Paradigma: A Distributed Framework for Parallel Programming

Sofien Gannouni, Ameur Touir, and Hassan Mathkour
College of Computer and Information Sciences, King Saud University, Saudi Arabia

Abstract: *Recent advances in high-speed networks and the newfound ubiquity of powerful processors have revolutionized the nature of parallel computing. It is becoming increasingly attractive to perform parallel tasks on distant, autonomous, and heterogeneous networked machines. This paper presents a simple and efficient new distributed framework for parallel programming known as Paradigma. In this framework, parallel program development is simplified using the Gamma formalism, providing sequential programmers with a straightforward mechanism for solving large-scale problems in parallel. The programmer simply specifies the action to be performed on an atomic data element known as a molecule. The workers compete in simultaneously running the action specified on the various molecules extracted from the input until the entire dataset is processed. The proposed framework is dedicated for fine-grained parallel processing and supports both the Simple program multiple data and multiple program multiple data programming models.*

Keywords: *Distributed systems, parallel programming, gamma formalism, single program multiple data, multiple program multiple data.*

Received March 5, 2015; accepted March 9, 2016

1. Introduction

The parallel solution of a large-scale computing problem consists of dividing the problem into independent tasks, distributing the tasks over the processes, dividing the data among the processes, and coordinating the various processes that are running simultaneously. The processes that execute the tasks are commonly known as workers. It is becoming increasingly attractive for the workers to operate on distant, heterogeneous, and autonomous machines that are connected through a network [15, 31]. Currently, only highly skilled programmers have the expertise to write parallel code for distributed machines. Although most sequential programmers are aware of the performance gains attainable through distributed parallel programming, the technologies involved are often prohibitively complex and require the mastery of a new and difficult programming style.

Various Parallel Programming Frameworks (PPFs) have been proposed to make parallel programming less complex and more widely accessible [11]. A framework can be treated as a set of components that constitute a puzzle in which certain pieces (the specific computations to be performed) are missing and must be provided by the end-user [23]. There is competition among the various frameworks on the market to decrease the effort required from the end-user by reducing the complexity of the missing pieces and providing transparency in the parallelism. Although existing PPFs require "little" work from the end-user, they nevertheless rely on his or her expertise in solving large-scale problems in a parallel fashion.

In this paper, we present a new simple and efficient framework for parallel programming known as Paradigma. Paradigma is an easy-to-use, reliable distributed framework for parallel programming that is dedicated for fine-grained parallel processing and supports both the Single Program Multiple Data (SPMD) and Multiple Program Multiple Data (MPMD) programming models. This framework reduces the complexity of parallel program development by adopting a programming technique known as the Gamma formalism. In the Gamma formalism [3, 4], a computation is described as a form of chemical reaction between various individual data elements. Parallelism is achieved by simultaneously performing the reaction on every data element, and the processing terminates when no further reactions are possible. We experiment the benefit of using the Gamma formalism in solving various problems such as normalization of a database schema [36], Quad tree and PM-Quad tree spatial index structures implementation [37]. Several attempts to implement the Gamma formalism were performed in order to simplify the parallel programming. Indeed, [36, 37] have adopted a centralized approach to implement their frameworks. Whereas, recently, [27] has developed the Gamma formalism on a cluster machine using International Business Machines (IBM) tuple space middleware. This implementation was extended by developing a parser to transfer the Gamma programs to Nvidia's Compute Unified Device Architecture (CUDA) architecture [25, 26].

We have developed a fully distributed sharing nothing framework for parallel programming based on

the Gamma formalism called Paradigma. Herein, we focus on describing the architecture of Paradigma. The remainder of this paper presents the main characteristics of the framework, describes the reference architectures of Paradigma and its various components, and explains how the proposed framework supports the SPMD and MPMD programming models.

2. Related Work

Several dedicated frameworks for parallel programming have been proposed in the literature, and we describe the most important ones here.

- Cactus [3, 4, 18] is an open-source environment that enables parallel computation on many different architectures. Thus, using Cactus, applications can run on clusters or supercomputers.
- PetSc [9, 13] is a library that is designed for parallel programs. This library provides an abstraction layer above MPI that enables users to focus on writing parallel programs rather than concerning themselves with low-level Message Passing Interface (MPI) operations.
- Cilk [12] is a C-based runtime system that supports the multithreaded parallel programming paradigm. Cilk includes a programming language that is an extension of C with additional primitives for expressing parallelism. The Cilk runtime system maps the expressed parallelism into parallel execution. Using these primitives, Cilk allows a function that calls another function to continue running simultaneously with the callee function.
- Swarm [24] (SoftWare and Algorithms for Running on Multicore) is an open-source portable parallel programming framework. Swarm provides basic features for multithreaded programming, such as synchronization, control and memory management, and collective operations.
- Condor [5, 20, 28, 35] is a meta-computing environment that resides on a cluster of machines (called a Condor pool) and observes their characteristics, such as the load average. In the Condor environment, idle machines are utilized for a significant fraction of the time. Users submit their jobs to Condor, which subsequently runs the jobs on idle machines, monitors their progress, and informs the user upon completion. When the owner of a machine returns, Condor either suspends or kills the job running on that machine.
- Master-Worker (MW) [19] is a framework that enables users to parallelize their applications in a straightforward manner using the Condor meta-computing environment. A large computation is divided into a number of tasks to be executed in parallel. The MW framework works with Condor to assign processing units (machines) to these tasks, handle the communication, reassign tasks if their current machines fail, and manage the global parallel computation.

Numerous recent parallel processing frameworks are Java-based. This is due to the fact that Java is an object-oriented, platform-independent, secure programming language that includes various ready-made packages for communication. Most of these frameworks have attempted to add layers, components, and/or services over the pre-existing technologies to overcome the limitations of Java in parallel processing. In the literature, the existing Java-based frameworks for parallel processing are usually classified into two main approaches:

1. Native MPI wrappers.
2. Pure Java implementations. We classify these frameworks into five main approaches.

- The first approach utilizes Java as a wrapper for existing MPI implementations. Prototypes such as mpiJava [8] and Java/DSM provide MPI communication by calling native methods (in C, C++, or Fortran) using the Java Native Interface (JNI). The major drawbacks of this approach are its lack of portability and interoperability.
- The second approach provides a portable message-passing implementation because the entire library is developed in Java. Prototypes such as MPJ [6, 7], Distributed Object Group Metacomputing Architecture (DOGMA) [32], Java Parallel Virtual Machine (JPVM) [33], Java Message Passing Interface (JMPI) [29], and Pure Java Implementation of Message Passing Interface (PJMPI) [39] have been proposed. This approach solves the portability and interoperability problems; however, the developed prototypes have unfortunately exhibited inferior performance compared with native MPI implementations.
- The third approach extends the Java language features to overcome the limitations of the second approach. Prototypes such as HPJava [14], JavaParty [25], Manta [30], Titanium [41], Java Object-Passing Interface (JOPI) [1, 2], and the do! [23] Have been shown to exhibit improved performance. Some of these prototypes extend certain Java features, such as inter-process communication, object serialization [21], and message passing [34, 38]. Others either provide new Java classes, enabling users to write parallel programs in a more straightforward fashion [23] or extend the Java syntax to describe how individual tasks are performed across multiple processes [14, 41].
- The fourth approach is mainly web-oriented; it uses the Java applet to execute parallel tasks. For example, Javelin [15] adopts this approach. Javelin is a web-based infrastructure for parallel computing that requires access to a Java-enabled web browser.

By simply pointing their browsers to a known URL, users can make their resources available for hosting portions of parallel computations. This is achieved by downloading and executing an applet that spawns a small daemon thread that waits and "listens" for tasks.

- The fifth approach is grid-based. Prototypes such as HPJava [14] (the latest version), Java Parallel Programming Framework (JPPF) [40], and Mpich-G2 [22] (a non-Java-based framework) adopt this approach. HPJava and Mpich-G2 extend existing parallel frameworks to allow users to run parallel programs across multiple computers. JPPF and Mpich-G2 rely on the Globus toolkit [17] for resource allocation, process creation, monitoring, control, and communication.

3. Characteristics of Paradigma

The main characteristics of the Paradigma framework are as follows:

- *Distributed parallel processing*. The parallel processing is performed in a fully distributed way. The framework relies neither on a central component to manage and synchronize the global processing nor on shared memory for exchanging data.
- *Loosely coupled*. The built-in knowledge that must be exchanged between the interacting components is minimal. Each component requires only the address of a directory service (registry) to join the framework and communicate with the other components.
- *No parallel programming*. The framework relieves the user from details of the communication between the framework components and from details of parallelism. The user can therefore concentrate his or her effort on the logic of the problem rather than focusing on aspects such as the splitting and gathering of data or its parallel processing.
- *Scalable*. To improve the responsiveness and performance of the framework, additional processing power can be added at run time. In addition, specific computations required to solve the problems can be entered by the users into the framework at run time.

4. Overview of the Gamma Formalism

The Gamma formalism [10, 11] is a programming model that is designed to make parallelism more accessible. Gamma allows the user to specify a task as an action on a data element (known as a molecule) that satisfies a criterion known as the reaction condition.

Parallelism is achieved by performing the action simultaneously on every molecule satisfying the reaction condition. Molecules are generated either from the data provided by the user (as input) or from the data produced by the action. The processing terminates when no further actions are possible, that is, when no remaining data elements satisfy the reaction condition.

5. Terminology and Specifications

- A worker (ω) is a processing unit.
- An atom is an elementary data item.
- A data set (Ω) is a set of atoms.
- A molecule (η) is a subset of a data set Ω.
  - The molecular cardinality of a molecule η, denoted by $|\eta|$, is the exact number of atoms that compose the molecule.
  - All molecules of the same type have the same molecular cardinality.
- A service (ς) is an action (specific computation of a user) to be performed on any given molecule η.
  - The execution of ς on the molecule η is denoted by $\varsigma(\eta)$. The result of this execution is two sets of atoms:*A* and *P* say.
    - *A* is the set of active atoms that will replace η in the data set Ω in further processing.
    - *P* is the set of *passive atoms* that are part of the final result and do not require further processing.
  - The capacity of a service ς, denoted by $|\eta(\varsigma)|$, is the minimum molecular cardinality required to run the action of the service ς.
- A molecule factory $\Psi(\Omega, c)$ extracts, whenever possible, a molecule η from Ω that satisfies $|\eta| = c$ where c is the molecular cardinality of η. Consequently, the size of the set Ω is reduced by the size of the set of atoms that composes η.

$\Omega' = \Psi(\Omega, \varsigma)$ Such that $\Omega' = \Omega - \eta$.

- Atoms integrator $\varphi(\Omega, X)$ inserts the set of atoms X into the data set Ω.
- A Gamma machine is a set of interconnected workers running a set of services. The machine is denoted by M(W,L,S) where:
  - W is a *nonempty* set of workers.
  - *L* is a set of links connecting the workers. A link is defined as a connected pair of workers, and L defines the network topology of the connected workers.
  - *S* is a set of *services*.
- A *task* $\tau(\varsigma, \Omega)$ is the input to the Gamma machine. A task is composed of a service ς and a data set Ω.
  - $\varsigma(\tau)$ is the *service* ς that is associated with τ
  - $\Omega(\tau)$ is the *data* set Ω that is associated with τ
- $\Gamma(\tau, \mathrm{M})$ denotes the parallel processing of τ on the machine M; it refers to the simultaneous execution of $\varsigma(\tau)$ by the various workers $\omega \in W$ of the

machineM on distinct molecules η, extracted by $\Psi(\Omega(\tau),\varsigma(\tau))$.

$$\Gamma(\tau,M): \bigcup_{\omega \in W} \Gamma(\tau,\omega) \quad (1)$$

$$\Gamma(\tau,\omega) = \bigcup_{m \subset \Omega(\tau)} (\varsigma(\tau))(m) \quad (2)$$

denotes the multiple execution of $\varsigma(\tau)$ on the worker ω with various molecules η extracted by $\Psi(\Omega(\tau),\varsigma(\tau))$:

$$\Gamma(\tau,M): \bigcup_{\omega \in W} \Gamma(\tau,\omega) = \bigcup_{\omega \in W} \bigcup_{m \subset \Omega(\tau)} (\varsigma(\tau))(m) \quad (3)$$

6. Paradigma System Architecture

Paradigma is a reliable distributed framework for parallel programming that is dedicated for fine-grained parallel processing that supports both the SPMD and MPMD programming models. As depicted in Figure 1, Paradigma consists of the following four components:

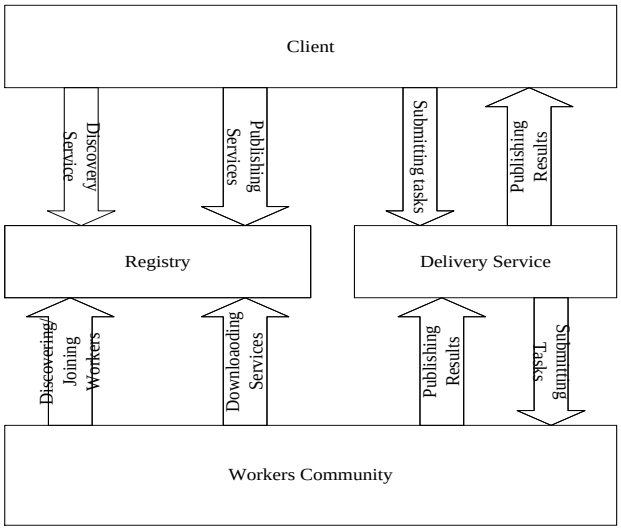

Figure 1. Paradigma reference architecture.

- *Client*. The client allows the user to

1. Define and publish his or her own services.
2. Discover existing services.
3. Submit tasks.
4. Receive the related results of the tasks in an asynchronous manner.

- *Gamma Machine*. As defined above, the Gamma machine is composed of a set of workers that are interconnected within a network of a given topology. These workers collaborate to accomplish the tasks submitted by the user and generate their results.
- *Delivery Service Mediator* (DSM). The DSM acts as a broker between the client and the Gamma machine. This component is responsible for delivering the tasks submitted by the client to the workers and transferring the results of the tasks to the client. Each client is assigned a DSM when it starts.
- *Registry Service*. The registry service is a repository in which all workers and services of the Gamma machine and delivery service mediators are recorded.

6.1. The Client

The client is mainly responsible for submitting the tasks of users and receiving their related results. This component provides a set of tools to handle the communication and synchronization between the components of the framework so that the users need not be concerned with these details. The client allows the users to specify and publish their own services and discover and utilize the already existing services, Figure 2 illustrates the client architecture.

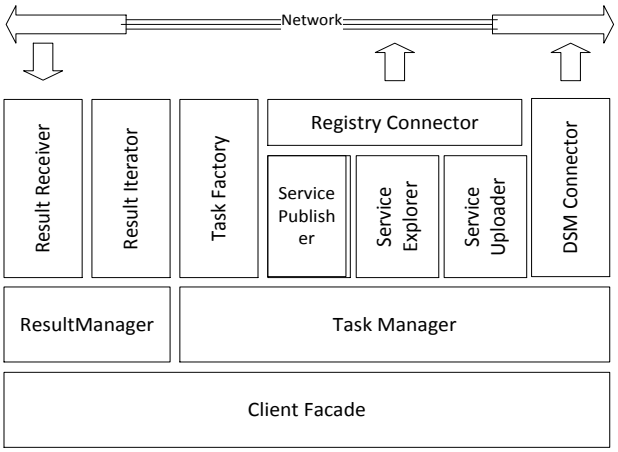

Figure 2. Client architecture.

As shown in Figure 2, the client consists of the following components:

- *The connectors*. There are two connectors: the registry connector and the DSM connector. These connectors provide remote access to the registry service and to the DSM assigned to the client. The registry connector is used basically for the publication and discovery of services.
- *The task manager*. This component is primarily responsible for defining and submitting the tasks of users and specifying and uploading the relevant services. The task manager consists of a service publisher, service uploader, service explorer, and task factory.
- *The service publisher*. publishes XML documents that describe the services deployed by the user (name, purpose, approach, context, input/output description, etc.,).
- *The service uploader*. uploads the services to the registry.
- *The service explorer*. enables the discovery of services that are already published in the registry.
- The task factory generates the tasks of the user to be processed by the Gamma machine.
- *The result manager*. This component consists of a result receiver and a result explorer. The result receiver receives the results of the submitted tasks in an asynchronous manner, while the result-

explorer browses the results and checks them for readiness.

- *The client façade*. This component ensures that the other three components of the client are well configured. The client façade may be viewed as the Paradigma user/application interface that encompasses all of the client components.

6.2. The Registry Service

The Registry Service (RS) is considered to be the entry point for clients and workers. The RS stores useful information that enables the various components of Paradigma to communicate, collaborate, and complete the tasks submitted by the users. The RS is composed of a workers repository, a services repository, and a mediator's repository.

6.3. The Delivery Service Mediator

The Delivery Service Mediator (DSM) acts as a broker between a client and the Gamma machine. Each client is assigned a DSM that is created when the client starts. The DSM

1. Forwards each received task to a worker selected randomly from the workers repository.
2. Receives the results from the workers.
3. Transfers the results back to the client.

Figure 3 illustrates the DSM architecture.

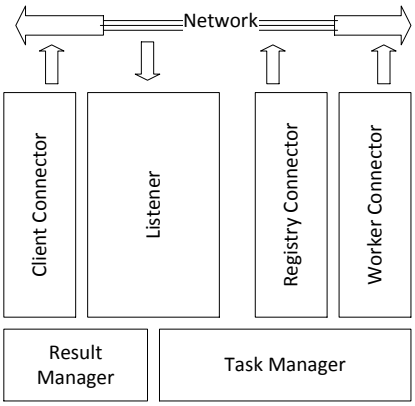

Figure 3. DSM architecture.

- The following three connectors:
  - *The registry connector*, which enables communication with the registry service.
  - *The client connector*, which transfers the results back to the relevant client.
  - *The worker connector*, which forwards the tasks to a worker of the Gamma machine.
- *The listener*. This component facilitates the receipt of tasks from the client and their corresponding results from the Gamma machine.
- *The result manager*. Because multiple tasks may be processed simultaneously and fragments of the results are often received in a random fashion, the result manager is responsible for assembling the fragments of a given result. When a result has been received in its entirety, it is transferred back to the client.

6.4. The Worker

The community of workers is considered to be the core unit of the Gamma machine. This component comprises the backend layer that constitutes the processing power of Paradigma. The worker community is composed of a scalable set of workers running on distant, autonomous, and heterogeneous interconnected machines. The workers are responsible for processing the tasks submitted by the various DSMs. They simultaneously run the expected services on disjoint portions of the data that are extracted fromthe received tasks. When a worker ω receives a task τ , it:

1. Extracts a molecule η from the data input $\Omega(\tau)$.
2. Forwards the remaining data to its neighbours.
3. Downloads and deploys the service $\varsigma(\tau)$ if necessary.
4. Performs $\varsigma(\tau)$ on η .

A worker may participate in the processing of a given task several times by running the same service on various molecules extracted from the input data. When the entire task has been processed, each worker ω sends the obtained result $\Gamma(\tau,\omega)$ to the DSM owner of the accomplished task. Figure 4 illustrates the worker architecture.

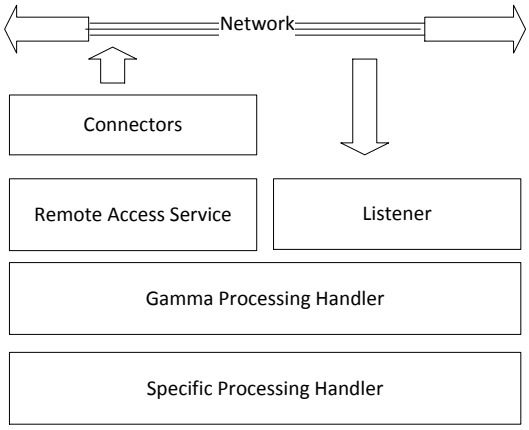

Figure 4. Worker architecture.

As depicted in Figure 4, the worker consists of the following components:

- The connectors:
  - *A registry connector*. that enables communication with the registry service (this connector enables the necessary services to be downloaded),
  - *A DSM connector*. that enables the results to be sent back to their corresponding DSMs and

  - *The worker. connectors*, which enable communication with the other workers.
- *The remote access service*. This component manages the network topology of the worker community and provides remote access to the various components of Paradigma through the connectors. The framework's underlying communication layer is designed in the form of patterns to enable various advanced network topologies, such as mesh, multidimensional torus, hypercube, or pyramid topologies, to be adopted with minimal effort. The current version of the framework fully supports ring, 2D and 3D torus network topologies.
- *The listener*. This component enables the receipt of messages (tasks, alerts, tokens, etc.) from DSMs and workers.
- *The Gamma processing handler*. This component is considered to be the core of the worker. The Gamma processing handler implements the Gamma formalism as a generic abstract machine.
- *The specific processing handler*. This component provides the Gamma processing handler with concrete problem-solving strategies.

6.4.1. The Specific Processing Handler

The Specific Processing Handler (SPH) provides uniform access to the specific services deployed by the users. For a given service ς, the SPH instantiates a single unique object of the class implementing that service. As depicted in Figure 5, the SPH consists of the following components:

- *The service loader*. This component is responsible for downloading services from the services repository of the RS. The download action is triggered whenever a worker attempts to run a non-available service.
- *The service manager*. This component handles all of the services that are downloaded by the service loader. Each service ς is made accessible through an instance, known as a Service Provider (SP), of the class implementing the service. An implementation of a service ς must override the following two methods:
  - Service Capacity (), which returns the capacity $|\eta(\varsigma)|$ of the service ς.
  - Action (η, A, P), the specific computation to be performed on any given molecule that satisfies the condition $|\eta| = |\eta(\varsigma)|$. This method implements $\varsigma(\eta)$ which produces two sets of atoms: *A* and *P*. Here, *A* is the set of active atoms that are generated by the action and require further processing, while *P* is the set of passive atoms that are generated by the action and have attained their final state.
- *The molecule factory*. This component implements $\Psi(\Omega, c)$. When a worker receives a task τ, the molecule factory extracts a molecule η from the data input $\Omega(\tau)$, with a molecular cardinality $|\eta| = c$, where $c = |\eta(\varsigma(\tau))|$.

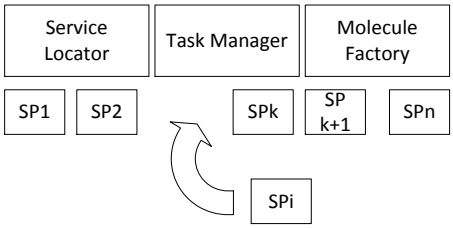

Figure 5. Specific processing handler architecture.

6.4.2. The Gamma Processing Handler

The Gamma Processing Handler (GPH) is the abstract machine that ensures the parallel processing of the tasks defined by the user. Given a received task τ, the GPH

a. Applies $\psi(\Omega(\tau), |\eta(\varsigma(\tau))|)$ to extract a molecule η from the input data $\Omega(\tau)$ that satisfies $|\eta| = |\eta(\varsigma(\tau))|$.
b. Delegates the remaining data to the neighbouring workers using the remote access service.
c. Performs $\varsigma(\tau)$ on η and obtains the sets of active atoms *A* and passive atoms *P*.
d. Delays the delivery of *P*, as a portion of the final result, until all of the workers have finished processing the task τ.
e. Applies $\phi(\Omega(\tau), A)$ to insert the set *A* into $\Omega(\tau)$.

As depicted in Figure 6, the GPH consists of the following components:

- *The processing manager*. This component determines whether the task τ must be processed locally or can be delegated depending on the status of the worker (busy or idle).
- *The local processor*. This component is responsible for the local processing of tasks τ. The local processor uses the SPH to obtain the service provider of τ and a molecule η from $\Omega(\tau)$ and triggers the service provider to take action on η. The remaining portion of $\Omega(\tau)$ is passed to the delegator. When the local processor finishes processing η, it continues working on the same task τ by extracting a new molecule η' from the available active atoms *A*. If no molecule could be generated, then the worker is set to idle.
- *The delegator*. This component enables the asynchronous submission of a data set Ω to the neighbouring workers.
- *The end detector*. This component is responsible for detecting the global termination of the parallel task

running on the distributed workers. In the end detector, the Dijkstra *et al.* [16] wave algorithm for termination detection on a ring network is extended to include 2D and 3D torus networks. The wave algorithm pays a token visit to each node and collects information regarding the status of the entire system. A full description of the algorithm is beyond the scope of this paper. However, we considered the following criteria in the specification of our termination detection algorithm:

- *Correctness*. The termination detection occurs if and only if the computation has terminated, and if the termination occurs, then it must be detected.
- *Number of detection iterations*. Detecting the global termination within a minimum number of iterations (turns) is important for the overall performance of the termination detection algorithm because it affects many factors, including the delay, resource consumption, and congestion.
- *Overhead*. The termination detection algorithm generates the minimum possible number of secondary messages.
- *Detection delay*. The termination detection delay is the time interval between the completion of the execution and the subsequent detection.

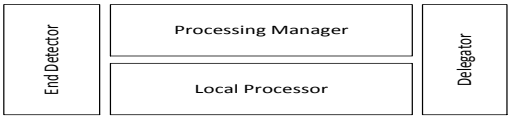

Figure 6. Gamma processing handler architecture.

7. Paradigma Parallel Programming Models

7.1. The SPMD Programming Model

In the SPMD programming model, the workers simultaneously perform the same service on different molecules. The molecules are extracted from the dataset provided by the task at hand. The task runs through all of the workers until all molecules are extracted and processed. Each worker creates an instance of the service and performs the service on the extracted molecules.

To submit a task, the user can implement his or her specific problem solution by extending an abstract class called Service. This class contains two fundamental abstract methods, action () and service Capacity (). The functionalities of these methods are described in Section 6.4.1. Once these two methods are implemented, the user must deploy the service in the framework by providing the service name and the URL of the jar file that contains the service implementation. Once these initial steps are complete, he or she can submit a task. The following tables show the various code fragments that must be implemented by the user to calculate the sum of a set of integers.

Algorithm 1 shows the description of the abstract class Service and focuses on the abstract methods that should be implemented by the user.

Algorithm 1: The definition of the abstract Service Class

```
// provided by the framework
1.  public abstract class Service {
2.   protected String serviceName;
3.   protected String serviceDescription;
4.   protected String serviceFileName;
5.   public Service(String svcName, String svcDesc,String
     svcFName) { … }
6.   public abstract void action(      LinkedList molecule,
7.         LinkedListactiveAtoms,
8.         LinkedListpassiveAtoms);
9.    public abstract intserviceCapacity();
10. }
```

Algorithm 2 shows how to implement the calculation of thesum of integers as a specific computation (a service). The sum service should be implemented as a concrete class of the abstract class Service.

Algorithm 2: Implementation of a specific computation of a user

```
    //Should be provided by the user
1.  public class SumService extends Service {
2.
3.    public void action(LinkedList molecule,
4.                       LinkedListactiveAtoms,
5.                       LinkedListpassiveAtoms) {
6.        int a, b;
7.        a =((Integer) molecule.poll()).intValue();
8.        b =((Integer) molecule.poll()).intValue();
9.        activeAtoms.add(new Integer(a+b));
10. }
11. public intserviceCapacity() {
12. return 2;
13. }
14. }
```

Algorithm 3 shows how the user should define his data set using the abstract class Data Set.

Algorithm 3:Definition of the abstract class Data Set and user defined data types.

```
    //provided by the framework
1.  public abstract class DataSet {
2.      private LinkedList<Object> data;
3.  public final LinkedList<Object>moleculeFactory(int
    cardinality) { … }
4.      public final void
    atomsIntegrator(LinkedList<Object> atoms) { … }
5.    …
6.      public final void add(Object atom) { … }
7.  public final LinkedList<Object>getData() {…}
8.  …
9.  }

    //Should be provided by the user if necessary
1.  public class MyData extends DataSet  {
2.    …
3.  }
```

Algorithm 4 shows how the user can deploy and submit the sum service as an SPMD program.

Algorithm 4: Deployment of a single service and the execution of a task asan SPMD program.

```
    //Should be provided by the user
1.  public class SPMD_Program {
2.     public static void main (String[] args) {
3.        ParadigmaClient cl  = new ParadigmaClient();
4.        DataSetmyData = new DataSet();
5.        SumService svc = new SumService();
6.
7.        for (inti=0; i< 10000000; i++)
8.              myData.add(new Integer(i));
9.
10. cl.deploy(svc);
11. Task myTask =  new Task(svc, myData) ;
12. cl.submit(myTask);
13. DataSetmyResult = cl.getResult(myTask);
14. }
15. }
```

7.2. The MPMD Programming Model

In the MPMD programming model, the workers simultaneously perform distinct tasks that perform different services. Each task runs as an SPMD program and benefits implicitly from the parallel capabilities of Paradigma.

Algorithm 5 shows an example of a client deploying distinct services and submitting several tasks for parallel processing.

Algorithm 5: Deployment of distinct services and the execution of several tasks as MPMD programs.

```
    //Should be provided by the user
1.  public class MPMD_Program{
2.     public static void main (String[] args) {
3.        ParadigmaClient cl = new ParadigmaClient();
4.        MyData1 myData1 = …;
5.        MyData2 myData2 = …;
6.        MyService1 svc1 = …;
7.        MyService2 svc2 = …;
8.        …
9.        cl.deploy(svc1);
10. cl.deploy(svc2);
11.
12. Task myTask1 =  new Task(svc1, myData1) ;
13. Task myTask2 =  new Task(svc2, myData2) ;
14. …
15. ArrayList<Task>myTasks = new ArrayList<Task>();
16. myTasks.add(myTask1);
17. myTasks.add(myTask2);
18.
19. cl.submit(myTasks);
20. ArrayList<DataSet>myResults = cl.getResult(myTasks);
21. …
22. }
23.  }
```

8. Conclusions

In this paper, we have presented a new distributed virtual parallel machine known as Paradigma.

Paradigma is composed of four components: the client, registry service, delivery service mediators, and workers. The client enables the users to submit SPMD and MPMD programs. The registry service enables the users to share their expertise in solving large-scale problems. The delivery service mediators ensure communication between the clients and workers. The workers execute the SPMD and MPMD parallel programs provided by the users. The workers are distributed and interconnected within a network. The proposed framework supports various connection topologies ranging from simple ring networks to multidimensional torus networks.

Most of existing parallel programming frameworks which rely on the expertise of the programmer to solve a large scale problem. In contrast to that, Paradigma adopts a programming technique known as the Gamma formalism which allows the programmer to concentrate his/her efforts on writing only the action (service) to be performed on a single data element known as a molecule. Paradigma automatically deploys the service on all of the available workers, which simultaneously perform the action on distinct molecules extracted from the task defined by the user.

Moreover, unlike existing frameworks, which distribute the tasks in an MPMD program to the workers for simultaneous execution with a single task assigned to each worker, Paradigma processes the tasks in a given MPMD program simultaneously as SPMD programs. The workers then compete to participate in the processing of each task. MPMD programs therefore benefit implicitly from the parallel capabilities of the proposed framework.

Acknowledgment

This work was supported by the Research Center of College of Computer and Information Sciences, King Saud University. The authors are grateful for this support.

References

[1] Al-Jaroodi J., Mohamed N., Jiang H., and Swanson D., "JOPI: A Java Object-Passing Interface," *in Proceedings of the Joint ACM-ISCOPE Conference on Java Grande*, Seattle, pp. 37-45, 2002.

[2] Al-Jaroodi J., Mohamed N., Jiang H., and Swanson D., "Middleware Infrastructure for Parallel and Distributed Programming Models in Heterogeneous Systems," *IEEE Transactions on Parallel and Distributed Systems*, vol. 14, no. 11, pp. 1100-1111, 2003.

[3] Allen G., Benger W., Dramlitsch T., Goodale T., Hege H., Lanfermann G., Merzky A., Radke T., and Seidel E., "Cactus Grid Computing: Review of Current Development," *in Proceedings 7th*

International Conference Euro-Parallel, Manchester, pp. 817-824, 2001.

[4] Allen G., Goodale T., Massó J., and Seidel E., "The Cactus Computational Toolkit and Using Distributed Computing to Collide Neutron Stars," *in Proceedings 8th International Symposium on High Performance Distributed Computing*, Redondo Beach, pp. 57-61, 1999.

[5] Bader D., Kanade V., and Madduri K., "SWARM: A Parallel Programming Framework for Multicore Processors," *in Proceedings of 1st Workshop on Multithreaded Architectures and Applications*, Long Beach, pp. 1966-1971, 2007.

[6] Baker M., Carpenter B., and ShafiA., "MPJ Express: Towards Thread Safe Java HPC," *in Proceedings IEEE International Conference on Cluster Computing*, Barcelona, pp. 1-10, 2006.

[7] Baker M., Carpenter B., and Shafi A., "MPJ: A New Look at MPI for Java," *in Proceeding of UK E-Science All Hands Meeting*, Nottingham, pp. 666-669, 2005.

[8] Baker M., Carpenter B., Fox G., Ko S., and Lim S., "MpiJava: An Object-Oriented Java Interface to MPI," *in Proceedings of International Parallel and Distributed Processing Symposium*, San Juan, pp. 748-762, 1999.

[9] Balay S., Gropp W., McInnes L., and Smith B., "Efficient Management of Parallelism in Object Oriented Numerical Software Libraries," *Modern Software Tools in Scientific Computing*, pp. 163-202, 1997.

[10] Banatre J., Fradet P., and Le Metayer D., "Gamma and the Chemical Reaction Model: Fifteen Years Later," *in Proceedings of International Conference on Membrane Computing*, Curtea de Arges, pp. 17-44, 2000.

[11] Banatre J., Fradet P., and Radenac Y., "Higher-Order Chemistry," *in Proceedings of International Workshop on Membrane Computing*, Tarragona, pp. 102-111, 2003.

[12] Blumofe R., Joerg C., Kuszmaul B., Leiserson C., Randall K., and ZhouY., "Cilk: An Efficient Multithreaded Runtime System," *in Proceedings of 5thACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, Santa Barbara, pp. 207-216, 1995.

[13] BrowneM., "The PETScLibrary: Portable, Extensible Toolkit for Scientific Computing," technicalreport,www.ichec.ie/support/tutorials/petsc.pdf, Last Visited, 2016.

[14] Carpenter B. and Fox G., "HPJAVA: A Data Parallel Programming Alternative," *Journal of Computing in Science and Engineering*, vol. 5, no. 3, pp. 60-64, 2003.

[15] Christiansen B., Cappello P., Ionescu M., Neary M.,Schauser K., and Wu D., "Javelin: Internet-Based Parallel Computing Using Java," *Concurrency: Practice and Experience*, vol. 9, no. 11, pp. 1139-1160, 1997.

[16] Dijkstra E., Feijen W., and VanGasteren A., "Derivation of A Termination Detection Algorithm For Distributed Computations," *Information Processing Letter*, vol. 16, no. 5, pp. 217-219, 1983.

[17] Foster I., "Globus Toolkit Version 4: Software for Service-Oriented Systems," *in Proceedings of IFIP International Conference on Network and Parallel Computing*, Beijing, pp. 2-13, 2005.

[18] Goodale T., Allen G., Lanfermann G., Mass´o J., Radke T., Seidel E., and Shalf J., "The Cactus Framework and Toolkit: Design and Applications," *in Proceedings of 5th International Meeting on High Performance Computing for Computational Science*, Porto, pp. 197-227, 2002.

[19] Goux J., Kulkarni S., Linderoth J., and Yoder M., "An Enabling Framework For Master-Worker Applications on the Computational Grid," *in Proceedings of 9th International Symposium on High-Performance Distributed Computing*, Pittsburgh, pp. 43-50, 2000.

[20] Goux J., Kulkarni S., Yoder M., and Linderoth J., "Master-Worker: An Enabling Framework for Applications on the Computational Grid," *Cluster Computing*, vol. 4, no. 1, pp. 63-70, 2001.

[21] Helen-Ma H. and YangL., "Improvement of Object Serialization in Java Remote Method Invocation," *in Proceedings 7th ACIS International Conference on Software Engineering, Artificial Intelligence, Networking, and Parallel Distributed Computing*, Las Vegas, pp. 35-42, 2006.

[22] Karonis T., Toonen B., and Foster I., "Mpich-G2: A Grid-Enabled Implementation of the Message Passing Interface," *Journal of Parallel and Distributed Computing*, vol. 63, no. 5, pp. 551-563, 2003.

[23] Launay P. and Pazaty J., "The Do! Project: Distributed Programming-Using Java," *in Proceedings of 1st UK Workshop on Java for High Performance Network Computing*, Europar, pp. 76-84, 1998.

[24] Leiserson C. and Plaat A., "Programming Parallel Applications InCilk," *Society of Industrial and Applied Mathematics News*, vol. 31, no. 4, pp. 122-126, 1998.

[25] Lin H., Kemp J., and Gilbert P., *Computer Engineering: Concepts, Methodologies, Tools and Applications*, IGI-Global, 2012.

[26] Lin H., Kemp J., and Gilbert P., "Computing GAMMA Calculus on Computer Cluster," *International Journal of Technology Diffusion*, vol. 1, no. 4, pp. 42-52, 2010.

[27] Lin H., Kemp J., and Molina W., "Parallel Computing in Chemical Reaction Metaphor with Tuple Space," *International Journal of Computer Science and Security*, vol. 4, no. 2, pp. 149-159, 2010.

[28] Meehean J. and Livny M., "A Service Migration Case Study: Migrating the Condor Schedd," *in Proceedings of the 38th Symposium on Instruction and Computing*, Wisconsin, pp. 456-472, 2005.

[29] Morin S., Koren I., and Krishna C., "JMPI: Implementing the Message Passing Standard in Java," *in Proceedings 16th International Parallel and Distributed Processing Symposium*, Ft. Lauderdale, pp. 1956-1962, 2002.

[30] Philippsen M. and Zenger M., "Javaparty-Transparent Remote Objects in Java," *Concurrency Practice and Experience*, vol. 9, no. 11, pp. 1225-1242, 1997.

[31] Ragab H., Sarhan A., Sallam A., and Ammar R., "Balanced Workload Clusters for Distributed Object Oriented Software," *The International Arab Journal of Information Technology*, vol. 12, no. 4, pp. 379-388, 2015.

[32] Snell Q., Judd G., and Clement M., "The DOGMA Approach to Parallel and Distributed Computing," *Scientific International Journal for Parallel and Distributed Computing*, vol. 2, no. 2, pp. 23-34, 2001.

[33] Sqalli M. and Sirajuddint S., "An Adaptive Load-Balancing Approach to XML-Based Network Management Using JPVM," *in Proceedings of 13th IEEE International Conference on Networks*, Kuala Lumpur, pp. 202-207, 2005.

[34] Taboada G., Tourino J., and Doallo R., "Performance Analysis of Java Message-Passing Libraries On Fast Ethernet, Myrinet and SCI Clusters," *in Proceedings IEEE International Conference on Cluster Computing*, Hong Kong, pp. 118-126, 2003.

[35] Thain D., Tannenbaum T., and Livny M., "Distributed Computing in Practice: The Condor Experience," *Concurrency and Computation: Practice and Experience*, vol. 17, no. 2, pp. 323-356, 2005.

[36] Touir A., Al-Athel D., and Mathkour H., "An application of Gamma Formalism to Database Design," *in Proceedings of International Conference on Computer and Communication Engineering*, Kuala Lumpur, pp. 974-977, 2008.

[37] Touir A., Al-Twairesh N., and Mathkour H., "A Gamma-Based PM-Quadtree Specification," *in Proceedings of the 9th World Multi-Conference on Systemics, Cybernetics and Informatics*, Orlando, pp. 215-221, 2005.

[38] Van Nieuwport R., Maassen J., Bal H., Kielmann T., and Veldema R., "Wide-Area Parallel Programming using the Remote Method Invocation Model," *Concurrency: Practice and Experience*, vol. 12, no. 8, pp. 643-666, 2000.

[39] WeiQin T., Hua Y., and WenSheng Y., "PJMPI: Pure Java implementation of MPI," *in Proceedings of 4th International Conference/Exhibition on High Performance Computing in the Asia-Pacific Region*, Beijing, pp. 533-535, 2000.

[40] www.jppf.org, Last Visited, 2009.

[41] Yelick K., Semenzato L., Pike G., Miyamoto C., Liblit B., Krishnamurthy A., Hilfinger P., Graham S., Gay D., Colelia P., and Aiken A., "Titanium: A high-performance Java dialect," *in Proceedings* of *ACM Workshop on Java for High-Performance Network Computing*, Stanford, pp. 825-836, 1998.

Sofien Gannouni received his Master degree in Computer Science from Paul Sabatier University (Toulouse III - France), and his PhD degree in Computer Science from Pierre & Marie Curie University (Paris VI - France). Currently, he is an Assistant Professor at College of Computer and Information Sciences, King Saud University. His main research interests include service-oriented computing, distributed computing, parallel processing, middleware grid computing and cloud computing.

Ameur Touir received his Master degree in Computer Science from Ecole National Superieure des Techniques Avancees (INSTA/Paris VI, Paris - France), and his PhD from Sup. Telecom. Paris. Currently, he is an Associate Professor at College of Computer and Information Sciences, King Saud University. His main research interests include spatial data structure and GIS, semantic web services, database.

Hassan Mathkour received his Master degree and PhD degree in Computer Science from the University of Iowa, USA. Currently, he is a Professor and former Dean of College of Computer and Information Sciences, King Saud University. His main research interests include service-oriented computing, distributed computing, artificial intelligence, bioinformatics, image processing and software engineering.