Space Based Architecture for numerical solving
Cyril Dumont, Fabrice Mourlin

To cite this version:
Cyril Dumont, Fabrice Mourlin. Space Based Architecture for numerical solving. IEEE. International Conferences on Computational Intelligence for Modelling, Control and Automation; Intelligent Agents, Web Technologies and Internet Commerce; and Innovation in Software Engineering, Dec 2008, Vienna, Austria. pp.309-314, 2008, IEEE Digital Library. <10.1109/CIMCA.2008.157>. <hal-00378342>

HAL Id: hal-00378342
https://hal.archives-ouvertes.fr/hal-00378342
Submitted on 24 Apr 2009

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L’archive ouverte pluridisciplinaire HAL, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d’enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.
Abstract

A strategy for the analytical solving of ordinary differential equations and a first implementation of it based on mobile agent community, using jini javaspace framework, are presented. This architecture is designed to support computation in a reconfiguration context. It means evolving network (blackout of processor) or internal event management (exception during computation). All the incidents are taken into account with our computing space and in the worst case; a state of the current computation is saved for a post mortem analysis or a further replay.

1. Introduction

This paper presents our approach for a computational strategy towards the analytical solving of ordinary differential equations (ODEs), and an implementation of our approach. The package consists of an ODE-solver based on an explicit parallel algorithm and dedicated software architecture.

We started two years ago the construction of an architecture based on mobile agents. Their tasks are to dynamically deploy and manage a computing case study. It is based on a parallel algorithm which already exists and has been chosen by users. Such a case is often based on large input data set, and the data balancing is an essential activity of our mobile agent. This activity is important as a pre treatment and also final for building an output set of results, but the data management is also a key feature throughout the run time. Because the resources are not always available, the data can move from one node of the network to another one. This work presents improvements of our architecture based on mobile space now and the chosen case study is a numerical computing with few input data but a strategic management of the workers.

The goal of our work is to solve linear ordinary differential equations of any order with constant coefficients. It can also solve many linear equations up to second order with no constant coefficients. This work builds general tasks for many of the nonlinear ordinary differential equations whose solutions are given in standard reference books such as [6]. Our approach applies template algorithms for general solutions for linear and weakly nonlinear partial differential equations. Truly nonlinear partial differential equations usually admit no general solutions.

We start our presentation with a brief explanation about the use of mobile agent in numerical simulation domain. Then, we present our case study and their objectives. Third, we describe our architecture based on mobile spaces. We analyze our results and finally we sum up about the scope of our chosen architecture and future evolutions.

2. Mobile agent and numerical simulation

Numerical simulation using computers or computational simulation has become a very important approach for solving complex practical problems in engineering and science. Numerical simulation translates important aspects of a physical problem into a discrete form of mathematical description, recreates and solves the problem on a computer and reveals phenomena virtually according to the requirements of the analysts.

Rather than adopting the traditional theoretical practice of constructing layers of assumptions and approximations, numerical approach attacks the original problems in all detail without making too many assumptions. This is an alternative tool for engineers but numerical techniques are not enough in a case where the computing environment is heterogeneous and unreliable.

Mobile agent is a software technology which allows engineer to add new properties to a numerical computing. Mobile agent means a piece of code which can move from one node of the network to another one. This code is a part of a more complex computing ap-
plication. In numerical simulation domain, these features provide three new facets: hot deployment, replay and adaptability. Hot deployment means the deployment of mobile agents can evolve during an execution. This is useful when the resources are shared between engineers. The size of the resource depends on the demand. Replay is essential when an anomaly occurs. Traditionally, this can involve lost of data and lost of time. Mobile agents allow engineer to keep track about the state of the interruption.

Adaptability remains the most powerful aspect. Because it means an agent can import other resource when its own task needs an add-on. We have applied this technology to a well known computing case called Runge Kutta method. This algorithm is often used into equation solvers.

Building distributed applications with conventional network tools usually entails passing messages between processes or invoking methods on remote objects. In JavaSpaces applications, in contrast, processes don't communicate directly, but instead coordinate their activities by exchanging objects through a space, or shared memory. A process can write new objects into a space, take objects from a space, or read (make a local copy of) objects in a space. When taking or reading objects, processes use simple matching, based on the values of fields, to find the objects that matter to them. If a matching object isn't found immediately, then a process can wait until one arrives. To modify an object, a process must explicitly remove it, update it, and reinsert it into the space. Spaces are object stores with several important properties that contribute to making JavaSpaces a powerful, expressive tool. Spaces are shared, persistent, transactionally secure [5].

3. Runge Kutta parallel algorithm

From the physical phenomena observed, mathematical models are established with some possible simplifications and assumptions. These mathematical models are generally expressed in the form of governing equations with proper boundary and initial conditions. The governing equation may be a set of ordinary differential equations (ODE), partial differential equations (PDE), integration equations or equations in any other possible forms of physical laws.

Ordinary differential equations play a prominent role in a range of application areas, including biology, chemistry, epidemiology, mechanics, microelectronics, economics, and finance. The Runge-Kutta algorithm is the magic formula behind most of the physics simulations. The Runge-Kutta algorithm lets us solve a differential equation numerically (approximately); it is known to be very accurate and well-behaved for a wide range of problems [1].

Consider the single variable problem: $x' = f(t, x)$ with initial condition $x(0) = x_0$. Suppose that $x_n$ is the value of the variable at time $t_n$. The Runge-Kutta formula takes $x_n$ and $t_n$ and calculates an approximation for $x_{n+1}$ at a brief time later, $t_{n+h}$. It uses a weighted average of approximated values of $f(t, x)$ at several times within the interval $(t_n, t_{n+h})$.

The formula is given by:

$$x_{n+1} = x_n + \frac{h}{6}(k_1 + 2k_2 + 2k_3 + k_4)$$

where

$$k_1 = f(t_n, x_n)$$
$$k_2 = f(t_n + \frac{h}{2}, x_n + \frac{h}{2}k_1)$$
$$k_3 = f(t_n + \frac{h}{2}, x_n + \frac{h}{2}k_2)$$
$$k_4 = f(t_n + h, x_n + hk_3)$$

This algorithm is not a parallel algorithm but we can make one. Indeed, we can observe that $x_{n+1}$ depends on $x_n$ and $k_1$, $k_2$, $k_3$, $k_4$. The way to parallelize this algorithm is to compute in the same time $k_1$, $k_2$, $k_3$, $k_4$. To do that, we have to develop the four equations: it is possible to replace $k_1$ by $f(t, x)$ and so on ($k_2$ in $k_3$, $k_3$ in $k_4$). The result is that is now possible to compute in parallel the four equations for each iteration. So the algorithm becomes parallel.

To run the simulation, we start with $x_0$ and find $x_1$ using the formula above. Then we plug in $x_1$ to find $x_2$ and so on.

With multiple variables, the Runge-Kutta algorithm looks similar to the above equations, except that the variables become vectors [2], [3].

4. A Space Based Architecture adapted to Numerical Simulation

Our architecture is inspired by a "space-based architecture", and then it seems obvious that its central point is a "space". Using the JavaSpace implementation "Outrigger" filled by API Jini, our "Space" is actually an over-layer of this implementation. There are a lot of similarities between this two API, but it had to be distinguish about the events management.

Our "Space", which will be henceforth called MCASpace (Mobile Computing Architecture), contains, as any other "Space", some entries (cf. Chapter
2). Those entries are, in our case, specific, because only from three types (Task, MCAProperties, Datahandler).

We will not study DataHandler in this paper because of two reasons: they are not useful in our present case, and they were already studied in a previous paper [4].

Two other component, strongly linked together, are essential in our architecture: ComputeAgent and ComputingWorker. Without them, a task execution would be impossible.

In this chapter, we are going to study in detail the role of the four components: MCASpace, Entries, ComputingWorker and ComputeAgent.

4.1. Entries: shared data

To use entries, a "Space" is necessary. The MCASpace allows to read them, to take them, to write them and listen to their presence and state. That is why we can consider that entries are shared data.

Those shared data are the heart of our architecture. We will expose the two principal entries: Task entries and MCAProperties entries.

4.1.1. Task entry.

It is a representation of a task which is a part of the computation case. A Task has the following properties:

- name: it must be unique. In a MCASpace, there cannot be two Tasks with the same name.
- parameters: it is the parameters list of the Task. It can be null.
- state: cf. Fig. 1
- computeAgentName: it is the name of the ComputeAgent needed to execute the Task (Cf.: 4.4).
- worker: it is the name or address of the ComputingWorker which executes or had executed the Task (Cf.: 4.3).
- result: it is the result of the Task. It can be null if the Task uses the DataHandler without giving any result.
- parentTask: it is the name of the Tasks which are needed to be executed before the current Task.

Figure 1. Lifecycle of a Task

4.1.2. MCAProperties entry

They are the shared properties of all different tasks. They are included in a Hashtable with "key value" couples. If a variable is shared by many tasks, it is more useful to put it there than in each Task's properties.

These properties can also permit to give a description of the computation case to know which the case in progress is. Then, it is enough to recover this entry and to read it.

4.2. MCASpace: The shared memory

The central component of our architecture is obviously the MCASpace. As a shared memory, it will contain the different entries, the shared data (Task and MCAProperties).

4.2.1. The TaskNotifierAgent.

To take part in the computation case, we could have considered that the ComputingWorker only needs to listen to all the tasks whose state is WAIT_FOR_COMPUTE. But if all the ComputingWorkers were warned in the same time that a waiting task was arriving, the computation case would have slowed down. To keep this possibility of warning the ComputingWorkers about the arrival of a new Task, we will need to use a mobile agent.

We have seen that, in the initializing step, the MCASpace starts a TaskListener. This listener watches the arrival of the WAIT_FOR_COMPUTE Tasks. To recover the waiting tasks, a ComputingWorker will register to the MCASpace, providing it the address and the port of its lookup (Cf.: 4.3). The TaskListener has the list of all the lookups. When it is warned about a new task, it will create a mobile agent, a TaskNotifierAgent, providing it this roadmap (the lookups list). This agent will travel from lookup to lookup to find a WAIT_FOR_COMPUTE ComputingWorker. When
it finds this ComputingWorker; it stops its road. This technique permits to start many mobile agent simultaneously in case of many tasks arrive in the same time. The order of the roadmap is random to optimize the role of this agent.

4.2.2. The AnotherTaskListener.

When a Task, which is in a WAIT_FOR_ANOTHER_TASK state, is written in the MCASpace, a listener is created: the AnotherTaskListener.

This listener, which was created specially for this task, will listen to the MCASpace and will be warned when the parentTasks will be in the COMPUTED state. When all the parentTasks will be computed, the WAIT_FOR_ANOTHER_TASK Task will be updated: the property parentTask will be null, and the Task’s state will be WAIT_FOR_COMPUTE.

It is important that it is the MCASpace which takes charge of that kind of task, instead of the ComputingWorkers, which would be unavailable for that role.

4.3. Workers: the adaptability

Our architecture aim is to be adaptive and this adaptability is really significant with the ComputingWorker presence. Indeed, in that case, it is not necessary to define, as the beginning of each computation case, the number of workers that will be used along this computation case. It is also possible to start a computation case without having any ComputingWorker available and to add some along the computation case. Moreover, a ComputingWorker can leave the computation case (on purpose or not) without stopping the computation process.

The ComputingWorkers do not change, depending on the computation case. They are identical and the same ComputingWorker can be used for different cases. The only key parameter, which can be change, is the MCASpace address. The principle is simple: at the beginning, the ComputingWorker starts a lookup and a TaskNotifierAgentListener (explained further). Standing for eventual changes, the ComputingWorkers registers to the MCASpace (Cf.: 4.2.1). A last listener to start: the AgentListener. This listener will permit to look on the network for the ComputeAgent required for the computation of the Task (Cf.: 4.4). Moreover, the ComputeAgent will permit the creation of a cache in order to prevent from looking for a busy cache (that would take a long time) and to keep update this cache while listening to the eventual update of the agent.

Now, the ComputingWorker is ready to recover tasks and to execute them. After the initialization, the ComputingWorker will check if there is any task in the state WAIT_FOR_COMPUTE. If there is any, it executes the task (we will go back over this point); if there is one, it listens to the TaskNotifierAgentListener, which will warn if a new WAIT_FOR_COMPUTE Task appears in the MCASpace.

How a ComputingWorker knows to execute a task while it is identical to the others computations? The answer is simple: the property compute_agent_name is recovered.

When the ComputingWorker has a Task, it switches to the state ON COMPUTING (it means that it doesn’t accept any new task) and puts the Task in the MCASpace with the state ON COMPUTING and initiates the property worker with the name (or address) of the ComputingWorker.

Thanks to the AgentListener, it recovers the right ComputeAgent and the Task is executed by the ComputeAgent. (Cf.: 4.4)

Two options are then possible:

- The ComputingAgent works perfectly and return a result. The ComputingWorker updates the task in the MCASpace by modifying its state (ON COMPUTING → COMPUTED) and result.
- The ComputeAgent starts an exception Wait-
ForAnotherTaskException (Cf.: 4.4). The ComputingWorker updates the task in the MCASpace by modifying its state (ON_COMPUTING → WAIT_FOR_ANOTHER_TASK) and add the task name required for parentTask property.

In both cases, the ComputingWorker, after having updated the Task in the MCASpace, looks for a new Task. Otherwise, it switches to stand by (ON_COMPUTING → WAIT_FOR_COMPUTE).

![Figure 3. Execution of a Task by a ComputingWorker](image)

4.4. ComputeAgent: the computation intelligence

A ComputeAgent is a mobile agent available on a lookup for all the ComputingWorkers. When a ComputingWorker needs one ComputeAgent, it gets a copy of it. This agent must implement an interface ComputeAgentInterface, and redefine the method execute. This method takes two parameters: one Task and one MCAProperties; it returns a result. It is here that we find the computation algorithm. We have to define the different ComputeAgent of the computation case to define the different Tasks.

Many interactions with the MCASpace are available for an agent:

- recover a task result.
- check if there are some computed tasks. The control become indispensable if a task result is needed to be recovered.
- add a task in the MCASpace to make the computation case progress.

We have to go back over to an important point: the control of the COMPUTED Tasks. If the control is well, i.e. that all the tasks needed for the computation of the Task in progress are COMPUTED, the execution will go on. On the other hand, if one needed task is not COMPUTED, the agent will throw an exception WAIT_FOR_ANOTHER_TASK_EXCEPTION, which will be caught by the ComputerWorker (Cf.: 4.3). During this control, if a Task does not exist in the MCASpace, the ComputeAgent will create it as a WAIT_FOR_COMPUTE Task and write it in the MCASpace.

5. Results

Our architecture is experimentally evaluated using a real application. We used a specific version of Runge Kutta method called RK4 [7]. We realized several experiments; the easiest one is about a short ODE:

\[
\frac{dx}{dy} = -2xy
\]

A right analytic solution is:

\[
y = \exp(-x^2)
\]

To compute this ODE, we need to make available six ComputeAgent: one for the function \(f\), one for the sum of each iteration and one for each component (\(k_1, k_2, k_3, k_4\)) (cf. Chapter 3). For this, we use one node with a lookup. Also we need to a Javaspace: one more node is required. Four tasks can be computed in parallel at least, four ComputingWorker are required to (four nodes). A total of six nodes are necessary to optimize our architecture. (It is also possible to use fewer nodes at the beginning of the computation case and add more during the case).

The configuration of each node is as follows:

- Pentium IV, 2.8Ghz, 512Mo/1Go RAM, HD40 Go
- Ubuntu 8.04

To start the computation case (here the ODE), we have to put on the MCASpace the Task to compute the
first iteration. As it will need the four components component \((k_1, k_2, k_3, k_4)\), the Task go back on the MCASpace at the state WAIT_FOR_ANOTHER_TASK and four Task (for the four components) will be put on the MCASpace.

At the end of the computation of this task, the Task will put a new Task to compute the next iteration (and so on).

We obtain with our simulation the same results with a precision \(10^{-2}\), it means for instance that for the iteration 1 when \(x = 0.1 \) and \(y = 0.990\) the four coefficients are:

\[
\begin{align*}
a &= 0.0198 \\
b &= -0.0294 \\
c &= -2.926 \times 10^{-2} \\
d &= -0.0384
\end{align*}
\]

All the other iterations follow the same correspondence with the analytic solution. Thus, we are confident into our approach and the applications of the technique.

6. Conclusion

This presentation is about new architecture software based on space architecture. It allows us to transform a classical parallel application into an application which is more reactive to its context. It means not only the execution context but also the need of each worker of the computing case. The main consequence of that approach is to provide new functionalities like the replay of execution or evolutive deployment.

References