loaded processor build an array for vertices that are to be imported to their local memories by overloaded processors. These arrays are used by the overloaded processors for writing in the same transparent way. Such arrays typically are remote arrays (i.e. defined at some location for being transparently accessed from some other remote ones).

Below is a code segment (extracted from the load-balancing procedure) that gives the flavor of how this can be written with the ARCH library.

```
RemoteArray<Load_picked_up> *export_data;
RemoteArray<Vertex_rcd> *import_data;

if(over_or_under_load > 0)
   //processor is overloaded. There are as much slots in array
   //export_data as there are underloaded processors
   export_data = new RemoteArray<Load_picked_up> (tag1, underloaded_nb);
else
   if(over_or_under_load < 0)
      //processor is underloaded. There are as much slots in array
      //import_data as there are vertices that need to be imported
```

```
         import_data = new RemoteArray<Vertex_rcd>(tag2, -over_or_under_load);

        //the members proc_id and ctrl_ptr have been initialized with the
        //null values NOPID, NOCP. offset with 0

//global synchronization for global data to be created everywhere before
//each processor starts getting access to them.
barrier();
.
.
.
//As the procedure is executed in the user-controlled-completion mode,
// RWCtrl data record is constructed and pointer to it is loaded in
//export_data's ctrl_ptr member.
RWCtrl ctrl( (func)upon_end_of_RW, (void *)rw_c);
export_data.set_ctrl(&ctrl);
.
.
.
//proc_id is only known dynamically. this line setup  proc_id
//that was initialized with  NOPID
export_data.set_proc_id(over_loaded_proc_id);
.
.
.
//counter in object rw_c is incremented and remote access is started
rw_c->more_writing(1);
//Load_requested record is built and written into the remote export_data
// array in slot attached there to this underloaded processor.
write_c(export_data(underloaded_rank),
Load_requested(self_address , import_data_indx, nb_of_vertices));

//program keep on executing
.
.
.
//the handler upon_end_of_RW() decrements counter in object rw_c
//wait here while counter value is !=0
wait_while(RW_not_complete);
```

7.4.2 Remote array declaration

Explicit declaration

Constructor

template<class T>
RemoteArray<T>::RemoteArray(int tag, unsigned d0,..., unsigned dn);

Declaration form

RemoteArray<type> name(t, d0, ..., dn);

declares a remote array of *type* data with name *name*. The array has n+1 dimensions. Currently, n is limited to 7 but can be increased at the expense of a minor expansion that would need to be performed on the code of the library.

t is the tag associated to the spread array as a SpreadArray is implicitly considered as global data. *t* **must** be a non 0 tag given as *DIR(integer)* (only attaching direct handle makes sense).

Global synchronization

A remote array construction or destruction needs be globally synchronized for the data to be defined before any processor starts performing access to them. For efficiency reasons, global synchronization is not automatically performed in the remote array constructor or destructor itself. This is left to the user who is expected to gather these operations as much as possible in order to reduce the cost of barrier synchronizations.

Notes

- Getting a *RemoteArray* class with implicitly synchronized constructors and destructors is straightforward by deriving a new class, say *RemoteArray_s*, from *RemoteArray* with constructors and a destructors that just perform a synchronization barrier.

- When performing a global (barrier) synchronization, one should use one of the functions: *void flush();* or *void flush(int n);* (see section 5). They perform the same job as a usual barrier but additionally polls so possibly not yet completed *read/write* functions can complete. The flush functions perform a distributed completion of *read/write* functions by reducing local completion upon a processor virtual tree and broadcasting global completion down the tree. The degree of reduction/broadcasting tree can be passed as an argument to the second function.

- When the non-synchronized declaration mode is used, one should be aware that the destructors are silently called at the end of blocks so global synchronization should explicitly be performed at the end of each block containing *RemoteArray* declarations. Making use of synchronized declarations and switching to non-synchronized ones at the performance optimization stage could be a good practice.

Examples

RemoteArray<Load_picked_up> *export_data;
RemoteArray<Vertex_rcd> *import_data;

Implicit declaration

Constructor

template<class T>
RemoteArray<T>::RemoteArray(int tag, unsigned *dim_ptr, unsigned dim_nb);

Declaration form

RemoteArray<type> name(t, dim_ptr, dim_nb);

declares a RemoteArray of *type* data with name *name*. The dimensions are supplied in the array pointed to by *dim_ptr* and the number of dimensions in *dim_nb*.

t is the tag associated to the spread array as a SpreadArray is implicitly considered as global data. *t* must be a non 0 tag given as *DIR(integer)*.

Example

the declaration below generates a 3-dimensional RemoteArray on each node.

int dim_ptr[2] = {4, 5, 2};
RemoteArray<int> SA(tag, dim_ptr, 3);

Getting the local base of a RemoteArray

template<class T>
T *RemoteArray<T>::to_local_ptr();

The function is to be used for access to the local section of a *RemoteArray*.

Data structures generated for a RemoteArray

The declaration of a RemoteArray gives rise to the generation of data structures for storing and retrieving data in the RemoteArray. Two data records are created for this purpose:

- a R-GPType pointer record,
- a RA_attributes record.

The internal structure of global pointer records is described in section 6.1. The members of the R-GPType global pointer record generated by a SpreadArray creation are set as follows:

- **proc_id**: NOPID (i.e. PROC_NB, an impossible value for proc_id),
- **tag**: non 0 and direct tag attached to the RemoteArray,
- **offset**: 0 (note this is not the null value NOFFST),
- **attributes_ptr**: pointer to the RA_attributes record attached to the RemoteArray,
- **ctrl_ptr**: NOCP (i.e. 0),

The *RA_attributes* record holds the set of data that describes the spread and internal dimensions of the RemoteArray. these data are used for index expression decoding.

7.5 Remote pointers

A R-GPType global pointer is automatically generated each time a *RemoteArray* is created which is more simply denoted as a *remote pointer*. A spread pointer is a restricted U-GPType global pointer for which:

- the tag must be associated a direct data handle and be different from 0,
- the attributes_ptr is set with the pointer to a RA_attributes record.

Any U-GPType global pointer can be turned into a remote pointer by setting these two members appropriately. This can be performed by assigning a RemoteArray or remote pointer identifier to the global pointer by using the global pointer constructors described in section 6.5 or by individually setting the global pointer members with the functions presented in section 6.12. The type of a remote pointer can be checked by using the function get_gptr_type() described in section 6.2.

Example

```
RemoteArray<int> RA(1, 4, 5, 2);
Star<int> RA_crt1(SA);
Star<int> RA_crt2 = RA;
```

7.6 Operations on remote pointers

It was noted ealier that a remote pointer is a R-GPType global pointer. All operators that apply to global pointers and are described in section 6 thus apply to remote pointers. Some operators however apply with different semantics and there is an additional indexing operator.

7.6.1 Remote pointer based predicates

== same proc_id and same tag and same offset and same attributes_ptr and same ctrl_ptr.

> same local_ptr and same tag and lhs offset > rhs offset and same attributes_ptr and same ctrl_ptr.

>= same local_ptr and same tag and lhs offset >= rhs offset and same attributes_ptr and same ctrl_ptr.

< same local_ptr and same tag and lhs offset < rhs offset and same attributes_ptr and same ctrl_ptr.

<= same local_ptr and same tag and lhs offset <= rhs offset and same attributes_ptr and same ctrl_ptr.

!= different proc_id or different tag or different offset or different attributes_ptr or different ctrl_ptr.

7.6.2 Remote pointer arithmetic operators

The semantics of usual operators: +, -, ++(pre/post), –(pre/post), +=, -=, follow the rules of the regular arithmetic.

7.6.3 Indexing '()'

A remote pointer attached to a remote array can be used for accessing the data held in it via indexing. Address computing is done according to regular arithmetic. Note that the current offset value of the remote pointer also takes part to the address computation. Indexing provides a Lval that can subsequently be used with *operator=()* (see section 6.7), *operator T()* (see section 6.8), or as an argument to a global *read/write* function (see sections 6.9 and 6.10).

8 User controlled send and recv

The *send/recv* functions that are described in the section 3 are completely monitored by the local scheduler. This is enough for machine with fast context-switching. For slower context-switching machines, it is a good strategy to avoid switching contexts as much as possible. In particular when the user program has only one thread. This section describes communication functions with *user-controlled completion* that helps achieve this objective.

8.1 ChanR_c class

Using a *ChanR_c* channel allows one to synchronize processes running on separate processors. As opposed to what is done for *ChanR* channels section 3, the completion of *send* and *recv* functions applied to *ChanR_c* channels is expected to be explicitly monitored by the user via a completion handler.

8.1.1 Constructor

ChanR_c(int tag, int remote_proc);

8.1.2 Declaration forms

ChanR_c name(tag, remote_proc, length);

This declares a channel intended to carry pure synchronization signals. Termination is controlled by the user at both ends of the channel. This is performed in the same spirit as for *read/write* functions (see section 5 and section 8.1.3 below).

8.1.3 Completion handler

ChanR_c channels are tools for building virtual communication networks over the set of processors. The data structure representing a *ChanR_c* on a node holds both a sender and receiver side record. The following functions allow both records to be loaded with the pointer to a completion handler.

typedef void (*HSR_ptr)(ChanR_c *);

void ChanR_c::set_ctrl_recv(HSR_ptr handler, void *arg);

This function sets the *ChanR_c* object it is applied to with two pointers one to a completion handler and another to its arguments. These data are loaded in the sender side record.

```
void ChanR_c::set_ctrl_send(HSR_ptr handler, void *arg);
```

This function sets the *ChanR_c* object it is applied to with two pointers one to a completion handler and another to its arguments. These data are loaded in the receiver side record.

in both cases *handler* is a function with the profile below:

```
void handler(ChanR_c *ptr);
```

Upon completion the system dynamically binds *ptr* to the *ChanR_c* channel supporting the *send/recv* operation. The handler can get access to all relevant information via *ptr* using the following *Chan_c* member functions:

- **on sender side:**

```
int get_remote_proc_s();          //processor message was sent to,
Thread *get_thread_ptr_s();       //thread in which execution takes place,
HSR_ptr get_func_ptr_s();         //pointer to send completion handler,
void *get_arg_s();                //pointer to additional arguments,
int get_data_size_s();            //size of last message sent (nb of bytes),
                                  //0 for ChanR_c, meaningful only for derived
                                  //templat<class T> ChanRD_c
```

- **on receiver side:**

```
int get_remote_proc_r();          //processor message was received from,
Thread *get_thread_ptr_r();       //thread in whitch execution takes place,
HSR_ptr get_func_ptr_r();         //pointer to recv completion handler,
void *get_arg_r();                //pointer to additional arguments,
int get_data_size_r();            //size of last message received (nb of bytes),
                                  //0 for ChanR_c, meaningful only for derived
                                  //templat<class T> ChanRD_c
```

- **on both sender and receiver side:**

```
int get_max_data_size();          //tag attached to ChanR_c,
int get_tag();                    //maximum size of messages ChanR_c can carry,
```

//(nb of bytes), 0 for ChanR_c, meaningful
//only for derived templat<class T> ChanRD_c

8.1.4 Synchronization functions

void ChanR_c::send_s();
void ChanR_c::send_r();
void ChanR_c::recv();

send_s returns immediately but completion occurs only when *recv* is executed at the other end of the *ChanR_c* both functions are applied to. *send_r* is delayed until *recv* is is executed at the other end of the *ChanR_c* channel both are applied. These follow the semantics of MPI *send/recv* functions (see MPI manual).

8.2 ChanRD_c class

These channels are intended for synchronization and/or message passing with a user controlled completion that is performed the same as we already described for *ChanR_c* channels.

8.2.1 Constructor

template <class T>
ChanRD_c::ChanRD_c(int tag, int remote_proc, int max_data_length=1);

8.2.2 Declaration form

ChanRD_c<type> name (tag, remote_proc, length);

This declares a channel that can perform synchronized and/or non-synchronized communications. On both ends of the channel, completion is controlled by a handler supplied by the user. The *length* argument is the maximum length of messages the channel is expected to carry.

8.2.3 Termination handler

ChanRD_c channels are tools for building virtual communication networks over the set of processors. The data structure representing a *ChanRD_c* on a node has both a sender and receiver side record.

template <class T>
void ChanRD_c::set_ctrl_recv(func handler, void *arg);

template <class T>
void ChanRD_c::set_ctrl_send(func handler, void *arg);

These functions set a ChanRD_c channel with completion handler and arguments in the same way as what was described for *ChanR_c* channels. Data can be used in *handler* that can be accessed via the set of functions already described above: section 8.1.3 complemented with the following three functions:

- on sender side: void *get_data_ptr_s(); //source address of message
- on receiver side: void *get_data_ptr_r(); //destination address of message
- on both sender and receiver sides: int get_Tsize(); //size of T type

8.2.4 Communication/synchronization functions

template <class T>
void ChanRD_c::send(T *msg_from_ptr, int l=1);

template <class T>
void ChanRD_c::send_b(T *msg_from_ptr, int l=1);

template <class T>
void ChanRD_c::send_s(T *msg_from_ptr, int l=1);

template <class T>
void ChanRD_c::send_r(T *msg_from_ptr, int l=1);

template <class T>
void ChanRD_c::recv(T *msg_to_ptr);

These communication and/or synchronization functions can be applied to a *ChanRD_c<type>* channel. The msg_from_ptr and msg_to_ptr arguments stand for the source and destination of the current message. The *l* parameter defines the length of the current message (as a number of T-typed items). This value is checked against the maximum value supplied at channel declaration time. All functions return immediately. The behavior of functions follows the semantics of similar MPI functions (see MPI manual).

8.2.5 Example

This example is borrowed from the image segmentation algorithm. The code fragments are parts of the load-balancing procedure that is expected to keep the load over the set of processors approximately uniform. The load of a node is the number of vertices of the distributed graph held by the node. The code fragments are part of the sequence that performs circulation of load offers.

```
chan_c = new ChanRD_c<Load_offered>(tag, (self_address+1)%PROC_NB)
.
.
.
SR_count sr_c;
chan_c->set_ctrl_send((func)upon_end_of_SR, &sr_c);
chan_c->set_ctrl_recv((func)upon_end_of_SR, &sr_c);
.
.
.
sr_c.more_SR(2);
chan_c->send(&load_offered_out);
chan_c->recv(&load_offered_in);
.
.
.
wait_while(SR_not_complete);
```

Acknowledgement

Special thanks to Marcy Rosenkrantz for the invitation for me to visist the Cornell Theory Center. Hubertus Franke from IBM-Yorktown kindly provided information about the MPI-F implementation. Chris Myers performed the difficult task of reading an early version of the manuscript and provided insightful suggestions. I had interesting discussions regarding the SP2 and the software system with Dave Schneider and John Zollweg. Donna Bergmark and Dave kindly accepted to spend time in installing additional Latex tools I needed to complete this report. Marcia Pottle helped me start with the SGI-Challenge in the project of porting the library to this machine. Many thanks also to the *runners group*, including Mike Hammill, Shannon Shen and Maurice Van Putten for providing friendly conversation and the necessary diversion from the work in this project. Mike read part of this report and gave helpful suggestions for improving the presentation.

Many thanks to Monique, by my side regardless of the distance.

References

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

[2] Adamo J-M. "Object-Oriented Parallel Programming: Library Design and Development for SPMD Programming", ICSI, Berkeley, TR-94-011, 1994.

[3] Adamo J-M., Anguita D., "Object-Oriented Design of Parallel BP Neural Network Simulator and Implementation on the connection machine CM-5", IWANN'95, Malaga, Spain, June 95.

[4] Alpern B., "The Myth of Scalability Analysis", Cornell Theory Center Seminar, August 29, 1995.

[5] Baden S. et al., "The LPARX User's Guide, V2.0", DCSE, Univ. California San Diego, Nov. 1994.

[6] Bader D.A. et al., "Parallel Algorithms for Image segmentation Enhencement and Segmentation by Region Growing with an Experimental Study", IACS, Univ. of Maryland, College Park, May 1995.

[7] 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.

[8] Chang Y-L. et al. "Adaptative Image Region Growing", IEEE Trans on Image Processing, 3(6), 868-872, 1994.

[9] Copty N. et al., "A Data Parallel Algorithm for solving the Region Growing Problem on the Connection Machine", JPDC, 21(1), 160-168, April 1994.

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

[11] Culler D.E. et al., "Parallel Programming in Split-C", Computer Science Dept. U.C. Berkeley, 1993.

[12] Culler D.E. et al., "Generic Active Message Interface Specification, Version1.1", Computer Science Dept. UC Berkeley, Nov. 1994.

[13] 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., pp268-269, April 1993.

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

[15] Franke H., "MPI-F An MPI Implementation for IBM SP-1/SP-2, Version 1.41", IBM, Watson Research Center, May, 1995.

[16] Franke H. et al, "MPI Programming Environment for IBM SP-1/SP-2", IBM, Watson Research Center, 1995.

[17] Gil J., "Renaming and Dispersion Techniques for Parallel Computers", Journal of Parallel and Distributed Computing, 23, 149-157 (1994).

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

[19] INMOS ldt., "OCCAM2, Reference Manual", Prentice, series in computer science, 1988.

[20] Kumar V. et al., "Scalable Load Balancing Techniques for Parallel Computers", Journal of Parallel and Distributed Computing 22, 60-79 (1994).

[21] Lippman S.B. "C++ Primer", Addison-Weysley, 1992.

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

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

[24] Stunkel C B. et al., "The SP2 Communication Subsystem", IBM, Watson Research Center, August, 1994.

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

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

A Pointers to the files in the library

A.1 Package presentation

The source files are in ./Src, the documentation file in ./Doc. I am providing a set of test programs in ./Test. I am also providing in ./RegGrow the files of the image segmentation algorithm that I have used to illustrate the documentation. The program shows how the library can contribute to developping real world parallel programs with dynamic load balancing involved. Please, note that I am still working on it. The whole file system compiles well. Some of the procedures are fully tested others need additional testing and some local data structures will probably change in the future.

A.2 Making the Library

Read ./Makefile and change the path-variable setting (include and libraries) to adapt to the system installation is performed on. Just enter "make" to make the ARCH library. The .o files, libarch.a and the include files are respectively created in ./ARCH_obj, ./Lib and ./Include.

Before making the library you should take a look at './Src/A.setconfig'. There are defined all the constants you might want to change to customize your installation. This file also provides a set of '#define' directives, each corresponding to a target processor. Selecting a particular one will allow the system to generate the context-switching procedure suitable to the platform you are installing on.

Two versions of the library are available. One runs makes use of MPI-F and Xlc and is intended for use on the IBM-SP2. The other one makes use of MPICH and g++ and should run on any system where these are available.

A.3 Using the library

Let <arch_lib_path> be the path to the base directory the library was created into. To use the library copy the Makefile you find in <arch_lib_path>/Test or in <arch_lib_path>/RegGrow. Change the INSTALL_DIR variable and the path variable setting to adapt to installation. You should first try some of the test programs to check the installation (take a look at the lines in Makefile corresponding to compiling and linking test programs). Next you can derive your own Makefiles.

Also take a look at the './Src/A.environment' file. There can dynamically be redefined all constants set up in the './Src/A.setconfig' file. This file also contains the definition of important global variables such as 'self_address' (processor rank in current run) and 'PROC_NB' (number of processors in current run).

The toy programs that appear in the next appendix section are in the files testdoci.C, with $1 <= i <= 11$. Playing with them could help you get started.

A.4 Source files

A.4.1 Threads, processes and scheduling

A.sched.h,
A.sched.C.

A.4.2 Synchronous message passing

A.syncSR.h,
A.syncSR.C.

A.4.3 User controlled message passing

A.ucontrSR.h,
A.ucontrSR.C.

A.4.4 Remote read/write

A.remoteRW.h,
A.remoteRW.C,
A.store.h,
A.store.C,
A.get.h,
A.get.C.

A.4.5 Polling, standard distributed termination procedures

A.poll-flush.h,
A.poll-flush.C.

A.4.6 Global-typed data

A.Global-data.h.

A.4.7 Spread/Remote Arrays and data structure

A.spread.interface.h,
A.spread.intern.C,
A.spread.intern.h.

A.4.8 environment files

A.environment,
A.setconfig,
A.class-declaration.

B sample programs

B.1 Sample program 1: Thread and processes

The nodes run independently from one another. The root process creates and runs two *Proc* instances that respectively run two *Proc* instances in turn (recursion on process creation). This code was one used to test the main scheduling functions.

The *A.environment* file brings down the library and a standard macro: *MAIN* that expends into the standard main() function. This function creates the scheduler data structures, starts the scheduler and schedules an instance of the *Root_proc* process class whose name is passed to the MAIN macro).

```
#include "A.environment"
#include "A.utilities"

//user program

class Proc: public Process{

   public:
        int pid;
        int step;

        Proc(int p, int s)
                : Process((Pmf0_ptr)&Proc::body){pid = p; step = s;}

        void body(){
               printf("node %d, in proc %d before loop at step %d\n",
                             self_address, pid, step);
               int i;
               for(i = 0; i < 20; i++){
                   printf("node %d, proc %d in for loop, i = %d\n",
                                self_address, pid, i);
                   reschedule();
               }
               printf("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[2] = {proc1, proc2};
                    int pri[2] = {1, 1};
                    par(2, proc_tabl, pri);
                    delete proc1;
                    delete proc2;
                }
                printf("on node %d, proc %d the end at step %d\n",
                       self_address, pid, step);
        }
};

class Root_proc: public Process{

   public:

        Root_proc()
                : Process((Pmf0_ptr)&Root_proc::body){}

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

MAIN(Root_proc, argc, argv)
```

B.2 Sample program 2: Threads, processes and synchronous communication

This code runs processes that communicate and synchronize via the synchronous message passing functions offered in the library. Process creation structure is partly similar to that in the previous sample program. The program below involves an additional process *Sync* which the *Proc* processes communicate and synchronize with. The first two levels *Proc* processes synchronize and communicate with the *Sync* processes respectively located on the left and right processors. The *Sync* processes handle non-deterministic message reception with the *alt* function. The four second level *Proc* processes also synchronize and communicate with the *Sync* processes but locally. The *Sync* processes once again handle non-deterministic message reception with the *alt* function.

```
#include "A.environment"

//user program

#define B 5

ChanLD<int> *chan[4];

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

ChanRD<Id> *proc_to_left;
ChanRD<Id> *proc_to_right;


class Proc: public Process{

   public:
        int pid;
        int step;
        Id id;

        Proc(int p, int s): Process((Pmf0_ptr)&Proc::body){
                pid = p;
                step = s;
```

```
        id.processor_Id = self_address;
        id.process_Id   = pid;
}

void body(){

        printf("node %d, in proc %d before loop, step %d\n",
                     self_address, pid, step);
        int i;
        for(i = 0; i < B; i++){
           printf("node %d, proc %d in for loop, i = %d\n",
                   self_address, pid, i);
           reschedule();
        }

        printf
        ("node %d, in proc %d after for loop, step %d\n",
         self_address, pid, step);

        if(step==0 && pid==1 && self_address>0){
               proc_to_left->send(&id);
               printf
               ("node %d, in proc %d, after send left, step %d\n",
                 self_address, pid, step);
        }

        if(step==0 && pid==2 && self_address<PROC_NB-1){
           proc_to_right->send(&id);
           printf
           ("node %d, in proc %d after send right, 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;
```

```
                    chan[index]->send(&pid);
                    printf
                    ("node%d,proc%d after send,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[2] = {proc1, proc2};
                    int pri[2] = {1, 1};
                    par(2, proc_tabl, pri);
                    delete proc1, delete proc2;
                 }

                 printf("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((Pmf0_ptr)&Sync::body){}

        void body(){
               printf("node %d, in Sync::body1\n", self_address);
               Alts *alt_table2[2];
               if(self_address > 0 && self_address < PROC_NB-1){
                     alt_table2[0] =
                          new AltRDrecv<Id>(proc_to_left,
                                            (Pmf2_ptr)&Sync::alt_body2,
                                            0,
                                            &id_left);
                     alt_table2[1] =
```

```
         new AltRDrecv<Id>(proc_to_right,
                           (Pmf2_ptr)&Sync::alt_body2,
                           0,
                           &id_right);
    AltCtrl * alt_ctrl2 = new AltCtrl(2, alt_table2);

    for(int i = 0; i< 2; i++){
       alt(alt_ctrl2);
   }

  delete alt_table2[0],  delete alt_table2[1];
  delete alt_ctrl2;

  printf("node %d, sync0\n", self_address);
}
if (self_address == 0){

   proc_to_left->recv(&id_left);

   printf
   ("node %d, sync1, id_left.prcr_Id = %d, id_left.prcs_Id = %d\n",
     self_address, id_left.processor_Id, id_left.process_Id);
}
if (self_address == PROC_NB-1){

   proc_to_right->recv(&id_right);

   printf
   ("node %d, sync2, id_right.prcr_Id = %d, id_right.prcs_Id = %d\n",
     self_address, id_right.processor_Id, id_right.process_Id);
}

Alts *alt_table4[4];
alt_table4[0] =
        new AltLDrecv<int>(chan[0],
                           (Pmf2_ptr)&Sync::alt_body4,
                           0,
                           &pid[0]);
alt_table4[1] =
```

```
            new AltLDrecv<int>(chan[1],
                                (Pmf2_ptr)&Sync::alt_body4,
                                0,
                                &pid[1]);
    alt_table4[2] =
            new AltLDrecv<int>(chan[2],
                                (Pmf2_ptr)&Sync::alt_body4,
                                0,
                                &pid[2]);
    alt_table4[3] =
            new AltLDrecv<int>(chan[3],
                                (Pmf2_ptr)&Sync::alt_body4,
                                0,
                                &pid[3]);

    AltCtrl *alt_ctrl4 = new AltCtrl(4, alt_table4);
    for(int i = 0; i< 4; i++){
       alt(alt_ctrl4);
    }

    printf("node %d, sync3\n", self_address);

    delete alt_table4[0],  delete alt_table4[1],
    delete alt_table4[2],  delete alt_table4[3];
    delete alt_ctrl4;
}


 void alt_body2(int rank, void *arg_ptr){
//notice the first implicit parameter

      if(rank==0)
          printf
          ("node%d,alt_body2.%d,id_left.prcr_Id=%d,id_left.prcs_Id=%d\n",
            self_address, rank, id_left.processor_Id, id_left.process_Id);

      if(rank==1)
       printf
       ("node%d,alt_body2.%d,id_right.prcr_Id =%d,id_right.prcs_Id=%d\n",
```

```
                self_address, rank, id_left.processor_Id, id_left.process_Id);

        }



        void alt_body4(int rank, void *arg_ptr){
            //notice the first implicit parameter

            printf("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((Pmf0_ptr)&Root_proc::body){}

        void body(){

                printf("node %d, in Root_proc::body1\n", self_address);
                proc_to_left = new ChanRD<Id>(1, self_address-1);
                               //synchronous
                proc_to_right = new ChanRD<Id>(2, self_address+1);
                               //synchronous

flush();

                chan[0]         = new ChanLD<int>;
                chan[1]         = new ChanLD<int>;
                chan[2]         = new ChanLD<int>;
                chan[3]         = new ChanLD<int>;

                Proc *proc1 = new Proc(1, 0);
                Proc *proc2 = new Proc(2, 0);
                Sync *sync  = new Sync;
```

```
                printf("node %d, in Root_proc::body2\n", self_address);

                Process *proc_tabl[3] = {proc1, proc2, sync};
                int pri[3] = {1, 1, 1};
                par(3, proc_tabl, pri);

                printf("node %d, in Root_proc::body4\n", self_address);

                delete proc1, delete proc2;
                delete proc_to_left, delete proc_to_right;
                delete chan[0],  delete chan[1],
                delete chan[2],  delete chan[3];
        }
};

//end of user program

MAIN(Root_proc, argc, argv)
```

B.3 Sample program 3: read/write, spread arrays and user-controlled completion

This program reads/writes from/to spread arrays. Completion of the *read/write* functions are monitored by a *user-supplied handler*.

```
#include "A.environment"

//user program

#define d0 4
#define d1 5
#define d2 2

class RW_count{
   public:
      int cnt;

      RW_count(){
         cnt = 0;
      }
      void reset(){
         cnt = 0;
      }
      void more_RW(int i){
         cnt += i;
      }
};

void
wait_while(int (*condition)()){
   while(condition())     //keep on polling condition
      poll();
}

RW_count *count;

void upon_end_of_RW(HRW_rcd *){
   count->cnt++;
}
```

```
int nc(){
   return (count->cnt < d0*d1*d2);
}

class Root_proc: public Process{

   public:
        Root_proc()
                : Process((Pmf0_ptr)&Root_proc::body){}

        void body(){
       printf("\n\n\nmain() is started\n\n\n");

           int LA0[d0][d1][d2];
           int LA1[d0][d1][d2];
           int LA2[d0][d1][d2];

           SpreadArray<int> SA0(DIR(1), d0, d1, d2, 2);
           SpreadArray<int> SA1(DIR(2), d0, d1, d2, 2);
           SpreadArray<int> SA2(DIR(3), d0, d1, d2, 2);
           //DIR is the only option for SpreadArrays

           flush();

           int *local_ptr0 = SA0.to_local_ptr();
           int *local_ptr1 = SA1.to_local_ptr();
           int *local_ptr2 = SA2.to_local_ptr();

           int d = (d0*d1*d2)/PROC_NB + 1;

           for(int z = 0; z < d; z++){
              local_ptr0[z] = 1000;
              local_ptr1[z] = 1001;
              local_ptr2[z] = 1002;
           }

           flush();

           RW_count c;
```

```
count = &c;

int (* not_complete)() = &nc;

RWCtrl ctrl(upon_end_of_RW, 0);
SA0.set_ctrl(&ctrl);
SA1.set_ctrl(&ctrl);
SA2.set_ctrl(&ctrl);

if(self_address == 0){
   for(int i = 0; i < d0; i++)
      for(int j = 0; j < d1; j++)
         for(int k = 0; k < d2; k++){
            write_c(SA0(i, j, k), 50);
         }
   wait_while(not_complete);
   flush();

   count->reset();
   for(i = 0; i < d0; i++)
      for(int j = 0; j < d1; j++)
         for(int k = 0; k < d2; k++){
            read_c(LA0[i][j][k], SA2(i, j, k));
         }
   wait_while(not_complete);
   flush();
}else{
   if(self_address == 1){
      for(int i = 0; i < d0; i++)
         for(int j = 0; j < d1; j++)
            for(int k = 0; k < d2; k++){
               write_c(SA1(i, j, k), 51);
            }
      wait_while(not_complete);
      flush();

      count->reset();
      for(i = 0; i < d0; i++)
         for(int j = 0; j < d1; j++)
```

```
            for(int k = 0; k < d2; k++){
               read_c(LA1[i][j][k], SA0(i, j, k));
            }
      wait_while(not_complete);
      flush();

   }else{ //self_address == 2
      for(int i = 0; i < d0; i++)
         for(int j = 0; j < d1; j++)
            for(int k = 0; k < d2; k++){
               write_c(SA2(i, j, k), 52);
            }
      wait_while(not_complete);
      flush();

      count->reset();
      for(i = 0; i < d0; i++)
         for(int j = 0; j < d1; j++)
            for(int k = 0; k < d2; k++){
               read_c(LA2[i][j][k], SA1(i, j, k));
            }
      wait_while(not_complete);
      flush();
   }
}

for(z = 0; z < d; z++){
   printf("node%d:local_ptr0=%d,local_ptr0[%d]=%d\n",self_address,
          local_ptr0, z,  local_ptr0[z]);
   printf("node %d:local_ptr1=%d,local_ptr1[%d]=%d\n",self_address,
          local_ptr1, z,  local_ptr1[z]);
   printf("node %d:local_ptr2=%d,local_ptr2[%d]=%d\n",self_address,
          local_ptr2, z,  local_ptr2[z]);
}

for(int i = 0; i < d0; i++)
  for(int j = 0; j < d1; j++)
     for(int k = 0; k < d2; k++){
        if(self_address == 0)
```

```
                        printf("node %d:  LA0[%d][%d][%d] = %d\n", self_address,
                                i, j, k,  LA0[i][j][k]);
                    if(self_address == 1)
                        printf("node %d:  LA1[%d][%d][%d] = %d\n", self_address,
                                i, j, k,  LA1[i][j][k]);
                    if(self_address == 2)
                        printf("node %d:  LA2[%d][%d][%d] = %d\n", self_address,
                                i, j, k,  LA2[i][j][k]);
                }
            printf("\n\n\nthe end\n\n\n");

            flush();
        }
};

//end of user program

MAIN(Root_proc, argc, argv)
```

B.4 Sample program 4: read/write, remote arrays and user-controlled completion

This program reads/writes from/to remote arrays. Completion of the *read/write* functions are monitored by a *user-supplied handler*.

```
#include "A.environment"

//user program

int count;

void handler(HRW_rcd *){
   count++;
}

class Root_proc: public Process{

   public:
        Root_proc()
                : Process((Pmf0_ptr)&Root_proc::body){}

        void body(){

           printf("from node %d: this is my rank\n", self_address);

           RemoteArray<int> RA(DIR(1), 3, 2);
           //DIR is the only possible option for RemoteArrays

           flush();

           Star<int> RA_crt = RA;

           RWCtrl ctrl(handler, 0);

           RA_crt.set_proc_id(self_address);
           RA_crt.set_ctrl(&ctrl);
           RA.set_proc_id(self_address);
           RA.set_ctrl(&ctrl);
```

```
    count = 0;
    for(int v = 0; v < 3; v++)
       for(int h = 0; h < 2; h++){
            write_c(RA_crt(v, h), self_address*10 + v*2+h);
       }

    while(count!= 6){
           poll();
    }

    int *RA_l_ptr = RA.to_local_ptr();

    printf("node %d: (1)RA =  %d, %d, %d, %d, %d, %d\n",self_address,
            RA_l_ptr[0*2+0], RA_l_ptr[0*2+1], RA_l_ptr[1*2+0],
            RA_l_ptr[1*2+1], RA_l_ptr[2*2+0], RA_l_ptr[2*2+1]);


    int *res = new int[2*3];

    for(v = 0; v < 3; v++)
       for(int h = 0; h < 2; h++){
          res[v*2+h] = 50+ v*2+h;
       }

    printf("node %d: (1)res =  %d, %d, %d, %d, %d, %d\n",self_address,
            res[0*2+0],res[0*2+1],res[1*2+0],res[1*2+1],
            res[2*2+0],res[2*2+1]);

    RA_crt.set_proc_id((self_address+1)%PROC_NB);
    RA.set_proc_id((self_address+1)%PROC_NB);

    count = 0;
    for(v = 0; v < 3; v++)
       for(int h = 0; h < 2; h++){
          read_c(res[v*2+h], RA(v, h));
          /*another possibility
  read_c(res[v*2+h], RA_crt(v, h));
          */
       }
```

```
            while(count!= 6){
               poll();
            }

            flush();

            printf("node %d: (2)RA =  %d, %d, %d, %d, %d, %d\n",self_address,
                    RA_l_ptr[0*2+0], RA_l_ptr[0*2+1], RA_l_ptr[1*2+0],
                    RA_l_ptr[1*2+1], RA_l_ptr[2*2+0], RA_l_ptr[2*2+1]);

            printf("node %d: (2)res =  %d, %d, %d, %d, %d, %d\n",self_address,
                    res[0*2+0],res[0*2+1],res[1*2+0],res[1*2+1],
                    res[2*2+0],res[2*2+1]);

            delete []res;

            printf("node %d: the end\n", self_address);
         }
};
//end of user program

MAIN(Root_proc, argc, argv)
```

B.5 Sample program 5: read, user-controlled completion and direct handle

This program reads from remote source. Completion is monitored by a *user-supplied handler*. *Global* data are defined with a *direct handle*.

```
#include "A.environment"

//user program

int count;

void handler(HRW_rcd *){
   count++;
}

class Root_proc: public Process{

   public:
        Root_proc()
                : Process((Pmf0_ptr)&Root_proc::body){}

        void body(){

           printf("from node %d: this is my rank\n", self_address);

           char source_data[21];
           char source_datax[21];

           for(int s=0; s<21; s++)
              source_data[s] = self_address;
           for(s=0; s<21; s++)
              source_datax[s] = self_address+50;

           char dest_data1[21];
           char dest_data1x[21];
           char dest_data2[21];
           char dest_data2x[21];

           Global<char> s_data(DIR(1), source_data);
```

```
            Global<char> s_datax(DIR(2), source_datax);

            flush();

            count = 0;

            RWCtrl ctrl(handler, 0);
            Star<char> remote_src1((self_address+1)%PROC_NB, s_data,
                                   (unsigned)0, NOAP, &ctrl);
            Star<char> remote_src1x((self_address+1)%PROC_NB, s_datax,
                                   (unsigned)0, NOAP, &ctrl);
            Star<char> remote_src2((self_address+2)%PROC_NB, s_data,
                                   (unsigned)0, NOAP, &ctrl);
            Star<char> remote_src2x((self_address+2)%PROC_NB, s_datax,
                                   (unsigned)0, NOAP, &ctrl);


           read_c(dest_data1[0],  *remote_src1, 21);
           read_c(dest_data1x[0], *remote_src1x, 21);
           read_c(dest_data2[0],  *remote_src2, 21);
           read_c(dest_data2x[0], *remote_src2x, 21);

           while(count!= 4){
              poll();
           }

           flush();

           for(int i=0; i<21; i++)
             printf("node %d: d1[%d]=%d, d1x[%d]=%d, d2[%d]=%d, d2x[%d]=%d\n",
                     self_address, i, dest_data1[i], i, dest_data1x[i], i,
                     dest_data2[i], i, dest_data2x[i]);
         }
};
//end of user program

MAIN(Root_proc, argc, argv)
```

B.6 Sample program 6: read, user-controlled completion and indirect handle

This program reads from remote source. Completion is monitored by a *user-supplied handler*. *Global* data are defined with an *indirect handle*.

```
#include "A.environment"

//user program
int count;

void handler(HRW_rcd *){
   count++;
}

char source_data[21];
char source_datax[21];

char* f(IDH_rcd *){
        return source_data;
}
char* fx(IDH_rcd *){
        return source_datax;
}

class Root_proc: public Process{

   public:
        Root_proc()
                : Process((Pmf0_ptr)&Root_proc::body){}

        void body(){

           printf("from node %d: this is my rank\n", self_address);

           for(int s=0; s<21; s++)
              source_data[s] = self_address;
           for(s=0; s<21; s++)
              source_datax[s] = self_address+50;
```

```
            Global<char> s_data(INDIR(1), f);
            Global<char> s_datax(INDIR(2), fx);

            char dest_data1[21];
            char dest_data1x[21];
            char dest_data2[21];
            char dest_data2x[21];
            flush();

            count = 0;
            RWCtrl ctrl(handler, 0);
            Star<char> remote_src1((self_address+1)%PROC_NB, s_data,
                                   (unsigned)0, NOAP, &ctrl);
            Star<char> remote_src1x((self_address+1)%PROC_NB, s_datax,
                                   (unsigned)0, NOAP, &ctrl);
            Star<char> remote_src2((self_address+2)%PROC_NB, s_data,
                                   (unsigned)0, NOAP, &ctrl);
            Star<char> remote_src2x((self_address+2)%PROC_NB, s_datax,
                                   (unsigned)0, NOAP, &ctrl);

            read_c(dest_data1[0],  *remote_src1, (int *)0, 21);
            read_c(dest_data1x[0], *remote_src1x, (int *)0, 21);
            read_c(dest_data2[0],  *remote_src2, (int *)0, 21);
            read_c(dest_data2x[0], *remote_src2x, (int *)0, 21);

            while(count!= 4){
               poll();
            }

            flush();

            for(int i=0; i<21; i++)
              printf("node %d: d1[%d]=%d, d1x[%d]=%d, d2[%d]=%d, d2x[%d]=%d\n",
                      self_address, i, dest_data1[i], i, dest_data1x[i], i,
                      dest_data2[i], i, dest_data2x[i]);
         }
};
//end of user program
```

```
MAIN(Root_proc, argc, argv)
```

B.7 Sample program 7: read, scheduler-controlled completion and direct handle

This program reads from remote source. *read* is considered as being performed in a *scheduler-controlled* setting. Completion is automatically monitored by the scheduler. *Global* data are defined with a *direct handle*.

```
#include "A.environment"

//user program

class Root_proc: public Process{

  public:

        Root_proc()
        : Process((Pmf0_ptr)&Root_proc::body){}


        void body(){

           printf("node %d: body is started\n", self_address);

           char source_data[4];
           char source_datax[4];
           for(int s=0; s<4; s++)
           source_data[s] = self_address;
           for(s=0; s<4; s++)
           source_datax[s] = self_address+50;

           Global<char> s_data(DIR(1), source_data);
           Global<char> s_datax(DIR(2), source_datax);

           flush();

           char dest_data1[4];
           char dest_data1x[4];
           char dest_data2[4];
           char dest_data2x[4];
```

```
          Star<char> remote_src1((self_address+1)%PROC_NB, s_data,
                                (unsigned)0, NOAP, NOCP);
          Star<char> remote_src1x((self_address+1)%PROC_NB, s_datax,
                                (unsigned)0, NOAP, NOCP);
          Star<char> remote_src2((self_address+2)%PROC_NB, s_data,
                                (unsigned)0, NOAP, NOCP);
          Star<char> remote_src2x((self_address+2)%PROC_NB, s_datax,
                                (unsigned)0, NOAP, NOCP);

          read(dest_data1[0],  *remote_src1, 4);
          read(dest_data1x[0], *remote_src1x, 4);
          read(dest_data2[0],  *remote_src2, 4);
          read(dest_data2x[0], *remote_src2x, 4);

          printf
          ("node%d:d1=%d,%d,%d,%d,d1x=%d,%d,%d,%d,d2=%d,%d,%d,%d,d2x=%d,%d,%d,%d\n",
            self_address,
            dest_data1[0], dest_data1[1], dest_data1[2], dest_data1[3],
            dest_data1x[0], dest_data1x[1], dest_data1x[2], dest_data1x[3],
            dest_data2[0], dest_data2[1], dest_data2[2], dest_data2[3],
            dest_data2x[0], dest_data2x[1], dest_data2x[2], dest_data2x[3]);

  flush();

          printf("node %d: the end\n", self_address);
        }
};
//end of user program

MAIN(Root_proc, argc, argv)
```

B.8 Sample program 8: read, scheduler-controlled completion and indirect handle

This program reads from remote source. *read* is considered as being performed in a *scheduler-controlled* setting. Completion is automatically monitored by the scheduler. *Global* data are defined with an *indirect handle*.

```
#include "A.environment"

//user program

char source_data[4];
char source_datax[4];

char* f(IDH_rcd *){
        return source_data;
}
char* fx(IDH_rcd *){
        return source_datax;
}

class Root_proc: public Process{

   public:
        Root_proc()
                : Process((Pmf0_ptr)&Root_proc::body){}

        void body(){

           printf("from node %d: this is my rank\n", self_address);

           for(int s=0; s<4; s++)
           source_data[s] = self_address;
           for(s=0; s<4; s++)
           source_datax[s] = self_address+50;

           Global<char> s_data(INDIR(1), f);
           Global<char> s_datax(INDIR(2), fx);
           flush();
```

```
            char dest_data1[4];
            char dest_data1x[4];
            char dest_data2[4];
            char dest_data2x[4];

            Star<char> remote_src1((self_address+1)%PROC_NB, s_data,
                                    (unsigned)0, NOAP, NOCP);
            Star<char> remote_src1x((self_address+1)%PROC_NB, s_datax,
                                   (unsigned)0, NOAP, NOCP);
            Star<char> remote_src2((self_address+2)%PROC_NB, s_data,
                                   (unsigned)0, NOAP, NOCP);
            Star<char> remote_src2x((self_address+2)%PROC_NB, s_datax,
                                   (unsigned)0, NOAP, NOCP);

            read(dest_data1[0],  *remote_src1,   (int *)0, 4);
            read(dest_data1x[0], *remote_src1x,  (int *)0, 4);
            read(dest_data2[0],  *remote_src2,   (int *)0, 4);
            read(dest_data2x[0], *remote_src2x,  (int *)0, 4);

            printf
           ("node%d:d1=%d,%d,%d,%d,d1x=%d,%d,%d,%d,d2=%d,%d,%d,%d,d2x=%d,%d,%d,%d\n",
             self_address,
             dest_data1[0], dest_data1[1], dest_data1[2], dest_data1[3],
             dest_data1x[0], dest_data1x[1], dest_data1x[2], dest_data1x[3],
             dest_data2[0], dest_data2[1], dest_data2[2], dest_data2[3],
             dest_data2x[0], dest_data2x[1], dest_data2x[2], dest_data2x[3]);

            flush();

            printf("node %d: the end\n", self_address);
        }
};
//end of user program

MAIN(Root_proc, argc, argv)
```

B.9 Sample program 9: read, tag 0 and direct handle

This program reads the absolute pointer to a remote source and next reads the data at the remote source via *tag 0. handle* is direct and Completion is automatically monitored by the scheduler.

```
#include "A.environment"

//user program

class Root_proc: public Process{

  public:

        Root_proc()
        : Process((Pmf0_ptr)&Root_proc::body){}


        void body(){

           printf("node %d: body is started\n", self_address);

           int *src_data_ptr;
           Global<int *>g_src_data_ptr(1, &src_data_ptr);

           if(!self_address){
              int *dest_data_ptr = (int *)44;
              int  dest_data = 11;
              Star<int*>remote_data_ptr(1, g_src_data_ptr,
                                         (unsigned)DIR(0), NOAP, NOCP);
              flush();

              read(dest_data_ptr, *remote_data_ptr);

              printf("node %d: dest_data_ptr = %d,\n",
                     self_address,  dest_data_ptr);

              Star<int>remote_data(1, (Global<int>)DIR(0),
                                   dest_data_ptr, NOAP, NOCP);
```

```
              read(dest_data, *remote_data);

              printf("node %d: dest_data_ptr = %d, dest_data = %d\n",
                      self_address,  dest_data_ptr, dest_data);

           }else{
              int src_data = 50;
              src_data_ptr = &src_data;

              printf("node %d: src_data_ptr = %d,\n",
                      self_address,  src_data_ptr);

              flush();
           }

  flush();

          printf("node %d: the end\n", self_address);
        }
};
//end of user program

MAIN(Root_proc, argc, argv)
```

B.10 Sample program 10: read, tag 0 and indirect handle

This program reads the absolute pointer to a remote source and next reads the data from the remote source via *tag 0*. *handle* is indirect and Completion is automatically monitored by the scheduler.

```
#include "A.environment"

//user program

int src_data;
int *handle(IDH_rcd *){
   return &src_data;
}
class Root_proc: public Process{

  public:

        Root_proc()
        : Process((Pmf0_ptr)&Root_proc::body){}


        void body(){

           printf("node %d: body is started\n", self_address);

           typedef int* (*handle_ptr)(IDH_rcd *);
           handle_ptr src_data_ptr;
           Global<handle_ptr>g_src_data_ptr(1, &src_data_ptr);

           if(!self_address){
              handle_ptr dest_data_ptr;
              int  dest_data;
              Star<handle_ptr>remote_data_ptr(1, g_src_data_ptr,
                                              (unsigned)DIR(0), NOAP, NOCP);
              flush();

              read(dest_data_ptr, *remote_data_ptr);

              Star<int>remote_data(1, (Global<int>)INDIR(0),
```

```
                                   dest_data_ptr, NOAP, NOCP);

              read(dest_data, *remote_data, (int *)0);

              printf("node %d: dest_data_ptr = %d, dest_data = %d\n",
                      self_address,  dest_data_ptr, dest_data);

           }else{
              src_data = 50;
              src_data_ptr = handle;

              flush();
           }

  flush();

          printf("node %d: the end\n", self_address);
        }
};
//end of user program

MAIN(Root_proc, argc, argv)
```

B.11 Sample program 11: user controlled send/recv

This program sends and receives via *Chan_R* and *Chan_RD* channels.

```
#include "A.environment"

//user program

class SR_count{

   public:
      int cnt;

      SR_count(){
         cnt = 0;
      }
      void reset(){

         cnt = 0;
      }
      void more_SR(int i){
         cnt += i;
      }
};

#define wait_while(condition) while(condition) poll();
#define not_complete  (c.cnt >0)

ChanRD_c<int> *chan;
ChanR_c *chan1;

void upon_end_of_SR_r(ChanR_c *chan){
   printf("node %d, in upon_end_of_SR_r, size = %d, Tsize = %d\n",
           self_address, chan->get_data_size_r(), chan->get_Tsize());
   ((SR_count *)(chan->get_arg_r()))->cnt--;
}

void upon_end_of_SR_s(ChanR_c *chan){
   printf("node %d, in upon_end_of_SR_s, size = %d, Tsize = %d\n",
           self_address, chan->get_data_size_s(),  chan->get_Tsize());
```

```
   ((SR_count *)(chan->get_arg_s()))->cnt--;

   chan->get_Tsize();
}


void upon_end_of_SR_r1(ChanR_c *chan){
   printf("node %d, in upon_end_of_SR_r1, size = %d, Tsize = %d\n",
           self_address, 0, 0);
   ((SR_count *)(chan->get_arg_r()))->cnt--;
}

void upon_end_of_SR_s1(ChanR_c *chan){
   printf("node %d, in upon_end_of_SR_s1, size = %d, Tsize = %d\n",
               self_address, 0, 0);
   ((SR_count *)(chan->get_arg_s()))->cnt--;
}


class  Root_proc: public Process{

   int recvd;

  public:
        Root_proc()
                : Process((Pmf0_ptr)&Root_proc::body){}

        void body(){

             printf("node %d, in Root_proc::body1\n", self_address);

             chan = new ChanRD_c<int>(1, (self_address+1)%PROC_NB);

             chan1 = new ChanR_c(2, (self_address+1)%PROC_NB);

             printf("node %d, in Root_proc::body2\n", self_address);

             MPI_Barrier(MPI_COMM_WORLD);
```

```
            SR_count c;

            chan->set_ctrl_send(upon_end_of_SR_s, &c);
            chan->set_ctrl_recv(upon_end_of_SR_r, &c);

            chan1->set_ctrl_send(upon_end_of_SR_s1, &c);
            chan1->set_ctrl_recv(upon_end_of_SR_r1, &c);

            printf("node %d, in Root_proc::body3\n", self_address);


            int sum_recvd = 0;
            for(int i=0; i<5; i++){
               int sent_data = self_address*10 + i;
               c.more_SR(2);
               chan->send(&sent_data);
               chan->recv(&recvd);
               wait_while(not_complete);

               c.more_SR(2);
               chan1->send_s();
               chan1->recv();
               wait_while(not_complete);

               int reduce_out;
               MPI_Allreduce(&recvd, &reduce_out, 1, MPI_INT,
                             MPI_SUM, MPI_COMM_WORLD);
               sum_recvd += reduce_out;

               printf("node%d, i=%d, recvd=%d, reduce_out=%d, sum_recvd=%d\n",
                      self_address, i, recvd, reduce_out, sum_recvd);
           }

           delete chan;
      }
};

MAIN(Root_proc, argc, argv)
```

Index