Performance Comparison of Parallel Programming Environments for Implementing AIAC Algorithms

JACQUES M. BAHI jacques.bahi@univ-fcomte.fr
SYLVAIN CONTASSOT-VIVIER sylvain.contassot-vivier@iut-bm.univ-fcomte.fr
RAPHAËL COUTURIER raphael.couturier@iut-bm.univ-fcomte.fr
Laboratoire d’Informatique de Franche-Comté (LIFC), IUT de Belfort-Montbéliard, BP 527, 90016 Belfort, France

Abstract. AIAC algorithms (Asynchronous Iterations Asynchronous Communications) are a particular class of parallel iterative algorithms. Their asynchronous nature makes them more efficient than their synchronous counterparts in numerous cases as has already been shown in previous works. The first goal of this article is to compare several parallel programming environments in order to see if there is one of them which is best suited to efficiently implement AIAC algorithms. The main criterion for this comparison consists in the performances achieved in a global context of grid computing for two classical scientific problems. Nevertheless, we also take into account two secondary criteria which are the ease of programming and the ease of deployment. The second goal of this study is to extract from this comparison the important features that a parallel programming environment must have in order to be suited for the implementation of AIAC algorithms.

Keywords:

Introduction

Iterative algorithms are very well suited for a large class of scientific problems and, in many cases, they are the single way to solve the problem. Their principle is to achieve the solution of a problem by successive approximations. Although some conditions must be verified to ensure that this process works well, it is not actually a great obstacle since these conditions are verified for many scientific problems.

Another interesting point is that these algorithms can be modified to run in parallel rather easily. Nevertheless, the classical parallel iterative algorithms are synchronous. We have shown in [5, 6] all the interest of using asynchronism in such algorithms especially in a global context of grid computing. These works together with [4] have led to the definition of a particular class of parallel iterative algorithms which we call AIAC. This acronym stands for Asynchronous Iterations Asynchronous Communications.

The first goal of this article is to find out if there is a particular parallel programming environment which is best suited to implement our AIACs algorithms. We have chosen to compare three environments which are well-known in the parallelism community and which propose rather different conceptual views: MPI, PM2 and Corba. In this comparison, we will discuss the ease of the implementation of AIACs, the ease of deployment over the grid and the efficiency aspects. The discussion will be illustrated by several experiments on two representative kinds of scientific problems, each of them using a different communication scheme.
Using these results, our second goal is to deduce the features required for a parallel programming environment to be suited for the implementation of AIAC algorithms.

The following section recalls the principle of asynchronous iterative algorithms and replaces them in the context of parallel iterative algorithms. In Section 2, we present a brief analysis of the programming environments used for the implementations of AIACs in our previous works. A brief description of the three tested parallel programming environments is given in Section 3. Section 4 gives the description of the two scientific problems chosen to make our experiments. Then, the comparison of the three parallel programming environments is presented in Section 5 together with experimental results. Finally, the important features required for an efficient implementation of AIACs are given in Section 6.

1. Parallel iterative algorithms

1.1. Synchronous parallel iterative algorithms

Iterative algorithms have the following structure

\[ x^{k+1} = g(x^k), \quad k = 0, 1, \ldots \text{ with } x^0 \text{ given} \]  

(1)

where each \( x^k \) is an \( n \)-dimensional vector, and \( g \) is some function from \( \mathbb{R}^n \) into itself. If the sequence \( \{x^k\} \) generated by the above iteration converges to some \( x^* \) and if \( g \) is continuous then we have \( x^* = g(x^*) \), we say that \( x^* \) is a fixed point of \( g \).

Let \( x^k \) be partitioned into \( m \) block-components \( X^k_i, \ i \in \{1, \ldots, m\} \), and \( g \) be partitioned in a compatible way into \( m \) block-components \( G_i \), then Equation (1) can be written as

\[ X^{k+1}_i = G_i(X^k_1, \ldots, X^k_m) \quad i = 1, \ldots, m, \text{ with } X^0 \text{ given} \]  

(2)

and the iterative algorithm can be parallelized by letting each of the \( m \) processors update a different block-component of \( x \) according to (2) (see [18]). At each stage, the \( i \)-th processor knows the value of all components of \( X^k \) on which \( G_i \) depends. It computes the new values \( X^{k+1}_i \) and communicates those on which other processors depend to make their own iterations. The communications required for the execution of iteration (2) can then be described by means of a directed graph called the dependency graph.

1.2. Asynchronous parallel iterative algorithms

Fully asynchronous networks including overlapping updating were characterized by Herz and Marcus in [12]. The model is as follows:

- the block nodes of the network may be updated in a random order and some nodes may not be updated at some times. Nevertheless, no block is permanently idle.
- at time \( t \), each node updates its own state using the last received information from its dependencies rather than waiting for those at time \( t - 1 \).