Trends in Combined Concurrency and Composition Modelling.

A. Djaoui
Particle Physics Department
STFC Rutherford Appleton Laboratory
abdeslem.djaoui@stfc.ac.uk

Abstract

The proliferation of commodity multi-core systems, clusters and Grids has the potential to enable more rapid and more complex applications, as new programming models emerge. In both science and business circles, there is a resurgence of interest in application models that can be applied to both local and distributed concurrency capabilities, thus going beyond the traditional dual approach to programming-in-the-small, for a single component or module, and programming-in-the-large, for a distributed application or a workflow. In this article we look at some recent advances in concurrency and composition programming models that are making building complex applications easier. We first provide an overview of concurrency, parallelism and distribution from different perspectives, and compare Grids and "Cloud Computing". We then look at a couple of frameworks for building applications that are composites of components or services, in particular their concurrency features. Finally we give an overview of a model that combines the same coordination and scheduling primitives for dealing with concurrency both locally and across distributed systems.

1. Introduction

The continuous increase in computing, storage and network bandwith capacity made available by scientific Grids such as EGEE[1] and utilily "Compute Cloud" (CC) platforms[2] has presented new challenges in distributed application programming. This challenge is exacerbated by the rapid proliferation of multi-core processors on desktops and even laptops, resulting in a fundamental shift towards "parallel programming for all". Although parallel techniques such as OpenMP[3] and the Message Passing Interface (MPI)[4] are widely used for solving complex problems on either massively parallel supercomputer or compute clusters, they are not sufficient, on their own, for the effective exploitation of Grids of multi-core processors.

Grids are currently mostly used to solve massively data-parallel problems. For more complex functional parallel applications (workflows) that are made of interacting components (processes, jobs or service instances), it is still common for developers to address parallelism, distribution and concurrency as separate issues on a line-by-line basis, targeting a specific architecture or system topology. This often results in applications that are not easily ported to another infrastructure topology. Ideally, a topology-independent model that allows instances of the application to adapt dynamically to the underlying infrastructure, would go a long way to easing the life of developers and democratising High Performance Computing.

In this article we look at the trends that are going beyond current practices, and examine the problem of composing/decomposing internet-scale distributed applications from/into smaller interacting components. We do not deal with scientific applications specifically, but look at the more general purpose Service Oriented Architectures (SOA)[5] frameworks that are common today. In SOA, an application is composed of services that have well defined interfaces and communicate using *interoperable* Web Services protocols. Services themselves may be composed of other (potentially distributed) services or may simply implement a single task or a set of tasks. As the Web Services specifications mature, new models and tools that can handle the whole spectrum from programming-in-the-large to programming-in-the-small[6] are emerging, allowing developers to concentrate on their applications rather than the underlying infrastructure topology.

In the next section we start by looking at parallelism, distribution and concurrency from an application decomposition point of view, since there is some confusion in the literature in the interpretation of these terms. We then briefly explore the differences and similarities between Grids and existing offerings of commercial Compute Clouds. In sections 4 and 5 we give an overview of two major SOA workflow frameworks that are gaining in popularity and adoption, namely WS-BPEL (Web Services Business Process Excecution Language)[7] and Windows Workflow (WF)[8]. We will not cover all the details but merely concentrate on their concurrency features and main difference between them. We then look at an example of a new breed of models, namely CCR/DSS (Concurrency and Coordination Run time /Decentralized System

Services)[9], that are bringing the idea of a single *concurrent* programming model for *local parallel* and *distribued* systems nearer to reality. We finish with a discussion on the trend towards productivity and ease of porting to different infrastructure topologies.

2. Concurrency, Parallelism and Distribution.

In theory, parallel computing is simple and consists of running independent tasks on multiple procesors or machines in order to increase throughput, thus saving time and money, and enabling previously unadressable problems to be tackled. In reality designing applications for parallelism is quite complex, with different approaches and different execution runtimes being used for clusters, grids and multi-core systems. This complexity has not been helped by the confusion in the use of some common terms such as concurrency, parallelism and distribution. We provide next a clarification of this terminology.

The term *concurrency* is commonly used today in reference to multi-core systems, but in general it refers to computational tasks that are executing at the same time and potentially interacting with each other[10]. It covers both tightly coupled *parallel* systems as well as loosely coupled *distributed* systems. But *concurrency* does not necessary imply *parallelism* or *distribution*, the "illusion" of *concurrency* can be achieved on a single sequential machines by use of time-sharing techniques as experienced by anyone who has used a modern operating system on a single CPU machine.

Technically *Parallelism*[11] is the subset of *concurrency* that is concerned with the execution of a complex task on multiple processors, multi-cores, or multiple computers, and therefore excludes the single processor "illusion" of concurrency. In practice *Parallelism* is most commonly used (but not exclusively) to describe fine grained parallel tasks that are executed on single multi-core or multi-processor systems. Only tasks that can be broken into pieces that are executed in parallel and then combined (e.g. adding two vectors) are suitable for parallelism. A range of programming techniques are used in this contex ranging from the complex and error-prone use of threads[12] to the high-level programming languages such as High Performance Fortran (HPF)[13]. Often parallelism is classified according to the granularity of the problem. Granularity refers to the ratio of computation to communication needed by a task or a sub-task. Fine Grained therefore refer to relatively small tasks or sub-tasks.

APPLICATION TYPE	TYPICAL PROGRAMMING MODEL	NOTES
Functional Parallel Distributed Applications	Workflow	Other models that fall in this category include Web 2.0 mash-ups.
Data Parallel Distributed Applications	Master/Worker frameworks	MapReduce[14] from Google and Yahoo is a popular model for data parallel applications. Dryad[15] from Microsoft is a general pupose distributed execution engine, also for data parallel applications.
Distributed memory Cluster Computing	MPI	While MPI is the traditional programming model for distributed memory clusters, new techniques are also being adopted including Workflow and Master/Worker frameworks.
Shared memory Multi-processor computing	OpenMP	Although OpenMP is typically used for shared memory programming and MPI for distributed memory, Some Partitioned Global Address Space (PGAS) based languages, such as Chapel[16], X10[17] and Fortress[18] combine features from both.

Table 1: Four levels of parallelism as embodied by popular programming, deployment and runtime models.

Distribution [19] is the subset of *parallelism* that uses multiple computers and therefore excludes *parallelism* on single multi-core and multi-processor computers. It correspond to the coarse-grained, loosely coupled end of the spectrum. A distributed application is usually spread over many machines at more than one location and its components tend to have a ratio of computation to communication that is much higher than in the finer-grained and more tightly coupled parallelism. In addition to communication reliability and

latency, the machines might also span different administrative domains. There are other new concerns for distributed applications, such as location of data with respect to computation and security. Coordination and synchronisation of concurrent distributed computation or data tasks, are also not trivial. From the programming models and tools perspective for distributed applications, we are still at the manual-approach stage, although a lot of progress is being made in SOA-based programming frameworks[9].

Table 1 illustrates four levels of parallelism that correspond roughly to existing programming and deployment models and their corresponding runtime, with the lowest level corresponding to a single machine, the middle levels to a cluster and the top level to a Grid of single machines, clusters or other Grids. This division is however only approximate, since, for example, MPI can equally be used on a multi-processor or multi-core system, workflows can be used on clusters and a functional language like Erlang[20] can be used for both concurrency and distribution.

3. Grids versus Compute Clouds

Before we look at some of the technical details, some clarification is also needed on the infrastructures these applications are targeting. In addition to scientific Grids, there are various offerings from the commercial Compute Clouds (CC) world with various capabilities, some exposing general purpose infrastructure such as compute and storage services (e.g. Amazon Web Services)[2], while others offer hosting services for web applications while hiding the compute and storage management concerns from the developer. Some detailed comparisons between Grids and CC are available [21].

Both Grid and Compute Cloud infrastructures are accessed through a number of well defined service interfaces, but differ in the amount of flexibility offered to the user and the approach to executing and administering running applications. Grids tend to focus more on the traditional concept of a batch job combined with a distributed file system. A workload management system takes care of running jobs on suitable resources and handling input and output data. However, deploying new services on to a Grid would require the intervention of an administrator and is not the norm on most current Grids. Often users need to incorporate their own services as part of their application, and might need to provision and administer their own infrastructure in order to do that.

Infrastructure Compute Clouds (CC) (e.g. AWS), rely heavily on virtualization[22] technologies to provide scalable virtual private machines (using Xen[23] in the case of AWS). AWS allows users to requisition specific machines, load them with custom application environments, manage access permissions and run instances of these applications [2]. As such, infrastructure CC computing provides more control and flexibility on the type of job or application instances that can be deployed and run, but leaves the burden of building and managing complex application instances to the user. Grid users looking for incorporating additional services with the Grid might find infrastruture CC a better alternative than the costly and lengthy process of buying and managing their own additional infrastructure.

The next stage in ease of use and lack of flexibility is the hosting CC type from the major vendors, such as Google's AppEngine[24]. The user writes, tests and deploys an application and the CC takes care of issues such as provisioning, scalability and fault tolerance. These type of CC offerings go beyond the traditional hosting platform by adding new capabilities such as BigTable[25] on Google. Being distributed over many machines, BigTable scales easily (from the user perspective) and allows the creation of tables with millions of entries, that can be queried by millions of users. Such proprietary capabilities also imply a lack of portability to other Grids and CC, since BigTable is only available on Google, and cannot currently be ported to another Grid or CC infrastructure. Other major vendors that have Cloud Computing offerings include IBM[26] and Microsoft[27].

In addition to capability, scalability and ease of use, there are other factors that can shift the balance between, Grids CC or private infrastructure. A major factor for most medical, industrial and financial applications are issues of security and privacy. Lack of portability to other infrastructures and the resulting dangers of vendor lock-in are becoming a major concern as the number of different CC flavours grows. In this context a SOA design approach to distributed applications has many advantages. By hiding the complexities of service implementation, SOA makes secure, distributed application programming much easier for developers, by allowing each service to be hosted on the best infrastructure that meets all

technical, regulatory and security requirements. Two general-purpose workflow programming models that are currently available for composing SOA applications are WS-BPEL[7] and Windows WF[8].

4. Web Services Business Process Execution Language (WS-BPEL)

WS-BPEL[7] was designed in order to help bring about the full potential of Web Services for distributed systems integration. This is achieved by augmenting the Web Services Description Language (WSDL)[28], which is essentially a stateless model for uncorrelated one way or request-reponse interactions, with stateful, long running interactions involving two or more parties. The ability to specify exceptional conditions and their consequence on distributed activities, including recovery sequences is an integral part of the language. At the heart of WS-BPEL is the concept of a *partner* with which the WS-BPEL process interacts. The interaction with each partner occurs through Web Service interfaces, using what is called a *partnerLink*. In order to achieve a business goal, a WS-BPEL process *orchestrate*s and *coordinates* such Web Service interactions with all the partners, and, in doing so, handles all the state and the logic necessary for this coordination. The process logic is centralized in one location, as opposed to being distributed across and embedded within multiple services, thus freeing partner services from implementing any complex coordination functionality. Another useful feature of WS-BPEL processes is the fact that they are themselves deployed as Web Services, allowing for recursive composition while exploiting the interoperability provided by the lower-level Web Services stack such as WSDL[28], SOAP[29] and WS-Addressing[30].

WS-BPEL has a rich set of activities and handlers that are typical of constructs found in standard programming languages. This resemblance to standard programming languages makes the development of large-scale distributed applications look and feel familiar to programming-in-the-small developers. A brief but incomplete overview of the major constructs in WS-BPEL is given next.

4.1. WS-BPEL Activities Overview

The major building blocks of WS-BPEL processes are activities. There are two types: structured and basic. Structured activities can contain other activities and define the business logic between them. In contrast, basic activities perform only their intended purpose, like receiving a message from a partner, or manipulating data.

Basic activities are used for invoking and providing Web Services operations ("*Invoke"*, "*Receive"*, "*Reply"*), for updating process variables ("*Assign"*), for signalling and propagating internal faults ("*Throw"* and "*Rethrow"*), for delayed execution ("*Wait"*) and for immediately ending a WS-BPEL process ("*Exit"*).

Structured activities cover more complex processing such as sequential processing ("*Sequence"*), Conditional behaviour ("*If"*), repetitive execution ("*While"*, "*RepeatUntil*"), selective event processing ("*Pick"*), parallel and control dependencies processing ("*Flow"*) and processing multiple branches ("*ForEach"*).

More details can be found in the WS-BPEL document [7]. We consider further here parallelism and concurrency features of the language, namely, the *"Flow"* activity, the parallel variant of the *"ForEach*" activity and Event Handlers. Note also that in WS-BPEL, there is no distinction between the terms "parallel" and "concurrent".

4.1.1. Concurrent Processing using the "*Flow*" activity

The *"Flow"* activity allows all activities nested within it to execute concurrently. A "*Flow*" completes when all its nested activities complete. Another powerful feature of the "*Flow*" activity is the ability to express synchronisation dependencies between activities.

4.1.2. Concurrent Processing using the parallel "*ForEach*" activity

The *"ForEach"* activity in WS-BPEL is one of three in the group of repetitive activities, the other two being "*While*" and "*RepeatUntil*". In its default behavior the "*ForEach"* activity iterates N times over a given set of nested activities. In the parallel variant of "*ForEach",* instead of performing each loop iteration sequentially, all loop iterations are started at the same time and processed in parallel.

4.1.3. Concurrent event processing

Event Handlers are invoked when a corresponding event occurs. There are two types of events; inbound messages and alarms that go off after a certain time. Message events represent Web Services operations exposed by a process. They are modelled as (`onEvent`) elements of the EventHandler. Timer events are modelled as (`onAlarm`) elements. Event handlers allow any number of events to occur and be processed in parallel.

4.2. Extension to WS-BPEL

WS-BPEL is focused on orchestrating interactions between Web Services only. For real-world applications it is often necessary to include human interaction with the process (such as the ability to interact with a real-time data analysis programme). The BPEL4People[31] and WS-HumanTask[32] specifications define a new type of basic activity that enable modelling of human interaction, which may range from simple approval or cancellation of a task, to complex scenarios such as separation of duties, and collaborative interactions involving ad-hoc data. The specification introduces the people activity as a new type of basic activity, which enables the formulation of human interaction in processes in a more direct way. These extensions allow WS-BPEL to be used beyond the domain of simply orchestrating Web Services.

5. Windows Workflow Foundations (WF)

Windows Workflow[8] is an integral part of Microsoft's (.Net 3.5) technologies that are used for building modern distributed applications. It provides a common foundation for building workflow-based applications on Windows, whether those applications coordinate interactions among systems (services), people, or both. WF is explicitly focused on addressing these two kinds of workflow in a unified way. To do this, it provides both sequential workflows, capable of executing activities in a pre-defined pattern, and event-based state machine workflows capable of responding to external events as they occur. State machines make it easy to model the more loosely defined nature of human workflow, while sequential workflow fits better with system workflows.

There are a lot of similarities between WS-BPEL (and its extensions) and WF, especially from a programming constructs point of view. But differences arise when looking at deployment and run-time features. While WS-BPEL is implemented and deployed on many platforms, WF is tightly coupled to the Microsoft Common Language Runtime (CLR). While this limits WF's portability, it has allowed Microsoft to address problems and rapidly add features to their platform on a quicker time scale, than WS-BPEL. Although these run-time issues are important, we will not expand on them here, but give a brief comparison between the two types of WF (sequential and state machine) and their WS-BPEL counterpart.

5.1. Sequential workflow

Sequential workflows are intended for applications where the workflow's activities are executed in a well-defined order, that can include loops, branches and other kinds of control flow, but the workflow nonetheless has a defined path from beginning to end. The sequential base activity library is very similar to WS-BPEL's and this is no surprise since they both were designed with system workflow in mind. In table 2, some typical WS-BPEL activities and WF activities (or classes) having similar functionality are shown. Note, however, that sometimes important differences exist between the activities on the left- and right-hand side of the table. For example, the WF "*Code*" activity is a powerful construct that allows the inclusion of any Visual Basic or

C# code in the workflow, while the WS_BPEL "*Assign*" activity is only used to copy data from one variable to another or to construct and insert new data using expressions.

For concurrency support, WF has the "*ParallelActivity*" class which can run a set of child activities in parallel.

WS-BPEL Activity	WF Activity
Receive	*Start or WebServiceReceive or Receive*
Reply	*Finish or WebServiceResponse*
Invoke	*InvokeWebService or Send*
Assign	*Code*
Throw	*Throw*
Wait	*Delay , Suspend*
Sequence	*Sequence*
If, ElseIf	*IfElse*
While	*While*
Scope	*Scope*
Flow	*Parallel*
Pick	*Listen*
Compensate	*Compensate*
Exit	*Terminate*
FaultHandlers	*ExceptionHandler/ExceptionHandlers*

Table 2: Similar WS-BPEL and WF activities (classes)

5.2. State Machine workflows

Unlike a sequential workflow, which structures its activities into a pre-defined order, a state machine workflow organizes its activities into a finite group of states and events that trigger transitions between these states. Organising a workflow's activities like this is useful when the exact sequence of events isn't known in advance, such as for event-driven workflows that coordinate the work of people rather than just applications. State machines can model similar concurrency scenarios to those that are achieved (in WS-BPEL[6]) with a combination of concurrent event processing and its (BPEL4People[31] and WS-HumanTask[32]) extensions. It is also worth mentioning that all parallel extensions of the .Net framework are available to the developer in addition to WF, for expressing concurrency at finer levels.

6. DSS/CCR

Workflow models such as WS-BPEL and WF address only the orchestration of loosely coupled, coarse-grained services but do not lend themselves to modelling the internal implementation details of services themselves, where fine-grained, tightly coupled tasks need to be coordinated. A separate programming and cordination model (i.e. programming-in-the-small) is usually used for that pupose. With the broad deployment of multi-core systems coinciding with the broad adoption of distributed Grid and Cloud Computing environments, there is renewed interest in a common programming model that can address orchestration and coordination accross the whole spectrum. One such model is the combination of the Decentralized System Services (DSS)[33] and the Concurrency and Coordination Run time (CCR)[34] from Microsoft.

While the CCR provides a scalable way to program asynchronous operations (both local and remote) and coordinates among multiple responses[35], DSS sits on top of CCR and provides a lightweight model (messaging and notifications) that is particularly suited for creating Web applications as compositions of services running in a distributed environment. In the next two sections we examine CCR and DSS in more detail.

6.1. CCR

CCR[34] is a Library that addresses the need to manage asynchronous operations, deal with concurrency, exploit parallel hardware and deal with partial failure. Synchronous calls such as I/O operations often need to wait a long time for the requests to finish, resulting in a lack of responsiveness and scalability. Asynchronous calls offer a significant benefit in this regard, but it is difficult to coordinate the many actions that are happening at the same time on many threads or many processes. CCR makes it much easier to deal with parallelism without the burden of coordinating multiple threads explicitly. Low-level threads assume that the primary communication between them is shared memory, forcing the programmer to use very explicit, error-prone methods to synchronise access to that shared memory. CCR uses higher level constructs that make such coordination much easier for the programmer.

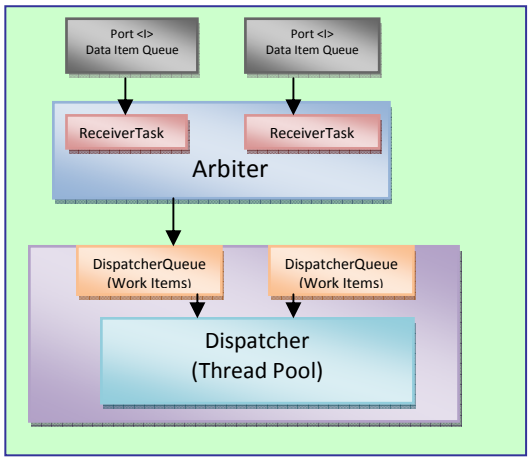

Fig. 1: CCR Class hierarchy

In CCR, I/O operations use the concept of a "port", which is an object that represents a queue of items. In addition to "ports", CCR uses two other primitives, "arbiters" and "dispatchers", the first for coordination and the second for scheduling tasks associated with the completion of messages. When items are posted to a "port", some appropriate tasks are scheduled for execution, by one or more "arbiters" being registered with that "port".

"Arbiters" are the feature that enables CCR's powerful coordination features. When there are no "arbiters" attached to the port, the item is simply added to a queue. If there are arbiters present, each one will be called to evaluate the item and decide if a task should be created and scheduled for execution.
The "Arbiter" coordination primitives provided by the CCR can be used for inbound requests or for responses from one or more outstanding asynchronous requests. One example in the first category is a Web Service listening for requests on some network port, using a CCR port to post all inbound requests and attaching handlers that wake up and serve each request independent of each other. One example in the second category is scattering multiple requests at once, then collecting all the responses using a single primitive, not caring in what order the responses arrive across a single or multiple response ports.

In CCR a "Dispatcher" object is used to manage a thread pool executing the tasks that are created and scheduled for execution by "Arbiter" objects. When you construct a "Dispatcher" object, you can pass to the constructor the number of threads you desire. By default, the "Dispatcher" creates one thread for every CPU in your computer. The CCR enables the creation of multiple thread pools or multiple "Dispatcher" objects. It is the "Dispatcher" object that shields the developer from the low level, error-prone multithreading. When

combined with the coordination features provided by the "Arbiter" objects, it makes CCR a powerful model for concurrent programming. DSS extends the benefits of CCR to distributed systems as explained in the next section.

6.2. DSS

DSS[33] builds on the existing Web model provided by HTTP augmented by the SOAP-based structured data manipulation and event notification. As in SOA, services are the key components of the DSS application model. Interaction with DSS services happen either through HTTP or the DSS protocol (DSSP)[33]. DSSP is a simple SOAP-based application protocol that defines a lightweight service model with a common notion of service identity, state and relationships to partner services, as pictured in Figure 2.

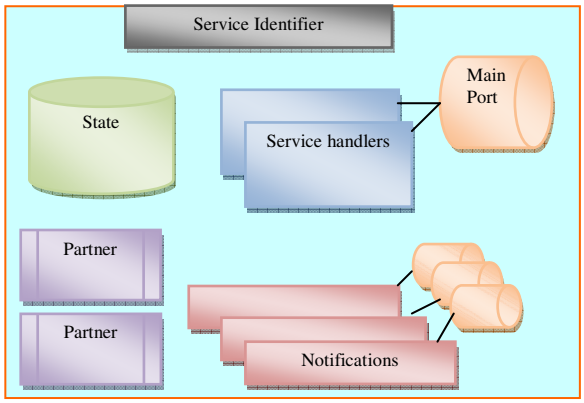

Fig. 2: Anatomy of a DSS service

Services are executed within the context of a DSS node. A DSS node is a hosting environment that provides support for services to be created and managed until they are deleted or the DSS node is stopped. Services are inherently network enabled and can communicate with each other in a uniform manner regardless of whether they are executed within the same DSS node or are distributed across the network.

The uniform model for communication and coordination within the same node (for tightly coupled, fine-grained tasks) or across the network (for loosely coupled, coarse grained tasks) is enabled through the use a main "port". The main "port" is a standard CCR "port" where messages from other services arrive. For each of the DSSP operations defined on the main "port", service handlers need to be registered to handle incoming messages arriving on that "port".

A common pattern used in DSS services is that of subscribing to other services. A service generates event notifications as a result of changes to its own state. For each subscription that a service has established with other services, the service will receive event notifications on separate CCR "ports". By using different "ports" for each subscription, it is possible to differentiate event notifications and determine which subscription each event notification is associated with. Furthermore, because event notifications arrive on "ports", it is possible to coordinate actions arising from event notification messages using the full spectrum of CCR primitives.

When a service instance is created within a DSS node, it is dynamically assigned a Uniform Resource Identifier (URI) by the constructor service. This service identifier refers to a particular instance of a service running on a particular DSS node. The service state is a representation of a service at any given point in time. One way to think of the service state is as a document that describes the current content of a service.

Partner services are other services that a service interacts with and possibly depends on in order to function properly. This concept is similar to one used by WS-BPEL. By declaring a set of other services as partners, a service can indicate to the runtime that it wants to be wired up with these services as part of the creation process of the service itself.

The combination of CCR and DSS has already found uses in a variety of fields ranging from robotics to high performance financial applications [36].

7. Discussion

Recent advances in computer technology and architectures have generated a resurgence of interest in parallel and distributed programming models and tools, beyond academic and computing specialist circles. In the drive for better performance and easier programmability, two approaches for developing internet scale applications are emerging, the first starting from programming-in-the-small for tightly coupled, fine-grained parallelism, and the other from programming-in-the-large for loosely coupled, coarse-grained parallelism. Both approaches attempt to cover the whole spectrum by incorporating more features.

In the first category, the availability of teraflop (and petaflop) systems and multi-core systems is seeing new advances in parallel (concurrent) programming, such as the new generation of high productivity programming languages Chapel[16], X10[17] and Fortress[18]. They allow easier programmability than the traditional Thread or MPI or openMP models, for both data-parallel and task-parallel problems, but it will be some time before they are ready (planned for 2010) and then widely adopted, if ever. Typically, they combine ideas from shared-memory and distributed-memory programming models and aim to provide the best of both worlds. In expectation of the continuous rise in the number of processors per computer, the new models are designed with scalability as a priority, and less attention to other important issues, such as how to deal with heterogeneous resources or security issues.

In the second category, Grid and Cloud Computing are enabling a new generation of distributed applications. A variety of programming models ranging from Master/Worker to workflow languages are used. When connecting, heteregeneous systems, interoperabity is handled by using SOA principles and technologies such as Web Services and security is also dealt with.

Beyond these two approches, new models that attempt to bridge the gap between them are now emerging. CCR/DSS, for example, handles concurrency and coordination uniformly for both local and distributed tasks, and significantly eases distributed, SOA-based applications. But being a Microsoft technology, it is not easily portable to other operating system platforms without cost. The new high-productivity languages try to make parallel programming as painless as possible, by supporting data and task distribution and many levels of parallelism, be it at the statement, function or module level, but they do not yet address many issues of programming-in-the-large for complex internet scale applications.

Currently, applications that take advantage of multi-core, multi-processor, cluster and Grid systems, need to be decomposed into components, with the target infrastructure topology in mind, as well as granularity and coupling considerations. Particular attention is paid to load-balancing parallel tasks, so as to avoid the problems of smaller tasks wasting resources while waiting for longer tasks to complete. Such an approach makes porting these applications between different Grid and Compute Cloud offerings a daunting task. New programming models are needed, that will make it possible to change the target infrastructure architecture or topology and distribute tasks across nodes, without needing to re-engineer the application from scratch. Already some exploratory projects (e.g. Microsoft Midori project [37]) for such adaptive technologies are underway but it will be some time before we see a working prototype that truly allows porting applications from one infrastructure architecture or topology to another without major changes.

Acknowledgement

Many thanks to Benyamin Aziz and David Kelsey for constructive comments on this article.

References

[1] Enabling Grids for EsciencE project: http://www.eu-egee.org/
[2] Amazon Web Services “Cloud Computing” platform: http://aws.amazon.com/
[3] Open Multi-Processing application programming interface: http://openmp.org/wp/
[4] Message Passing Interface applicaion programming interface: http://www.mpi-forum.org/

[5] Newcomer, Eric; Lomow, Greg (2005). *Understanding SOA with Web Services*. Addison Wesley. ISBN 0-321-18086-0
[6] Frank DeRemer and Hans Kron. "*Programming-in-the-Large Versus Programming-in-the-Small*", IEEE Trans. on Soft. Eng. 2(2), 1976.
[7] Web Services Business Process Execution Language: http://docs.oasis-open.org/wsbpel/2.0/wsbpel-v2.0.pdf
[8] Bruce Bukovics: *Pro WF: Windows Workflow in .NET 3.0*, Apress, 19 February 2007, ISBN 1-59059-778-8
[9] Xiaohong Qiu; Fox, G.C.; Huapeng Yuan; Seung-Hee Bae; Chrysanthakopoulos, G.; Nielsen, H.F., *High Performance Multi-paradigm Messaging Runtime Integrating Grids and Multi-core Systems, 3rd* IEEE International Conference on e-Science and Grid Computing, 10-13 Dec. 2007, http://grids.ucs.indiana.edu/ptliupages/publications/CCRSept23-07eScience07.pdf
[10] Roscoe, A. W. (1997). *The Theory and Practice of Concurrency*. Prentice Hall. ISBN 0-13-674409-5
[11] Parallel Computing: http://en.wikipedia.org/wiki/Parallelism_(computing)
[12] Bill Lewis: *Threads Primer: A Guide to Multithreaded Programming*, Prentice Hall, ISBN 0-13-443698-9
[13] High Performance Fortran: http://hpff.rice.edu/
[14] MapReduce: http://labs.google.com/papers/mapreduce.html
[15] Dryad Home: http://research.microsoft.com/en-us/projects/dryad/
[16] Cray's Chapel language: http://chapel.cs.washington.edu/
[17] IBM's X10 language: http://www.research.ibm.com/X10
[18] Sun's Fortress language: http://projectfortress.sun.com
[19] Attiya, Hagit and Welch, Jennifer (2004). *Distributed Computing: Fundamentals, Simulations, and Advanced Topics*. Wiley-Interscience. ISBN 0471453242.
[20] Joe Armstrong (Pragmatic Bookshelf, 2007, ISBN 978-1-9343560-0-5). "Programming Erlang, Software for a Concurrent World".
[21] Clouds and grids - evolution or revolution? https://edms.cern.ch/file/925013/3/EGEE-Grid-Cloud.pdf
[22] Virtualization http://en.wikipedia.org/wiki/Virtualization
[23] Xen http://en.wikipedia.org/wiki/Xen
[24] Google Appengine http://code.google.com/appengine/
[25] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E. Gruber *Bigtable: A Distributed Storage System for Structured Data* http://labs.google.com/papers/bigtable.html
[26] IBM blue Cloud: http://www.ibm.com/developerworks/websphere/zones/hipods/
[27]Microsoft Windows Azure: http://www.microsoft.com/azure/windowsazure.mspx
[28]Web Services Description Language (WSDL) 1.1 specification: http://www.w3.org/TR/wsdl
[29] Simple Object Access Protocol (SOAP): http://www.w3.org/TR/soap/
[30] Web Servicess Addressing (WS-Addressing): http://www.w3.org/2002/ws/addr/
[31]WS-BPEL Extension for people specification version 1.0 (BPEL4People) http://xml.coverpages.org/BPEL4People-V1-200706.pdf
[32]WS-HumanTask http://xml.coverpages.org/WS-HumanTask-V1-200706.pdf
[33] Decentralized Software Services (DSS) Protocol – DSSP/1.0 http://download.microsoft.com/download/5/6/B/56B49917-65E8-494A-BB8C-3D49850DAAC1/DSSP.pdf
[34] Concurrency and Coordination Runtime (CCR) http://msdn.microsoft.com/en-us/library/bb648752.aspx
[35] Concurrent Affairs http://msdn.microsoft.com/en-gb/magazine/cc163556.aspx
[36] CCR and DSS case studies: http://www.microsoft.com/ccrdss/#CaseStudies
[37] Microsoft's Midori project: http://www.sdtimes.com/link/32627