A Service Centred Approach to Concurrent and Parallel Computing

Hervé Paulino, Artur Miguel Dias, Maria Cecília Gomes, and Jose C. Cunha

CITI / Departamento de Informática
Faculdade de Ciências e Tecnologia, Universidade Nova de Lisboa
2829-516 Caparica, Portugal

Abstract. The rise of the multi-core architectural design has pushed concurrent and parallel programming, particularly the shared-memory paradigm, to the forefront of mainstream software development. However, as the number of cores increases, the distributed nature of these architectures tends to favour the scalability and modularity of the message-passing paradigm over shared-memory. Moreover, the diversity and heterogeneity of the currently available processors, namely in the layout of their memory hierarchy, is more and more driving applications to dynamically adapt their execution to the underlying runtime systems and architecture.

In this context, we propose a model that builds on the promotion of the service abstraction from the system to the language level. The goal is to provide a uniform abstraction for the construction of concurrent dynamic and scalable systems by composing services published by the application, by the runtime system or by third-parties (such as Web accessible services). The model decouples service specification from service provision, defining a two-level development methodology which, at a macro-level, addresses service interaction and coordination, and, at a micro-level, the internals of the service providers. This approach provides a good framework for: a) expressing different types of parallelism at different levels, b) accessing services offered by the runtime system to tune or steer the execution, c) dynamically swap or adjust the service providers available for a given application, and d) treat both local and remote resources uniformly at the language level. This paper presents an overview of the proposed model, illustrating its features by modelling some known parallel programming paradigms, namely SPMD, Pipeline and Master/Worker.

Keywords: Concurrent and Parallel Programming; Service-oriented Computing

1 Introduction

Modern applications now encompass multiple software components with complex interactions and mapped onto heterogeneous distributed infrastructures. As a consequence applications exhibit different levels of concurrency, parallelism, and distribution.

In order to provide adequate functionalities for application programming and development with the minimal required knowledge about the internals of the computer mappings and management, multiple approaches identify two distinct levels
of abstraction: i) Resource abstractions, for logical access to types of remote, heterogeneous, parallel and distributed resources, such as computation, storage, network, and to any other kind of system provided functionalities that can be made accessible via some interface e.g. data sets and information repositories, or specialized physical devices; and ii) User abstractions, aiming to provide higher level concepts and interfaces to ease application programming and development tasks. Examples can be found, e.g. in the concepts of parallel virtual machine [3,9], or in Grid middleware services (resource management, discovery, scheduling, information registry and querying, authentication, security, management of computation, data and communication), as well as in the Grid user-level abstractions for single-sign on and access to pools of logical resources. This also applies to the service concepts in the Web services standards, with useful functionalities available through middleware-level approaches. Similar analysis applies to the Cloud, where service concepts were pushed forward and promoted to first-level entities.

Unfortunately, there was less progress concerning the adequate higher level programming models. Overall, there is still a big gap from the adequate programming abstractions for application development and the existing service-oriented interfaces to access the parallel and distributed infrastructures of varying performance, scale and behaviour.

As applications became pervasive due to the Internet, they have increasingly been designed as compositions of logically distributed entities (processes, objects, agents, components, services, and so on) that interact and cooperate, and are spatially distributed and mapped to heterogeneous hardware architectures. Each of these entities is usually designed as a local name space, with a specific internal computation model (cf single- or multithreaded), and interacts with others through logical communication abstractions, shared- or distributed memory. This leads to a two-level view of a programming model, where at a global macroscopic level the application is viewed as a collection of individual logically distributed and interacting entities, and at a local microscopic level the internal computation model of each individual entity is considered. On the bridge from the global to the local level, lies the interface defined by each entity, allowing to decouple the identification of the adequate abstractions and functionalities, which are exposed to the global view of the application, and on the other hand to consider their implementation, internal to each entity, as a separate concern, which can be hidden to the global view.

Attempts to capture this view can be found in object-oriented and component-oriented programming [2,3,6], and in service-oriented computing [15]. However, as we have discussed above, there is still a big gap from the programming interfaces provided at middleware level, to access the corresponding services, and the programming abstractions that should be offered to the application programmer. This motivates our proposal for a unified approach, encompassing the above concerns, centred on the concept of service.

For this purpose, we propose a model that promotes the service abstraction from the system to the language level, as a mean to construct concurrent dynamic and scalable systems that can easily compose services from both the application and the system realm. The model decouples service specification from service provision, growing from the two-level development methodology to: (a) express different types
of parallelism at different levels; (b) accessing services offered by the runtime system to tune or steer the execution; (c) dynamically swap or adjust the service providers available for a given application, and (d) treat both local and remote resources uniformly at the language level.

The following section presents the model’s foundations and constructs. Section 3 demonstrates how the model can be used to code some of the most well known paradigms of parallel programming; Section 4 compares our work to others in the field, and; finally Section 5 presents our concluding remarks.

2 The Service Model

Services are autonomous, platform-independent entities that can be described, published, discovered and loosely coupled in novel ways [15]. The proposed model builds on these characteristics to express code suitable for concurrent execution. Such code is encapsulated into operations that compose an active software module, the service, which defines the application building block.

The model supplies two main programming abstractions: service: a set of operations that specify the interface of a service, and; service provider: a concrete implementation of a service interface. This distinction decouples service specification from service provision, defining a two-level development methodology which, at a macroscopic level, addresses service interaction and coordination, and, at a microscopic level, the internals of the service providers.

An application is designed as a set of service providers that offer functionalities through an interface, encapsulating both state and implementation. Providers may be active, in the sense that they perform computations autonomously, or passive, only responding to external stimulus (invocation of service operations). There is no centralized initial point of execution, active providers begin their concurrent execution automatically from the moment that they are deployed. Moreover, communication is established through the invocation of service operations, as is typical in service-oriented computing. Invocation is decoupled from execution, a characteristic of actor and active object models [11] that enables the concurrent execution of operations within services. The conjunction of both these features provide functional parallelism by design. An application is inherently parallel, offering an good tool for the development of concurrent code by non-experts.

Service clients bound themselves to abstract services and not to concrete providers, following a contract-based binding mechanism based on the service’s interface. The choice of the actual providers is postponed to the deployment stage, providing the means to tune the application according to the target architecture, hence enhancing modularity and scalability. Such behaviour can be observed at the language level of Figure 1. The term client (or pure client) refers to software modules that require functionalities offered by other services but do not offer a functionality themselves.

The available libraries of service providers can be interpreted as a repository of the services available to the application. The find and bind stages of the service-oriented computing paradigm are mostly performed at deployment-time, when services are mapped onto providers. Nonetheless, to cope with the dynamicity of today’s applications, new providers and even new services can be added at runtime.
The use of a uniform abstraction for both application and system services enables applications to make use of both transparently. The runtime system can control the degree of interference it allows from the upper layers of the software stack by adjusting the number and nature of the services it publishes. Figure 1 illustrates a scenario where an application resorts to four services, being that one is natively implemented by the runtime system.

2.1 The Macroscopic (Inter-service) Level

Service Definition. Service interfaces establish a protocol between service providers and clients. They specify the set of operations that must be implemented by the former and are accessible to the latter. As is common in service-oriented computing, and contrasts with more coupled paradigms, such as the object-oriented, only service interfaces define types - providers are not addressable in a program.

We begin by presenting a simple example of a service that specifies a set of operations over a list (Listing 1). All examples resort to a Java-like syntax.

The interface induces a stateless interaction model, not resorting to any internal state whatsoever. This has the advantage of performing well in large-scale applications, but imposes several limitations. Consider, for instance, a Producer/Consumer example with multiple producers, each with a queue accessible to multiple consumers. To express such behaviour with stateless interactions is, at least, complex. We have therefore introduced support for multi-party stateful sessions, as a mean to express an interaction requiring state preserving between a provider and multiple clients. The state associated to these sessions must be stored at the server side of the interaction [12]. Note, however, that it is only preserved as long as the session is running. Once the session concludes, the state is eliminated.

Sessions are completely orthogonal to the service itself being stateless or stateful. When in the presence of multiple providers of a given service, the runtime system can easily map a given session to a specific provider. Thus, multiple sessions can be mapped into multiple providers, enabling scalability. Stateful services do not support such modular mapping, since, in order for all providers to respond the same result to a given operation invocation, the state must be shared by all providers, which degrades scalability.
Proceeding with the Producer/Consumer example. The modelling with multi-party sessions is quite simple - each producer queue is mapped into a session available to all interested clients. The service’s interface is depicted in Listing 2. Sessions constructors and operations are delimited by a session block, being that the constructors are used by clients to configure the session’s parameters. The service’s implementation will be given ahead, in Subsection 2.2.

The Application Configuration File The application configuration file specifies the mapping of services into providers. The application loader receives a set of pure clients, the repository of service providers, and an configuration file that indicates which service providers to be loaded. The configuration file is akin to a choreography in the sense that it specifies the services, and respective providers, that will compose the application and communication channels between them. Naturally, the successful loading of the application requires the fulfilment of all provider dependencies. An application file is declarative and contains rules of the following types:

**service ServiceID : List of ProviderID(List of values)** - deploys a set of providers of the given service. Each provider can be annotated with mapping information, such as the number of instances and the location of (each) instance. As will explained in Subsection 2.2 providers can be parametric, hence the list of input values.

**session SessionID: newsession ServiceID(List of values)** - establishes a session with a given service and stores it SessionID.

**client: List of ProviderID(List of values)** - deploys a set of parametrizable clients.

An example of the configuration file will be presented in Subsection 2.2.

### 2.2 The Microscopic (Intra-service) Level

Service definitions are materialized by service providers. Internally, a provider is composed of a private addressing space, which may be partitioned among several sessions, and a set operations that execute asynchronously within that state. To this extent, the internals of a provider must handle two fundamental issues: the actual implementation of service operations, which may use provider-wide variables, and the coordination of the execution of these operations.

**Service Implementation** The structure of a provider follows the one of its service interface, complemented with global state variables (bound or not to a session) and private auxiliary operations, for code structuring’s sake. Active services must implement method `void main()`, the one that will be automatically executed as soon as they are deployed.

**Basic Communication:** The invocation of operations borrows the method selection syntax from object-oriented languages, but its semantics relies on implicit futures. Any invocation implicitly returns a future, allowing the invoker to proceed its execution until the result of the operation is needed. An access to the variable
holding the result will automatically block the invoker’s thread until the value is available. Nonetheless, there are situations where a synchronous invocation is preferable, for instance to cope with side effects. For that purpose, we define the sync keyword.

Stateful Communication: Sessions are requested through the newsession construct. The operation returns a communication channel compliant with the session’s type, denoted by S.session, where S is the identifier of the target service. Session termination is delegated on a runtime garbage collector, and thus not handled at the model level.

Coordination: The concurrent execution of operations within a service’s address space brings to the table the fundamental problems of managing shared states. How these concurrency concerns are expressed is orthogonal to the remainder of the model, and can be delegated in the host language.

We are however researching on the separation of the concurrency and functionality concerns. The goal is to segregate the concerns of concurrency, namely process synchronization from the program’s logic. Such allows specifying and reasoning about the concurrency requirements of a service implementation in terms of method composition, despite their concrete implementations. This work grows from Dijkstra’s guarded commands [7] and Campbell’s path expressions [4].

In a simplified version, service operations are guarded by boolean expressions and a path annotation indicates which operations must execute atomically. For the Queue provider of Listing 3 we specify the following expression: put | get | (isFull isEmpty) which must be read as: put or get or an arbitrary sequence of concurrent executions of isFull and isEmpty.

Data Parallelism: As stated in the introduction to this section the model provides task parallelism by design. Nonetheless, the data parallelism dimension can also be explored in the implementation of service operations. The invocation of an operation in this setting is executed by multiple concurrent execution flows, each operating on a different partition of the original dataset. These flows run in conformity to a variation of the SPMD execution model, which we have baptised as Single Operation Multiple Data. Figure 2 illustrates the whole process that follows a Distribute-Map-Reduce parallelism strategy that comprises the following stages:

- Distribute: applies a distribution to a given input parameter to obtain a collection of elements of the same type as the given parameter.
- Map: applies the operation to each element of the previously created array.
- Reduce: applies a reduction to combine all the results generated by the previous stage to compute the final result.

This behaviour is expressed through a set of annotations that specify the data distribution and reduction polices to apply. Both distributions and reductions are native services that specify the prototype for the implementation of distribution and reduction policies. Segregating them to services allows for reuse and distinct implementation strategies, e.g. parallel or sequential. Subsection 3 will present an application example.

2.3 Putting it All Together - The Producer/Consumer Example

Consider a simple instance of the Producer/Consumer example composed of the producer and the consumer (both clients), and an intermediary queue (provider
of service `Queue`). In order to share the same queue, both the producer and the consumer must belong to the same session. References to sessions can be passed among providers. For instance, the producer can create the session and allow the consumers to interrogate it, in order to retrieve the reference. For that purpose, it is obliged to implement a service to feature a publicly accessible operation.

To avoid such approach we resort to the faculty of specifying communication channels in application configuration files, enabling the producer and consumer clients to receive the session through a parameter. Listings 3, 4 and 5 depict, respectively, a provider for the `Queue` service, the producer and resulting application configuration file.

3 Modelling Known Parallel Programming Paradigms

This section illustrates how the proposed service-based model can be used to code known paradigms of parallel programming, namely: Single Program Multiple Data (SPMD), Pipeline and, Master/Worker\(^1\).

**Single Program Multiple Data (SPMD)** The SPMD paradigm maps seamlessly in the SOMD support. We substantiate such claim with an adaptation of an implementation of the classic Mandelbrot Set (Listing 6\(^2\)). Annotations `dist` and

---

1 Due to space restrictions we cannot present the entirety of the code, referring the reader to [http://asc.di.fct.unl.pt/serco/europar2012](http://asc.di.fct.unl.pt/serco/europar2012)

2 Original code retrieved from [Core Java Performance Examples](http://code.google.com/p/core-java-performance-examples/).
**reducewith** denote, respectively, the application of distribution and reduction policy. Parameters `out` and `Cib` are distributed according to standard array partitioning (explicit policies can be defined), and the partial results are reduced by the `reduce` operation of `MandelReducer.session` - Listing 7 depicts the native `Reduction` service. Implementations of `MandelReducer` combine the partially computed results.

```
service MandelBrot { void runSimulation(int n, String outputFile); }
provider MBProvider implements MandelBrot {
    out = computeMandelArea(out, Crb, Cib);
    [...]
    byte[][] computeMandelArea(dist byte[][] out, double[] Crb, dist double[] Cib) reducewith MandelReducer.session() {
        for (int i = 0; i < out.length; i++)
            for (int xb = 0; xb < out[0].length; xb++)
                out[i][xb] = getByte(xb * 8, i, Crb, Cib);
        return out;
    }
    byte getByte(int x, int y, double[] Crb, double[] Cib) {... }
}
```

Listing 6. The MandelBrot service and a provider

```
service Reduction<T> { T reduce(T[] partials); }
```

Listing 7. The Reduction native service

**Pipeline** A pipeline is a sequence of connected processing stages, usually organized in a linear fashion, where the output of one stage is passed to the input of the next stage. Pipelines are useful when the computation can be organized in terms of a stream of data flowing through a sequence of independent processing stages. In a concurrent setting, the stages execute concurrently and the communication between two contiguous stages can be implemented resorting either to synchronization or to buffering. We have chosen buffering the input and the output of each stage. Therefore each stage holds an input and an output (concurrent and bounded) queue.

Each stage of the pipeline maps values of type `I` to values of type `O`. Our solution aimed at allowing the use of varying types across the pipeline. This goal largely determined the design of the solution that regards the notion of pipeline as having two variants: the variant `Stage` (Listing 8) represents atomic pipelines mapping values from type `I` to type `O`; the variant `Compound` Listing 9) represents pipelines that result from the concatenation of two pipelines, respectively mapping values from type `I` to type `U` and mapping values from type `U` to type `O`.

Method `main()` of provider `StageProv`, consumes the input from the input queue, processes them and places the result on the output queue. Method `main()` of provider `CompoundProv` copies the output on the first sub-pipeline to the input of the second sub-pipeline, as is illustrated in Figure 3.

Concrete stages must specify how the values are transformed by implementing method `process()`. When building a non-trivial pipeline, the stages are represented as sessions over the various `Stage` subservices and the stages are joined together using sessions over the service `Compound`.

**Master-worker** In a master-worker paradigm, the master is a single execution flow that solves problems by means of running multiple tasks on multiple worker flows. The master coordinates the execution of the workers, which includes receiving the individual results and sending those results, sometimes after extra processing, back to the caller.
In the following example, we implement a very simple server for the concurrent execution of independent tasks (Java objects compliant with the Callable interface). The master receives a task to execute and returns the computed result to the caller. Internally, it coordinates a adjustable set of workers and uses an individual worker to run each task. There is an internally managed concurrent queue of free workers, upon which the master must wait for a worker to become free in order to run the next task.

To fully understand the method run of provider MasterProv, you must recall that the methods of any service implicitly return futures. Notice, therefore, the use of the sync construction inside the method run(). Its purpose in the example is to allow the method to wait for the completion of the task and therefore to detect the moment at which the worker becomes free. To gain access to all providers of the Worker service, MasterProv resorts to a runtime service (ProviderManager) that offers, among others, operation getAll(). The operation receives the service’s interface and an handler to be executed for each provided retrieved.

**Dynamic Adaptability of the Number of Workers:** Service ProviderManager features operations to control the number of providers of a given service at the application level: newInstance() and removeInstance(). Another runtime service (ProviderInspector) allows providers to inspect the wait queue associated to each of their methods. Provider MasterProv makes use of both these feature to dynamically adapt the number of its workers.
service Worker { O run(Callable task); }
service Master { session << O run(Callable task); } 
service ProviderManager { 
<< S >> getNewInstance(S service, ServiceEventHandler session<< handler); 
<< S >> newSingleInstance(S service, ServiceEventHandler session<< handler); 
<< S >> void removeInstance(S service); 
}
service ServiceEventHandler { session<< void handle(S instance); } 
service WorkerHandler extends ServiceEventHandler { 
session extends ServiceEventHandler{Worker} 

void setQueue(Worker, WorkerHandler) 
}

provider WorkerProv implements Worker { 
<< O >> O run(Callable task) { return task.call(); } 
}

provider MasterProv implements Master { 

session { 
Queue session<< Worker >> freeWorkers; 
( int nWorkers) { ProviderManager.<Worker>getAll(Worker, WorkerHandler); } 
<< O >> ! checkFreeWorkers() O run(Callable task) { 
Worker worker = freeWorkers.get(); 
O result = sync worker.run(task); 
freeWorkers.put(worker); 
adaptNumberWorkers(); 
return result; } 

boolean checkFreeWorkers() { 
if ( ProviderInspector.waitForWaitQueueSize("run") >= queuedTasksUpperBound) 
ProviderManager.newInstance(Worker, WorkerHandler); 
return freeWorkers.isEmpty(); 
} 

void adaptNumberWorkers() { 
if ( freeWorkers.size() > freeWorkers.capacity()/2) 
ProviderManager.removeInstance(Worker); 
} } 

}

provider WorkerHandlerProv implements WorkerHandler { 
session extends Handler<Worker> 
void handle(Worker instance) { freeWorkers.put(instance); } 
void setQueue(Worker >> queue) { freeWorkers = queue; } 
}

Listing 10. The Master/Worker example

4 Related Work

Having its base of service-oriented computing, our work shares a lot with component-based software development, namely in its macro and micro vision, but removes itself from the port model and its provides/uses binding mechanism.

The Common Component Architecture (CCA) [2] is a component model with a clear research statement on high-performance computing. However, its support for parallel programming is mostly restricted to the SPMD paradigm. The focus is fundamentally on the provision of interoperability across languages, enabling the construction of parallel applications by combining existing software modules.

The ProActive [3] Java middleware also builds up from the component-based programming model to assemble sets of active objects. It distinguishes itself by allowing the hierarchical composition of components, which permits it to build recurring patterns similar to the ones addressed in Section 3. This however is not done programmatically, we are once more at the integration rather than at the programming level. Components are defined through XML files that specify component interface, composition, and requirements.

Regarding service-oriented computing in particular, Web services are currently the de-facto standard. They instantiated the concept with Web standards to provide
an Internet-scale interoperable platform, with the purpose of constructing applications by composing deployed services. The service abstraction is used exclusively at a middleware level as a system integration tool.

Another model that applies the notion of interconnected isolated entities to concurrent programming is the actor model, introduced by Hewit [11]. It provides a good framework to express basic concurrency but lacks in hierarchical organization, which requires the incorporation of other programming paradigms to develop complex applications. Nonetheless, the actor model can be seen as the basis for all the models based on active objects, be it concurrent or distributed, and thus for service-oriented computing in general. Among the languages that grow from the actor model are Scala [10], Erlang [1] and Axum [14].

Other models do not resort to the same component or service approach, but also aim at high-level programming of parallel architectures, namely, the PGAS (Partitioned Global Address Space) model, in UPC [8], Titanium [13], as well as X10 [6] and Chapel [5].

5 Conclusions

We introduced a programming model building on the service abstraction to provide a high-level model for the programming of concurrent and parallel applications, as well as the means for applications to dynamically adapt their execution to the runtime and architecture characteristics. The use of an established and intuitive abstraction such as the service, provides an adequate framework for designing the parallel code, the decoupling of resource awareness from the functional concerns, and for adapting the runtime system API to the application.

The distinctive characteristics of the model are related to how the following functionalities build upon the service abstraction: i) Decoupling invocation from execution, a feature proper of actor and active object models that support parallelism by design. Service providers have no pre-determined execution order, i.e., they execute concurrently - a service may be an active entity, with its own flow of execution, and, moreover, the execution of a service operation runs concurrently with the remainder of the application. Thus, an application is inherently parallel since its deployment; ii) Decoupling service definition from service provision, forcing a contract-based software development methodology. The choice of the providers for the required services is postponed to the deployment stage, providing the means to tune the application to specific target architecture without going back to the development stage, hence increasing modularity and scalability; iii) Providing the means to construct elastic and application tailored runtime APIs. The application/runtime frontier is narrowed by allowing the runtime system to contribute to the set of providers available to the application. The service presents itself as a uniform abstraction for both the application and the system services. This meets the more so often requirements of applications to dynamically adapt their execution to the underlying runtime systems and architecture; iv) Exploring different types of parallelism at different levels. The composition of service providers supplies task parallelism by design, however parallelism can be further exploited internally to a service, namely through data parallelism in the implementation of its operations.
The paper illustrates the expressiveness of the model to codify some of the most
known parallel programming paradigms. Moreover, we illustrated the advantages
of having a unified abstraction at both application and runtime levels. The master-
worker example clearly benefits from the access to specific details of the underlying
runtime support, allowing to dynamic adapt its configuration to the also dynamic
application characteristics.

The model allows programmers without a deep knowledge of concurrent pro-
gramming to have their applications take advantage of parallel hardware without
being burdened with resource number and locality, nor with the mapping of the
problem’s decomposition onto localities. The model is currently instantiated in the
Java programming language, fostering on previous knowledge of the language to
smooth and speed the learning curve.

We are applying the model for more flexible implementations of MapReduce,
concerning the handling of issues such as parallelization, data partitioning and dis-
tribution, load balancing, and failures.

References

   in ERLANG. Prentice Hall, 1996.
2. R. C. Armstrong et al. The CCA component model for high-performance scientific
   2006.
3. L. Baduel et al. Grid Computing: Software Environments and Tools, chapter Pro-
   gramming, Deploying, Composing, for the Grid. Springer-Verlag, January 2006.
4. R. H. Campbell and A. N. Habermann. The specification of process synchronization
   by path expressions. In Symposium on Operating Systems, volume 16 of Lecture Notes
5. B. L. Chamberlain, D. Callahan, and H. P. Zima. Parallel programmability and the
7. E. W. Dijkstra. Guarded commands, nondeterminacy and formal derivation of pro-
9. A. Geist et al. PVM: Parallel virtual machine: a users’ guide and tutorial for networked
10. P. Haller and M. Odersky. Scala actors: Unifying thread-based and event-based pro-
11. C. Hewitt, P. Bishop, and R. Steiger. A universal modular actor formalism for artificial
12. N. Josuttis. SOA in Practice: The Art of Distributed System Design. O’Reilly Media,
15. M. P. Papazoglou, P. Traverso, S. Dustdar, and F. Leymann. Service-oriented com-