TECHNISCHE UNIVERSITÄT MÜNCHEN

INSTITUT FÜR INFORMATIK

Sonderforschungsbereich 342:
Methoden und Werkzeuge für die Nutzung
paralleler Rechnerarchitekturen

The ALDY Load Distribution System

Thomas Schnekenburger

TUM-I9519
SFB-Bericht Nr.342/11/95 A
Mai 1995

The ALDY Load Distribution System

Thomas Schnekenburger

Institut für Informatik, Technische Universität München
Orleansstr. 34, D-81667 München, GERMANY
email: schneken@informatik.tu-muenchen.de

1 Introduction

Load distribution is a central problem for parallel programming in distributed systems. Load distribution has to manage the assignment of service demands of a parallel program to distributed resources of the system. More generally, load distribution has to manage the mapping of *distribution objects* to *distribution units*. Distribution objects may correspond for example to processes, threads, or data structures whereas distribution units may correspond for example to computing nodes or application processes. Methods for load distribution can be classified into two groups:

- *System based load distribution* is used by operating systems or runtime systems. Distribution objects are implicitly defined by the load distribution system and automatically assigned to distribution units. Usually, system based load distribution realizes distribution mechanisms transparently for the application.
- *Application based load distribution* is used directly by parallel applications: Load distribution is realized usually by assigning distribution objects to *processes* of the program. The implementation of distribution objects depends on the individual application.

Both methods have their advantages and disadvantages: System based methods can be implemented transparently for the parallel program; therefore implementation effort for load distribution in individual parallel programs is small. The disadvantage of system based methods is that they require a fixed model of distribution objects. In addition, the *migration* of existing (active) objects often causes considerable overhead. System based load distribution is extensively investigated in the literature and a large variety of systems realize system based load distribution for parallel programs. The implementation of distribution objects in these systems ranges from processes [Zho92] and threads [KK93] to data structures [CS93]. Nevertheless, there is a variety of parallel applications where system based methods do not achieve sufficient efficiency. The main reason for poor efficiency is that application objects which should be considered as distribution objects cannot be mapped appropriately to distribution objects of the load distribution system: adequate load distribution methods for these applications have to consider the specific implementation of objects for load distribution and also specific correlations among these objects.

Application based methods are far more flexible, because load distribution mechanisms and strategies can be adjusted to the implementation of distribution objects (for example threads or data domains). The disadvantage of application based methods is the extra implementation effort which has to be spent to realize load distribution mechanisms. Unlike system based methods, application based load distribution is mainly investigated and implemented for individual applications. In most cases, mechanisms and strategies for load distribution are directly integrated into the application. Due to the large efforts for design and implementation of various strategies, only a small number of possible strategies is usually implemented and tested for an individual application.

This report describes ALDY (Adaptive Load Distribution SYstem). ALDY is a combination of system integrated and application integrated load distribution. It can be used either directly by a parallel program or by the runtime system of a parallel programming environment. The flexible object model and system independent load distribution strategies ensure that ALDY is portable over a large range of programming environments and a large range of hardware platforms. In addition to the conventional *task farming* paradigm used by most load distribution systems (for example, Dynamo [Tär94], and Mentat [Gri93]), ALDY supports the migration of application objects. Another ALDY feature is the realization of *adaptive* load distribution: Methods for adaptive load distribution are robust with respect to varying availability of system resources due to heterogeneity or processes which are generated by other users.

Section 2 surveys the functions and design objectives of ALDY and describes the ALDY load distribution model. Section 3 presents the ALDY library interface. The basic concepts of the ALDY load distribution strategies are explained in section 4. In section 5, three application examples, each representing a large class of similar applications are described. Section 6 gives a summary and an outlook to our future work.

2 Concepts

This section surveys basic concepts of ALDY. After describing the functions for load distribution support which are performed by ALDY, we discuss design objectives and explain the integration of ALDY into a parallel application.

2.1 The Functions of ALDY

ALDY is not designed as a complete parallel programming environment but as a system which *supports* the implementation of load distribution. The following list summarizes the functions for load distribution support which are provided by ALDY.

- **Load distribution strategy:** ALDY manages load distribution objects and controls the assignment of objects: ALDY decides whether and where an object is assigned or migrated.

- **State information:** ALDY collects global information about states of objects, needed by the load distribution strategy.

- **Communication control:** Realizing messages between migratable objects is usually very complex and can be a main source of errors in parallel programs. ALDY offers an efficient protocol for location transparent communication between migratable objects.

- **Initialization and termination protocols:** ALDY provides protocols for initialization and termination of the application.

2.2 Objectives

The design objectives for ALDY can be classified into four different aspects of portability:

Portability w.r.t. object implementation

The basic concept of the ALDY design is to be independent from a specific implementation of distribution objects. The application maps its own objects to *virtual objects* of the ALDY load distribution model and informs ALDY about attributes and states of virtual objects. Based on this information, a load distribution strategy is used to decide about mechanisms for load distribution. Since ALDY has no information about the implementation of distribution objects, these mechanisms have partially to be realized by the application.

Portability w.r.t. load distribution strategies

The suitability of individual strategies depends strongly on the characteristics of the application. It is not advisable to try to develop a very general, but in many cases moderate or poor strategy. Therefore, another ALDY feature is to provide a collection of various load distribution strategies. The selection of a strategy can be done at run time; it is not necessary to recompile the program.

Portability w.r.t. system parameters

An interesting idea for the design of ALDY is the restricted usage of system information by the load distribution strategy: strategies of ALDY use only information about the states of virtual objects in the load distribution model. System states like the load and memory usage of computing nodes are not used. As a result, the ALDY strategy is completely independent from the individual resources considered for load distribution and their specific properties. Nevertheless it is possible to realize efficient strategies on top of this restricted information, as explained in section 4.

Portability w.r.t. communication and synchronization

The multitude of environments for parallel programming has led to a large number of mechanisms for communication and synchronization. To be not fixed to a specific system, a concept of ALDY is to provide just a kernel system for communication and synchronization and to require individual interface components which implement routines for communication and synchronization based on the given programming environment. By using the specific communication and synchronization routines of the application, ALDY has the possibility to integrate the exchange of internal information for load distribution into the communication of the application program.

2.3 Integration of ALDY

Figure 1 illustrates the integration of ALDY into parallel programs. Load distribution with ALDY is performed on application level. Therefore, from the operating systems point of view, the parallel application and ALDY together form a single parallel program, consisting of several processes. The ALDY system is linked to each process of the parallel program. There is only local interaction between the application and the corresponding ALDY instance.

Parallel programs and ALDY are connected by an interface which implements communication and synchronization for ALDY. Furthermore, information about the implementation of

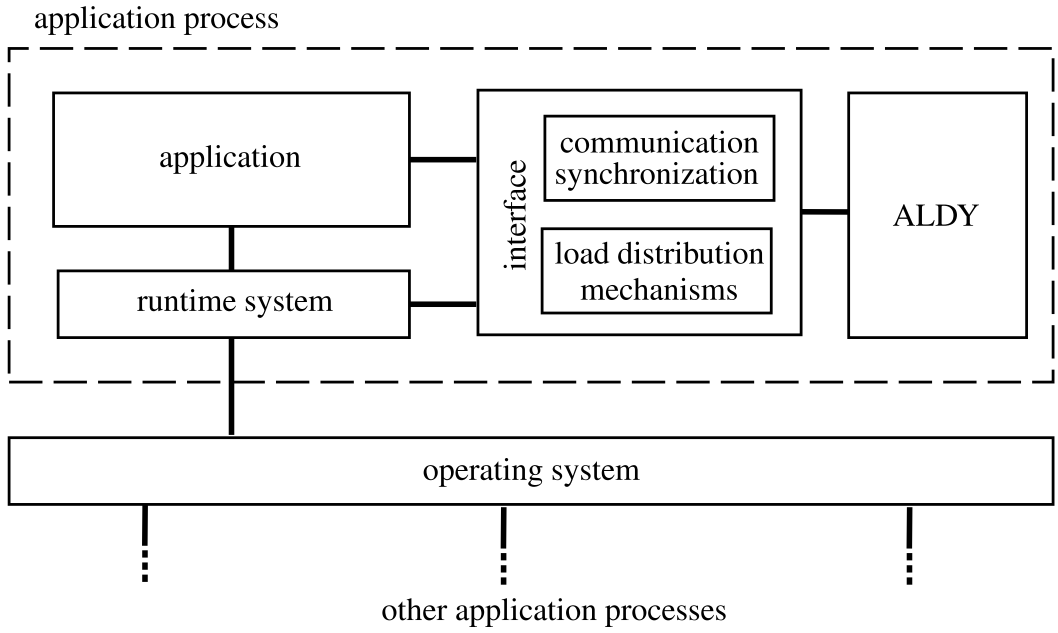

Figure 1: Integration of ALDY

application objects (and therefore the routines for load distribution mechanisms like sending and migrating objects) is assumed to be integrated into this interface. Routines for several load distribution mechanisms are implemented as *call back* functions, so that the ALDY strategy can transparently perform these mechanisms. Global communication among different ALDY instances is realized using the existing communication mechanisms of the application.

2.4 Load Distribution Model

Generally, load distribution is performed by assigning distribution objects to distribution units. Therefore, a system which supports load distribution needs a model of distribution objects and distribution units, which is called the *load distribution model*. The load distribution model specifies the assignment of distribution objects to distribution units. For example, the load distribution model of many systems consist of computing units and processes which are assigned preemptively or non-preemptively to the computing units; the Linda load distribution model consists of Linda tasks which are assigned dynamically and non-preemptively to Linda workers [CGMS94].

In Linda and most other systems which support load distribution, the load distribution model corresponds directly to the programming model of the parallel programming environment. In contrast to these systems, the ALDY concept consists of *virtual* objects: ALDY knows about the existence of virtual objects and their attributes (assignment directives, actual assignment, activity etc.), but knows nothing about their implementation. There is a clear separation between real objects of the parallel program (for example processes, threads, data objects) and the corresponding virtual objects. Load distribution is realized not directly by applying mechanisms on real objects: ALDY manages load distribution by directing the application to perform certain load distribution mechanisms for virtual objects.

The ALDY load distribution model is called the PWT (Process, Worker, Task) model. It consists of three classes of virtual objects. The first class are *virtual processes*, representing the distribution units, that means load distribution is realized by *assigning* virtual objects to virtual processes. These objects are divided into two classes: *virtual workers* and *virtual tasks*. The three classes are now explained more detailed. To illustrate flexibility, the mapping the three very different parallel applications (or programming models, respectively) to the PWT model is considered:

1. A data parallel application where each process is responsible for several partitions of a distributed data structure, for example blocks of a matrix. Data migration is realized by moving data to another process. Data which are necessary to work on the matrix in the other process has to be migrated. Furthermore, messages which refer to the matrix have to be sent to the correct process.

2. A system consisting of several processes running on a distributed system where each process may consists of a number of local threads. Load distribution is achieved by migrating threads to other processes. To migrate a thread, it has to be terminated in the sending process and to be restarted in the receiving process. As above, data which are necessary to run the thread in the other process must be provided and messages which are directed to the migrated thread must be sent to the correct process.

3. A distributed file system. Load distribution can be realized by migrating file partitions to other disks. For example, [HY91] describes a specific load distribution concept for a distributed file system.

Virtual processes

As mentioned above, load distribution has to manage the assignment of service demands of a parallel program to distributed resources of the system. In the PWT model, these distributed resources correspond to *virtual processes*, in the following just called processes. Usually a virtual process will correspond to a real application process (example 1 and 2). Nevertheless, in example 3, a virtual process can correspond to a disk of the distributed file system.

Virtual workers

Virtual workers (in the following just called workers) are objects which are assigned *preemptively* to processes, that means they can be migrated dynamically to another process. There may be several workers which are assigned simultaneously to a process. The general load distribution strategy for workers is to migrate workers dynamically from *overloaded* to *underloaded* processes. In example 2, virtual workers can be used to represent threads. In example 1 and 3, virtual workers represent partitions of the data: in example 1 a worker corresponds to a block of the matrix and in example 3 a worker corresponds to a file partition.

Virtual tasks

Virtual tasks (in the following just called tasks) are objects which are assigned non-preemptively to workers (not directly to processes!). As long as a task is assigned to a worker, we say that the worker *processes* the task. There is no parallelism within workers, that means tasks are processed one after another by workers. Tasks have two basic purposes:

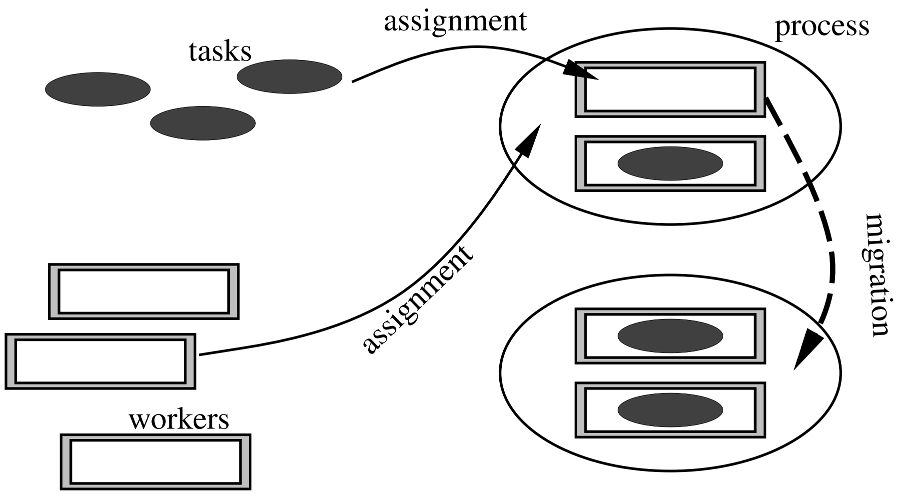

Figure 2: PWT model

- Tasks represent messages which are assigned to workers. The load distribution strategy has to ensure that a task which is assigned to a worker “finds” the right process even if the worker has been migrated.
- Tasks represent operations which are processed by workers. Tasks restrict the possibilities of worker migration and therefore simplify the implementation of worker migration: If a worker processes a task, this worker is not migrated. The motivation for this concept is that workers generally will repeatedly wait for messages and apply an internal algorithm on the received data. In most applications it is relatively easy to migrate a worker during the waiting for messages but it is often difficult to migrate a worker computing some internal algorithms.

Again, there are several possibilities for the correspondence of tasks and real application objects as described by the three examples.

1. If workers correspond to data, a task may correspond to a message or a remote procedure call which refers to this data. Processing messages or execution of a remote procedure call then corresponds to the processing of a task.
2. If workers are realized as threads, a task may correspond to a message for a worker. Sending of a message corresponds to the generation of a task. Receiving of messages corresponds to the assignment of tasks. Performing operations relating to the message corresponds to processing of the task.
3. In [HY91], a tasks corresponds to a file operation for a specific file partition. In conformity with the PWT model, file partitions which actually process a file operation are not migrated.

Summing up, the PWT model consists of three classes of objects (see figure 2). Processes are distribution units which own several workers. Each worker sequentially processes tasks. A worker which is not processing a task may by dynamically migrated to another process.

3 Library Interface

The ALDY library interface consists of functions for describing virtual objects and their attributes and functions which are used by ALDY to inform the application about load distribution mechanisms. This section describes the ANSI-C library interface.

3.1 Identifiers

Virtual objects have identifiers which are determined by the application. That means ALDY does not return identifiers of newly generated objects but the application has to tell ALDY identifiers for newly generated objects. At first sight that seems to be uncomfortable but there is an important advantage: The application can *plan* the global identification of objects in advance and needs not to implement protocols for exchange of object identifiers. Identifiers for ALDY objects have to be **positive integers** (`int` values > 0). Processes, workers and tasks should have distinct identifiers.

3.2 Initialization

To implement initialization, termination and other distributed protocols, ALDY requires a specific *topology* of virtual processes: One process has to be marked as *main*-process. It will be used as central instance for several internal distributed protocols. Furthermore, all virtual processes have to be arranged in a *ring*. For initialization, each process has to call the function

```
void ALDY_Init(int myPid, int mainPid, int nextPid);
```

where `myPid` is the identifier of the calling process, `mainPid` is the identifier of the *main*-process and `nextPid` is the identifier of the next process in the ring. The `ALDY_Init` function terminates when the ring is complete.

Example: The processes of an application are named $1, \ldots, P$, where P is the number of processes. Process 1 is the main process and the ring successor of process p is $p + 1$ if $p \neq P$ and 1 if $p = P$. Then each process p has to call

```
ALDY_Init(p, 1, (p!=P) ? p+1 : 1);
```

3.3 Communication

Sending messages for ALDY

Internal communication of ALDY is implemented on top of the application's communication mechanisms. Therefore, the application has to implement two call back functions:

```
void ALDY_CB_SendInfo(int pid, int * addr, int n);
int ALDY_CB_SendObject(int ob, int pid, int * addr, int n);
```

`ALDY_CB_SendInfo` should send an array of `n` values of type `int`, starting at address `addr` to process `pid` where `pid` is the identifier which was defined by the receiving process in the `ALDY_Init` call. `ALDY_CB_SendObject` should send a message to process `pid` consisting of two parts: The first part is an array starting at address `addr` with `n` values of type `int`. The second part are data which are necessary to send object `ob` to process `pid`. Since ALDY is not aware of the implementation of objects, the application programmer has to define these data. ALDY_CB_SendObject should return the number of additional bytes which are used for sending ob.

Example: If PVM 3 [SGDM94] is used as message passing interface, the function ALDY_CB_SendObject may by implemented like this:

```
int ALDY_CB_SendObject(int ob, int pid, int * addr, int n) {
  pvm_initsend(PvmDataDefault); /* initialize send-buffer */
  pvm_pkint(&n,1,1); /* put length of ALDY-message */
  pvm_pkint(addr,n,1); /* put ALDY-internals */
  /* assuming Object has 1000 Bytes starting at address obAddr */
  pvm_pkbyte(obAddr,1000,1); /* put the object into the buffer */
  /* assuming that the array tids contains pvm-identifiers */
  /* send ALDY message, tag 999 is used to mark ALDY-messages */
  pvm_send(tids[pid-1],999);
  return 1000; /* return number of bytes of object */
}
```

Receiving messages for ALDY

When a message for ALDY is received by the conventional message passing mechanism, the application has to call

```
void ALDY_PutInfo(void * addr, int n);
```

Here, addr should be the address of the received message and n should be the number of int values of the message. When calling this function, ALDY reads the information which is contained in the message. When the function call returns, the buffer for the message can be deleted. In some cases, ALDY requests the application to receive a message for it. This is realized by the call back function

```
void ALDY_CB_Receive(int block);
```

block specifies whether the receiving is "blocking" (block==ALDY_BL), that means ALDY definitely expects a message, or "non-blocking" (block==ALDY_NBL), that means the application may receive a message for ALDY.

Example: The PVM 3 implementation of ALDY_CB_Receive may look like this:

```
void receive(void) { /* receives an ALDY-message */
  int n;
  pvm_upkint(&n,1,1); /* read number of int-values */
  pvm_upkint(buffer,n,1); /* write message into a buffer */
  ALDY_PutInfo(buffer,n);
}

void ALDY_CB_Receive(int block) {
  if (block==ALDY_BL) { /* wait for a message */
    /* assuming that tag 999 is used to mark ALDY-messages */
    pvm_recv(-1,999); receive(); }
  /* look for further messages */
  while (pvm_nrecv(-1,999)) receive();
}
```

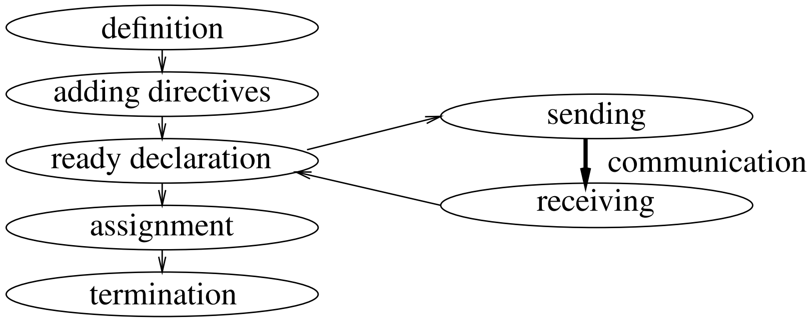

Figure 3: Definition and assignment of objects

If an object was sent together with the internal ALDY information (using ALDY_CB_SendObject), `ALDY_PutInfo` will invoke the call back function

```
void ALDY_CB_RecvObject(int ob, void * addr, int n);
```

where `ob` is the identifier of the sent object, `addr` is the address of the message part containing the object, and `n` is the number of bytes which where returned from the corresponding call of `ALDY_CB_SendObject`.

3.4 Workers and Tasks

Figure 3 shows a diagram representing the steps of definition and assignment of virtual workers and tasks. The following paragraphs explain the corresponding steps.

3.4.1 Definition

First, virtual workers and tasks have to be *defined*, that means the application declares new identifiers to be a representation of objects. This is done by calling the function

```
void ALDY_DefineObject(int ob, int obType);
```

where `ob` is the identifier of an object and `obType` either is `ALDY_WORKER` if the object is a worker or `ALDY_TASK` if the object is a task.

3.4.2 Directives

After defining an object, the application has to specify directives for the load distribution mechanisms which can be applied to this object. Note that directives for an object have to be specified by the process which defines it. There are six different relations specifying directives for load distribution. The function for defining all directives is

```
void ALDY_AddDirective(int dirType, int ob1, int ob2);
```

When this function is called, object `ob1` should be previously defined (see above) by the calling process. For each object, an arbitrary number of directives may be specified. All directives are *global*, that means ALDY will consider the directives even if object `ob2` is (subsequently) defined by another process. Depending on the value of `dirType`, `ALDY_AddDirective(dirType,ob1,ob2)` has the following semantics:

- dirType==ALDY_TD (task-dependency): ob1 should be defined as task. ob2 is specified as *predecessor* of ob1, that means ob1 is not assigned to a worker before the task ob2 terminates.
- dirType==ALDY_TW (task-worker-assignment): ob1 should be defined as task. ob1 can be assigned to worker ob2. If more than one worker is specified for assignment of a task, ALDY selects a worker according to the load distribution strategy.
- dirType==ALDY_WD (worker-dependency): ob1 should be defined as worker. ob1 is not assigned to a process before worker ob2 terminates.
- dirType==ALDY_WP (worker-process-assignment): ob1 should be defined as worker. ob1 can be assigned to process ob2. If more than one process is specified for assignment of a worker, ALDY selects a process according to the load distribution strategy.
- dirType==ALDY_WM (worker-process-migration): ob1 should be defined as worker. ob1 can be migrated to process ob2. This directive is mainly used to exclude specific processes from worker migrations.
- dirType==ALDY_WN (worker-neighborship): ob1 should be defined as worker. ob2 is specified as *neighbor (see below)* of worker ob1.

The semantics of the worker-neighborship directive has to be explained in detail. In many applications there exists a specific *topology* of workers, that means communication among workers has a specific pattern and is not arbitrary. Therefore, communication amount may be reduced by trying to assign *tightly coupled* communicating workers to the same process. The neighborship-directive supports this concept: Workers for which it is advantageous to be assigned to the same process can be specified as neighbors. The ALDY load distribution strategy migrates a worker w to a process p' only if there exists a worker w' on process p' which is a neighbor of w. In addition, if a process has no workers, ALDY may migrate a worker to that process although there can be trivially no neighbors of local workers.

Example: Figure 4 shows a common worker topology: Workers communicate with their left and right *neighbors* according to a one-dimensional partitioning of application data. In that case, communication overhead is reduced by migrating workers only among adjacent processes.

3.4.3 Setting Objects Ready

By calling the function

```
void ALDY_ReadyObject(int ob);
```

the application informs ALDY that all directives for object ob are specified and that the object can be assigned. The call of ALDY_ReadyObject may involve sending of the object to another process (using ALDY_CB_SendObject).

3.4.4 Assignment of Objects

Object assignment consists of the assignment of workers to processes and the assignment of tasks to workers. Workers are assigned to processes by the call back function

```
void ALDY_CB_AssignWorker(int wid);
```

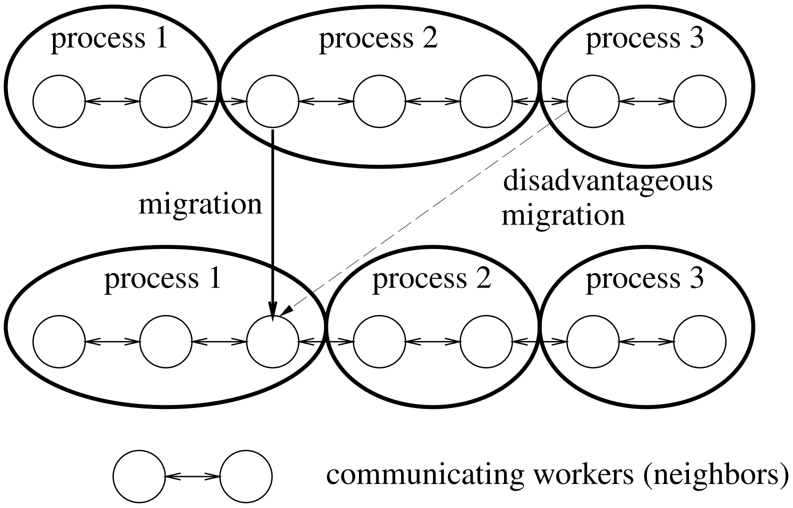

Figure 4: Example for a worker topology

This function tells the application, that the worker `wid` is assigned to the actual process. If a process "knows" that it has to wait for the assignment of a specific worker (according to the directives for worker assignment), it may wait for this assignment using the function

```
void ALDY_AwaitWorker(int wid);
```

If the worker `wid` is assigned to the local process, then the function call to `ALDY_AwaitWorker` returns immediately. If not, `ALDY_AwaitWorker` returns after the worker is assigned, that means after ALDY has called the function `ALDY_CB_AssignWorker(wid)`.

In contrast to worker assignment, task assignment is realized by a user interface function which is called by the application requesting a task:

```
int ALDY_NextTask(int block, int wid, int wanted);
```

This function returns either a task identifier or 0, if no task is assigned. `wid` specifies the worker which wants to get a task. If `wanted==0`, ALDY may assign an arbitrary task to worker `wid` (with respect to the assignment directive). If `wanted!=0`, `wanted` specifies the task which has to be assigned to worker `wid`. `block` specifies whether ther request is blocking (`block==ALDY_BL`) or non-blocking (`block==ALDY_NBL`). A blocking requests waits (via call backs of `ALDY_CB_Receive`) until an appropriate task is found. A non-blocking request either returns an appropriate task or returns 0, if no appropriate task is found.

3.4.5 Termination of Objects

ALDY is informed about the termination of a worker by calling the function

```
void ALDY_EndWorker(int wid);
```

Termination of tasks is implicit: A task is terminated by calling `ALDY_NextTask(block,wid,wanted)`. This call informs ALDY that the task which is currently assigned to worker `wid` is terminated.

3.5 Worker Migration

Worker migration is the key concept of ALDY's load distribution mechanism. When ALDY decides to migrate a worker, it invokes the call back function

```
int ALDY_CB_MigrateWorker(int wid, int pid, int * addr, int n);
```

`wid` specifies the worker which has to be migrated. Similar to the function `ALDY_CB_SendObject`, `addr` specifies the address of an array of `n` values of type `int`, which has to be placed in front of the worker. The message consisting of the array and the worker has to be sent to the process `pid`. The migrated worker will be assigned to process `pid` by the call back function `ALDY_CB_AssignWorker`. According to the PWT model, ALDY ensures that `ALDY_CB_MigrateWorker` is called only if currently no task is assigned to the worker.

3.6 Load and Time Information

A load distribution strategy needs load information to decide whether a distribution unit is *overloaded* or *underloaded*. In the PWT model, distribution units are virtual processes. To decide whether a virtual process is overloaded or underloaded, ALDY uses the number of *active* workers within a process as basic measure for the load distribution strategy. According to the flexible implementation of virtual objects, ALDY provides a flexible mechanism for defining a worker as *active*. The functions

`void ALDY_StartAction(int wid);` *and*

`void ALDY_EndAction(int wid);`

are used by the application to define the worker `wid` as active or non-active, respectively. When a worker is assigned to a process, it is implicitly non-active. Usually, a worker should be defined as active, if it consumes the resource which is considered for load distribution. For example, this resource may be the CPU or a local disk.

The basic load distribution strategy of ALDY requires information about the elapsed time of a virtual process. To provide optimal portability and flexibility, the ALDY system does not use a specific library function but uses the call back function

```
double ALDY_CB_Time(void);
```

This function should return the elapsed time (not CPU time) of the actual process. The time unit which is used is arbitrary since the returned value is *directly* used by the load distribution strategy. The application of the load and elapsed time of processes for the load distribution strategy is described in section 4.

3.7 Termination

ALDY provides a distributed termination protocol. The termination problem for a system with migratable objects is to globally decide whether there are still objects in the system. The ALDY termination protocol is based on workers. The application can test for termination using the function

```
int ALDY_Terminate(void);
```

`ALDY_Terminate` returns 0 if there is (globally) any worker in the program which is defined but not yet terminated, else it returns 1. The call of `ALDY_Terminate` is always non-blocking and may be repeated as long as it returns 0. After `ALDY_Terminate` has returned 1, ALDY assumes that the calling process will terminate and the application should not execute further calls to any of the ALDY library functions.

3.8 Complementary Functions

The functions described so far are sufficient to realize all load distribution mechanisms with ALDY. The following functions can save some programming effort.

3.8.1 The Group Concept

In many applications, objects used in the specification of directives can be divided into groups of "equivalent" objects. For example, if a task can be assigned to all workers of the application, these workers can be collected in a single group and we can say that the task has to be assigned to this group. ALDY supports the group concept using the function

```
void ALDY_PutInGroup(int ob, int gr);
```

where `ob` is the identifier of an object which is declared to be in a group `gr`. `ob` has to be previously defined by the calling process. `gr` is implicitly declared as identifier of a group. *Mixed* groups are not allowed, that means groups consist either of tasks, workers or processes. Identifiers of groups can be used for the definition of directives: In a call of the function

```
void ALDY_AddDirective(int dirType, int ob1, int ob2);
```

`ob2` can be not only a single object, but also an identifier of a group of objects. Note that `ob1` has to be a single object; therefore the definition of a directive among groups is not possible. Similar to directives, groups are handled globally.

3.8.2 Object Pointers

Usually an application has to realize pointers from virtual ALDY objects to real objects. To save the application from the work of implementing a dynamic table containing pointers from virtual to real objects, ALDY provides the functions

`void ALDY_PutObjectAddr(int ob, void * addr);` *and*

`void * ALDY_GetObjectAddr(int ob);`

For both functions, `ob` has to be an object which is defined or assigned locally. Using `ALDY_PutObjectAddr`, the application can assign an address `addr` to object `ob`. Then `ALDY_GetObjectAddr(int ob)` returns this address. The result of `ALDY_GetObjectAddr(int ob)` is undefined, if `ALDY_PutObjectAddr` was not previously called for object `ob` or if `ob` is terminated.

3.8.3 Worker List

Processes hold a dynamically varying number of workers. To implement a "loop" across all workers which are assigned to the local process, the application can use the function

```
int ALDY_NextWorkerPos(int wid);
```

If `wid==0`, `ALDY_NextWorkerPos` returns an arbitrary worker from the internal list of assigned workers. If `wid` is the identifier of an assigned worker, `ALDY_NextWorkerPos` returns "the next" worker from the internal list of assigned workers. If the function is used to implement a loop like

```
for(wid=ALDY_NextWorkerPos(0);wid;wid=ALDY_NextWorkerPos(wid)) ...
```

it is guaranteed that the loop repeatedly covers all workers which are assigned to the process. If no workers are assigned to the process, `ALDY_NextWorkerPos` returns 0.

3.8.4 Avoiding Migrations

In some applications there are *critical sections* (for example, computation of an internal algorithm) where it would be difficult to implement worker migrations. Since worker migration is realized by call back functions, migration within a critical section may only happen if an ALDY library function is called within the critical section. Furthermore, to avoid migration of a worker during a critical section, a task representing execution of this section can be generated and assigned to the worker. To save the programmer even from implementing these *avoid-migration-tasks* the functions

`void ALDY_NoMigration(int wid);` *and*

`void ALDY_PermitMigration(int wid);`

can be used. The call `ALDY_NoMigration(wid)` tells ALDY that worker `wid` cannot be migrated until `ALDY_PermitMigration(wid)` is called.

3.9 Optimization

This section presents some further functions which may be useful to optimize the application program.

3.9.1 Information Integration

In some applications, communication between application processes is not completely mapped to the exchange of ALDY objects. In that case, the number of messages (and therefore usually the communication overhead) can be reduced by integrating application messages and ALDY messages. When the function

`int * ALDY_GetInfo(int pid, int * n);`

is called, ALDY returns the address of an array of type `int` which should be integrated into the application message to process `pid`. The call returns 0, if there are no data for integration. `n` should be a pointer to an integer. After the call, `n` references the length of the array. After receipt of a message with additional ALDY data, the application should use `ALDY_PutInfo` to pass the data to the ALDY system.

3.9.2 Information about Worker Locations

To understand the use of the following function, it is necessary to know that ALDY uses an *optimistic* protocol for the management of information about worker locations: Each process has an internal table of worker locations. When a process sends a message (probably including a task) relating to a specific worker, the table entry for that worker is used to determine the target process for this message. If the location information is not available, ALDY implements a (currently centralized) protocol for the exchange of worker locations. If the location information is available but wrong because the worker was migrated in the meantime, the receiving process forwards the message to the target process of the migration and informs the sending process about the new worker location. Although this protocol ensures that each message relating to a worker will sometimes be received by that worker, the forwarding of messages and sending of new locations causes a communication overhead. This overhead may be reduced by the function

```
void ALDY_WorkerLoc(int w, int p);
```

The call `ALDY_WorkerLoc(w,p)` tells ALDY, that worker `w` is presumably located at process `p`. For example, if the directives specify that a worker has to be started on a specific process, it is guaranteed that this worker will be located at this process until it terminates or migrates. If this is known to other processes, they can use `ALDY_WorkerLoc` to tell ALDY the expected location of the worker and therefore improve the efficiency. If `p` is not the correct location of worker `w`, the call `ALDY_WorkerLoc(w,p)` will not cause an error but just some overhead.

3.10 Summary

The following table summarizes the ALDY library interface. The first part lists the library functions which can be called by the application. The second part lists all call back functions which have to be provided by the application if the ALDY library is linked.

ALDY_AddDirective	add a directive for a defined object (3.4.2, 3.8.1)
ALDY_AwaitWorker	wait for assignment of worker (3.4.4)
ALDY_DefineObject	task/worker definition (3.4.1)
ALDY_EndAction	declare worker to be non-active (3.6)
ALDY_EndWorker	declare worker as terminated (3.4.5)
ALDY_GetInfo	integrate information (3.9.1)
ALDY_GetObjectAddr	get address of object (3.8.2)
ALDY_Init	initialization (3.2)
ALDY_NextTask	request for task; termination of previous task (3.4.4, 3.4.5)
ALDY_NextWorkerPos	loop across all local workers (3.8.3)
ALDY_NoMigration	do not migrate the worker (3.8.4)
ALDY_PermitMigration	cancel ALDY_NoMigration (3.8.4)
ALDY_PutInfo	put a received message to ALDY (3.3, 3.9.1)
ALDY_PutInGroup	put object into group; possibly declare new group (3.8.1)
ALDY_PutObjectAddr	assign an address to object (3.8.2)
ALDY_ReadyObject	declare object as ready for assignment (3.4.3)
ALDY_StartAction	declare worker to be active (3.6)
ALDY_Terminate	termination of process (3.7)
ALDY_WorkerLoc	inform ALDY about location of worker (3.9.2)
ALDY_CB_AssignWorker	worker assignment (3.4.4, 3.5)
ALDY_CB_MigrateWorker	worker migration (3.5)
ALDY_CB_Receive	receive message for ALDY(3.3)
ALDY_CB_RecvObject	receive data of object (3.3)
ALDY_CB_SendInfo	send ALDY information to other process (3.3)
ALDY_CB_SendObject	send object and ALDY information to other process (3.3)
ALDY_CB_Time	returns real time (3.6)

4 Load Distribution Strategy

One of the key ideas of ALDY is to separate the application program from a specific load distribution strategy. Furthermore, ALDY implements not a fixed strategy, but a generic load distribution concept. Using this generic concept, a collection of several parameterized specific load distribution strategies can easily be integrated. An ALDY load distribution strategy consists of an *internal* and an *external* strategy:

- The internal strategy is the efficient realization of a distributed information management: Since directives about load distribution mechanisms are global (see 3.4.2), ALDY has to provide global information about states, directives and actual locations of virtual objects.

- The external strategy is the concrete load distribution strategy. Based on the global object information provided by the internal strategy, the external strategy has to decide when and where objects are assigned or migrated, respectively.

The individual strategy and its parameters are not specified by the library interface but by a specific parameter file which is scanned by the ALDY library function `ALDY_Init`. This way, the parameters or even the load distribution strategy itself may be changed without having to recompile the program.

4.1 Internal Strategy

Although knowledge about the internal strategy is not necessary for using ALDY, it may be useful to improve the efficiency of the program. Of course it would be too complicated to describe the complete internal strategy. We emphasize only the aspect which is most important for an efficient implementation of the internal strategy.

One part of optimization is already contained in the library interface: Directives and group memberships can be specified only for locally defined objects. The reason for this restriction is that it allows valuable optimizations using the following classification of virtual objects:

- *Local directed objects*: Objects which have to be assigned to the local process. These are workers which have to be assigned locally, and tasks which have to be assigned to a local worker. Management of a local directed object can be implemented without communication overhead (and therefore very efficient) if all directives for this object relate to local objects. (Here *local* means that the objects are defined by the same process.)

- *Remote directed objects*: Objects which have to be assigned to another specific process. These are workers which have to be assigned to a specific non-local process, and tasks, which have to be assigned to a specific worker which is assigned to another process. Exchange of information about a remote directed object can be transformed directly into a single message containing the object if directives of this object relate to objects of the sending process or objects of the receiving process.

- *Undirected objects*: Objects which can be assigned to several processes. For undirected objects and objects which use "global" directives, ALDY has to realize a distributed object management.

The classification shows that the application generally should prefer local directed objects or remote directed objects without global directives. In some cases, global directives can be avoided by analyzing the application. For example, task dependencies (`ALDY_TD`) may be avoided by generating tasks only when their *predecessors* have terminated or by using the `wanted` parameter of `ALDY_NextTask` (see 3.4.4).

Summing up, although the internal strategy is hidden from the application programmer, he should be aware that management of undirected objects and objects with global directives is more expensive than management of directed objects. He should try to map his applications objects to virtual ALDY objects so that needless global directives are avoided.

4.2 External Strategy

The key issue of every load distribution strategy is the *load index*: This (usually multi dimensional) value is a representation of the *load* of a distribution unit. ALDY uses a very simple load index: The actual load

$$l(t)$$

of a process is defined as the number l of *active* workers at time t, where t is determined by ALDY using the call back function `ALDY_CB_Time` (see 3.6). Based on the actual load, several derived load values can be defined. They are used as parameters for specific load distribution strategies. For example:

- At time t, a process is *idle for x time units*, if $l(u) = 0$ for all $u \in [t - x, t]$.
- The *activity* $a(t)$ of process p is 1 if $l(t) > 0$ and 0 if $l(t) = 0$.
- At time t, the *average activity over x time units* is $\frac{1}{x} \int_{u=t-x}^{t} a(u)du$ where $a(u)$ is the activity.

As mentioned in section 3.6, a worker should be defined as *active* if it consumes the resource considered for load distribution. At first sight, the CPU would be usually the resource to consider for load distribution. Although this is true for simple parallel architectures, the reality of parallel computing is more complicated: For example, if a distributed system consisting of workstations is used, then the resource to be considered is not only the CPU, but also the main memory and the behavior of the local disk. In addition, processes of other users (*external load*) may have a severe influence on the run time of individual parts of the application. Here the problem is to find a load distribution strategy which can consider all these influences.

The solution of this problem using ALDY is rather simple: The external ALDY strategy always tries to assign or migrate objects (and therefore workload) to *underloaded* processes. Usually processes will be underloaded because their progress is faster (due to faster CPU, more main memory, less external load, etc.) than the progress of other processes. If workers are defined to be active as long as they execute a time consuming part of the application (just leaving unnoticed the physical resource which is used), the ALDY strategy will result in an *adaptive* strategy which can compensate for *all* factors which have an influence on the execution time of the application. The following example shows the strategy specification for a simple (but often very efficient) receiver-initiated strategy:

```
aldyStrategy = Receiver
aldyRecvTS = idle(1.5)
aldySendTS = avg(0.6,5)
aldyWaitRepeat = 0.5
```

`aldyStrategy = Receiver` specifies that ALDY uses a receiver-initiated strategy. This strategy works as follows: If the load value of a process p is below a *threshold* `aldyRecvTS`,

p looks for a neighbor process p', that means a process p' holding a worker w' which is neighbor of a worker w assigned to p. If there is such a neighbor process p', p sends a request message to p'. After receiving the request message, p' checks whether its load exceeds a threshold `aldySendTS`. If the threshold is not exceeded, p' sends a rejection message to p, and p has to look for further neighbors. If the threshold is exceeded, p' migrates the neighbor worker w' to p. If p cannot enforce a worker migration, it does not immediately repeat the request for workers but waits for a time `aldyWaitRepeat`. This waiting time is very important to reduce the communication overhead for information exchange. Two different load values are used in the example. `aldyRecvTS = idle(1.5)` means *process p is idle since 1.5 time units*. `aldySendTS = avg(0.6,5)` means *the average activity of process p' over 5 time units is 0.6*.

5 Examples

To illustrate the use of the ALDY library, three typical applications are regarded. Each of these applications is shown in three stages. Stage 1 is the conventional application program. Stage 2 is the transformed program according to the PWT model. That means, processes, tasks and workers representing objects of the application are formed. Stage 3 is the application program with integrated calls to the ALDY library.

The first and second examples are a task farming program and a program consisting of replicated servers, respectively. Both examples are thought to represent a large class of similar applications. Therefore, details which are not important with respect to the use of ALDY are omitted or simplified. The third example is an almost complete non-trivial application.

5.1 Task Farming

A task farming program consists of a number of independent subproblems which can be solved by any process of the program. Although a specific example is used, it is straightforward to extend it to "real" task farming problems like distributed backtracking [FM87] and many numerical algorithms [WL88]. The basic mechanism for dynamic load distribution is the dynamic assignment of tasks to processes. Migration is not necessary.

The sample program has to compute

$$\sum_{i=1}^{100} f(i);$$

where $f(i)$ can be computed independent from other values. It consists of a *central* process generating function calls and collecting results, and several *replicated* processes receiving and evaluating function calls and returning results. Termination is implemented by sending an *end-of-program* call to each replicated process. The conventional version of the application can be sketched like this:

```
/* Central Process */
main() {
  int i;
  for (i=1; i<=100; ++i) assign function calls to processes
  wait for all results
  send end-of-program call to each process
  sum up results
}
......
/* Replicated Process */
main() {
  do {
    receive a call
    if (call is end-of-program) break;
    evaluate function call and return result
  } while (1);
}
```

Task Farming Program, Stage 1

To integrate the ALDY library, processes and function calls have to be mapped to objects of the PWT model. The idea is to map function calls to virtual tasks and to define a single virtual worker within each application process. This way, the ALDY mechanism of dynamic task assignment to workers can be used for the assignment of function calls. This leads to the modified application program:

```
/* Central Process */
main() {
  int i;
  for (i=1; i<=100; ++i) generate tasks representing function calls
  wait for all results
  send task representing end-of-program call to each worker
  sum up results
}
......
/* Replicated Process */
main() {
  define a worker which represents this process
  do {
    wait for a task
    if (task represents end-of-program) break;
    evaluate task (function call) and return result
  } while (1);
}
```

Task Farming Program, Stage 2

Before getting the final version of the program, identifiers for all virtual objects have to be selected. The following table shows the identifiers assuming that there are 10 replicated processes and one additional central process.

1	central process
2...11	replicated processes
102...111	worker w located at process $w - 100$
200	group consisting of all workers
1001...1100	task t representing call $f(t - 1000)$
2002...2011	task t representing end-of-program call for process $t - 2000$

A group consisting of all workers (200) is used to simplify the specification that each function call can be assigned to each worker. Returning and collecting results from function calls has not to be changed if ALDY is used, therefore this part remains as pseudo-code. This leads to the final version of the program, representing stage 3.

```
/* Central Process, Id==1 */
main() {
  int i;
  ALDY_Init(1,1,2); /* see 3.2 */
  for (i=1001; i<=1100; ++i) { /* generate tasks representing function calls */
    ALDY_DefineObject(i,ALDY_Task);
    ALDY_AddDirective(ALDY_TW,i,200);
    ALDY_ReadyObject(i);
  }
  wait for all results
  /* generate task representing end-of-program call to each worker: */
  for (i=2002; i<=2011; ++i) {
    ALDY_DefineObject(i,ALDY_Task);
    /* bind end-of-program task i to the worker i-2000+100: */
    ALDY_AddDirective(ALDY_TW,i,i-2000+100);
    ALDY_ReadyObject(i);
  }
  sum up results
  while (!ALDY_Terminate()); /* wait for termination */
}

......
/* Replicated Process, Id is p */
#define w (100+p) /* w is my local worker */
main() {
  ALDY_Init(p,1,(p==11)?1:p+1); /* see 3.2 */
  /* define worker w which represents this process: */
  ALDY_DefineObject(w,ALDY_WORKER);
  ALDY_PutInGroup(w,200); /* make group of all workers */
  ALDY_AddDirective(ALDY_WP,w,p); /* assign worker locally */
  ALDY_ReadyObject(w);
```

```
  ALDY_AwaitWorker(w); /* wait for the assignment of worker w */
  do {
    int task = ALDY_NextTask(ALDY_BL,w,0); /* wait for a task */
    if (task>2000) break;
    ALDY_StartAction(w);
    evaluate function call f(task - 1000) and return result
    ALDY_EndAction(w);
  } while (1);
  ALDY_EndWorker(w);
  while (!ALDY_Terminate()); /* wait for termination */
}
```

Task Farming Program, Stage 3

Call backs

The implementation of the call back functions for this example is straightforward. Since workers are not migrated and worker assignment needs no specific treatment, the functions `ALDY_CB_AssignWorker` and `ALDY_CB_MigrateWorker` can be implemented with an empty body. The functions `ALDY_CB_Receive` and `ALDY_CB_SendInfo` can be implemented (if PVM 3 is used) similar to section 3.3. The function `ALDY_CB_SendObject` can be implemented similar to `ALDY_CB_SendInfo` since it will not be called for workers (workers are assigned only locally) and tasks are completely specified by their identifiers. Therefore, the function `ALDY_CB_RecvObject` can also be implemented with an empty body. It remains to realize the function `ALDY_CB_Time` using the conventional operating system calls.

Conclusion

At first sight, it seems to be toilsome to use ALDY for such a relatively simply structured application. But consider what we have gained by integrating the ALDY library: Now the program can easily use a large number of different load distribution strategies without need of recompiling. We just have to specify another strategy in the ALDY parameter file.

5.2 Replicated Servers

The example for a replicated server application is the implementation of a distributed data structure, which is assumed to be split into 100 *partitions*. There are 10 *server processes* managing the 100 partitions. A *client process* performs 10000 asynchronous accesses to the partitions. To implement load distribution, it is obviously not possible to use a similar approach as for the task farming program above, because an access to a partition has to be executed by the server process which is responsible for that partition. For this application, load distribution has to be implemented by dynamically changing the mapping of partitions to processes. Again, it is straightforward to extend this example to "real" problems. These are for example distributed simulations and iterative algorithms on distributed data structures. Returning and collecting of results is omitted since it is similar to the previous example. Furthermore, program termination can be realized similar to the next, more comprehensive example and therefore is not described. This is the first stage of the program:

```
/* Client Process */
main() {
  int i;
  for (i=1; i<=10000; ++i)  access to partition i% 100 (modulo)
}

......
/* Server Process */
main() {
  initialize local partitions
  do {
    receive an access to local partition
    evaluate access and return result
  } while (1);
}
```

Replicated Server Program, Stage 1

For stage 2, application objects have to be mapped to objects of the PWT model. As mentioned above, load distribution can be realized by migrating partitions among processes. Therefore a partition is mapped to a *worker*. (Generally, several partitions can be combined to a single worker, if the granularity of one partition is too fine.) Accesses to the partitions are mapped to *tasks*. Now the program can be described as follows:

```
/* Client Process */
main() {
  int i;
  for (i=1; i<=10000; ++i)  generate task representing access to worker i% 100
}

......
/* Server Process */
main() {
  initialize local workers
  do {
    receive a task representing access to local worker
    evaluate access and return result
  } while (1);
}
```

Replicated Server Program, Stage 2

To realize stage 3, a mapping of application objects to ALDY identifiers is required.

1 2...11 20	client process server processes group consisting of all server processes
100...199 200	worker w representing partition $w - 100$ group consisting of all workers
10001...20000	task t representing access number $t - 10000$ of client

The partitions are numbered from 0 to 99. The initialization of partitions is arranged so that server process p initializes the 10 partitions $10(p-2)$ to $10(p-2)+9$ for $p = 2, \ldots, 11$. A group consisting of all server processes and a group consisting of all workers is used to facilitate specification of the directives. This is stage 3 of the replicated servers program:

```
/* Client Process, Id==1 */
main() {
  int i;
  ALDY_Init(1,1,2); /* see 3.2 */
  for (i=10000; i<=20000; ++i) {
    ALDY_DefineObject(i,ALDY_Task);
    /* task i has to be assigned to partition i%100: */
    ALDY_AddDirective(ALDY_TW,i,100+i%100);
    ALDY_ReadyObject(i);
} }
......
/* Server Process, Id==p */
main() {
  int w,task;
  ALDY_Init(p,1,(p==11)?1:p+1); /* see 3.2 */
  ALDY_PutInGroup(p,20); /* process group */
  for (w=100+(p-2); w<=100+(p-2)+10; ++w) {
    initialize partition w-100 (application specific)
    ALDY_DefineObject(w,ALDY_WORKER);
    ALDY_PutInGroup(w,200);
    ALDY_AddDirective(ALDY_WP,w,p); /* start worker locally */
    /* worker can be migrated to each server process: */
    ALDY_AddDirective(ALDY_WM,w,20);
    /* worker is neighbor of all other workers: */
    ALDY_AddDirective(ALDY_WN,w,200);
    ALDY_ReadyObject(w);
  }
  for (w=ALDY_NextWorkerPos(0); ; w=ALDY_NextWorkerPos(w))
  if (w) { /* caution: w may be 0 if no worker is locally assigned */
    if (task = ALDY_NextTask(ALDY_NBL,w,0)) { /* look for an access */
      ALDY_StartAction(w);
      evaluate access task to partition w-100 and return result
      ALDY_EndAction(w);
} } }
```

Replicated Server Program, Stage 3

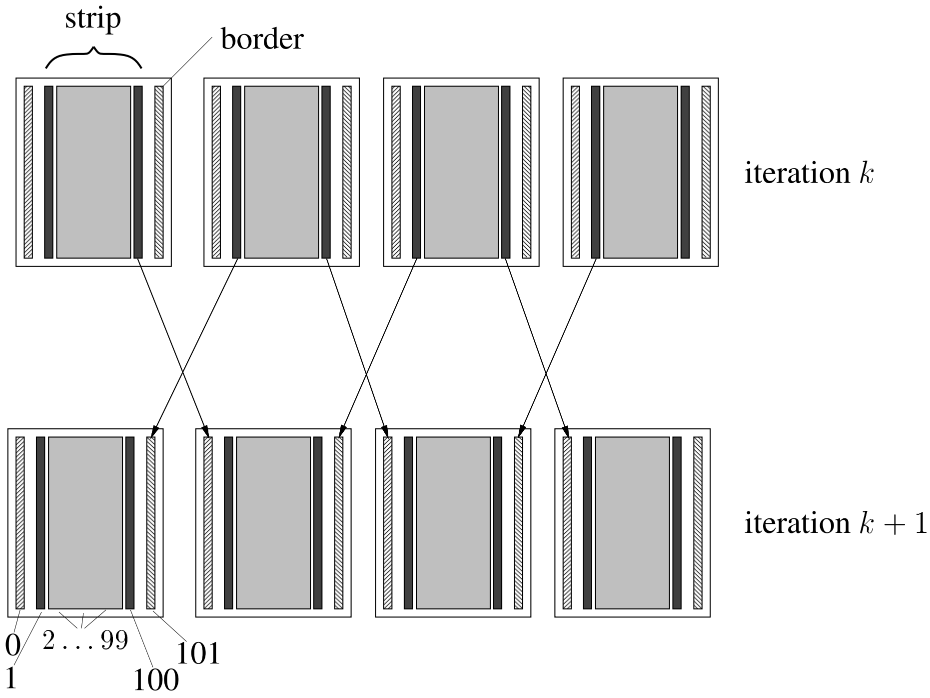

Figure 5: Iterative synchronized computation

The call back functions can be implemented similar to those in the next example. Summing up, we can make the same conclusion as for the task farming example: The integration of ALDY transforms the application program to a basis for a large number of different load distribution strategies, although it requires some work.

5.3 Synchronized Iterative Computation

The third example is an iterative synchronized computation. An iterative algorithm is performed on a rectangular data domain (a matrix). The function value of an element of the matrix depends on its value in the previous iteration and the values of its horizontal and vertical neighbors. The application is parallelized by partitioning the data domain into vertical *strips*, that means several (adjacent) colums of the matrix are combined to a strip. To compute an iteration for a strip, the algorithm needs of course the strip values of the previous iteration and also the previous *border values* of the left and right strip (see figure 5). Obviously, the task farming approach is not useful for realizing dynamic load distribution for this application: Since a process needs all strip values of the previous iteration, it would be too expensive to assign the computation of an iteration to an arbitrary process. The computation of a strip has usually to be performed by the process which "owns" the data of the strip. The basic concept for dynamic load distribution for this application is to assign several strips to one process. By *migrating* strips dynamically from *overloaded* to *underloaded* processes, the application can control the workload for each process. The following program represents stage 1 of the application. The example is slightly simplified because it is assumed that all processes are *inner* processes, that means that each process has a left and right neighbor. In the complete application, there would be two additional processes which are similar to the inner processes but which have only one neighbor.

```
/* I'm process p */

void compute_iteration(int s, int i) {
  compute iteration i of strip s
  if (not last iteration) send local left and right borders to left and right neighbor strip
}

main() {
  for all strips s which are initialized by process p do {
    initialize data of strip s (including borders)
    compute_iteration(s,0);
  }
  for (i=1; i <= max_iterations; ++i) {
    for all strips s which are locally assigned do {
      wait for left and right border
      compute_iteration(s,i);
    }
  }
}
```

Synchronized Iterative Computation, Stage 1

Stage 2 introduces the objects of the PWT model. Obviously strips are represented by workers, since load distribution should be realized by migrating strips. The communication between strips is a communication between migratable workers. That means it would be difficult for the application to implement communication between workers directly. Therefore the application should represent the communication between strips by tasks. Furthermore, workers should not be migrated during the computation of an iteration. Therefore, tasks should also represent the computation of an iteration for one strip. There are several possibilities to realize this concept. In this example, two different types of tasks are used: *left-tasks* which are sent to the left neighbor strip represent only a message containing a border, and *right-tasks* which are sent to the right neighbor strip represent a message containing a border *and* the computation of an iteration. Correct synchronization is achieved by computing the left task (just inserting the border) and then computing the right task (inserting the other border and performing the iteration step). The `for` loop across the iterations has to be changed: If workers are migrated, it cannot be guaranteed that workers are computing the same iteration. Therefore, the loop is replaced by an information for each worker which describes the actual iteration number for the worker.

```
/* I'm process p */

void compute_iteration(int w, int i) {
  compute iteration i of strip of worker w
  if (last iteration) terminate worker w;
  else generate left-tasks and right-tasks for neighbor strips
}
```

```
main() {
  for all workers w which are initialized by process p do {
    initialize data of strip w (including borders)
    compute_iteration(w,0);
  }
  while (no termination)
  for all workers w which are locally assigned do {
    determine appropriate task for worker w using worker information
    look for appropriate task for worker w
    if (task is here) {
      insert border into data structure
      if (task is right-task) compute_iteration(w,iteration number);
    }
  }
}
```

Synchronized Iterative Computation, Stage 2

The following table shows the identifiers for the objects. There are 10 processes and 100 workers computing 100 iterations.

1...10 20	the processes group consisting of all processes
100...199 200	worker w representing strip $w - 100$ group consisting of all workers
10000...19999	left-task t represents left border message after iteration (t mod 100) of strip $(t - 10000)/100$ (in other words: left border message after iteration i if strip s is encoded as $10000 + 100s + i$
20000...29999	right-tasks; similar to left tasks

The initialization of strips is arranged similar to the previous example: process p initializes strip $10(p-1)$ to $10(p-1)+9$. Again, a group consisting of all processes and a group consisting of all workers is formed. In contrast to the previous example, the concrete implementation of data structures which are mapped to virtual objects is shown. A strip is realized as an array of columns. The borders to the strip are integrated in this array (see figure 5). In addition, the components `right_next` and `iteration` specify whether the next task is a right task and the number of the previous iteration, respectively. When the first iteration is computed, the worker should not be migrated during the call of `compute_iteration()`. Therefore the functions `ALDY_NoMigration` and `ALDY_PermitMigration` are used.

```
/* I'm process p */
typedef struct { float c[1000]; } column; /* a column of the matrix */
typedef struct {
      column matrix[102];
      int right_next, iteration; } strip;

void * copy_border(int w, int is_right_border) {
  /* this function allocates memory, copies the left or right border of worker w */
  /* into the allocated area, and returns a pointer to this area: */
  int index = (is_right_border)?100:1; /* see figure 5 */
  void * m = (void *)malloc(1000*sizeof(float));
  return memcpy(m,((strip *)ALDY_GetObjectAddr(w))->matrix[index],
      1000*sizeof(float));
}

void compute_iteration(int w, int i) {
  ALDY_StartAction(w);
  compute iteration i of strip of worker w
  ALDY_EndAction(w);
  ++((strip *)ALDY_GetObjectAddr(w))->iteration;
  if (i==99) ALDY_EndWorker(w);
  else { int t = 10000+100*(w-100)+i;
    ALDY_DefineObject(t,ALDY_TASK); /* define left-task */
    /* left-task has to be assigned to the left strip; */
    ALDY_AddDirective(ALDY_TW,t,w-1);
    /* save the border and assign address to the task: */
    ALDY_PutObjectAddr(t,copy_border(w,0));
    ALDY_ReadyObject(t);
    /* do the same for the right-task: */
    t+=10000;
    ALDY_DefineObject(t,ALDY_TASK);
    ALDY_AddDirective(ALDY_TW,t,w+1);
    ALDY_PutObjectAddr(t,copy_border(w,1));
    ALDY_ReadyObject(t);
  }
}
```

```
main() {
  int w;
  ALDY_Init(p,1,(p==10)?1:p+1); /* see 3.2 */
  ALDY_PutInGroup(p,20); /* process group */
  /* optimization: inform ALDY about all initial worker locations: */
  for(w=100; w<=199; ++w)
    ALDY_WorkerLoc(w,(w-100)/10+1);
  /* initialize data structures of all local strips: */
  for(w=100+10*(p-1); w<=100+10*(p-1)+9; ++w) {
    strip * s = (strip *)malloc(sizeof(strip));
    s->is_right_border = 0; /* wait first for left border */
    s->iteration = -1; /* number of last iteration */
    initialize strip matrix s->matrix;
    ALDY_DefineObject(w,ALDY_WORKER);
    ALDY_PutInGroup(w,200);
    ALDY_AddDirective(ALDY_WP,w,p); /* start worker here */
    ALDY_AddDirective(ALDY_WM,w,20);
    /* migrate worker only to left or right neighbor: */
    ALDY_AddDirective(ALDY_WN,w,w-1);
    ALDY_AddDirective(ALDY_WN,w,w+1);
    ALDY_PutObjectAddr(w,(void *)s);
    ALDY_ReadyObject(w);
    ALDY_AwaitWorker(w);
    ALDY_NoMigration(w); /* compute first iteration; prevent from migration: */
    compute_iteration(w,0);
    ALDY_PermitMigration(w);
  }
  for (w=ALDY_NextWorkerPos(0); !ALDY_Terminate();
      w=ALDY_NextWorkerPos(w)) if (w) {
    strip * s = (strip *)ALDY_GetObjectAddr(w);
    int t = 10000+10000*s->right_next+(w-100)*100+s->iteration;
    if (ALDY_NextTask(w,ALDY_NBL,t)) {
      /* insert border into data structure, see figure 5: */
      memcpy(s->matrix[(s->right_next)?101:0],
        ALDY_GetObjectAddr(t),1000*sizeof(float));
      if (s->right_next) compute_iteration(w,iteration-number);
      s->right_next = 1-s->right_next;
    }
  }
}
```

Synchronized Iterative Computation, Stage 3

This example shows the advantage of user-directed identification of virtual objects: Since all identifiers are planned (according to the table above), the application "knows" the identifiers of requested tasks and can use this information to wait for specific tasks.

Call backs

It remains to implement the call back functions. The implementation of `ALDY_CB_Time` is not shown, because this function depends only on the environment and not on the specific application. `ALDY_CB_AssignWorker` can be implemented with an empty body. For the other call back functions, PVM 3 is used as sample environment. `ALDY_CB_SendInfo` and `ALDY_CB_Receive` can be implemented similar to section 3.3. `ALDY_CB_SendObject` can be implemented as follows (cf. section 3.3):

```
int ALDY_CB_SendObject(int ob, int pid, int * addr, int n) {
  pvm_initsend(PvmDataDefault); /* initialize send-buffer */
  pvm_pkint(&n,1,1); /* put length of ALDY-message */
  pvm_pkint(addr,n,1); /* put ALDY-internals */
  if (ob<200) /* if object is worker */ {
    /* send worker data (strip) and free allocated memory: */
    strip * s = (strip *)ALDY_GetObjectAddr(ob);
    pvm_pkfloat(s->column,1000*102,1);
    pvm_pkint(& s->right_next,1,1); pvm_pkint(& s->iteration,1,1);
    free(s);
  }
  else /* if object is a task */{
    /* send task data (column) and free allocated memory: */
    pvm_pkfloat((float *)ALDY_GetObjectAddr(ob),1000,1);
    free(ALDY_GetObjectAddr(ob));
  }
  /* assuming that the array tids contains pvm-identifiers */
  /* send ALDY message, tag 999 is used to mark ALDY-messages */
  pvm_send(tids[pid-1],999);
  /* return number of bytes of object: */
  return (ob<200) ? sizeof(strip) : sizeof(column);
}
```

The function `ALDY_CB_RecvObject` has the opposite job of `ALDY_CB_SendObject`, that means it has to read the data structure of workers and task out of the message buffer. The implementation therefore is straightforward. It remains to implement the function `ALDY_CB_MigrateWorker`. Here we profit from the migration concept: Analyzing the locations in the code where a worker may be migrated, we find that it is sufficient to send the same data as within `ALDY_CB_SendObject` for a worker migration. Then we have moved all data necessary for the worker to the remote process. Therefore, the implementation of `ALDY_CB_MigrateWorker` is straightforward too.

Conclusion

The third example is an almost complete (with respect to communication and synchronization) but not trivial application. It demonstrates, that integration of ALDY is almost straightforward and can be performed systematically. Although we have to implement the basic call back routines for sending and receiving of objects, all difficult parts for realizing dynamic load distribution based on data migration are provided by the ALDY system. Again, stage 3 represents a basic implementation for which we now can easily apply many different load distribution strategies, ranging from simple receiver-initiated strategies to rather complex strategies like for example microeconomic strategies [FYC88] (see below).

6 Summary and Future Work

This report describes the ALDY load distribution system. ALDY has several new concepts which were not provided by previous load distribution systems:

- A large range of applications (or even parallel run time systems) with very different distribution objects can be supported by ALDY using the *virtual objects* concept.
- The *PWT* programming model supports not only the task farming programming style, but also migration mechanisms.
- Portability of the application is not reduced by using ALDY since all internal data exchange of ALDY is implemented on top of the applications communication mechanisms.
- ALDY provides a collection of several parametrized load distribution strategies which can be easily exchanged without having to recompile the application.

The ALDY library interface for ANSI-C and several examples illustrating the use of the interface are presented. The examples show that although integration of ALDY requires some effort, a large part of work for integrating ALDY can be done straightforward. The main effort is to find a suitable mapping from real application objects and their correlations to virtual ALDY objects and directives for the load distribution strategy.

Currently, ALDY is implemented as a prototype. Out of the large variety of existing load distribution strategies, the ALDY prototype supports only some basic strategies like the receiver-initiated strategy (as described in section 4.2) and a corresponding sender-initiated strategy. Although these strategies usually appear to be very efficient, it will be interesting to test other strategies like for example microeconomic strategies [FYC88] or strategies using physical analogies [HS92]. In addition, we plan to extend the library interface to allow dynamic addition and termination of virtual processes.

Another feature is the introduction of *weighted directives*: The assignment directives of the current version are strict: the load distribution strategy has to consider them absolutely. This concept can be generalized by considering weighted directives not as absolute directives but only as (weighted) hints for the load distribution strategy. The disadvantage of this concept would be a more complicated and more obscure load distribution strategy.

The ALDY library interface is independent from a concrete implementation of the load distribution system. On principle, it can be used as a common interface to all load distribution systems based on the process-worker-task programming model. Therefore the ALDY user interface is a step towards a standard interface to load distribution systems, similar to MPI [Wal94] as a standard interface to message passing systems.

References

[CGMS94] N.J. Carriero, D. Gelernter, T.G. Mattson, and A.H. Sherman. The Linda alternative to message-passing systems. *Parallel Computing*, 20(4):633–656, 1994.

[CS93] C.H. Cap and V. Strumpen. Efficient Parallel Computing in Distributed Workstation Environments. *Parallel Computing*, 19(11):1221–1234, 1993.

[FM87] Raphael Finkel and Udi Manber. DIB—A Distributed Implementation of Backtracking. *ACM Transactions on Programming Languages and Systems*, 9(2):235–256, 1987.

[FYC88] Donald Ferguson, Yechiam Yemini, and Nikolaou Christos. Microeconomic Algorithms for Load Balancing in Distributed Computer Systems. In *Proceedings of the 8^{th} International Conference on Distributed Computing Systems, San Jose, California*, pages 491–499, 1988.

[Gri93] Andrew S. Grimshaw. Easy-to-use Object-Oriented Parallel Processing with Mentat. *Computer*, 26(5):39–51, 1993.

[HS92] Hans-Ulrich Heiss and Michael Schmitz. Distributed Load Balancing Using a Physical Analogy. Technical Report 5/92, Universität Karlsruhe, Fakultät f"ur Informatik, 1992.

[HY91] Kien A. Hua and Honesty C. Young. A Cell–Based Data Partitioning Strategy for Efficient Load Balancing in A Distributed Memory Multicomputer Database System. Technical Report RJ 8041, IBM Research Report, 1991.

[KK93] L.V. Kalé and S. Krishnan. Charm++: A portable concurrent object oriented system based on C++. In *Conference on Object Oriented Programming Systems, Languages and Applications*, 1993.

[SGDM94] V.S. Sunderam, G.A. Geist, J. Dongarra, and R. Manchek. The PVM concurrent computing system: Evolution, experiences, and trends. *Parallel Computing*, 20(4):531–546, 1994.

[Tär94] Erik Tärnvik. Dynamo - A Portable Tool for Dynamic Load Balancing on Distributed Memory Multicomputers. *Concurrency: Practice and Experience*, 6(8):613–639, 1994.

[Wal94] David W. Walker. The design of a standard message-passing interface for distributed memory concurrent computers. *Parallel Computing*, 20(4):657–673, 1994.

[WL88] Robert A. Whiteside and Jerrold S. Leichter. Using Linda for Supercomputing On a Local Area Network. In *Proceedings of the Supercomputing '88, Orlando, Florida*, pages 192–199, 1988.

[Zho92] Songnian Zhou. LSF: load sharing in large-scale heterogeneous distributed systems. In *Workshop on Cluster Computing, Tallahassee, Florida*, December 1992.