Abstract – This paper presents DPC++, an extension of C++ dedicated to object-oriented parallel and distributed programming on clusters of SMPs. DPC++ follows the active object model and provides language constructs for the exploration of both fine- and coarse-grain parallelism. The main goal of the language is to encourage intuitivity in concurrent programming by means of implicit parallelism. We present an overview of its main characteristics and describe the implementation of a parallel algorithm using the proposed extensions.

Keywords: Active objects, Cluster computing, Implicit concurrency

1 Introduction

Cluster computing is currently a common practice in the field of parallel processing around the world, thus it is desirable that the use of such computing platforms be accessible to an as wide as possible group of users. This motivation has led, in the last years, to the introduction of concurrency in ordinary programming languages as a means to smooth the edges between sequential and parallel programming.

Nowadays there are a number of parallel languages which resulted from the introduction of parallel extensions to existing languages, especially C++ [5, 8] and, more recently, Java [6, 2].

In our point of view, an adequate parallel programming language for cluster computing should present some important characteristics:

Simplicity Experience shows that it is difficult to introduce and establish new designs in terms of computer programming, this is why most parallel languages are actually extensions of C++ or Java. The same can be applied to introducing new language elements such as modifiers or operators, so a parallel language should try not to rely on too many new and/or complex language constructs;

Expressivity It is well understood that suiting a language or programming model to every kind of problem is a hard or even impossible task, but it is still crucial that a parallel language provide resources flexible enough to be used directly or composed into more elaborate structures, especially in the case of synchronisation;

Efficiency Since performance is the primary motivation for doing parallel computing, a programming language should be able to efficiently explore the available computational power of the underlying parallel machine. This includes providing suitable mechanisms to explore all levels from fine- to coarse-grain concurrency, and also design language constructs that can be efficiently adapted to shared-memory and message-passing environments. This is especially important for workstation clusters, where both environments are typically present.
Intuitivity Intuitive constructs collaborate for the design of algorithms and better understanding of a language. We believe that, in parallel programming, intuitive mechanisms are closely related to implicit parallelism.

We are convinced that the previous characteristics are effectively achieved by current parallel programming languages. Our work is then motivated by our understanding of intuitivity. Most of today’s parallel languages offer explicit mechanisms to achieve concurrency (e.g., explicit creation of threads), and this can be understood as intuitive since sequential programming is traditionally the “natural” approach. But one can still choose the point of view that concurrency is present in the real world, and as such it should be “naturally” applied in computer programming (Ian East exposes similar ideas [4]).

While this discussion lies on a more philosophical area, we were convinced that it should be worth to have a parallel programming language that encourages implicit concurrency and still offers the other mentioned characteristics. This has led to the current design of DPC++ — Distributed Processing in C++\(^1\). In this paper we present an overall description of DPC++ and illustrate the use of its language constructs in the design and implementation of a parallel algorithm.

The next section presents an overview of current parallel programming languages and their relation with object-oriented programming; DPC++ and its implicit concurrency model are then presented in Section 3; Section 4 describes the implementation of a parallel application with the proposed syntax and, finally, Section 5 brings our conclusions and perspectives.

---

\(^1\)Not to be confused with Data Parallel C++ [5]; unfortunately this coincidence has only been noticed after a rather stable stage of the project, which discouraged us to change the abbreviation.

2 Object-oriented parallel programming

Parallel programming languages can be divided in three distinct classes [5], according to the programming model they follow and the relationship between objects and concurrency. The first class, or task parallel model, corresponds to the classical model where parallelism is achieved by running multiple tasks (processes) that interoperate through message passing, and is usually not linked with objects. Examples of languages within this class are CC++ and Presto. The second class, or data parallel model, is used to implicitly associate activities to distributed data. In general, a call to a parallel method of an object will be executed over a distributed collection of data. Languages like pC++ and C++* follow this model. In the last class, corresponding to the active object model, parallelism is generated by the creation of objects on distinct computational nodes, communicating through remote method invocation. An active object in this case is, in opposition to a passive object, an object that is associated with an independent thread of execution. UC++ and C++// are examples of languages that follow this model.

All the mentioned languages are based on C++, and many more still exist. It is however mandatory to mention the importance of Java in this parallel programming scenario. Java includes a rather complete support for parallel programming with the explicit creation of threads and provision of synchronisation by monitors. Additionally, the language includes support for distributed computing through the Remote Method Invocation API. Within the three categories, Java could be classified as a task-parallel language. Extensions towards the active object model can be found in Parallel Java [6] and Java// [2].

What can be noticed is that fine- and coarse-grain parallelism are usually not present within the same language, and concurrency must frequently be expressed explicitly. Our research is intended to experiment a different point of
view in terms of concurrent programming, as presented next.

3 The DPC++ language

The DPC++ project started in 1993 [3] with a main focus on distributed computing (hence the name Distributed Processing in C++). With the evolution and establishment of cluster computing, the original model of distributed objects proposed for the language has been restructured and extended to explore fine-grain parallelism.

DPC++ follows the active object model. Objects can be instantiated on remote nodes and communicate through remote method invocations. The whole distribution mechanism of DPC++ is completely transparent to the application level; the user only has to declare which classes should produce active objects.

3.1 Language level

The achievement of concurrency within DPC++ is done by means of syntactic and semantic extensions to C++. In order to follow the desired characteristics of simplicity and intuitivity, only a few new keywords are introduced, being most extensions implemented semantically.

3.1.1 Declaration and creation of active objects

The decision upon which objects should become active is taken by the programmer by declaring the classes of these objects as *dclasses*:

```cpp
dclass Person
{
    // regular attributes and methods
};
```

Any instance of a class declared with the *dclass* keyword becomes an active object. The creation of such objects is done normally, as in C++:

```cpp
Person p0();
Person *p1;
...  
p1 = new Person();
```

Concurrency in DPC++ is achieved when methods are called, in two distinct ways: By asynchronous method calls and by multiple method handling within an active object.

Method calls are either synchronous or asynchronous, depending on the return type declared in the class body. *Void* methods are always called asynchronously; methods that return any kind of information are called synchronously. In the previous example, both methods `walk()` and `whistle()` would be called asynchronously, while `rememberPhoneNumber()` would force the caller to wait until execution is completed, since it must return some information. This is our first step towards implicit parallelism.

Objects created from regular classes are always instantiated locally. As usual in active object-based languages, such objects make part of, and are limited to, the scope of one individual active object.

The internal definition of a *dclass* follows the same rules as in traditional C++. Thus, simple applications can be easily ported from C++ to DPC++ by simply changing the declaration of the appropriate classes.

An important remark on DPC++ active objects is related to the term “active”: Although it follows the active object model, an active object in DPC++ does not have an independent, internal flow of execution as presented, for instance, by C++// (method `Live()`). Every activity within an active object in DPC++ must be originated by a remote method call.

3.1.2 Remote method calls

Methods of active objects in DPC++ are declared and called in the same way as in regular C++:

```cpp
dclass Person
{
    Person(char *name);
    void walk();
    void whistle();
    int rememberPhoneNumber(char *name);
}
...  
Person p("John");  
p.walk();
```

Concurrency in DPC++ is achieved when methods are called, in two distinct ways: By asynchronous method calls and by multiple method handling within an active object.

Method calls are either synchronous or asynchronous, depending on the return type declared in the class body. *Void* methods are always called asynchronously; methods that return any kind of information are called synchronously. In the previous example, both methods `walk()` and `whistle()` would be called asynchronously, while `rememberPhoneNumber()` would force the caller to wait until execution is completed, since it must return some information. This is our first step towards implicit parallelism.
If an active object receives two simultaneous method calls (e.g. from two distinct objects, or by consecutive calls to void methods), they are handled concurrently by two independent execution flows (threads). In this way, the methods of an active object are potentially concurrent in relation to each other.

These two features provide the basis for the achievement of concurrency in DPC++. The example given, though merely illustrative, tries to follow the idea of intuitivity by defining “tasks” that can be naturally understood as potentially concurrent. Our intention with DPC++ is that such a vision be also followed in the design of real parallel applications. In Section 4 these ideas are illustrated in the implementation of a more practical algorithm.

### 3.1.3 Additional synchronisation: monitors

It has been shown that synchronisation between distinct active objects is ruled by different semantics in remote method calls. But one might still run into race conditions within an active object, due to inter-method concurrency. As a solution to this problem, we add the behaviour of monitors to objects in DPC++.

In order to guarantee mutual exclusion, methods in DPC++ can be prefixed with the modifier `atomic`. This forces, on every method call, the acquire and release of a lock implicitly associated to each object.

In addition, it is possible to declare *conditional variables* as regular attributes of a class, which is done with the keyword `cond`. Conditional variables provide the primitives `wait()` and `signal()`.

The next example shows a piece of the implementation of a Bounded Buffer using these constructs:

```cpp
class BoundedBuffer
{
    ... cond empty, full;
    ...
    atomic void produce(Data item) {
        while (count == MAXITEMS) { // do some internal management here...
            full.wait();
            empty.signal();
        }
    }
    ... }
```

Notice that BoundedBuffer is not declared as a `dclass`; monitor synchronisation can be applied in regular objects as well.

We have chosen to offer the mechanism of monitors because it can be suitably incorporated in the logical structure of a class, as seen on the example, and also due to the high expressivity this mechanism provides. The above language constructs can be easily used to build other synchronisation mechanisms such as semaphores and mutexes. The implementation described in Section 4 provides additional examples on the use of these features.

### 3.2 Operational level

The operational level of DPC++ is implemented on top of the *DECK* environment [1]. DECK provides many functionalities useful to parallel and distributed programming, such as threads and mail boxes, running in an SPMD approach.

The main feature in the operational level of DPC++ is the *cluster*. The cluster is in practice a DECK application (i.e. a single process image started on each of the nodes) that provides objects with the capacity of sending and receiving messages to and from remote nodes.

The traditional approach in the active object model is to map each active object to a dedicated process, which then provides the “active” part. In DPC++, the active part of several objects is concentrated in a single entity called the *dispatcher*. The dispatcher is the initial thread of a cluster, being dedicated to the reception of remote method invocations.

Upon receipt of a such a message, the dispatcher immediately spawns a new thread of control to treat it and goes back to listening the channel. The spawned thread unpacks the message and performs the desired operation, which is either the creation of a new active object or a method invocation. In the first case, a local
instance of the referred class is created and the thread returns a reference to the caller. Multiple active objects are thus allowed to exist simultaneously within the cluster’s address space.

The returned reference is used in the future in order to locate a specific active object. This is the case of the second kind of request, where the caller includes the reference additionally to the method identification and the needed parameters. In case the method has a return value, the thread is also responsible for returning it to the caller when the method invocation is completed. Figure 1 shows the internal view of a cluster handling multiple invocations in the DPC++ model.

3.3 Critical analysis

Many parallel languages follow the active object model due to the adequate match that can be achieved between object orientation and distributed computing. This was also the motivation for the original DPC++ model. With the addition of fine-grain concurrency, it became suitable for hybrid environments such as clusters of SMPs. Moreover, we believe that the implicit parallelism of DPC++ encourages a more natural approach to concurrent programming.

Two levels of concurrency can be noticed within a DPC++ cluster: a first level where multiple methods of a single object can be executed in parallel and a second level where concurrency is achieved between multiple active objects. The immediate benefits from this two-level concurrency are the actual parallelization of tasks in multiprocessor architectures and the overlapping of communication and computation through the use of multiple threads of control.

One might see an inconsistency in our philosophy of implicit parallelism from the fact that active objects are defined explicitly. In this case we must admit that it is more practical (and probably more efficient) to let the user design the distribution of an application rather than to delegate this task to a complex program analysis implemented within the compiler.

The implementation described in the next section furnishes a more practical view of the features we have proposed.

4 Application

Additionally to short examples as the bounded buffer shown before, we have designed and implemented a real parallel application in order to validate the DPC++ model. The application consists on a parallel implementation of the Mandelbrot fractal generation algorithm [7].

The application is based on two classes, MandelMaster, which delivers regions of the fractal to be calculated and organises those already done, in order to compose the final picture, and MandelWorker, which performs the calculation of a single region. Figure 2 shows a functional view of the application.

The declaration of the class MandelMaster, which concentrates the use of the proposed features, is shown below:

dclass MandelMaster
{
   private:
      ...
}
cond allRegionsReceived;
atomic void incRegionsIn();

public:
MandelMaster(...);
atomic int getNewRegion();
void regionDone(...);
atomic void saveFinalPicture(...);

During execution, the Worker objects obtain a region to be calculated by calling the method getNewRegion(). When the calculation is done, the Master is signaled through method regionDone() and the Worker is able to request another region. When all the regions have been delivered, the Master returns a special flag and the calling Worker exits.

Concurrency is originated within the Master by simultaneous calls to method regionDone(). This method reads a file (generated by the Worker) with the results calculated to the corresponding region, and fills the appropriate part of the final picture. In this way, a higher number of Worker objects tends to increase concurrency within the Master, and consequently performance. Naturally, concurrency is also achieved between distinct (distributed) Workers.

The proposed synchronisation mechanism plays a very important role in the application, since it controls the correct delivery and reception of regions, and signals the end of computation. Updates on the number of delivered and received regions are synchronised by declaring methods getNewRegion() and incRegionsIn() atomic (this last method is internally called by regionDone()).

To correctly detect the end of the application, we use a conditional variable allRegionsReceived. This variable is signaled when the last region is received. Method saveFinalPicture() is intended to be called by the main application, after having created and started the Master and the Workers. Its implementation first checks whether all regions have been received; if not, it performs a wait() on the conditional variable.

This design reveals a suitable use of DPC++ features in the modelling of a parallel application. Concurrency has been introduced in a simple and intuitive way, keeping the algorithm clear, and synchronisation is also achieved clearly.

This application has been implemented on a 4-node Dual Pentium cluster connected by Fast Ethernet, running Linux 2.2.10. The obtained results are shown in Figure 3. The graph on the left side shows the speed-up obtained with distribution alone (i.e. between the active objects), while the other shows only the gain from internal concurrency in the Master object. The different curves correspond to variations in the size of the regions. As expected, larger regions result in coarser-grained parallelism, which provides better results in terms of distribution. On the other hand, fine-grain concurrency is better suited to smaller regions. The balance between them is better achieved with 6 Worker objects and regions in the range of 500 to 1000 pixels.

5 Conclusions and perspectives

The main intention in the DPC++ extension to C++ is to encourage a more natural approach to parallel programming with objects. We try to achieve that by integrating a concise and intuitive language level with efficient support in the operational level. Concurrency is expressed implicitly by asynchronous method calls and multiple method handling inside an active object. Synchronisation and data coherency are obtained from remote method calls and the use of monitors.

Our conclusions from the design and implementation of parallel algorithms with DPC++ is that it does achieve its main objective. We have shown the implementation of a Mandelbrot fractal generation algorithm where DPC++ language constructs are used in a simple and intuitive manner. The obtained practical results were considered very good; even with a rather heavy use of synchronisation, the application was able to reach a total speed-up in the range of 6 in a 4-node cluster, with more than 70% of efficiency.
We are currently working on the implementation of additional applications (e.g. NAS benchmarks) with DPC++, as well as extending and fine-tuning the DECK environment for Myrinet and SCI networks. Our expectation is to obtain additional feedback on the use of DPC++ by introducing it in academic activities of graduate and under-graduate studies.

Acknowledgements

This work has been partially supported by grants from the Brazilian entities CNPq and CAPES.

References


