Experience of Using mpC to Improve Performance of CORBA-Based Distributed Applications on Heterogeneous Networks

Alexey Lastovetsky
University College Dublin, Ireland

Abstract *The paper demonstrates that performance of CORBA-based distributed programs can be easily and drastically improved with the mpC parallel programming system. It presents a typical distributed application, parallel mpC implementation of its remote computationally intensive operations and performance results on a network of workstations.*

Keywords: CORBA, distributed systems, parallel processing, mpC, heterogeneous networks

1. Introduction

CORBA [1] is widely used to develop and integrate highly complex distributed technical applications in industries as diverse as health care, telecommunications, banking, and manufacturing. CORBA is supported on almost every combination of hardware and operating system in existence, available from a large number of vendors, and supports a large number of programming languages. CORBA-based distributed applications support the "client-server" programming paradigm. A typical CORBA-based application server provides a number of operations that can be invoked by remote clients.

One of the most important qualities of the service, provided by the remote server, is the execution time of the remote operations. The total time of execution of a remote operation includes the time of communication between client and server and the time of computation on the server side. In case of computationally intensive operations, maximal effect would come from acceleration of computations on the server-side hardware, which is typically a heterogeneous network of diverse computers. There are two ways typically used to speedup execution of remote operations on a network of computers. The first way is to balance the workload of available computers. This means that the CORBA implementation tries to start up the server providing the requested operation on the computer that is the fastest at the moment of the receipt of the request. The second way is a multithreaded implementation of the remote operation and its execution on a shared-memory multiprocessor computer if the latter is available.

One more way is a parallel implementation of the remote operation and its execution on the network of computers treated as a distributed-memory parallel computer system. Theoretically, this can provide much more essential speedup independent on availability of a multiprocessor computer. Practically, this way is not used due to two main reasons. The first one is the lack of experience of integration of distributed-memory parallel computing into CORBA-based distributed applications. The second and the most important reason is that technologies and tools for parallel computing on heterogeneous networks of computers only take first steps and are not as mature and widespread as those for homogeneous multiprocessors.

The paper demonstrates that computationally intensive remote operations in CORBA-based distributed applications can be easily and drastically accelerated with help of tools for parallel computing on heterogeneous clusters. Orbix [2] was used as a particular implementation of CORBA, and the mpC language [3,4] was used for parallel implementation of such remote operations.

The paper is organized as follows. Section 2 describes a sample CORBA-based distributed application. Section 3 briefly introduces the mpC language and outlines the mpC implementation of one of the computationally intensive operations provided by the application server. Section 4 presents experimental results. Section 5 concludes the paper.

2. Chain of supermarkets distributed application

The following sample distributed application is used in the paper. Let there be a chain of supermarkets. Let each cash register send information about every basket of items purchased by a customer to the central information center (CIC), where the information is stored. A single basket of items is stored in the form of a file record, so that a single file contains data about a fixed number of baskets. All those files together accumulate the full information about the structure of customer baskets during some period of time. This data dump is used for extraction of diverse useful information, and any supermarket can require the CIC for one or another information. That service is implemented by a CORBA-based application server providing a set of corresponding remote operations. In particular, a supermarket can require the CIC for recommendation on optimal distribution of different items over a given number of sections. A fragment of the CORBA IDL specification of the full service relevant to this particular request is depicted in Figure 1.

A client invokes operation *BasketOfItems* to add a basket to the server's data store. A basket is just a sequence of codes of purchased items. To get an optimal distribution of items over a number of sections, the client invokes the operation *getDistribution* passing this number as an input parameter. The operation returns a sequence of sequences of items representing the requested distribution. To initialize a session with the remote server the client invokes operation *Hello*. To finalize the session, it invokes operation *Bye*.

Operation *getDistribution* is computationally intensive and implements the following algorithm. First, it reads all files one by one and forms a vector S and a matrix P representing mappings $S:I->N$ and $P:IxI->N$ respectively, where I is a set of all items, N is a set of positive integers, $S(i)$ is the total number of baskets containing item i, and $P(i,j)$ is the total number of baskets containing both item i and item j. Then it uses the mappings to divide set I into M non-intersecting subsets $I_0,\ldots, I_{M-1}$, where M is the number of sections. M most frequently bought items head separate sections. The rest items are distributed over the sections in the following way:

```
delete i_0,…,i_M-1 from I
for(k=0; I is not empty; k=(k+1)%M)
{
  find i: P(i,i_k)==max{P(I,i_k)}
  add i to I_k
  delete i from I
}
```

Here $i_0,\ldots,i_{M-1}$ are leaders of the sections, and the operation % computes the modulus of two integers.

Intuitively, the algorithm tries to make each section have at least one very popular item surrounded by the items that most often accompany this popular one. It is assumed that such a distribution will stimulate customers to buy items of secondary necessity and reduce the total shopping duration. Due to the data store is very large and input/output operations are relatively slow, the execution time of the algorithm is practically equal to the time of computation of mappings S and P. Obviously, more accurate distribution can be obtained by taking into account the popularity of triplets of items and so on, not only that of single items and their pairs. But this leads to much more

```
typedef short Item;
typedef sequence<Item> Basket;
typedef sequence<Item> Section;
typedef sequence<Section> Distribution;
interface  central_office {
  void BasketOfItems(in Basket b);
  Distribution getDistribution(in short number_of_sections);
  void Hello();
  void Bye();
  ...
};
```

Figure 1. CORBA IDL specification of the chain of supermarkets application server.

computationally intensive and slower algorithms. Thus, although the presented algorithm is computationally intensive, it performs the minimal volume of computation that is necessary to solve the problem.

Both client and server parts of the described distributed application were originally implemented in Orbix 3 C++ programming system.

3. Parallel implementation of remote operations in mpC

The traditional purely serial implementation of the presented application server causes the operation *getDistribution* to be very slow from the client's point of view. At the same time, the application server normally runs on a network of computers whose total performance is very high. Therefore, a parallel implementation of the operation *getDistribution* that enables it to use effectively all available performance potential could essentially accelerate the operation.

Let the data store consist of a big number, F, of files each of which containing B basket records. Let the network of computers to execute the operation be heterogeneous one possibly running some other applications as well. Under those assumptions, an obvious parallel modification of the original serial algorithm consists in parallel computing mappings S and P by all available processors. Namely, *i*-th processor computes mappings S_i and P_i by processing a subset of the full set of files. The number of files in subsets processed by different processors is proportional to the relative performance of the processors. The resulting mapping $S(P)$ is obtained by means of the merge of mappings S_i (P_i). Mathematically, the merge is nothing more than computation of the sum of all vectors S_i (matrices P_i). Thus, the parallel algorithm looks as depicted in Figure 2.

The parallel algorithm is hard to be implemented in a portable form using traditional parallel tools like PVM [5] or MPI [5], which are oriented on dedicated homogeneous distributed-memory parallel systems, but can be easily implemented in mpC – a language specially designed for parallel computing on common heterogeneous networks of computers. The mpC language is a strict extension of the ANSI C language; therefore the corresponding parallel mpC code is obtained by very slim modification of the original serial C code used in the Orbix C++ implementation of the application server.

The core of the mpC code is a description of such features of the implemented parallel algorithm that influence the running time. The description looks like a description of an abstract heterogeneous network executing the algorithm. From the mpC language's point of view, that description defines a parameterized type of abstract networks and is called the network type definition. Thus, the key fragments of the parallel mpC code looks as depicted in Figure 3.

```
detect the total number of available processors
detect relative performances p_i of the processors
compute number n_i of files processed by i-th processor
compute mappings S_i and P_i in parallel
merge S_i into resulting mapping S and P_i into P
compute the distribution of I over sections based on S and P
```

Figure 2. Parallel algorithm of distribution of items over a given number of sections.

In this code, the network type definition introduces the name *ParallelDataMining* of the network type, a list of parameters – integer scalar parameter *n* and vector parameter *f* of *n* integers, and coordinate variable *I* ranging from *0* to *n-1*. Finally, it associates abstract processors with this coordinate system and

declares relative volumes of computations to be performed by each of the processors. It is assumed that *i*-th element of vector *f* is equal to the number of files processed by *i*-th abstract processor.

Execution of the statement *recon* is that all physical processors running the program execute in parallel some test code, and the time elapsed by each of the real processors is used to refresh the estimation of its performance.

The library function *MPC_Get_processors_info* returns the number of available physical processors (in variable *nprocs*) and their relative performances (in array *powers*).

Based on the number and relative performances of the actual processors, the library function *Partition* computes how many files of the data store each actual processor will process. So, after this call *files[i]* holds the number of files processed by *i*-th actual processor. In general, *MPC_Partition* divides a given whole (specified in the above call with *num_of_files*) into a number of parts in accordance with the given proportions.

Next key line of the code defines the abstract network *the_net* of type *ParallelDataMining* with actual parameters *nprocs* – the actual number of physical processors, and *files* – an integer array of *nprocs* elements containing actual numbers of files to be processed by the processors.

The rest computations and communications will be performed on this abstract network. The mpC programming system maps abstract processors of the abstract network *the_net* to real parallel processes constituting the running parallel program. This mapping is based, on the one hand, on information about the performance of physical processors of the real network executing the program, and on the other hand, on the above information about the parallel algorithm to be performed by the defined abstract network. The programming system does the mapping at run time and tries to minimize the execution time of the parallel algorithm.

The rest modifications of the original Orbix implementation of the application server are minor and rather technical. They are aimed at smooth integration of the mpC parallel environment into the Orbix distributed environment. Code implementing operations *Hello* and *Bye* is modified to initialize and finalize the mpC programming environment respectively. In addition, the code implementing operation *getDistribution* is modified to enable passing input data (the number of sections) from the Orbix framework of the application server to the mpC inserted component and output data (the recommended distribution) from the mpC component back to the Orbix layer. The input data is passed to the mpC program as an external argument, and a temporary file is used for passing the results

```
nettype ParallelDataMining(int n, int f[n]) {
  coord I=n;
  node {I>=0: files[I];};
};
...
repl nprocs, num_of_files, *files;
repl double *powers;
...
recon;
MPC_Get_processors_info(&nprocs, powers);
Partition(nprocs, powers, files, num_of_files);
{
  net ParallelDataMining(nprocs, files) the_net;
  ...
}
```

Figure 3. mpC implementation of parallel distribution of items .

computed by the mpC program to the main Orbix body of the application server. In general, the modifications integrating the mpC parallel application into the Orbix distributed application are pretty obvious and easy to make. Although there is possible some deeper integration of the two technologies, say, on the language level, it does not look reasonable. Such a deeper integration would be much more sophisticated and, at the same time, would

provide no visible improvement of the quality of services compared to the above light-weight integration scheme.

4. Experimental results

This section presents some results of experiments with the chain of supermarkets application.

A small network of workstations was used for the experiments. The client ran on an IBM RS6000 workstation, the serial application server ran on a 4-processor Sun E450 workstation, and the parallel application server ran on a network of two 4-processor Sun E450 workstations and one 6-processor HP 9000/K570 workstation. Table 1 shows relative performances of the three computers obtained automatically by an mpC utility that executed some serial test code on each of the computers.

The data store consisted of 60 files each containing 8000 basket records. Up to 100 different items could appear in a single basket. The client code invoked the remote operation *getDistribution* to get an optimal distribution of the 100 items over 5 sections and measured the execution time of the operation. This time obtained for different configurations of the application server are presented in Table 2. There were 4 configurations that only differed in the way of execution of the operation *getDistribution*:

- The original serial version of this operation was executed on a Sun workstation;
- The mpC parallel version of the operation was executed on the same 4-processor Sun workstation as the serial one;
- The mpC version was executed on the cluster of the two 4-processor Sun workstations;
- The mpC version was executed on the cluster of the two 4-processor Sun workstations and one 6-processor HP workstation.

One can see that parallel configurations of the application server demonstrated much better performance.

Workstation's number	1	2	3
Model	Sun E450	Sun E450	HP 9000/K570
Number of processors	4	4	6
Relative performance	2658	3280	6067

Table 1. Cluster of computers executing the application server.

Workstations involved in execution of the remote operation	1	1	1,2	1,2,3
Mode of the remote operation	Serial	Parallel	Parallel	Parallel
Execution time (seconds)	332	168	96	42

Table 2. Execution time of the remote operation *getDistribution* invoked to calculate an optimal distribution of 100 items over 5 sections.

5. Conclusion

The paper has presented an experience of integration of the mpC-based technology of parallel computing on heterogeneous clusters into the CORBA-based technology of distributed computing. It has demonstrated that the presented light-weighted integration techniques is easy to make and does not require any changes in the combined technologies. At

the same time, the integration essentially improves performance of application servers providing computationally intensive operations and running on networks of computers.

References

[1] Object Management Group, The Common Object Request Broker: Architecture and Specification, Revision 2.3, 1999.

[2] Sean Baker, CORBA Distributed Objects – Using Orbix, ACM Press, Adison Wesley, 1997.

[3] A.Lastovetsky, D.Arapov, A.Kalinov, and I.Ledovskih. A Parallel Language and Its Programming System for Heterogeneous Networks. *Concurrency: Practice and Experience*, 12(13): 1317-1343, 2000.

[4] D.Arapov, A.Kalinov, A.Lastovetsky, and I.Ledovskih. A Language Approach to High Performance Computing on Heterogeneous Networks. *Parallel and Distributed Computing Practices*, 2(3): 87-96, 1999.