Mathematical Modelling and Numerical Analysis ESAIM: M2AN
Modélisation Mathématique et Analyse Numérique M2AN, Vol. 36, N° 5, 2002, pp. 923–935
DOI: 10.1051/m2an:2002040

SOME NEW TECHNICS REGARDING THE PARALLELISATION OF ZÉBULON, AN OBJECT ORIENTED FINITE ELEMENT CODE FOR STRUCTURAL MECHANICS

Frédéric Feyel[1]

Abstract. A finite element code, called ZéBuLoN was parallelised some years ago. This code is entirely written using an object oriented framework (C++ is the support language). The aim of this paper is to present some problems which arose during the parallelization, and some innovative solutions. Especially, a new concept of message passing is presented which allows to take into account SMP machines while still using the parallel virtual machine abstraction.

Mathematics Subject Classification. 65Y99.

Received: 26 November, 2001. Revised: 27 June, 2002.

1. Introduction

ZéBuLoN [1][2] is a finite element code devoted to non-linear structural mechanics. One of its specificities is to be entirely developed in a Object Oriented framework, using C++ as the support language. It contains now about 350 000 lines of C++ code.

The code was parallized some years ago [3], using a domain decomposition method named FETI [2]. This parallized code now relies on the SPMD paradigm: several copies of the code are concurently running, operating on several data sets.

The objective of this paper is to present the general parallel architecture of the code, to describe some problems we focused on, and the innovative object oriented solutions we found to overcome these difficulties. All the following technics have been developed for ZéBuLoN, but are very general and can be applied to any object oriented domain decomposition code.

2. Parallel architecture

2.1. Presentation

ZéBuLoN is an innovative research code and is also a commercial product sold in Europe and United States. It then has been ported to a lot of different architectures, including all Unix-like machines and Windows NT.

As opposed to a master–slave architecture, ZéBuLoN relies on the SPMD paradigm. However, it uses also another task which is responsible for watching the computation. This task is not responsible for any usefull

Keywords and phrases. Parallel computing, finite element method, message passing, multithreading, object oriented, I/O.

[1] ONERA, DMSE–LCME, 29 Avenue de la division Leclerc, BP 72, 92322 Châtillon, France. e-mail: `feyel@onera.fr`

[2]`http://www.nwnumerics.com/Zebulon`

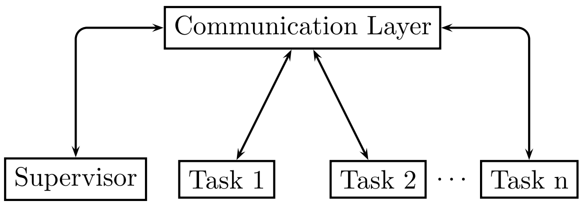


FIGURE 1. General parallel architecture of ZéBuLoN.

computation, but takes care of the stability of the underlying virtual machine. It is automatically warned if one task of the code fails or becomes unreachable due to some network problem.

This component (called the *supervisor*) works as a request server: any task of zebulon, but also the operating system and the message passing library, can send a request to the supervisor. This request, for instance, can notify an host or task failure. Figure 1 presents this parallel architecture.

The main operation of the supervisor is to wait for any message. The supervisor maintains a list of `HANDLER_MP` instances. Besides some utilities methods, class `HANDLER_MP` just declares a virtual method named `int action(int)` which will be called if a task requests an action of a specific handler. Each handler also provides the supervisor with a list of integer describing all requests type supported. During its initialization, the supervisor asks the object factory (see Sect. 3.3) to get all `HANDLER_MP` derivatives and creates a single instance of them.

When a message is received by the supervisor, the list of all handlers is scanned to check if a handler is able to respond to the message tag:

1. if found, the message tag is passed to the `action` method of the handler;
2. if not found, a message is printed to warn the user that an unknown message tag was received.

For efficiency reasons, the handler list is pre-scanned during the initialization to optimize step 1.

It appears that the supervisor is also a good place to provide other services. The most important of these services is a networked file server.

2.2. A micro networked file server

File access is usually a tricky task in the framework of SPMD programming. The easiest solution is to use a classical external networked filesystem such as NFS which allows all copies of the code to access the same input data files. However, it is not always possible for the end-user to activate this solution, because it needs system administrator priviledges. Another drawback of such approach is that the performances of these very general networked filesystems are not so good when a lot of processes access to the same file. To bypass these problems, two solutions are usually used:

- data file *replication*: all input data files are replicated (copied) on all hosts in a local filesystem. The question is then: how to be sure that all these copies are consistent? The end-user often forgot to update these files or update only some of them ...
- use of an *internal networked* database: the files are read once by the first task which is responsible for sending the definition of the problem to be solved to all other tasks. This way of programming is very intrusive: all input functions of the code have to be modified which leads to a specific parallel version of the code. Two versions of the same code have to be maintained: one for sequentiel computations, another for parallel computations.

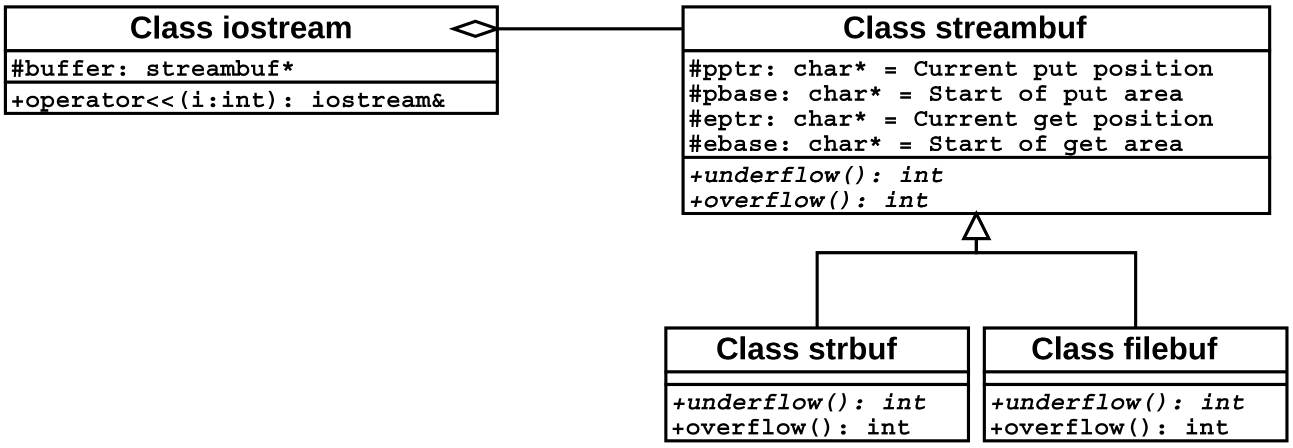


FIGURE 2. C++ general I/O classes.

Thanks to the object oriented scheme and to the standard C++ library, we suggest another approach in ZéBuLoN, which ensures that:

- all tasks read a *consistent* input data set;
- the code responsible for reading this input data set is *not modified* and is exactly the same for parallel and sequential computations.

2.2.1. *C++ I/O overview*

The C++ norm defines a lot of classes to handle input/output. Figure 2 shows the general architecture of C++ I/O classes.

I/O classes are divided in two parts. Some classes, deriving from class `iostream` are responsible for the formating of the output (*i.e.* how to write the integer 123 as the 3 characters '1', '2' and '3'), whereas other classes deriving from class `streambuf` are buffers responsible for reading/writing datas from/to the real "hardware". We put "hardware" between quotes, because it is not necessarily a real physical disk: one may connect a C++ file (`iostream`) with a character string, and will use class `strbuf` instead of class `filebuf` (this later class accesses a file on a disk).

Thanks to the polymorphism of C++, one can derive its own `streambuf` implementation to provide access to new capabilities. This is exactly what we do in ZéBuLoN: we derive a class named `Zbuffer` from `streambuf`. This new class allows to handle networked file operations. It works with a specific handler in the supervisor which transforms this I/O request into real I/O calls.

Each time a task opens a file for reading or writing during a parallel computation, the `iostream` object creates a buffer of type `Zbuffer` instead of `filebuf`. The open request is then sent to the supervisor and the handler opens the file and sends back a file identifier (or an error code, if something goes wrong) for later use. All following I/O operations are done by sending requests to the supervisor. Figure 3 illustrates this technique by showing what happens when a developer opens a file (`f`) and writes an integer (`i`).

One can identify the following advantages of such technique:

- all these operations are heavily buffered (the size of the buffer is a parameter) to avoid too much network traffic;
- when several tasks open the *same* file for reading, the supervisor only opens the file once and works by maintaining buffers for each task; it thus avoids I/O jams;
- the code used by developers in the whole code is nothing but a classical C++ code for doing I/O: nothing new, nothing complex.

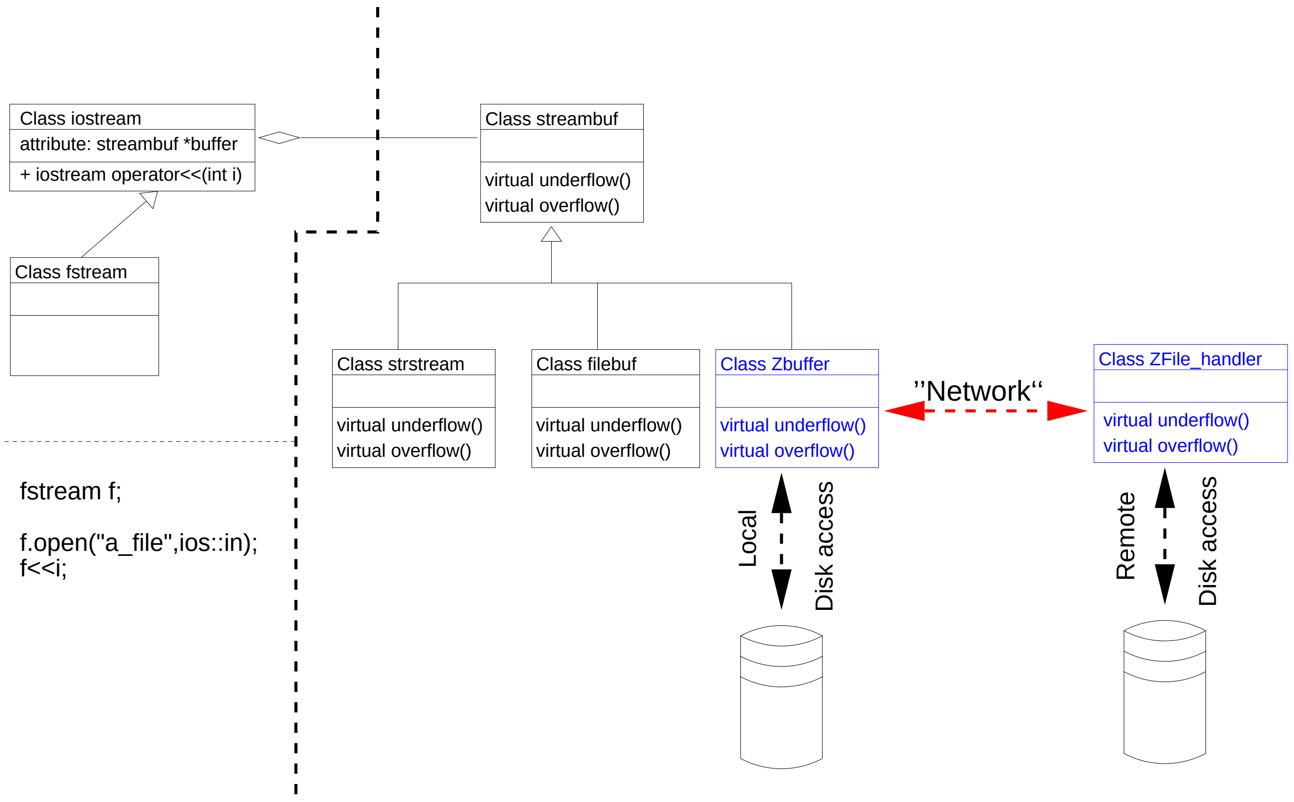


FIGURE 3. What happens when a developer opens a file in ZéBuLoN using classical C++ calls?

We did not implement anything to ensure consistent writings to the same file. The main reason is that we do not need it in ZéBuLoN, because all tasks store their results on a local disk.

Some libraries provide parallel input/output capabilities (for instance MPIv2). We made the choice of *not* using these capabilities, because they are certainly too specific to this standard, and will be perhaps missing in future message passing libraries. As we will see in the following section, one of the lightmotiv of ZéBuLoN is to be independant of third party library as much as possible, and then to avoid the use of highly specialized capabilities such as MPI parallel I/O.

3. PORTABILITY

Portability is then a major worry in the ZéBuLoN environment.

This term *portability* hides several points:

- the *same* source code must be complied and run on a lot of heterogeneous architectures;
- the *same* binary executable must be used regardless of the kind of computation (*i.e.* parallel or sequential computation);
- the code must be independent of any third-party library.

We will focus in this section on the third item.

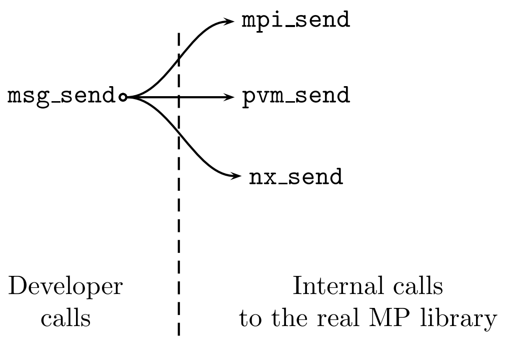


FIGURE 4. A *virtual* interface pattern applied to message passing function calls.

3.1. The problem

Distributed parallel computing requires the use of a message passing library which allows the programmer to use friendly and well designed interface routines to exchange messages between all instances of the code. These libraries avoid the use of low level Input-Ouput operations such as *socket* manipulations. They also provide an interface which is hardware independant: the same functions are used to access Fast-Ethernet network or highly optimized private networks (Myrinet for instance).

In the past, several different libraries were developed:

- NX (message passing library of the Intel Paragon);
- PVM (Parallel Virtual Machine);
- MPI (Message passing interface).

MPI seems to become the today standard library for message passing, but PVM is still maintained, developed and used by a lot of codes. Other message passing libraries will certainly exist in the future.

The challenge for all developers is then to be able to build a code which is independent of a specific message passing library as much as possible.

3.2. ZéBuLoN's solution

This is done in ZéBuLoN by the use of a specific object oriented pattern which we called a *virtual interface* [4]. The basic idea is to:

(i) identify all needed features. In our case we basically need to send and receive a packed message, and to use some reduction operation (min, max, sum). It is important to notice that scientific codes usually only require a limited number of message passing function whereas all libraries provide a huge number of different routines. This step is then nothing but find the basic MP functions common to all libraries;
(ii) build an internal message passing library;
(iii) use only this internal library which will transform internal MP calls into function calls compliant to the real message passing library.

Figure 4 illustrates this idea: the developer only uses internal call (`msg_send`) which are transformed into the right MP call (`mpi_send` or `pvm_send` for instance).

This pattern basically works by using an abstract base class. This base class contains only abstract methods and provides all internal message passing calls:

```
class MESSAGE_PASSING
{
  public :
    virtual void send_message(void*)=0;
    virtual void recv_message(void*)=0;
};
```

specific message passing library classes are then derived from this base class to use the user-chosen message passing library:

```
class MP_PVM : public MESSAGE_PASSING
{
  public :
    virtual void send_message(void*) {
      pvm_send(...);
    }
    virtual void recv_message(void*) {
      pvm_recv(...);
    }
};
```

```
class MP_MPI : public MESSAGE_PASSING
{
  public :
    virtual void send_message(void*) {
      MPI_Send(...);
    }
    virtual void recv_message(void*) {
      MPI_Recv(...);
    }
};
```

Thanks to the polymorphism of C++, this kind of abstraction allows the developer to use a MESSAGE_PASSING instance without knowing wether it is a MP_PVM or a MP_MPI instance. The main code thus only contains calls such as:

```
MESSAGE_PASSING *messenger;

...

messenger->send_message(p);

...
```

3.3. Dynamic linking through an *Object Factory*

The main question is then the following: how to create the right instance of MESSAGE_PASSING? One could imagine that a single specific global function is used for that purpose and initializes a global static object:

```
MESSAGE_PASSING* Initialize_message_passing_object(String &lib)
{
  if(lib=="mpi") return(new MP_MPI);
  else if(lib=="pvm") return(new MP_PVM);
  else ERROR("bad mp lib type");
}
```

this is in fact not a so good idea, because it requires to modify the code each time a new message passing library is taken into account: the developer not only has to derive a new class from MESSAGE_PASSING but also has to modify the code somewhere else to "register" the new capability.

ZéBuLoN massively uses an extension of what Stroustrup calls an "Object Factory". This pattern avoids any imbricated statements, and replaces them by a single call:

```
MESSAGE_PASSING *mp;

if(lib=="mpi") mp=new MP_MPI;
else if(lib=="pvm") mp=new MP_PVM;
else ERROR("bad mp lib type");
```

```
MESSAGE_PASSING *mp;
mp=Create_object(MESSAGE_PASSING,lib);
```

An object factory is usually nothing but two lists: a list of keywords and a list of pointers function. Every keyword in the first list is associated with a function creator in the second list. This function just creates an instance and returns a pointer to the base class.

A couple `(keyword,creator_function)` can be added in the two lists by several means:

1. *direct* addition to the two lists:

   ```
   typedef BASE_CLASS* (*fc)();

   LIST<STRING> keywords;
   LIST<fc> creators;

   BASE_CLASS* f1() { return(new DERIVATIVE); }

   ...

   keywords.add("derivative"); creators.add(f1);
   ...
   ```

2. *automatic extension at link/run time*: this second way is more subtle and is based on the creation of static objects at runtime. It is stated in the C++ norm that static objects are created after the creation of all static "simple" variables (such as `static int` variables for instance), but *before* entering the `main` function. The creation of these static objects follows the classical C++ scheme: especially a constructor is called.

   The constructor of these static objects is then a good place to register any class in the object factory. Some preprocessor macros have been defined in ZéBuLoN to make the registration as simple as possible. The developer has only to use the following code to declare that class `B` is a derivative of class `A` associated to the keyword `"plastic"`:

   ```
   class B : public A { };

   DECLARE_OBJECT(A,B,plastic);
   ```

   The macro called `DECLARE_OBJECT` is expanded by the declaration of a static object in the constructor of which the class is registered:

   ```
   DECLARE_OBJECT(A,B,plastic);
   ```

   ```
   OF_A_B_OBJECT()  return (new B());
   OFP OFP_A_B_OBJECT("A","plastic",OF_A_B_OBJECT);
   ```

   `OFP` is a class which is responsible for the registration itself. This is done, as explained before, in the creator which takes three arguments: two character strings and a pointer function.

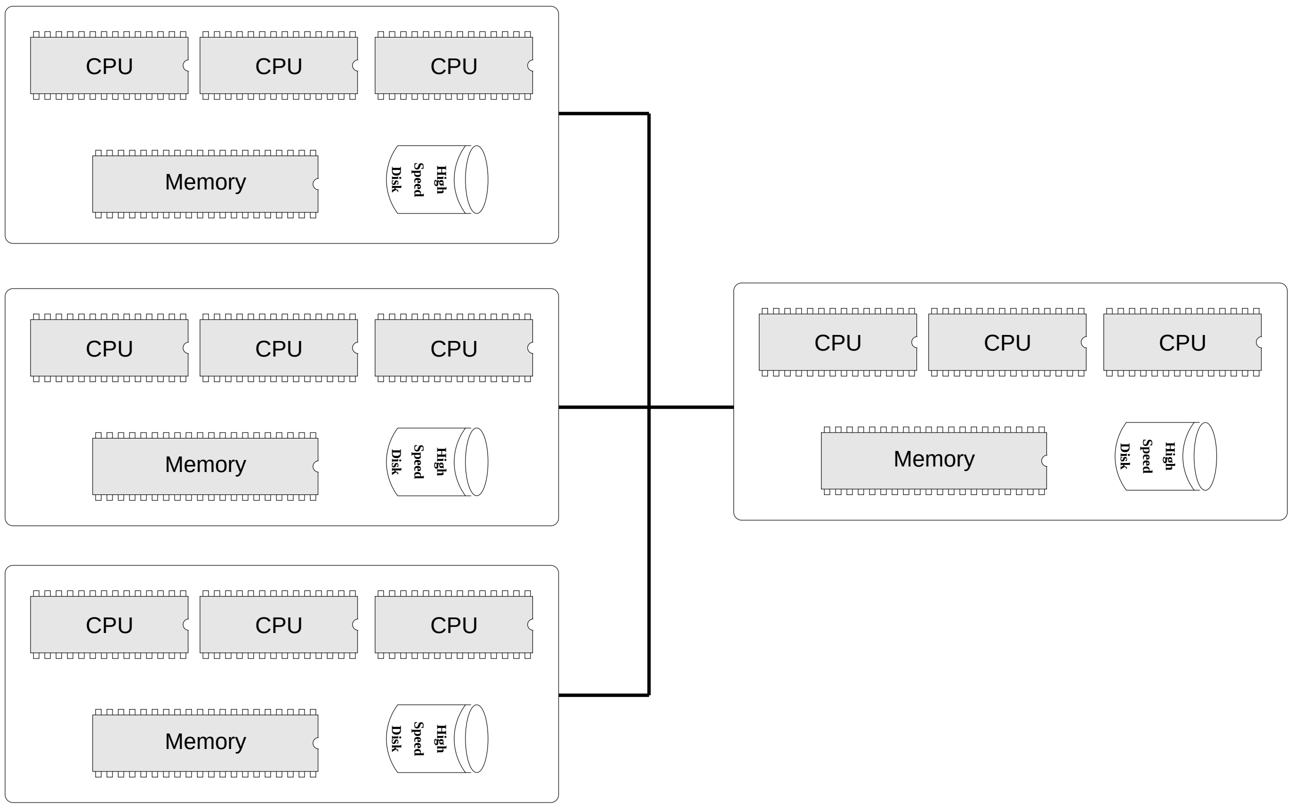


FIGURE 5. A cluster of SMP boxes.

The second technic is used inside ZéBuLoN due to its great flexibility. One has also to notice that the creation of static objects in dynamic libraries follows the same order: static objects are created just after the loading of any dynamic library. It is then possible to use the same pattern to create "plugins" (*i.e.* dynamic libraries) in which some class derivatives are automatically registered during the dynamic link, just after the loading of that library.

4. Symmetric multi processing *vs.* message passing processing

4.1. **Existing hardware**

It seems that supercomputer hardware architecture is on the way to converge. Most of the supercomputers provided today are based on a cluster of SMP machines. The unit brick of such supercomputer is an SMP box which contains several CPU (usually between 2 and 8), a local memory and some disks. A high speed network links all these SMP boxes together. It can be anything from Fast Ethernet to some very efficient private networks (*e.g.* Myrinet). Figure 5 shows this architecture.

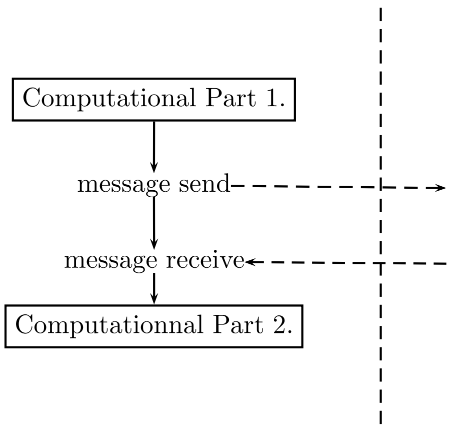
This kind of architecture is thus somewhere between a distributed environment (one might consider each SMP box as a single CPU machine) and a shared environment. It is then necessary to find new paradigm to take this architecture into account.

It is still possible to rely on the parallel virtual machine assumption: each SMP box can be seen as multiple machines, and can concurently run several processes. This way of programming is however not the best one, because it simply ignores the underlying hardware. This kind of approach can also be used on single CPU machines: each CPU runs several copies of the code (using the time sharing capabilities of the operating system).

This is usually a bad idea and produces dramatic results: the CPU time is only consumed by the operating system to swap between these huge processes and not to make usefull floating point calculations. This is mainly because the standard time slice of most operating systems is about $t_s \approx 10^{-2}$ s: each t_s second, the operating system stops the running process and tries to "activate" a new process for execution. The time required for this "activation" varies according to a lot of parameters and depends on the memory size of the process: it increases with the memory size.

4.2. A new scheduler appropriate for message passing applications

It is possible to define a new scheduling algorithm which is well suited for message passing applications and resolved the previous difficulty. We only assume there that we take into account a *real* application which contains some message passing stages. Finishing part 1, a given task sends some results and waits for data from other tasks before starting part 2:



We assume that the task can not go on before the receive operation complete. It means that our application is *data synchronous*: the synchronization between all tasks is done automatically because some tasks need datas from other tasks to continue. This is the case of a lot of scientific application, including ZéBuLoN.

The new scheduling algorithm is then:

1. let N be the number of "active" task at any time. A task is said to be "active" when it can run without receiving any message (for instance during FE computation, a task solving the local sparse linear tangent system is active because it can do this local operation without exchanging any information);
2. when an active task wants to receive a message:
   (a) mark the task as inactive, and stop it;
   (b) choose another inactive and eligible task (a task which is waiting for a message, **and** this message has already arrived), based for instance on a round robin algorithm, mark it as active, and run it.

This new scheduling algorithm is based on the data synchronism. Deadlock can occur when all tasks are waiting for a message (but none received a good message to go on), but this is also the case in classical message passing programming.

There are many advantages of such algorithm, especially:

- It allows to dynamically fit the number of active task to the number of CPU. For a given number of tasks, it is then possible to run the same computation starting from (i) a "sequential" computation (all tasks are run one after the other, in a sequential order: $A \to B \to C \cdots \to A$) to (ii) a fully parallel computation (the machine has the same number of CPU as the number of tasks).

- The time sharing scheduling is *not* done by the operating system, but by the computation. It allows a much larger granularity which is automatically adapted. This is sometimes called a cooperative time sharing.

The main idea is that the application developer better knows when to swap tasks than the operating system which has to be able to run a large collection of different processes with different requirements.

4.3. **ZéBuLoN and multithreading**

Most modern operating systems are built on a mulithread kernel. In these operating systems, the execution unit is not a process, but a thread. All processes consist in a number of threads executed concurently sharing the same memory space. Each thread however has its own stack and heap (which means for instance that each thread has a private dynamic memory allocation system).

The main advantage of the thread concept is that a number of threads can be anchored on a single copy of the code, and also that the swap between threads is much more efficient than the swap between processes. Due to their structure, threads are often called *lightweight processes*.

A standard library exists and provides generic functions to manipulate threads: the POSIX thread library [5]. It allows the developer to create new threads in the current thread of execution (each process contains at least a main thread which is automatically created when the process is started), to destroy running threads and to start/stop threads using semaphore. A stopped thread is completely ignored by the system which only watch the condition required to wake up it again: a stopped thread does not consume any CPU time, and is not taken into account by the system time sharing algorithm.

We will see in the following that multithreading is a convenient framework to implement the algorithm described in Section 4.2.

In the usual way of SPMD programming, a task is nothing but a copy of the code running with its own datas. Each task is then referenced by a task number starting from 0. In this new architecture, a single copy of the code runs on each SMP boxes and contains a number of execution threads. A single task number is not enough since it only points out a copy of the code which contains a number of threads.

To bypass this ambiguity, we called a *universe* each process and a *task* a thread which is inside a universe. Each task is then designated by two numbers: (u, t) its universe number (u) and its task number (t) inside this universe. Figure 6 summarizes these notations.

A universe is in fact a process, all tasks inside a given universe will exchange messages using simple memory to memory copy operations. Tasks in different universes will use the chosen message passing library to exchange messages.

We want all message passing operations to be detected and used to schedule the execution of all tasks. To do that, a special task called the *post-master* is responsible for all data exchanges. All tasks are assigned a mail-box in the post-master. A mailbox is a list of pointers, each pointer points to a message.

Because all tasks in a single universe (including the post-master) share the same memory space, there is no useless memory copy: only pointers are transfered to/from the post-master, not datas. Each task owns a semaphore instance which allows to put them into sleep mode and also to wake up them after a sleep period.

4.3.1. *Sending datas*

When a task wants to send a message, the following operations are executed:

1. datas are packed together. Packing is always used in ZéBuLoN, because a message usually does not contains a single information;
2. the post-master is waken up (by increasing its semaphore reference count), and a pointer to packed datas is given to it (by copying it to a specific location);
3. the post-master is then responsible for handling the message: the task continues its execution. The post-master seeks the destination of the message:
   (a) if it is inside its universe, then the data pointer is copied to the mail-box of the destination task;

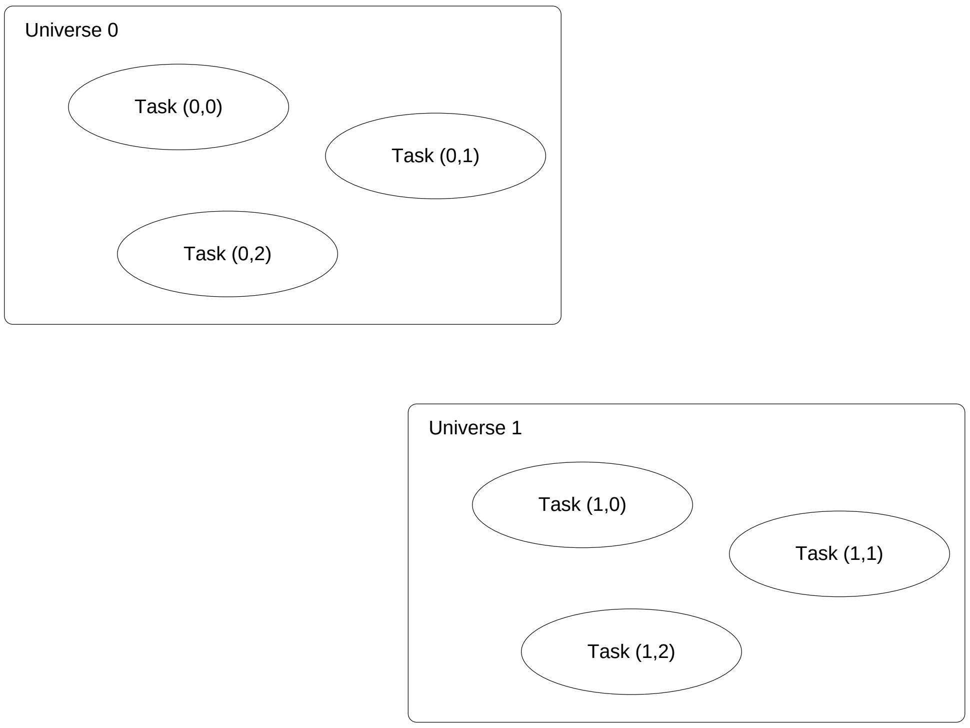


FIGURE 6. The notion of universe and tasks.

   (b) if it is not inside its universe, then the post-master contacts the remote post-master and sends all datas using classical message passing operations.

4. the post-master then goes by itself in sleep mode, waiting for other task requests.

4.3.2. *Receiving datas*

When a task wants to receive a message,

1. the task seeks in its post-master mailbox to see if the wanted message has already arrived or not, depending on the message tag and the message source:
   (a) if the message is in the mailbox, it is used and unpacked. The task continues its execution without interruption;
   (b) if the message is not in the mailbox, the task goes into sleep mode. It will be waken up later by the post-master when a matching message arrives. The post-master chooses another inactive task which is ready to run and places it into active mode.

Finally Figure 7 shows message exchanges in this new framework.

4.4. **Initialization of the system**

The initialization of this system requires the creation of a post-master instance for each universe. The post-master instance will create as many mail-boxes as needed.

This initialization stage is, in fact, completely transparent for the developer. We use the virtual interface pattern (see Sect. 3.2) combined with the dynamic object factory (see Sect. 3.3).

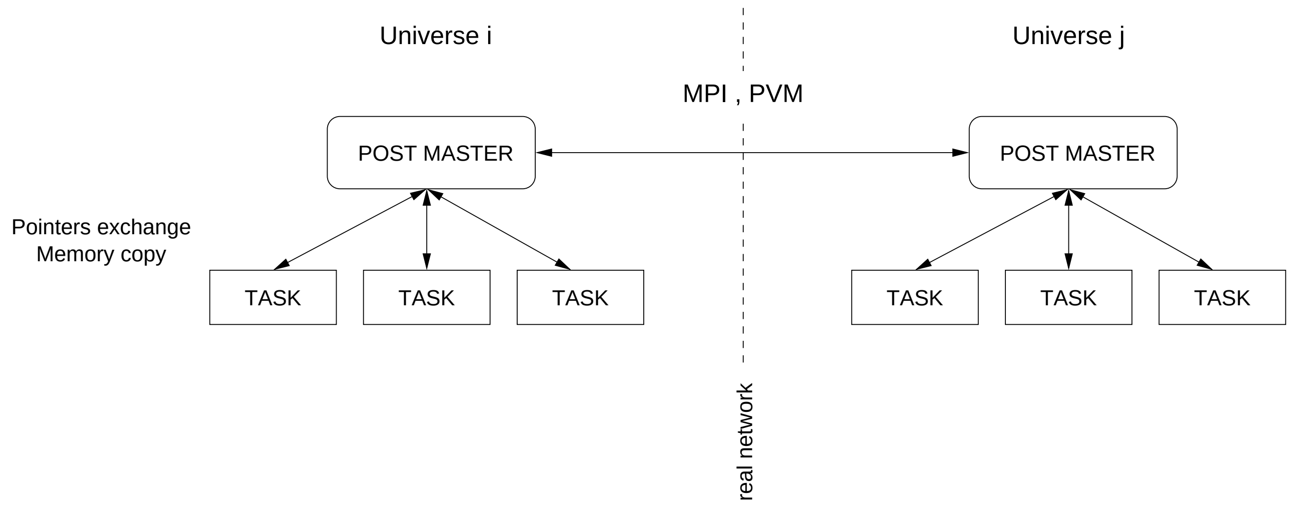


FIGURE 7. Message passing inside / between universes.

A specific message passing interface is derived from the base class (“mt” stands for multithread):

```
class MP_MT : public MESSAGE_PASSING
{
  public :
    virtual void send_message(void*);
    virtual void recv_message(void*);
};

DECLARE_OBJECT(MESSAGE_PASSING,MP_MT,mt);
```

Both methods are used to code send and receive operations described in the previous sections. During the first message passing call, a test is done to check wether the post-master is initialized or not.

End users and developers of ZéBuLoN have then only to specify that the message passing library is “mt” instead of “mpi” or “pvm”. The dynamic object factory will then create the required interface.

These functions are today implemented in ZéBuLoN using a plugin which is dynamically loaded at run time. It shows that it is completely independant of the main code and that it can be ignored by everyone but the developer who made it.

5. CONCLUSION

We presented in this paper the general parallel architecture of ZéBuLoN, an innovative finite element code developed following an object oriented framework, and using C++ as the support language. The parallel mode of operation relies on the Single Program Multiple Data paradigm, but uses a specific task called the supervisor to watch the computation, and also to serve specific requests such as file input/output.

In a second part, we presented a new technique trying to take into account a cluster of SMP machines while remaining in the virtual machine framework. We introduced a new scheduling algorithm which is based on a cooperative time sharing approach in order to avoid any useless time consumed by an operating system managing a high number of processes. This new feature is hidden for the programmer which still develops in the single program multiple data framework. Another advantage of this approach is that it allows any combination of sequential-parallel computations (where all tasks are chained and executed in a sequential order) and full-parallel computations (where all tasks are run simultaneously).

References

[1] J. Besson and R. Foerch, Large scale object-oriented finite element code design. *Comput. Methods Appl. Mech. Engrg.* **142** (1997) 165–187.

[2] F.-X. Roux and C. Farhat, Implicit parallel processing in structural mechanics. *Computational Mechanics Advances* **2** (1994) 1–124.

[3] F. Feyel, *Application du calcul parallèle aux modèles à grand nombre de variables internes.* Ph.D. thesis, École Nationale Supérieure des Mines de Paris (1998).

[4] E. Gamma, R. Helm, R. Johnson and J. Vlissides, *Design Patterns, Elements of Reusable Object-Oriented Software.* Addison-Wesley (1995).

[5] IEEE/ANSI Std. *Portable Operating System Interface (POSIX)–Part 1: System Application: Program Interface (API) [C Language].* IEEE/ANSI Std (1996).