
Arnaud Defrance, Stéphane Vialle, Morgann Wauquier
Firstname.Lastname@supelec.fr
Supelec, 2 rue Edouard Belin, 57070 Metz, France

Philippe Akim, Denny Dewnarain, Razak Loutou, Ali Methari
Firstname.Lastname@summithq.com
Summit Systems S.A, 62 Rue Beaubourg, 75003 Paris, France

Abstract

This paper introduces a generic n-tier distributed architecture for business applications, and its basic component: a computing server distributed on a PC cluster. Design of this distributed server is relevant of both cluster programming and business standards: it mixes TCP client-server mechanisms, MPI programming and database accesses from cluster nodes.

The main topics investigated in this paper are the design of a generic distributed TCP server based on the MPI paradigm, and the experimentation and optimization of concurrent accesses to databases from cluster nodes.

Finally, this new distributed architecture has been integrated in the industrial Summit environment, and a Hedge application (a famous risk analysis computation) has been implemented and performances have been measured on laboratory and industrial testbeds. They have validated the architecture, exhibiting interesting speed up.

1 Motivations

Typical HPC scientific applications run long computations in batch mode on large clusters or super-computers, and use flat files to read and write data. Classical MPI paradigm is well adapted to implement these applications on clusters. But business applications are different. They use client-server mechanisms, are often 3-tiers, 4-tiers or n-tiers architectures [2], access databases, and need to be run immediately in interactive mode. Users run numerous short or medium computations from client applications, at any time and need to get the results quickly (on-demand computations). They need to run their computations on available servers in interactive mode. Moreover, business application architectures are evolutive: new functionalities can be added easily, just including new server processes in the architecture.

Considering all these industrial constraints, we have designed a parallel architecture including on demand parallel computing servers. These computing servers run on PC clusters and mix distributed programming paradigm, client-server mechanism and database accesses. This parallel architecture has been evaluated on financial risk analysis based on Monte-Carlo simulations[6], leading to intensive computations and including time constraints. For example, traders in market places need to run large risk computations and to get the results quickly to make their deal on time!

The financial application we wanted to speed up and to size up used a distributed database, several scalable computing servers and some presentation servers, but still needed to increase the power of its computing servers (applicative servers). The new solution uses a generic distributed server architecture, some standard development libraries, and has been integrated in the industrial Summit environment.

Section 2 introduces our complete generic architecture and its basic component (a distributed on-demand server), section 3 focusses on the mix of TCP client-server paradigm and MPI distributed programming, and section 4 introduces our investigations about database accesses from cluster nodes. Then, section 5 describes the implementation of a complete financial application on our distributed architecture, and ex-
hibits some experimental performances in an industrial environment. Finally, section 6 summarizes the achieved results and points out the next steps of the project.

2 Architecture overview

To parallelize our business applications we consider a large pool of PCs, organized in clusters (see figure 1), and a two level distributed architecture. The high level takes into account the partitioning of the pool of PCs into clusters. These clusters can be statically defined or dynamically created and cancelled. They can have fixed size or on demand size. These topics are still under analyze, the high level of our architecture is not totally fixed.

The low level defines the basic component of our architecture: the cluster architecture of our distributed on-demand server, and its distributed programming strategy. This strategy will be TCP- and MPI-based and will have to interface databases, TCP clients and distributed processes on cluster nodes (see figure 2).

In the next sections this paper focusses on the design of the basic component of our distributed architecture, and introduces our first experiments on two different testbeds.

3 MPI-based TCP server

We have identified three important features for the MPI based TCP server we want to design (see figure 2):

1. The server node has to be clearly identified (ex: "PE-03" on figure 2) and fixed by users, not by the MPI runtime mechanism.

2. The TCP server node has to be a worker node, listening for client request and processing a part of this request.

3. The server has to remain up after processing a client request, listening for a new request and ready to process it.

3.1 TCP server identification

A usual MPI program is run specifying just the number of processes needed and a file listing all available machines to host these processes[7]. We do not make any assumption on the numbering of the MPI processes and their mapping on the available machines: they are fixed by the MPI runtime mechanism. So, we decided the TCP server process will not be a MPI process with a specific number but will be the MPI process running on a specific node. The TCP server node will be specified in a config file or in the command line (ex: mpirun -np 10 -machinefile machines.txt prog.exe -sn PE-03, where -sn specifies the server name). It will be identified at execution time, during the initialization step of the MPI program.

Top of figure 3 illustrates this initialization step. Each MPI process reads the name of the TCP server node, gets the name of its host node, detects if it is running on the server node, and sends this information to the MPI process 0. So, the MPI process 0 can identify the rank of the MPI process hosted by the server node, and broadcasts this rank to all MPI processes. Finally, each MPI process knows the rank of the server process. Moreover, this algorithm chooses automatically the MPI process with the lower rank if several MPI processes are hosted by the server node.

3.2 Distributed server algorithm

When the server process is identified, the first computation step starts. The server process initializes and listens a socket, waiting for a client request, while other MPI processes wait for a message of the server process, as illustrated on the left-bottom part of the figure 3. Each time the server process receives a client request, it reads this request, may load some data from the database, and splits and load balances the work on all the MPI processes (it broadcasts or scatters a mission to each MPI process).

Then each process receives its mission, usually loads data from the database and starts its work on all the MPI processes (it broadcasts or scatters a mission to each MPI process).

This algorithm has been successfully implemented using MPICH and the CommonC++ socket library[1] on Windows and Linux. Next experiments introduced in this article use this client/distributed-server architecture.
Figure 1: Global architecture: a N-tiers client-server architecture and a pool of PCs split into clusters. Applicative servers are distributed on PC clusters.

Figure 2: Detail of the basic component of the architecture: a distributed and parallel TCP applicative server.

4 DB access experiments

Business Application use databases, so our distributed architecture needs to access databases from cluster nodes. In the next sections we identify three main kinds of database accesses and several access strategies from cluster nodes. Then we try to point out a generic and efficient strategy for each kind of access.

4.1 Three kinds of DB accesses

We can easily identify three kinds of data accesses from concurrent processes:

1. Independent data reading: each MPI process reads specific data. For example, financial computing processes load different lists of trades to evaluate.

2. Identical data reading: all MPI processes need to read the same data. For example, financial computing processes need to load the same market data before to evaluate their trades.

3. Independent data writing: each MPI process writes its results in a database.

4.2 Possible DB access strategies

From a MPI parallel programming point of view, we can adopt different implementation strategies. Database accesses can be distributed on the different processes, or can be centralized (gathered) on a process that will forward/collection the data to/from other processes, using MPI communications. When centralized on a process, database accesses and data forward/collection can be serialized or overlapped depending on MPI communication used. When database accesses are distributed in all MPI processes, their executions can be highly concurrent or scheduled using MPI synchronization mechanisms.

All these strategies have been experimented on the three kinds of database access identified previously, and on two testbeds using different databases.
4.3 MySQL database experiments

Figures 4, 6 and 8 show the times we have measured on our laboratory testbed: a 6 node Gigabit Ethernet PC cluster accessing a MySQL database across the native MySQL API. This database was hosted on one devoted PC with SCSI disks.

When reading independent data from each node (ex: loading some trades to evaluate), it is really faster each node makes its own database access than centralizing the requests on one node (see figure 4). The database did not slow down when all the cluster nodes emit concurrent requests, and there is no improvement when trying to schedule and serialize the different accesses. At the opposite, when reading identical set of data from each node (ex: loading some common market data on each processor) the most efficient way is to make only one database access on one node and to broadcast the data to all other nodes (see figure 6). A MPI broadcast operation makes it easy to implement. Finally, when writing independent results from each node in the database, it is more efficient to make concurrent write operations than to centralize the data on one node before to write in the database (see figure 8). This was similar to read independent data, but concurrent writes in our MySQL database were a little bit more efficient when scheduled and serialized to avoid contention on the database (see the distributed-scheduled curve). MPI synchronization barriers...
make this scheduling easy to implement.

4.4 Sybase database experiments

This section introduces experiments on a Sybase database, accessed across the Summit API (used by all Summit applications), and hosted on one devoted PC connected to the cluster across a Gigabit Ethernet network. However, this PC had only IDE disks, leading to longer execution times than previous experiments on the laboratory testbed.

For each kind of database access, the distributed & concurrent strategy appeared the most efficient, or one of the most efficient (see figures 5, 7 and 9). Then it has been the favorite, because it is efficient and it is very easy to implement. It seems the industrial Sybase database (accessed across the Summit API) supports any kind of concurrent accesses from a small number of nodes.

4.5 Best strategies

Experiments on industrial and laboratory testbeds have led to point out different best strategies to access a database from a cluster (see table 1). Finally, it is possible to identify the best common strategies for the two testbeds we have experimented, considering only efficient strategies and looking for good compromises. But we prefer to consider real business applications will use only industrial databases (like Sybase), and to focus on the distributed-concurrent strategy.

However, performances and optimal strategies could change when the number of processors will increase. So, our next experiments will focus on larger industrial systems.

5 The Hedge application

The Hedge computation is a famous computation in risk analysis. It calls a lot of basic pricing operations (frequently based on Monte-Carlo
Table 1: Best database access strategies from cluster nodes

<table>
<thead>
<tr>
<th>Database Access</th>
<th>Independent data reading</th>
<th>Identical data reading</th>
<th>Independent data writing</th>
</tr>
</thead>
<tbody>
<tr>
<td>MySQL - native MySQL API</td>
<td>distributed &amp; concurrent</td>
<td>centralized &amp; broadcast</td>
<td>distributed &amp; scheduled</td>
</tr>
<tr>
<td>Sybase - Summit API</td>
<td>distributed &amp; concurrent</td>
<td>distributed &amp; concurrent</td>
<td>distributed &amp; concurrent</td>
</tr>
<tr>
<td>Best common strategies (interesting compromises)</td>
<td>distributed &amp; concurrent</td>
<td>centralized &amp; broadcast</td>
<td>distributed &amp; concurrent</td>
</tr>
</tbody>
</table>

simulations) and consider large sets of market data. It is time consuming and a good candidate for distributed computing, like many financial risk computations ([8, 4]). The following sections introduce its implementation in the industrial Summit environment, according to our distributed TCP-server architecture.

5.1 Introduction to the Hedge

The Hedge is the computation of a set of discrete derivatives, based on finite differences. The market data are artificially disturbed, some trade values are re-computed, and the difference with their previous values measure their sensitivity to the market data changes. These computations allow to detect and reduce the risks associated to trades and to market data evolution.

The distributed algorithm of our Hedge computation is illustrated on the figure 10. The server process receives a client request (a request for a Hedge computation) (1), loads from the database some informations on the trades to process (2), and splits the job for the different cluster nodes, according to a load balancing strategy (3). Then the MPI server process sends to each MPI process a description of their job, using a MPI collective communication (broadcast or scatter operation). So, each MPI process receives a description of a task to process (4) and then loads data from the database: a list of trades to process (4) and a list of up to date market data (5). Then all cluster nodes shift the market data (6) to artificially disturb these data, and re-evaluate all their trades (7). Finally, the server process gathers these elementary results (8) (executing a MPI gather operation) and returns the global result to the client (9).

5.2 Optimized implementation

We have implemented the distributed algorithm introduced in the previous section in the Summit environment with different optimizations.

First optimization has been to implement the database accesses with respect to the optimal strategies identified on an industrial database in section 4.5. Each processor need to read the trades and the market data it has to process (step 4 and 5 of algorithm 10). The corresponding database accesses are achieved according to the distributed-concurrent strategy: each processor accesses its data concurrently to other processors.

The second optimization has been to call a specific function of the Summit API to read
short trade descriptions on the server, in place of complete descriptions, to achieve the step 2 of the algorithm (see figure 10). This specific function was available but not used in sequential programs. This optimization has decreased the serial fraction of the *Hedge* computation of 30%. Its impact will be great on large clusters, when the execution time of the serial fraction will become the most important part of the total execution time, according to Amdahl's law[3].

The third optimization has been to improve the load balancing mechanism (step 3 on figure 10) taking into account the trades are heterogeneous. Now the server process sorts the trades according to their kinds or to their *estimated processing times*, and distribute the trades on the different processors applying a round-robin algorithm. To estimate the *processing time* of the trades, some benchmarks have to be run before the *Hedge* computation to measure some elementary execution times. These measures are stored in a database and read by the server process. However, trades with the same kind have similar processing times, and optimizations of the load balancing based on the kind of the trades or on their estimated execution time have led to close improvements. Finally, compared to a basic load balancing of the amount of trades, the total execution time has decreased of 35% on 3 PCs, when processing 5050 trades of 5 different kinds.

5.3 Experimental performances

We have run an optimized distributed *Hedge* computations on a 8-node industrial cluster, with a Sybase database and the Summit environment.

We have processed a small portfolio of 5000 trades and a larger one with 10000 trades, both were heterogeneous portfolios including different kinds of trades. The curves on the figure 11 show the speed up reaches approximately 5 on 8 PCs for the small portfolio, and increase until 5.8 for the large portfolio, leading to an efficiency close to 73%.

These first real experiments are very good, considering the entire *Hedge* application includes database accesses, sequential parts and process heterogeneous data. Moreover, performances seem to increase when the problem size increases, according to Gustafson’s law [5]. Our distributed architecture and algorithm are promising to process larger problems on larger clusters.
6 Conclusion

This research has introduced a generic n-tiers distributed architecture for business applications, and has validated its basic component: a distributed on-demand server mixing TCP client-server mechanism, MPI programming and concurrent accesses to databases from cluster nodes.

We have designed and implemented a distributed TCP server based on both TCP client-server and MPI paradigms, and we have tracked the optimal strategies to access databases from cluster nodes. Then we have distributed a Hedge computation (a famous risk analysis computation) on an industrial testbed, using our distributed TCP server architecture and optimizing the database accesses and the load balancing mechanism. We have achieved very good speedup, and an efficiency close to 73% when processing a classical 10000-trade portfolio on a 8-node industrial testbed. These performances show our distributed approach is promising.

Implementation of the basic component of our distributed architecture has been successful under Windows on the Summit industrial environment. We have both used the CommonC++ and MPICH-1 libraries, a Sybase database and the Summit proprietary toolkit.

Next step will consist in running bigger computations on larger clusters, and testing the scalability of our distributed architecture.

Acknowledgment

Authors want to thank Region Lorraine that has supported a part of this research.

References