SharedOnRead Optimization in Parallel Object-Oriented Programming

Denis Caromel[1], Eric Noulard[2], and David Sagnol[1]

[1] OASIS - Joint Project CNRS / INRIA / University of Nice Sophia Antipolis
INRIA - 2004 route des Lucioles - B.P. 93 - 06902 Valbonne Cedex, France
FirstName.LastName@sophia.inria.fr
[2] Société ADULIS – 3, rue René Cassin – F-91742 Massy Cedex – France
E.Noulard@adulis.fr

Abstract. This paper presents how a specific technique (SharedOnRead) can be used to improve performances of a distributed objects language. We present results for a network of workstations, providing comparisons between MPI and C++// implementations, both on homogeneous and heterogeneous platforms. The results could be applied to other models of distribution, such as RMI.

1 Introduction

This paper presents a mechanism for sharing objects when two processes (active objects [1] in our object-oriented framework) want to access the same object. This mechanism is implemented using the C++// [2, 4] library. In C++//, active object cannot share passive ones, and there is a deep copy of all the parameters of a function call, between 2 active objects (the same behavior occurs in RMI). This can be a bottleneck when active object are implemented using threads.

This article proposes a solution to solve that problem: the SharedOnRead mechanism. A SharedOnRead object implements a well-known mechanism allowing to have both constant semantics and efficiency, whether objects are located in the same address space or not.

We test the SharedOnRead mechanism on basic linear algebra operations commonly used in iterative methods [9]. The key point for efficient parallel implementation of iterative methods are good performance of the distributed sparse matrix/vector product, and distributed dot product since these operations are at the heart of all basic Krylov algorithms [8]. In this context it is crucial to avoid unwanted copies in matrix operations. Explicitly dealt with in MPI-based parallel numerical applications, we want to achieve this in a more implicit and transparent way in an object model.

The next section gives a quick survey of the active object model we are using, presenting the main features of the C++// library. Section 3 introduces the SharedOnRead extension to the model, and section 4 presents some performance results for matrix operation in linear algebra. Section 5 concludes the paper.

2 The active object model

The C++// language was designed and implemented with the aim of importing reusability into parallel and concurrent programming. It does not extend the language syntax, neither requires to modify the C++ compiler, since C++// is implemented as a library and a preprocessor (a Meta-Object Protocol [7]), and uses Nexus [6] library as the transport layer for communications.
The C++// model adopts the following principles:

- heterogeneous model with both passive and active objects (actors, threads),
- active objects which act as sequential processes serving requests (i.e. method invocations) in a centralized and explicit manner by default,
- systematic asynchronous communications towards active objects,
- wait-by-necessity (automatic and transparent futures),
- automatic continuations (a transparent delegation mechanism),
- no shared passive objects (call-by-value between processes),
- centralized and explicit control by default,
- polymorphism between standard objects, active objects, and remote objects.

As shown in Figure 1 (page 3), C++// programs are organized in subsystems: one subsystem contains *one* active object and *several* passive objects. Active objects are active by themselves, with their own thread of control, unlike passive objects that are normal C++ objects. An active object is the only one that can be referenced by objects in other subsystems; there are no shared passive objects. The absence of sharing is enforced by the programming model since all passive objects used as parameters are automatically transmitted by copy (deep copy of objects, alike serialization in Java RMI) when communication occurs towards an active object (another subsystem).

With respect to this paper, a crucial point is where active objects are created. A C++// programmer has several choices to determine the machine (or node) where a new active object will be created: (1) give the machine name, (2) provide an existing active object in order to use the same machine. We do not detail this mechanism here. But central to the issue is that, in both cases, the programmer has two options to create the new active object:(a) in an existing address space, (b) in a new address space. In case (a), several active objects will be able to share the same address space — threads are used to implement active objects. Let us note that even when two active objects in the same address space communicate, passive objects are still transmitted by copy. This potentially time and space consuming strategy is mandatory if we want the program semantics to be constant, whatever the mapping is. However, in some cases, sharing is actually possible, and copying large objects could be avoided.

The SharedOnRead mechanism was defined to make possible such optimization, and to provide a general strategy that keeps the semantics constant when architecture and mapping vary.

3 The SharedOnRead framework

3.1 Strategy

The general idea of the strategy can be stated as follows: upon a communication between two subsystems that are within the same address space, instead of copying a SharedOnRead parameter, we just share it; there is no change to the copy semantics if the two subsystems are not in the same address space (cf Figure 1). While this scheme is quite simple, a mechanism is needed in order to maintain the same semantics that occurs when the two subsystems are mapped in two different address spaces. We can notice that objects will be able to be

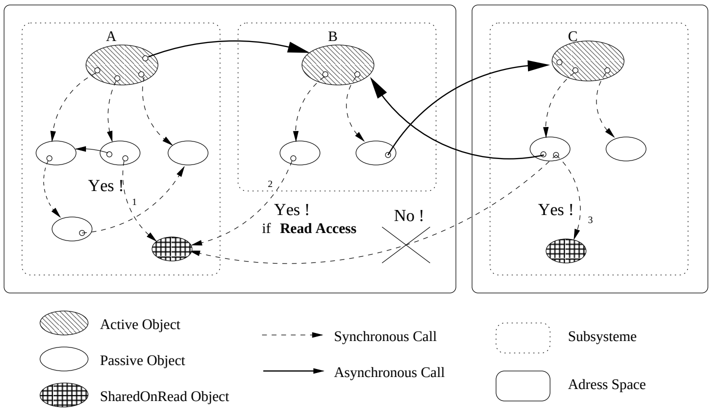

Fig. 1. SharedOnRead objects within several address spaces

shared by several subsystems only as long as they are not modified by one of the subsystems. In order to be effective, the strategy needs to make a distinction between *read* and *write* accesses, and also needs to know when a subsystem *forgets* a SharedOnRead object (the subsystem suppresses his reference to the SharedOnRead object).
The algorithm can be summarized as follows:

- When a SharedOnRead object is used as a communication parameter between two subsystems being in the same address space, the original object is not copied, but instead a new reference is memorized (a counter is incremented within that object).
- A *read* access is freely achieved (from both the subsystem of the owner or another one).
- Upon a *write* access (from both the subsystem of the owner or another one), if the counter value is more than 1, a copy of the object is made. The modification applies to the copy. The counter of the previously existing object is decremented; the counter of the copy is set to 1.
- Upon a *forget* operation, the counter is just decremented. When reaching zero, the object is automatically garbaged.

3.2 Programmer interface and Implementation

As a design decision, we choose to give users the control over which objects should be SharedOnRead and which one should have the standard copy behavior. A SharedOnRead object is an instance of a class that inherits directly or indirectly from the C++// SharedOnRead class (Figure 2, class SharedOnRead). This class provides several member functions:

- *read_ access* and *write_ access* are both used to specify how the data members are accessed by a given method. If *read_ acess* (respectively *write_ access*) is selected for a method, we ensure that, for this function, data members are never (respectively can be) changed. They take a *mid_ type* parameter which is unique for all the member functions in the program; the *mid* function[1] provides that unique identifier for a function name.
- The *access_ declaration* function has to be redefined in order to specify for each public member function its read or write nature: write access is the default behavior. SharedOnRead user has to take care with that and check for each function in his class which ones are leaving the object in the same state and which ones are making modification to the object state.
- Finally, *forget* signals that we do not use the object anymore. This have to be dealt with explicitly because SharedOnRead object can not be aware that they are not referenced anymore.

```
class SharedOnRead : public EC_Reflect {
public:
    virtual void read_access(mid_type);
    virtual void write_access(mid_type);
    virtual void access_declaration();
    virtual void forget();
};
class Block { // A Matrix block
public:
    virtual void reach(int** ai, int** aj, double** a);
    virtual void update(int**& ai, int**& aj, double**& a);
};
class SorBlock: public SharedOnRead, public Block {
public:
    SorBlock();
    virtual void access_declaration() {
       read_access(mid(reach));
       write_access(mid(update));
    }
};
```

Fig. 2. Programming a SharedOnRead block in a matrix example

[1] this function is part of the C++// library

The implementation of the SharedOnRead class is based on the reification mechanism provided by C++//. Inheriting from the class *EC_Reflect* (cf Figure 2, class SharedOnRead), all the SharedOnRead member functions are reified: the functions are not directly executed but go first in a specific *proxy* where all the necessary implementation is defined and achieved (update of the counter, copy when necessary, etc.). After this step, the *proxy* executes the function.

When a C++// programmer wants to get the SharedOnRead optimization, he just has to define a new class that inherits from an existing class and the SharedOnRead one, the only constraint being to redefine the *access_declaration* function. For instance, in the case of a *Block* class used to implement a matrix, a new class *SorBlock* can be defined, inheriting from both *Block* and SharedOnRead (Figure 2). The function *access_declaration* specifies that *update* can make modification to the block, while *reach* cannot. This point is delicate since we can not prevent a user from declaring a function with read access that modify the object.

3.3 Simple example

The purpose of this section is to present a simple application using the SharedOnRead mechanism. The *Buffer* class (Figure 3) has two virtual functions: one to get a particular element (the *Element* object can be any kind of standard C++ object) and one to modify an element. At the beginning of the program,

```
class Buffer : public SharedOnRead {
public:
    virtual Element* getElem(int i); // return element i
    virtual void updateElem(int i, Element* e); // update element i
    virtual void access_declaration() {
         read_access(mid(getElem));
         write_access(mid(updateElem));
    }
};
```

Fig. 3. A SharedOnRead buffer

this buffer is created by an active object (named e.g. *active1*) and is filled with different elements. At a moment of the calculation, another active object (named e.g. *active2*) wants to acquire a reference to this buffer object. In normal C++// (without SharedOnRead), it would have acquired a deep copy of this buffer, but as *Buffer* inherits from SharedOnRead (and since *active1* and *active2* belong to the same address space) it acquires a direct pointer to the buffer. As long as the two active objects only use *getElem* functions, they can share the buffer. But if one makes an *updateElem*, it automatically acquires a copy before the function is executed. After that, *active1* and *active2* objects are working on two different buffers.

This example demonstrates that the main use of SharedOnRead objects is when we do not want to replicate huge objects: several active objects can share the same SharedOnRead object. However, for really small objects, it can be better to not use SharedOnRead mechanism. Suppose you want to perform a lot of function calls to such a small object: because of reification overhead (around 5 times the cost of a virtual call) it can be more interesting to send a copy and after that, to work on this copy. The choice depends on your primary goal, reduce memory usage or increase speed.

4 Benchmark application

4.1 Parallel linear algebra

We will apply our SharedOnRead mechanism to the implementation of efficient matrix/matrix operations for (block) iterative methods. As defined in [8] we may focus on a reduced set of operations:

- full SAXPY, $Y := \alpha X + \beta Y$ with $Y, X \in \mathbb{R}^{n\times p}, \alpha, \beta \in \mathbb{R}$.
- full or sparse matrix product $Y := \alpha A \cdot X + \beta Y$ with $Y \in \mathbb{R}^{m\times p}, A \in \mathbb{R}^{m\times n}, X \in \mathbb{R}^{n\times p} \alpha, \beta \in \mathbb{R}$

Parallel numerical linear algebra is concerned with data distribution of the matrix arguments in the above operations. In our case we will only consider a block distribution scheme of matrices which is widely used [3] and well suited for our applications. With these kind of distributions, we have to do full matrix saxpy operation and full or sparse matrix product.

These operations are implemented as a method of a class `Matrix` representing a block partitioned matrix. The methods are:

- `Matrix::scal(double alpha, Matrix* B, double beta)`
  This method performs the matrix saxpy operation $this = \beta \cdot this + \alpha \cdot B$.
- `Matrix::axpy(double alpha,Matrix* A,Matrix* X,int mm)`
  This method performs the matrix operation $this = \alpha \cdot A \times X + this$, and $this = \alpha \cdot A \times X$ if `mm==1`.

Here the distributed objects are the blocks of the matrix which may be called CSC since they are Compressed Sparse Column (eventually sparse) matrices.

4.2 From sequential to parallel matrices

A sequential matrix contains a list of CSC objects, each of these objects holding a *Block*. This *Block* is responsible for allocating all the arrays representing the matrix. If we want to parallelize these classes using C++//, we only have to redefine the *Matrix* constructors. These constructors create the distributed CSC objects and associate (or not) the *SorBlock* depending if we want to use the SharedOnRead mechanism or not. A distributed CSC object (*CSC_ll* object) can be created by inheriting from CSC and the C++// class *Process*. This class implements all the mechanism we need to create active objects (remote creation, automatic marshalling, ...). That's the only part of the sequential code that we have to modify. All the functions presented above (scal, axpy, ...) come from the sequential classes.

Figure 4 presents the sequential version for a full axpy function. Because of polymorphism, the two A and X variables can be *CSC_ll* objects. In this case, the *block()* function returns a *SorBlock* object (if SharedOnRead is used): we get direct access to the object implementing the matrix block. It is sufficient to access *bl1* and *bl2* in read mode (if they are located in the same address space, we use them directly, otherwise we get a local copy). On the other hand, the *mine* variable, which represents the local *Block* of the *CSC* object, has to be modify so must be "declared" in write mode.

```
void CSC::axpy(double alpha, CSC* A,
               CSC* X, double beta) {
   Block* bl1 = A->block();
   Block* bl2 = X->block();
   Block* mine = block();
   bl1->reach(&tia, &tja, &ta);
   bl2->reach(&tix, &tjx, &tx);
   mine->update(&tiy,&tjy,&ty);
   ...
}
```

Fig. 4. Sequential full CSC_Block product

The MPI implementation requires to redefine all the functions to take into account that we do not make the same operations depending the processor we are executing on: MPI is very intrusive. This means that you have to add a lot of MPI calls in the initial sequential code in order to obtain the parallel version. Moreover, this makes it difficult to go back and forth from sequential to parallel version for the application developer. Furthermore, MPI is a message passing library thus parallelism is explicit, and the programmer has to directly deal with distribution and communication. Another advantage of the C++//version is that we do not require to use a temporary matrix for reduction step. In the examples in the next section, this temporary matrix can take up to 7MB.

4.3 Performances

Even if C++// is not a pure data parallel language, we choose to compare our results with MPI [5]. MPI is a well known and efficient standard library to program parallel applications. The following tests were performed on a network of 4 Solaris Ultra 1 with 128MB of memory and a 10Mb Ethernet link. The MPI tests use Lam library, now developed at the university of Notre Dame [2].

Figure 5 presents our result for a `scal` (cf 4.1) calculation. Matrices used during these tests were rectangular matrices with 90449 rows and a variable number of columns. The use of SharedOnRead objects demonstrates a speed-up

[2] http://www.mpi.nd.edu/lam/

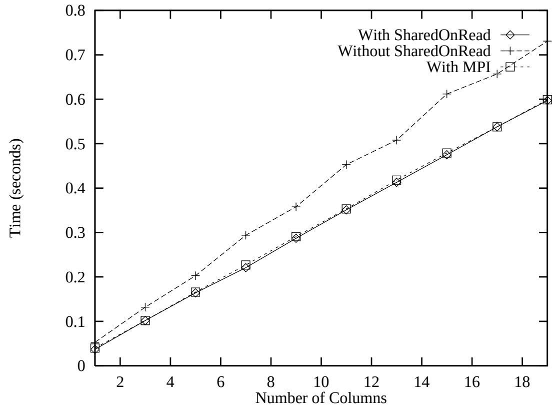

Fig. 5. SAXPY with 4 computers

between 20 and 25% compared to the normal C++// version. When compared with the MPI version, we cannot distinguish any difference between C++// with SharedOnRead and MPI. One important point to notice is the fact that the normal C++// version (without SharedOnRead objects) presents more and more overhead when the matrix size is growing. The main reason being that this version requires many communications between the different active objects that are located in different processes. For the SharedOnRead version, the two blocks (two active objects) are mapped within the same process, so we avoid communication and copy.

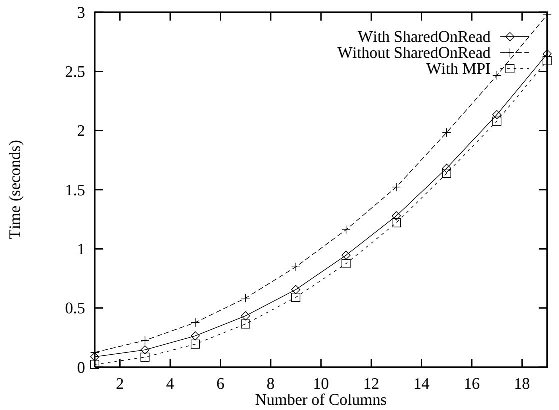

Fig. 6. Full matrix product with 4 computers

Figure 6 presents results for a full matrix product. Again, the normal version is 30% slower than MPI. Between the C++// version and the MPI one, the overhead is constant. The SharedOnRead version and the MPI one do not present the same result in this test because this operation requires communication during the reduction phase. The C++// solution sums the local matrix in a sequential way because it reuses the sequential code.

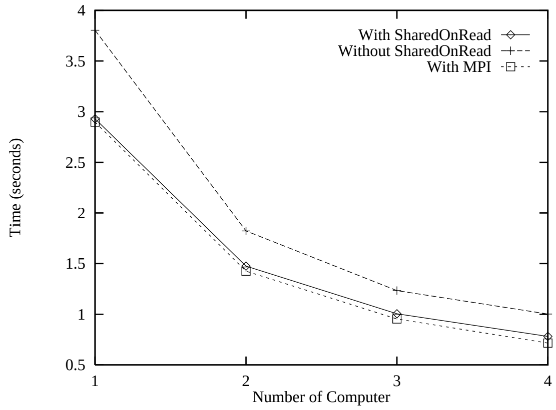

Fig. 7. Full matrix product with (90449x10) Matrix

Figure 7 presents the speed-up obtained with 4 computers for the 3 different configurations. All calculations on full matrices are perfectly scalable ; for 4 machines, the speed-up is around 3.9. We can observe that the overhead of the normal C++// version is constant whatever the number of computers we use.

Finally, Figure 8 presents results for a sparse matrix product. A first point to notice in such a case is that we had to rewrite the *add* function of the matrix. The reduction being critical in this benchmark, it was important to achieve it in parallel.

The second important aspect of this benchmark deals with the platform architecture. All the previous tests were made on homogeneous computers: the same CPU, at the same frequency, with the same amount of memory. The current test was performed on both homogeneous and heterogeneous platforms.

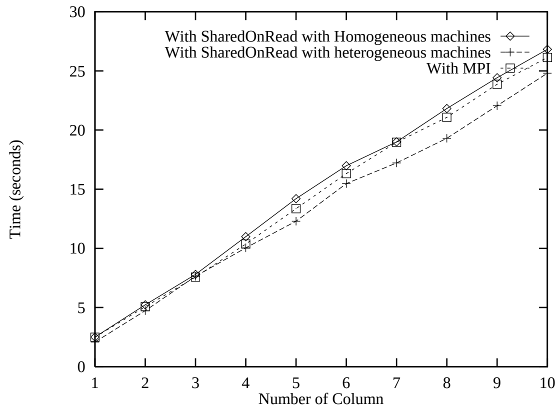

Fig. 8. Sparse matrix product with 4 computers

Two Ultra 1 were replaced with two Ultra 2 with 192 MB of memory. Within this new architecture, the MPI program has the same result we had before. In the C++// case, the benchmark demonstrates that the object-oriented version is actually more efficient than the MPI one. While MPI is subject to synchronization barriers, the C++// advantage comes from the asynchronous model. In that case, the reduction of the two local matrices of the fastest computers can start even if the computation on the slowest computers is not finished.

5 Conclusion

This paper presented how a SharedOnRead mechanism can help a distributed object-oriented language to be competitive with MPI. In some cases, we can even overcome MPI thanks to C++// asynchronism and wait-by-necessity mechanisms. When comparing the MPI implementation with the C++// one, it also appears that the later allows us to avoid writing large amount of code to obtain a parallel version.

This technique of passive object sharing could be implemented in any object-oriented language with distributed objects that do not have a mechanism to share passive object upon copy semantics. It allows these object environments to keep their advantage of simplicity and reusability, and also improving their performances.

References

1. Gul Agha. *A Model of Concurrent Computation in Distributed Systems.* MIT Press, 1986.
2. Denis Caromel, Fabrice Belloncle, and Yves Roudier. The C++// Language. In *Parallel Programming Using C++*, pages 257–296. MIT Press, 1996. ISBN 0-262-73118-5.
3. J. Choi, J. Dongarra, S. Ostrouchov, A. Petitet, D. Walker, and R.C. Whaley. A proposal for a set of parallel basic linear algebra subprograms. Technical Report Lapack Working Note 100, May 1995.
4. Caromel D., Dzwig P., Kauffman R., Liddell H., McEwan A., Mussi P., Poole J., Rigg M., and Winder R. Ec++ – europa parallel c++ : A draft definition. In *Proceedings of High-Performance Computing and Networking (HPCN'96)*, volume 1067, pages 848–857. Springer, Lectures Notes in Computer Science (LNCS), April 1996. ISBN 3-540-61142-8.
5. Message Passing Interface Forum. MPI: A message passing interface standard. Technical Report CS-94-230, University of Tennessee, Knoxville, TN, March 1994.
6. Ian Foster, Carl Kesselman, and Steve Tuecke. The nexus approach to integrating multithreading and communication. *Journal of Parallel and Distributed Computing*, 37:70–82, 1996.
7. G. Kiczales, J. des Rivières, and D.G. Bobrow. *The Art of the Metaobject Protocol.* MIT Press, 1991.
8. Eric Noulard, Nahid Emad, and Laurence Flandrin. Calcul numérique parallèle et technologies objet. Technical Report Rapport PRISM 1998/003, ADULIS/PRiSM, Juillet 1997. Révision du 30/01/98.
9. Yousef Saad. *Iterative Methods for Sparse Linear Systems.* PWS Publishing Company, New York, 1996.