ENHANCING MIDDLEWARES WITH PARALLEL PROGRAMMING *

FARAHNAZ POORFARD
Computer Engineering and IT Department, Amirkabir University of Technology
(Tehran Polytechnic), 424 Hafez Ave, Tehran, Iran

HOSSEIN PEDRAM
Computer Engineering and IT Department, Amirkabir University of Technology
(Tehran Polytechnic), 424 Hafez Ave., Tehran, Iran
{fz-poorfard, pedram}@aut.ac.ir

Middleware is necessary for distributed systems; in fact all applications in a netware world request their services through a middleware. In modern applications the time for accessing to services through a middleware is important and even critical. Access time not only depends on network speed, but it also depends on runtime of middleware by itself, so we often try to minimize the execution time of middlewares. This paper presents the idea of merging middlewares with parallel programming techniques. We study merging OpenCOM, which is a reflective adaptive middleware, with MPI as a favorite technique in parallel programming. We focus on enhancing the runtime of configuration and reconfiguration operations of this middleware.

1. Introduction

Middleware has emerged as an important architectural component in supporting distributed applications. The role of middleware is to present a unified programming model to application writers and to mask out problems of heterogeneity and distribution [4]. Today, we know that doing this task has been very hard for a middleware than previous, although we have had great progresses in the network world, but this means that there are many newer services like multimedia, real-time, wireless and mobile, that a well-developed middleware should be able to support. In addition because of increasing number of application domains requiring distribution, and the consequent increase in distributed-systems requirements, users might need to extend or restrict a distribution model [18]. It is an aspect of flexibility, especially in configuration and reconfiguration operations.

Generally, to apply any changes to a middleware we need adaptivity in our model. The need for adaptation comes from the fact that the environment in which distributed applications are executed, changes continuously [1]. The term “changes” in an environment sees all different aspects.

Today, more important roles of a middleware are: adaptivity, (re)configurability and flexibility. Hence, we are looking for some models in middleware that help us to have well-developed and ubiquitous applications. One of the useful models that offer this goal is the component-based model. The component-based model has a rich adaptivity feature. In this model, a middleware is just composed of a few components, each of which does a few tasks, and composition of these tasks can make the proposed aims of the determined middleware. Activation of components must occur in real-time, which means that component initialization must not become a bottleneck. Thus, on-demand linking/unlinking mechanisms are necessary to (re)configure component implementations dynamically. The lifecycle for linking/unlinking of these components must be optimized using reflective middleware techniques to minimize footprint, prolong battery life, maximize extensibility, and meet key application QoS requirements more adaptively [25].

Thus, we need an additional feature for our model: reflection, which defines the rules for cooperating among components to do the whole task. Clearly, we will need operations to generate and delete component(s), and (un)establish connections among components to reach a complete model.

Reflection refers to the ability of a system to reason about and act upon itself. More specifically, a reflective system is one that provides a representation of its own behavior which is amenable to inspection and

adaptation, and is casually connected to the underlying behavior it describes. Casually-connected means that changes made to the self-representation are immediately mirrored in the underlying system’s actual state and behavior, and vice-versa [23].

Not only the new and progressive features of a middleware, but also the speed of accessing to its services especially in real-time and/or critical applications, is very important. This is the main reason that encouraged us to work on enhancing middleware’s runtime. We decided to apply such enhancement by adding parallel programming to middleware. The remaining of this paper is organized as follows: In the next section, a survey of related works is presented. In section 3 we describe two main tools for our inspection, named OpenCOM and MPI. In section 4 there is an explanation of our experiments. We also present our main results in section 5 and conclude the paper in section 6.

2. Related Works

There are different works to make progresses on middleware, obviously each of them consider different aspects of middlewares. For example in [15] a combination of aspect oriented programming (AOP) with reflective middlewares has been explored. This combination can put together the advantages of customizing a middleware implementation (by AOP) and the ability to alter runtime behavior (by reflection).

Ref. [21] introduced DAJ, a system for encapsulating middleware services to mask invasive and pervasive middleware functionality. This technique can help to increase a middleware design process.

Javier Fernandez et al. introduce GridExpand, a new approach to data access in computational grids. It is a parallel I/O middleware that integrates heterogeneous data storage resources in grids [9].

There are also some individual designs of middlewares, e.g. Jgroup which is a middleware system that integrates group technology with distributed objects. It supports a programming paradigm called object groups and enables development of dependable network services based on replication [22].

ProActive is a Java middleware library for parallel, distributed and multithreaded computing. It is a programming model to simplify the programming and execution of parallel applications on multi-core processors, distributed Local Area Network, clusters and Data Center Servers, and GRIDs [24].

Al-Jaroodi et al. in [17] introduce a middleware infrastructure that provides software services for developing and deploying high-performance parallel programming models and distributed applications on clusters and networked heterogeneous systems. Their middleware infrastructure utilizes distributed agents residing on the participating machines and communicating with one another to perform the required functions.

Among all these trends there is not a direct combination of middlewares with parallelism. We show in this paper that this kind of combination can result in better execution time.

3. Background

In section 1 we enumerated typical features of new generation of middlewares. In studying different existing models, we have chosen the OpenCOM [19] which is a general-purpose and language independent component-based systems building technology [2]. We first tried to evaluate it, so we did some experiences for measuring the runtime of its different operations and considering these timings against each other, for evaluating the overhead/benefit trade off. After all, we tried to join OpenCOM with some existing mechanisms for parallel processing in order to reduce the main timing overheads, if it does. MPI has been used for this.

The following subsections provide a quick review on OpenCOM and MPI respectively.

3.1. OpenCOM

OpenCOM is a lightweight and efficient in-process component model, built atop a subset of Microsoft’s COM [5]. A core of COM has been used as: the binary-level interoperability standard (vtable); the Microsoft’s IDL; COM’s GUIDs; and the IUnknown interface.

OpenCOM has been built by using this core and considering three additional concepts: explicit dependencies among components; mechanism-level functionality for reconfigurations, such as mutual exclusion locks; and intercepting on component operations, as pre-method call or post-method call.

Fig. 1 shows the main concepts in OpenCOM: interface, receptacle and binding.

The architecture of OpenCOM is very simple and straightforward. OpenCOM uses two types of
components: One (and only one) OpenCOM runtime component that is the header component for other components in an address space, and some other components which can exist in an arbitrary number. The operations and connections between these later components make our middleware environment and its functionality.

Figure 1. Main concepts of OpenCOM [3]

In addition, it is possible to encapsulate some components in a component framework (CF) for applying security and control policies, simplifying the design and maintenance of middleware, and also increasing the speed of reconfiguration by using more coarse grained components in design process (an advanced form of CFs than its primary one, with IAccept interface, has been shown in [10]). Each CF can also accept additional plug-in components that change or extend its behavior. In OpenCOM, components or CFs can be connected hierarchically, but we are able to have a layered approach for our design by applying the necessary restrictions. This layering which is just a virtual view for actual hierarchies helps us understand the middleware operations better. We choose layering since it matches our networks architectures (for more details see [5] and [6]).

By using components (and CFs) as major entities in OpenCOM, we can use reflection: using different combinations of components and using different interfaces of them which reflects different views of our middleware infrastructure. OpenCOM has been used as an infrastructure layer for some more progressive middlewares too, like GridKit ([12] and [13]), RUNES ([7] and [8]) and ReMMoC ([10] and [11]).

3.2. MPI

Parallel programming is a well-known way for enhancing the efficiency of programs’ runtime. We can use parallel programming by distributing the run of different parts of program on separate processes/threads on one CPU, or even on separate processes/threads on several separate CPUs. Different tools for parallel programming have been built and among them is MPI (Message Passing Interface). MPI supports both point-to-point and collective communication operations. The primary goals of MPI are efficient communication and portability [16]. In addition, MPI has a rich library for both primary and complex operations and its library has been prepared for many alive programming languages such as C, C++ and Fortran. MPI can be installed on a wide range of hardware/software platforms, so now we are able to use MPI on all systems from super computers to PC clusters. It is also simple and straightforward to develop and debug a program on it [16], [20] and [14]. For all these, to enhance the timing of (re)configuration phase in our middleware, we chose MPI (more accurately saying, MPICH) as our parallel programming tool.

4. Experiments

In studying the OpenCOM, we did some timing evaluations and noticed that at the configuration phase of middleware and at each reconfiguration operation, there is a time overhead. We measured time for different operations accurately and noticed that it is the creation of the components that contributes mainly to this overhead. It is of note that we should directly create different instances of our proposed components/CFs and define their connections for creating our middleware. Similarly, at each reconfiguration operation, we may have to create some new components/CFs. To reach some equilibrium, we tried to apply a parallel programming technique on our middleware.

First, by expanding the OpenCOM’s demo, we measured time for creating and activating some sample components. Then we defined some considerations for ourselves: 1) we did not consider the ‘CoInitialize’ operation, because it runs just once; 2) we did not consider the creating and activating operations for the main runtime OpenCOM component; 3) to create a component, we used ‘createInstance’ function and to activate it we obtained its ILifeCycle interface and then called ‘startup’ function; 4) we consider a createInstance operation as an atomic operation.

5. Results and Discussion

In this section we will describe and discuss our results especially considering the speedup. The main goal is to reduce the runtime and increase the speedup. Our
experiments are basically divided into two modes: uniprocessor-multiprocess and multiprocessor.

In our experiments we have 6 components, for two of them we called two more interfaces than just ILifeCycle interface, but the effect of this is negligible. The results are presented in the next two subsections.

5.1. Uniprocessor-multiprocess

In the first phase, we ran the merged OpenCOM and MPI middleware on a single processor system, and then increased the number of processes one by one. In this case, as we expected, increasing the processes, increases the total time of initiating the middleware. The growth will continue until the number of processes would be equal to the number of components. The reason is for the atomicity of the createInstance operation.

We then applied in two phases some complexity in generating components and considered the results. By increasing the complexity the results would be better especially on one processor. We call these 3 modes of complexity as: 1) normal complexity, 2) a few more complexity, 3) more complexity.

Fig. 2 shows the results for these three modes of complexity on one processor.

Afterwards, we first did the experiments with 2 processors, and then increased them to 6. The results are shown in the following subsection.

5.2. Multiprocessor

With 2 processors we obtained more interesting results. As before, we started with all 6 components on one processor and continued first by adding a second processor. After adding the second processor we increased the number of processes one by one, up to six. In each step we tried with all different number of processes on our 2 processors, for example when we reached to 2 processors and 4 processes we examined 3 cases separately:

- 1 process on the first processor and 3 processes on the second,
- 3 processes on the first processor and 1 process on the second,
- 2 processes on each processor.

Then we selected the best results (minimum timings). We did all experiments for our previous three modes: “components with normal complexity” (mode 1), “components with a few more complexity” (mode 2) and “components with more complexity” (mode 3). Fig. 3 shows the results. As it is seen:

- There is a sudden enhancement in fig. 3 for distributing all components between processes. Of course, our two processors were not homogeneous.
- Although both mode 2 and 3 have acceptable shapes, mode 2 has greater speedup (2.734/1.391 = 1.965) than mode 3 (2.368/1.386=1.709), so it is better to control the complexity of components from being too expensive.

By increasing the number of processors to 6, we see a considerable decrease in complete distribution mode (6 processors), (depicted in the following figures) which is very similar to complete distribution of components among 6 processes on 2 processors with normal complexity mode (mode 1). Figures 4, 5 and 6 are the integration of 1, 2 and 6 processors and are respectively for our three familiar modes of complexity. The similarity of decreasing runtime with full distribution on 2 and 6 processors in the following figures shows that in distributing components, the best case is a complete distribution (not partial).

Speedups for 6 processors are: 3.121 / 0.771 = 4.048 for mode 1, 2.367 / 0.47 = 5.036 for mode 2, and 2.347 / 0.463 = 5.069 for mode 3, that is a bit greater than 5.036, but we believe that it is not considerable.
compared to the cost of complexity. In the other word, we vote to a middle complexity (“a few more complexity” mode).

![Figure 4. Results for “normal complexity” mode](image)

As it is shown, the best results belong to multiprocessor mode. Both the 2 processors-multiprocess mode and the 6 processors-uniprocess mode have acceptable functionality, but just when all components have been distributed (full distribution).

![Figure 5. Results for “a few more complexity” mode](image)

![Figure 6. Results for “more complexity” mode](image)

Fig. 7 shows the speedups according to our 3 modes of complexity. We see that results for 6 processors seem better because of their both better speedup and more stability in variation of speedups. But if the usage of “more complexity” mode (mode 3) would not be acceptable for its complexity and time, we suggest the “a few more complexity” mode (mode 2).

![Figure 7. Comparing the speedups according to complexity](image)

6. Conclusion

In this paper, we proposed the idea of distributing the operation in the middlewares aiming lower execution time.

We showed that providing a middleware with parallel processing can enhance the runtime of (re)configuration operation. In selecting a suitable mode for parallelism degree, consideration of speedup can help us. According to our experiments a middle complexity level leads to a better speedup.

Acknowledgments

We would like to specially thank Gordon Blair and Paul Grace, Lancaster University, for their feedback, insights and help.

We also thank Ehsan K. Ardestani, University of California Santa Cruz for reviewing the paper and for his feedbacks and truly help.

References


