USING REFLEXD FOR A GRID SOLUTION TO THE N-QUEENS PROBLEM: A CASE STUDY

Rodolfo Toledo – Éric Tanter* – José Piquer
University of Chile, Computer Science Dept.
Avenida Blanco Encalada 2120, Santiago, Chile.
{rtoledo,etanter,jpiquer}@dcc.uchile.cl

Denis Caromel – Mario Leyton
INRIA Sophia-Antipolis, CNRS, I3S, UNSA.
2004, Route des Lucioles, BP 93, F-06902 Sophia-Antipolis Cedex, France.
{First.Last}@sophia.inria.fr

Abstract

Aspect-Oriented Programming (AOP) promotes better separation of concerns in software systems by introducing aspects for the modular implementation of crosscutting concerns. It is therefore not surprising that the distribution concern is recently being addressed from an AOP point of view, and finally materialized in AOP frameworks like ReflexD or languages like AWED. The motivation of this paper is to compare the implementation of a solution to the N-Queens problem in a distributed environment, when using the ReflexD library for distributed AOP and when using the ProActive middleware. Although the paradigms and target environments are different, precisely in this difference lies the interest of the exercise: this comparison is one of the necessary steps in a path to validate the applicability of AOP in a distributed setting. Beyond the software engineering interest of an AOP-based solution over a standard middleware approach, this paper highlights the good performance of the ReflexD platform.

Keywords: Aspect-oriented programming, distribution aspect, Grid, ReflexD, ProActive.

1. Introduction

Aspect-Oriented Programming (AOP) provides means for proper modularization of crosscutting concerns [5], *i.e.* concerns that cannot be cleanly modularized using traditional programming paradigms. Typical examples of such concerns are *non-functional* concerns such as monitoring, security, concurrency, etc., but also *functional* concerns such as observation relationships and,

*É. Tanter is partially financed by the Millenium Nucleus Center for Web Research, Grant P04-067-F, Mideplan, Chile.

in general, coordination between different modules. Without AOP, the implementation of such concerns is scattered across several modules. AOP also helps *adaptation* of software systems: for a given concern to be adaptable, it first has to be modularized.

The relation between AOP and distributed computing is interesting. Even though AOP is used in application servers [6], aspects are defined and applied locally to enhance the implementation of the application server. In other words, *AOP in distributed systems is NOT distributed AOP*. Extensions to OOP to the world of distribution, like Java RMI, do not help for distributed AOP: a distributed aspect cannot be modularly implemented. This is why proposals like JAC [8], DJcutter [7], and AWED [3] have appeared, addressing the distribution concern from an AOP point of view. ReflexD [10] also targets distributed AOP. Being a versatile kernel for distributed AOP, ReflexD offers the basic building blocks for distributed aspects, on top of which (domain-specific) distributed aspect languages like AWED can be implemented. Distributed AOP is being applied in various settings, like unit testing [7], sophisticated distributed cache policies, and checking of architectural constraints in distributed systems [2].

In this paper we consider a traditional Grid computing problem, and see if and how ReflexD can help in turning a non-distributed implementation into a distributed one, running on a Grid. An existing approach to such a problem is to use a middleware library like ProActive [1], which allows the original solution to be run on a Grid, with some modifications to the source code. Although quite minimal, these modifications to the source are undesirable from a software engineering point of view. We rather adopt the AOP approach, abstracting the distribution aspect and using a distributed AOP library (like ReflexD) to implement it, hence avoiding direct modifications to the source code.

More precisely, we consider an implementation of the N-Queens problem using ReflexD and a comparison of this solution with a variation of the implementation that won the "ProActive N-Queens GRID Contest" in 2004 by Cansado and Leyton [12], based on ProActive. The N-Queens problem consists in finding the number of ways N queens can be placed on an $N \times N$ board without any of them attacking each other. This problem is known to be $O(n^n)$, which makes it impossible to solve for big values of N. In fact, for values greater than 20 of N, the algorithm takes more than one day to finish the calculation in an average computer (Pentium IV of 3.0 GHz and 1Gb of RAM).

In addition to the software engineering benefits of using aspect orientation, the aim of this work is to concretely compare the efficiency of the ReflexD platform versus the well-known ProActive middleware. The hypothesis is that ReflexD, being more lightweight than ProActive, should have gains in terms of performance. However, the considered problem does not entail much communication between nodes, so the gain should be marginal. The results of the benchmarks outweigh our expectations, first by validating that ReflexD is

more efficient for solving the N-Queens problem, and also by indicating a great potential for more complex problems that entail much more communication between nodes.

This paper is organized as follows: Section 2 introduces the algorithmic solution to the N-Queens problem, and how it is parallelized. Then, Section 3 briefly introduces ProActive and the ProActive-based solution to the N-Queens problem. The ReflexD approach is shown in Section 4. Finally, Section 5 presents and analyzes the results of benchmarking the two solutions; Section 6 concludes and discusses future work.

2. Extended Parallel Takaken's Algorithm

The fastest sequential algorithm to solve the N-Queens problem nowadays is the Takaken's algorithm (TA). The TA is based on the symmetry of the N-Queens problem to avoid unnecessary calculations besides of making intensive use of bitwise operations to test all the possible configurations of the board.

The Extended Parallel Takaken's Algorithm (EPTA) is an extension of the TA that works on multiples CPUs. In [12], EPTA is used to refer to the distributed implementation, but in this paper we use EPTA for just the concurrent (multi-cpu, non-distributed) algorithm, since it is the base for our experiment. We therefore assume that the original EPTA implementation is correctly implemented for a non-distributed, but concurrent environment (hence issues such as thread safety are addressed).

The methodology used by the EPTA algorithm is to divide and conquer: it reduces the complexity of a problem of size N in several problems of a certain size $M < N$, which can be computed in a single computer in reasonable time. This division is achieved by pre-placing $N - M$ queens on the board, which reduces the number of rows and columns the remaining queens can use. It is important to notice that as $N - M$ increases, the number of sub-problems we generate also increases because the number of possibilities of placing the $N - M$ queens is bigger.

The principal entities in this algorithm are the *manager* and the *workers*, which interact in the following way:

1. The manager divides the problem into sub-problems, called *tasks*, by fixing the board positions of some queens.

2. The workers, running on different CPUs, continuously *a)* ask the manager for a task to solve, *b)* solve it, *c)* submit their result back to the manager upon completion.

3. The manager receives the sub-results calculated by the workers and finally joins them to find the complete solution.

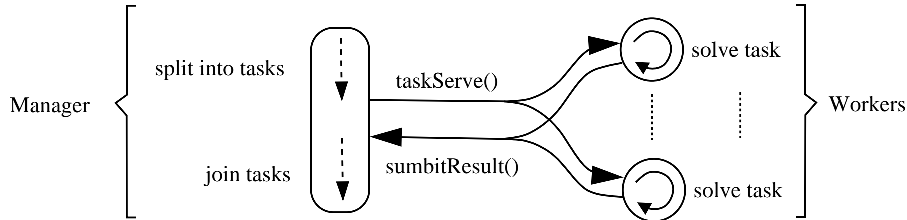

Figure 1. Extended Parallel Takaken's Algorithm

Figure 1 illustrates the basic working of EPTA. The `taskServe` and `submitResult` methods from the communication protocol between the workers and the manager.

3. ProActive Solution

In this section we briefly describe some of the elements present in ProActive to later explain the implementation of the distributed version of EPTA that uses it. Finally, we discuss the advantages and disadvantages of this implementation.

3.1 ProActive in a Nutshell

ProActive is a middleware for seamless parallel, distributed, and concurrent computing in Java [1]. ProActive is based on a model of *active objects*. An active object encapsulates its own execution thread, and has a queue of requests based on which it can decide the request to serve, either based on the request type (e.g. calls to the `stopImmediately` method) or based on some request property (e.g. the oldest unattended request). An active object is remotely accessible in ProActive, and interactions between active objects is asynchronous.

In order to support return values of asynchronous calls, ProActive supports transparent *futures*. A future is a placeholder for the result of an asynchronous call, which is immediately returned to the sender. The sender can then proceed in parallel with the server active object, as long as it does not need to access the result. Upon access, either the future has already been resolved to the actual result of the server activity, in which case the sender proceeds as normal, or the sender is blocked till the server activity completes. This mechanism, known as wait-by-necessity [4], is therefore a convenient mean to augment parallelism.

3.2 ProActive Implementation

The ProActive implementation we consider in this work is a variation of the implementation that won the "ProActive N-Queens GRID Contest" in 2004.

We say "variation" because we have actually disabled the optimization that allows the solution to run on very large Grids[1]. This change does not invalidate our study, because the core of the implementation itself is not changed at all, and our benchmarks are performed on a Grid of limited size.

The ProActive implementation is simply the non-distributed (although concurrent) implementation we discussed previously. In order to run it on a Grid, we need to modify the source code to create active objects explicitly in some remote nodes. The code below illustrates the modified code when creating workers:

```
1  Node node = ...; // obtain desired node for worker
2  Object[] args = ... ; // include reference to the manager
3  Worker worker =
4          (Worker) ProActive.newActive("Worker", args, node);
5  // ...proceed with worker...
```

With ProActive, this is all that needs to be modified: apart from worker creation, which has to be done explicitly on remote nodes by calling a ProActive library method after obtaining a `Node` reference, the core of the algorithm is the same, since other issues arising in distribution are handled by ProActive. Note that also, the manager itself has to be created as an active object in order to be remotely accessible.

3.3 Discussion

Although a reasonable amount of changes in the original code is necessary, these changes have bad software engineering properties. As we have to modify the code in order to explicitly create nodes objects and active objects using the `newActive` method, the resulting code is tightly coupled to the ProActive library, and is no more runnable in a non-distributed, non-ProActive setting. In other words, the concern of deploying workers on nodes using ProActive is *tangled* with the actual implementation of the algorithm, thereby compromising its reuse in other contexts. Furthermore, using ProActive implies using the whole object model, with support for active objects with their request queues, and futures (whose implementation in Java is not trivial). This has impact on the performance as validated later on. Furthermore, it has to be considered that in the case of the N-Queens problem, the place of worker instantiation in source code is quite localized, but in more complex problems, this may imply making modifications is several places spread out over the original code.

[1]Basically, the optimized version works in a hierarchical way: it defines intermediary entities between the manager and the workers. One of the purposes of these entities is to avoid the overhead that the workers produce over the manager when they communicate with it (when asking for new tasks to solve and when submitting their results).

4. ReflexD Solution

In order to solve the software engineering issue of the ProActive solution, we now consider a solution based on ReflexD, a platform for distributed aspect-oriented programming [10].

4.1 ReflexD in a Nutshell

ReflexD [10] is a platform for distributed AOP in Java. It is an extension to distribution of the Reflex AOP kernel [11], which relies on the notion of an explicit *link* binding a cut to an action, in order to specify aspects. The *cut* refers to the basic elements one wants to intercept with an aspect (such as calls to particular methods or instantiations of particular classes), and is specified in a *hookset*. The *action* to undertake upon occurrences of the events of interest is implemented in an object usually called a metaobject, but which is indeed a normal Java object. Finally, the *binding* is the specification of the exact protocol between the cut and the action (that is, which method to call on the metaobject, passing which pieces of information from the interception operation occurrence), along with a number of customization parameters. A link is the first-class entity that packages this piece of information defining an aspect. This is illustrated as needed in the following of the section. For more details we refer the reader to the above-mentioned papers.

ReflexD extends Reflex with notions such as distributed cut, remote action, and distributed binding. These advanced mechanisms are however not needed here. We just rely on the remote object creation service of ReflexD and on the Remote Consistency Framework for dealing with different types of remote communications. This is presented hereafter.

4.2 ReflexD Implementation

The two requirements we need to meet in order to turn EPTA into a distributed version is *a)* provide a way to execute the workers on a different host, and *b)* provide a way to let them communicate with the manager in order to submit the partial solutions.

Transparent Creation of Workers in Remote Hosts. An important factor of motivation for using ReflexD is to *transparently* create workers in remote hosts, rather than having to explicitly modify the actual source code as in the case with ProActive. This is done in ReflexD by specifying a link binding all instantiations of `Worker` objects occurring in any class to a remote creation service:

```
1 link RemoteWorkerCreation {
2   on Instantiation in named(*) where named(Worker)
3   around : RemoteCreator.createRemote(#args)
4   by shared(new RemoteCreator(''Worker''))
5 }
```

The above code uses the concrete syntax of Reflex presented in [9], because of the enhanced readability it provides. All occurrences of the instantiation operation (that is, calls to `new`) occuring in any class (`*`) where the instantiation class is named `Worker` are *replaced* (`around`) by a call to the `createRemote` method of a `RemoteCreator` object, passing it as parameters all the arguments of the original instantiation (`#args`). The `RemoteCreator` instance is unique and shared by all objects creating workers (`by shared`), and is created by giving it the name of the worker class. The `RemoteCreator` class is a utility class, whose simplified implementation is as follows:

```
1 public class RemoteCreator {
2   String classname; // initialized in constructor
3   public Object createRemote(Object[] args){
4     RHost host = RHosts.get(...); // determine appropriate host
5     return host.create(classname, args);
6 } }
```

The `createRemote` method first gets a reference to the remote host in which the worker should be created (4). Then the `create` method of the `RHost` is used to create an object in that host with the same original parameters, and the reference to this remote object is returned (3). It is important to note that from now on, the application will use this remote reference in the same way as a local reference. Remote communication is managed behind the scene by Reflex. For space reasons, we deliberately skip the whole issue of determining the target hosts for worker creation, both in the ProActive and ReflexD solutions.

Communication Between Workers and Manager. An element that we have not addressed yet is the fact that the manager is passed as a constructor parameter to the workers to allow them to ask for tasks and to notify their results; and with the link definition above, it is now passed through the network, when instantiating remotely a worker (line 5 in the code above). If nothing is done, the code above will fail because the `Manager` class has no reason to be serializable. Making it serializable does not solve the problem neither, because the default semantics of Java is to pass objects by copy. As a result, each worker will have a local reference to a copy of the manager.

Fortunately, ReflexD includes a Remote Consistency Framework[2], which makes it possible to specify custom policies when turning non-remote objects into remote ones. The policy we need in the present case is known as a SlaveToMasterPolicy in ReflexD, meaning a remote copy of a Manager should act as a slave delegating to the original instance (residing on the host in which the remote creation service was invoked). Hence, the code below configures ReflexD appropriately:

```
ReflexD.handleConsistency("Manager", new SlaveToMasterPolicy());
```

Consistency policies in ReflexD are customizable on a per-method basis (allowing some methods to be executed locally while others remotely), but in the current case, all calls from a Worker to the Manager should be dealt with in the same way: by forwarding the call to the original manager. With the above configuration, embedded in the definition of the aspect, we ensure that the tasks are obtained from the host where the manager resides in and that the result submission is forwarded there.

4.3 Discussion

In this section we have reviewed an implementation of the N-Queens problem with a distributed version of EPTA using ReflexD. As expected, it was possible to turn the non-distributed EPTA implementation into a distributed one with no modification to the EPTA code at all. All that is needed is to express the deployment of workers and their communication as an aspect in ReflexD. This difference with the ProActive solution is, from a software engineering point of view, fundamental: we have isolated the distribution aspect from the original problem and even more, implemented it as an independent module, without having to change a single line of the original EPTA code. Just applying the aspect embedding the remote instantiation and the specification of the consistency policy of the manager turns the original EPTA code into a distributed version, runnable on a Grid.

5. Performance Evaluation

In this section we explain the benchmarks we run and the cluster configuration we used. We then present the results we obtained from benchmarking the two implementations, and compare them.

[2]The Remote Consistency Framework of ReflexD is described in details on the Reflex website. http://reflex.dcc.uchile.cl/.

5.1 Benchmark Setting

Our experiment consisted in comparing the performance, in terms of time, of ProActive versus ReflexD when solving the N-Queens problem. The ProActive solution is the EPTA code modified to use ProActive in order to communicate the different workers through the network. The ReflexD solution is the unmodified, original EPTA code plus the ReflexD configuration presented before.

Configuration. We ran the two algorithms in the following configuration: an Itanium 2 cluster with 15 nodes, 2 CPUs per node and 1.9 GB RAM, all of them connected through a 10 Mbit LAN and Red Hat Linux as the operating system. It is worth noticing that we took some precautions like changing the names of certain classes in order to ensure that the same amount of data were transmitted through the network (due to the Java serialization mechanism).

Benchmarks Parameters. In this experiment, we considered the M parameter equals to 16, which implies that for N equals to 20 we are resolving 22.000 tasks approximately. We chose such a small M on purpose: the more traffic we produce on the manager, the bigger will be the differences in terms of message processing time between both solutions.

In order to avoid the pollution of the measured times due to startup activity, we only took into account the time necessary to calculate the number of solutions to the N-Queens problem: it starts after all workers are initialized and when the first task is served; and finishes when the last pending task is received by the manager. In this measured time it is also included the manager activity because it is done incrementally as the solutions for the different tasks arrive. Finally, and to obtain more accurate results, we ran each algorithm 5 times, removing the best and worst result and taking as a final result the average of the remaining three.

5.2 Benchmark Results

ProActive Results. Figure 2 shows the performance of ProActive for N equals to 18, 19 and 20 *vs.* the time required (in seconds) to solve the N-Queens problem.

We can see in this graph that augmenting the number of nodes involved in the calculation of partial solutions speeds up the application: the required time is reduced to approximately the half if we go from 5 nodes to 10 nodes. Although, if we look at the gain obtained when the number of nodes reaches 15 and 20, we can see that it is marginal, only a small time percentage is reduced.

ReflexD Results. Figure 3 reveals the results of the same experiment we did for ProActive using ReflexD. Similarly to the previous case, the addition of

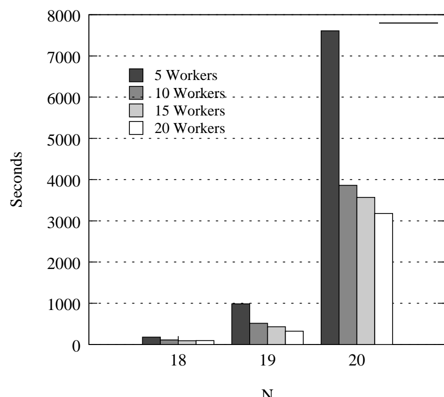

Figure 2. ProActive benchmark results.

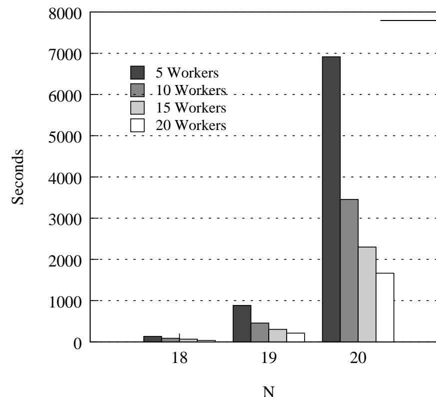

Figure 3. ReflexD benchmark results.

the nodes involved in the calculation diminishes the time required to solve the problem. First, we have a 50 percent reduction (like before), but the subsequent gains are bigger than in the previous case. We can see how the time is still reduced to the half when the number of nodes is 20 (compared to when there are 10 nodes).

Comparison of the Results. Figure 4 shows the average speedup (for N equals to 18, 19 and 20) of both solutions. ReflexD always performs better than ProActive. Although the scalability of both implementations seemed similar before, this last graph shows that the actual performance of ReflexD is superior: the speedup of ProActive is the 50 percent of the speedup of ReflexD in when 20 nodes are used.

As stated before, in this benchmark we artificially induced a bottleneck in the master by using small tasks. The difference in scalability is explained by the mechanisms used to access the master. ProActive has a lower performance because the active object programming model puts the worker's requests in the master's request queue, which requires memory allocation and modification of the queue structure. On the other hand, ReflexD uses the regular java object synchronization mechanism to access the master, which yields a more scalable approach for this kind of applications.

6. Conclusion and Future Work

Comparing ReflexD and ProActive as a whole does not make sense, because both platforms have different objectives, and rely on different paradigms. ProActive is a full-fledged middleware based on active objects, while ReflexD is a powerful platform for distributed AOP. However, it is interesting, and it has

Figure 4. Average speedup, for different values of N.

been the subject of this work, to compare solutions to the same problem (the N-Queens in this case) with both systems, both in terms of software engineering and performance. This represents an important milestone in the research agenda of validating distributed AOP.

This exercise has shown that, as expected, an aspect-oriented approach to distribution leads software engineering benefits, associated to a better modularization of concerns. It however still remains to be explored whether ReflexD can be used in a much wider range of problems for the Grid. In this respect, we feel that ReflexD has sufficient extensibility to address a wide range of problems, and most of the features of ProActive can indeed be implemented as library aspects on top of ReflexD.

The benchmarks we carried out validates the current implementation of ReflexD: it does perform reasonably well, actually better than ProActive although the gain in the considered problem is not of orders of magnitude. Still, it has practical value, all the more when considering the time it takes to compute solution of the N-Queens problem with big values on N. The overhead of ProActive, displayed by the benchmarks, is due to the supported advanced features. Mainly, the active object programming model, and the transparent futures with wait-by-necessity. Our experiments suggest that these capabilities add a performance penalty in applications where such features are not exploited. Two conclusions derive from this: first, it has to be expected, in problems that serve a large number of requests (usually problems with high communication rate), that ReflexD is going to scale better than ProActive; second, it still remains to be proven that ReflexD can support advanced features of ProActive efficiently, in cases where those features are necessary. This represents the main agenda for future work in this area.

Acknowledgments. We thank Ángel Núñez and Guillaume Pothier for their work on ReflexD, and the anonymous reviewers for their useful remarks.

References

[1] L. Baduel, F. Baude, D. Caromel, A. Contes, F. Huet, M. Morel, and R. Quilici. *Grid Computing: Software Environments and Tools*. Springer-Verlag, January 2006.

[2] L. D. Benavides Navarro. Dhamaca – an aspect-oriented language for explicit distributed programming. Master's thesis, Vrije Universiteit Brussel, Belgium, 2005.

[3] L. D. Benavides Navarro, M. Südholt, W. Vanderperren, B. De Fraine, and D. Suvée. Explicitly distributed AOP using AWED. In *Proceedings of the 5th International Conference on Aspect-Oriented Software Development (AOSD 2006)*, pages 51–62, Bonn, Germany, Mar. 2006. ACM Press.

[4] D. Caromel. Toward a method of object-oriented concurrent programming. *Communications of the ACM*, 36(9), 1993.

[5] T. Elrad, R. E. Filman, and A. Bader. Aspect-oriented programming. *Communications of the ACM*, 44(10), Oct. 2001.

[6] JBoss AOP website, 2004. http://www.jboss.org/developers/projects/jboss/aop.

[7] M. Nishizawa, S. Chiba, and M. Tatsubori. Remote pointcut – a language construct for distributed AOP. In K. Lieberherr, editor, *Proceedings of the 3rd International Conference on Aspect-Oriented Software Development (AOSD 2004)*, pages 7–15, Lancaster, UK, Mar. 2004. ACM Press.

[8] R. Pawlak, L. Seinturier, L. Duchien, G. Florin, F. Legond-Aubry, and L. Martelli. JAC: an aspect-oriented distributed dynamic framework. *Software Practice and Experience*, 34(12):1119–1148, 2004.

[9] É. Tanter. An extensible kernel language for AOP. In *Proceedings of AOSD Workshop on Open and Dynamic Aspect Languages*, Bonn, Germany, 2006.

[10] É. Tanter and R. Toledo. A versatile kernel for distributed AOP. In *Proceedings of the IFIP International Conference on Distributed Applications and Interoperable Systems (DAIS 2006)*, volume 4025 of *Lecture Notes in Computer Science*, pages 316–331, Bologna, Italy, June 2006. Springer-Verlag.

[11] É. Tanter and J. Noyé. A Versatile Kernel for Multi-Language AOP. In *Proceedings of the 4th ACM SIGPLAN/SIGSOFT Conference on Generative Programming and Component Engineering (GPCE 2005)*, volume 3676 of *Lecture Notes in Computer Science*, pages 173–188, Tallinn, Estonia, Sep/Oct 2005. Springer-Verlag.

[12] I. O. Team and ETSI. Report on ProActive User Group and Grid Plugtests, 2004. http://www-sop.inria.fr/oasis/ProActive/plugtest_report.pdf.