International Journal of Advances in Engineering and Management (IJAEM)
Volume 6, Issue 06 June 2024, pp: 120-123 www.ijaem.net ISSN: 2395-5252

Aspect-Oriented Support for Modular Parallel Computing

Dr. Niharika Prasanna Kumar, Dr. Kirankumar K
Dept of Information Science & Engineering, RV Institute of Technology and Management
Dept of Information Science & Engineering, RV Institute of Technology and Management

Date of Submission: 28-05-2024 Date of Acceptance: 06-06-2024

ABSTRACT- This paper presents Aspect Oriented Support for Modular Parallel Computing using AspectJ coding. It addresses the benefits and motivation behind Aspect Oriented Programming in the context of single-core and multi-core CPU architectures and how AOP can help solve parallelization concerns. Examples are given to demonstrate how parallelization of code can be achieved using AOP techniques. The study identifies AspectJ advantages on the modularization concept to achieve parallelism and parallelization concern. Modularization through AspectJ also provided a platform for reusing code to make other parallel applications without so much complexity that would ordinarily be faced in dealing with conventional concurrent application development using other traditional languages.

Keywords: AspectJ , Multi-Core, Parallel Computing.

I. INTRODUCTION

Parallel computing applications are on an increasing demand thanks to the rapidly growing popularity of multi-core CPU architectures, which need the power of concurrent programming to unleash high parallel processing proficiencies [1]. On the other hand, concurrent programming could lead to application execution time overheads on systems using single cores. Grid systems attempt to offer a solution to this problem but they are very sophisticated to develop because of the multi-faceted nature of computing resources, network bandwidths, and latencies. Even so, grid systems may be integrated in multi-core CPUs [2].

Conventional parallel applications face the problem of classic tangling where different modules of the application encounter parallelization challenges. This problem arises from an attempt to achieve parallelization together with core functions such as the domain logic [3]. Issues such as partitioning of work into smaller parallel tasks, executing of tasks concurrently, distributed processing of tasks, and synchronization of many different parallel accesses to data structures that are shareable have been without a solution. The code associated with these issues is highly tangled, making it difficult to read and understand, or even reuse to advance the fundamental functionality and the related parallel code of software. For this reason, conventional parallel programming has always concentrated on addressing performance concerns [4].

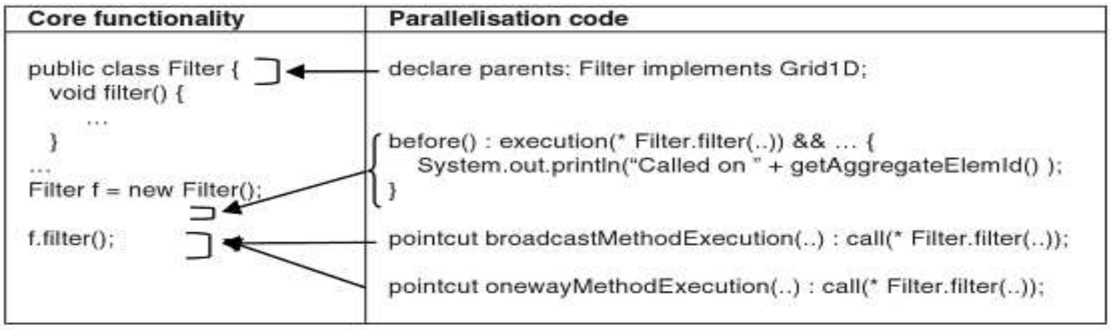
Aspect Oriented Programming, or AOP for short, bridges the gap between high performance and high level computing through effective modularization of these issues. This research presents a method that offers effective support for modular parallel computing. This approach involves developing effective parallel applications that have parallelization issues/concerns modularized into different aspects.

1.1 AspectJ: Overview

AspectJ is Java extension for backward compatibility, which supports AOP. The extension has several expressions that are encapsulated in an exceptional class called an aspect. An aspect can be used to change base code behavior through advice or extra behaviors at different join points and point cuts. Moreover, an aspect can be used to alter a class by adding new members and even parents. It provides support for two types of crosscutting composition: dynamic and static. Static crosscutting has the function of permitting type-safe changes to the static structure of an application that consist of type-hierarchy modification and member addition. An inter-type declaration is a mechanism of AspectJ that allows for adding new members such as methods, constructors, and fields. The type-hierarchy modifications of AspectJ enable making additions to interfaces and super-types that target certain classes [5].

II. METHODOLOGY

This study proposes the use of modularization of parallelization concerns using AOP in a bid to improve overall modularity and potential for code/module reuse in developing parallel applications. This, therefore, requires a base application to be open to parallelization because our center of interest is in modularizing only current parallel applications. The approach this research takes entails using conventional object-oriented techniques to achieve core functionality of an application and the support for parallelization with AOP. For this purpose, we benefit from the grouping of parallelization concerns into three categories: distribution, data partition and/or functional, and concurrence. To realize this, each aspect module will be responsible for the implementation of a single parallelization concern. The concerns are therefore easy to plug onto the core functionality of an application or even unplug them from the application.



```
Core functionality                 Parallelisation code

public class Filter {              declare parents: Filter implements Grid1D;
  void filter() {
     ...
  }                                before() : execution(* Filter.filter(..)) && ... {
...                                   System.out.println("Called on " + getAggregateElemId() );
Filter f = new Filter();           }

f.filter();                        pointcut broadcastMethodExecution(..) : call(* Filter.filter(..));

                                   pointcut onewayMethodExecution(..) : call(* Filter.filter(..));
```

Fig.1. Example of a simple application.

Objects are replicated transparently and method calls are managed by the partition module. The partition aspect is responsible for managing replicated objects and their lifecycle. Such objects are known as aspect managed objects. Common partitions are such as divide and conquer, heartbeat, and farming. Method calls are executed in respect of specific aspect managed objects already sequenced in a pipeline. Perhaps, of more concern in aspect oriented development of modular parallel computing is the concurrency module, which is responsible for invoking methods asynchronously.

In this case, caller objects continue running whereas called objects run the method requested by the caller object. In the context of Java software development language, the requested method is handled using a new method. Invocations of asynchronous methods might as well need synchronization to keep shared objects safe, circumvent data races, as well as make sure that execution occurs in a specified order. Similarly, synchronization codes gets into the concurrency aspect before proceeding to a synchronized block.

Concurrency and partition code is implemented in different modules. The logic behind this is to first create a partition module then create a concurrency module. This approach makes it possible to plug or unplug concurrency to allow for debugging and assist in circumventing inheritance inconsistency issues since it is possible to reuse partition code independently of any concurrency restrictions.

The approach this paper takes is founded on utilizing distributed objects that can be set up across many machines. Concerns with respect to object distribution are applied in their module too. This way, there are two advantages. First, concurrency and partition modules become easier to develop as they do not require addressing issues about object distribution. That is, distributed objects are developed to execute in shared memory or single processor machines. In addition, there is evident ease of switching between various foundational implementations of the core middleware to support distribution concerns like Java and MPI.

III COMPOSING MODULES

Let us consider a code segment that implements funds from a source bank account to a destination source account. The class Account implements the features of the objects sourceAcc and destinationAcc, which represent the where the funds will move from and to, respectively:

```
void fundsTransfer(Account sourceAcc, Account destinationAcc, int transferAmount) throws Exception {
if (sourceAcc.getAccountBalance() < transferAmount)
throw new InsufficientFundsException();
sourceAcc.fundsWithdraw(transferAmount);
destinationAcc.deposit(transferAmount);
}
```

The following example is an attempt to handle the considerations or concerns in the above implementation. First, for an application of this kind to work, it requires proper security checks necessary to conduct verification of details such as current user and his/her authority to do a funds transfer involving the selected accounts. Also, this application needs some way of connecting to a transaction database. Essentially, transaction details for funds transfer require to be logged in for later audit or query resolution. In response to this, the following code snippet attempts to address these issues:

```
void fundsTransfer (Account sourceAcc, Account destinationAcc, int transferAmount, User customer, Logger logger) throws Exception {
logger.info("Transferring funds…");
if (!isUserAuthorised(customer, sourceAcc)) {
logger.info("Customer has no authorization.");
throw new UnauthorisedUserException();
}
if (sourceAcc.getAccountBalance() < transferAmount) {
logger.info("Not enough funds.");
throw new InsufficientFundsException();
}
sourceAcc.fundsWithdraw(transferAmount);
destinationAcc.fundsDeposit(transferAmount);
database.commitChanges(); // Atomic process.
logger.info("Transaction was successful.");
```

Unfortunately, as the example above shows, security and database connectivity issues have been added to the code segment but this has resulted in the code being entangled in a manner that core functionality or business logic concern has been mixed with the other concerns thereby yielding what is popularly known as cross-cutting concerns. This approach is not recommended because of poor reusability of code, if any. For instance, changing security aspects on this code requires great effort because security code is appearing in several sections of the code.

To solve this problem Aspect Oriented Programming comes in handy. Using AspectJ, for example, to solve this problem would result in leaner code that is easier to understand, easier to debug, and promotes reusability. See below:

```
aspect Logger {
void Bank.fundsTransfer(Account sourceAcc, Account destinationAcc, int transferAmount, User customer, Logger logger) {
logger.info("Transferring money…");
}
void Bank.getFundsBack(User customer, int transactionId, Logger logger) {
logger.info("Customer requested funds back.");
}
// Supplementary cross-cutting code.
}
```

What the above code does is that with modular parallel programming using AspectJ, for instance, the security concerns in the first code segment are removed from the various sections or lines of code and written in a single independent module known as an aspect. For instance, the security aspect in the last (previous) code shows that code for user verification is done all from one module. The logging aspect is well taken care of from this single module. Therefore, AOP indeed proves to work like a debugging or user-level tool.

IV. CONCLUSION

This research report has discussed the Aspect Oriented method for supporting modularization of parallelization concerns. The concerns discussed include object partitioning, object distribution, and concurrency management. The study exploited the benefits of AOP including the technology’s ability to exert control based on the higher compositional proficiencies of AOP to achieve improved potential for code or module reuse from these parallelization concerns.

Putting together non-reusable aspects is achieved through the exploitation of aspect oriented programming capabilities. On the other hand, this study has not attempted to derive a sufficient model to put together reusable aspects and this may be a topic for future studies.

REFERENCES

[1] D. J. Pearce, M. Webster, R. Berry, and P. H. J. Kelly, "Profiling with AspectJ," Softw. Pract. Exp., vol. 37, no. 7, pp. 747-777, 2007.

[2] T. B. Ionescu, A. Piater, W. Scheuermann, and E. Laurien, “An aspect-oriented approach for the development of complex simulation software,” The Journal of Object Technology, vol. 9, no. 1, pp. 161–181, 2010..

[3] C. A. S. D. Cunha, Reusable aspect-oriented implementations of concurrency patterns and mechanisms [M.S. thesis], University of Minho, Braga, Portugal, 2006.

[4] T. B. Ionescu, A. Piater, W. Scheuermann, and E. Laurien, “An aspect-oriented approach for the development of complex simulation software,” The Journal of Object

Technology, vol. 9, no. 1, pp. 161–181, 2010.

[5] Sérgio Soares , Paulo Borba , Eduardo Laureano, Distribution and persistence as aspects, Software—Practice & Experience, v.36 n.7, p.711-759, June 2006