Applied Mechanics and Materials Vols. 263-266 (2013) pp 1315-1318
© (2013) Trans Tech Publications, Switzerland
doi:10.4028/www.scientific.net/AMM.263-266.1315

A Parallel Block Predictor-Corrector Method by Python-Based Distributed Computing*

Kun-Ming Yu[1,a], Ming-Gong Lee[2,b]

[1]Department of Computer Science and Information Engineering

[2]Department of Leisure and Recreation Management

[2]Ph.D. Program in Engineering Science

Chung Hua University

Hsinchu, Taiwan 300, R.O.C.

[a]yu@chu.edu.tw, [b]mglee@chu.edu.tw

Keywords: Ordinary differential equations (ODEs), Parallel computing, Block Predictor-Corrector formulas (block method), PC Cluster, MPI, Python

Abstract. This paper is to discuss how Python can be used in designing a cluster parallel computation environment in numerical solution of some block predictor-corrector method for ordinary differential equations. In the parallel process, MPI-2(message passing interface) is used as a standard of MPICH2 to communicate between CPUs. The operation of data receiving and sending are operated and controlled by mpi4py which is based on Python. Implementation of a block predictor-corrector numerical method with one and two CPUs respectively is used to test the performance of some initial value problem. Minor speed up is obtained due to small size problems and few CPUs used in the scheme, though the establishment of this scheme by Python is valuable due to very few research has been carried in this kind of parallel structure under Python.

Introduction

Numerical computation for ordinary differential equations (ODEs) is importance in scientific computation, as they were widely used to model the real world problems. The common methods used to solve ODEs are categorized as one-step (multistage) methods and multistep (one stage) methods, which Runge-Kutta methods represent the former group, and Adams-Bashforth-Molton method represents the later group. We will consider a class of explicit and implicit multistep and multistage methods for solving ordinary differential equations; we can call it block method [1]. This method can obtain a block of new values simultaneously which makes it competitively, and especially its implicit type method can be used in solving stiff ODEs efficiently. In this paper, we set up a Python-based cluster parallel computation environment to implement a parallel computation. A MPI-2 (Message Passing Interface) is used as a standard of MPICH2 (Message Passing Interface Chameleon) to communicate between CPUs, such as data sending and receiving. Comparison of using one and two CPUs, respectively, is studied to show the effect of parallel computation of a block predictor-corrector method.

The organization of this paper is as follows: In Section 2, we introduce some multistage and multistep methods. In Section 3, we give the system structure which implements the parallel block computation, including MPI, MPI-2, Python, and mpi4py of Python. In Section 4, we give numerical results of one block formula under this parallel scheme. Section 5 is the numerical results and some discussion, and Section 6 is the conclusion.

Block Multistage and Multistep method

In numerical solution of ordinary differential equations, we wish to approximate the solutions of a differential equation with an initial value, we call it the initial value problem,

$$y'(x) = f(x, y(x)), \qquad y(a) = y_a. \tag{1}$$

In the interval of [a, b], suppose the right-hand side function is continuous and satisfies a Lipschitz condition on $[a,b]\times(-\infty,\infty)$, and guarantees the existence of a unique solution $y(x) \in C^1[a,b]$. For $h \in (0, h_0]$, let $x_k = a + kh$, a sequence of block numerical values y_k which approximates $y(x_k)$ can be obtained by a block-type method. A block type formula with multi-stages and multi-steps will be introduced in the next section. Its implementation in $PECE$ scheme or $PE(CE)^k$ can be used to solve non-stiff ODEs problems, and interested readers can refer to [1,2,3]. A Simple Block Method is as follows.

Given a s stages and m steps integration formula [1,3], where $h^{-1}(\sum_{j=0}^{m} k_j^q y_{i+s-j})$ approximates $y'(x_{i+q})$ of order $O(h^m)$ $q = 1, 2, ..., s$. Two formulas are given in the following: For s=m=1, it is exactly the same as the classical BDF formula [2]. For s=2, m=3, explicit and implicit formulas are defined as [1]:

$$\begin{aligned} 2y_{n+2} - 9y_{n+1} + 18y_n - 11y_{n-1} &= 6hf(x_{n-1}, y_{n-1}) \\ -y_{n+2} + 6y_{n+1} - 3y_n - 2y_{n-1} &= 6hf(x_n, y_n) \end{aligned} . \tag{2}$$

$$2y_{n+2} + 3y_{n+1} - 6y_n + y_{n-1} = 6hf_{n+1}$$

$$11y_{n+2} - 18y_{n+1} + 9y_n - 2y_{n-1} = 6hf_{n+1} . \tag{3}$$

The other similar higher order block formulas, readers can refer to [1]. These formulas can also be implemented in a predictor-corrector scheme, for example Eq. 2 as a predictor, and Eq. 3 as a corrector, we call it s=2, m=3 PECE scheme [1].

System Structure

3.1 MPI – 2

MPI (Message Passing Interface) [4] represents an interface between information, and MPI is just the standard for the transmission. MPI-2 support almost all high efficiency environments, such as Linux, Mac OS x, and Windows. It can send data to the designated CPUs and collect results from certain CPUs back to the main computer to do next assignment. It includes several hundreds of functions for users to communicate with the target CPUs efficiently.

MPI in MPICH2 means Message Passing Interface, and CH means Chameleon. MPICH2 is an open source code and is widely used in high computing efficiency computers, it includes some basic functions, such as, MPI_COMM.send, MPI_COMM.recv, MPI_COMM.rank, MPI_COMM.size, and some other setting and control functions to make parallelization easy. In a multicore system, MPI contains some automatic detection ability to make the communication easy.

3.2 MPI for Python

Since MPI is defined by C language, but Python does not have the ability to communicate with MPI, instead a mpi4py [4] is used as a parallel communication tool in Python. The mpi4py actually uses the functions of MPI to do communication and the simplicity of Python to do the programming. As a result, mpi4py enables the parallelization for Python with the utilization of MPICH2. Some usages of mpi4py are introduced:

1. MPI.COMM_WORLD.Get_size
2. MPI.COMM_WORLD.Get_rank

MPI.COMM_WORLD.Get_sizeis is used to obtain how many CPU are used in the group to do parallel computation; and MPI.COMM_WORLD.Get_rank is used to obtain exactly the numbering of the current CPU. The numbering of CPU starts with zero, so the Get_rank of the first CPU is 0 instead of 1, and the second CPU is 1 instead of 2, etc. The number of CPU in the concurrent computation can be demanded in advance, and the actual numbers can be more than the existed numbers. That means some of the CPU may have different number, when it is called by Get_rank. The detail implementation can refer to the manual of mpi4py. Other important functions used in mpi4py are listed below as:

4. MPI.COMM_WORLD.send
5. MPI.COMM_WORLD.recv

As the name expressed, MPI.COMM_WORLD.send is used to define the CPU to send data, while the MPI.COMM_WORLD.recv is the CPU to receive the data. To accomplish the job, one "sender" must be matched with another "receiver".

3.3 Signal Receiving Processing Component

The way that CPU get their job is done by the following, first use the function Get_rank() to get the number of each CPU, and to determine their work by "if" and "elif" statements, sample code is given here for reference:

```
mycomm = MPI.COMM_WORLD
myrank = mycomm.Get_rank()
if myrank ==0 :
    (job of CPU0)
elif myrank ==1 :
    (job of CPU1)
```

After the job assigning task is done, only one CPU owns values of parameters and variables, so another CPU need to receive these information. As a result, the functions of "send" and "recv" are used to do further. The following sample codes are listed for reference:

For CPU0	For CPU1
mycomm.send([y,MPI.FLOAT],1,1)	mycomm.recv([y,MPI.FLOAT],0,1)
mycomm.send([t,MPI.FLOAT],1,2)	mycomm.recv([t,MPI.FLOAT],0,2)
mycomm.recv([predictory,MPI.FLOAT],1,3)	mycomm.send([predictory,MPI.FLOAT],0,3)

Numerical Experiments

A blocked predictor-corrector method for s=2, m=3 as in Eqs. 2, and 3 are given to do numerical experiment. Some more examples are also tested, and readers can refer to [1,3] for those test problems. The numerical results given here is to implement with one CPU, and secondly to use two CPUs. Their difference in elapsed time and error will be compared. Some parameters in computation are given, for example, h=0.005, t_begin=0, and t_end=20. Define Eq. 2 as Eq. 4, and Eq. 3 as Eq. 5 with matrices A, B, C, and D. Different coefficient matrices can be defined by different block methods [1].

$$\begin{bmatrix} y_{i+1}^{p} \\ y_{i+2}^{p} \end{bmatrix} = [A]\begin{bmatrix} y_{i-1} \\ y_{i} \end{bmatrix} + [B]\begin{bmatrix} 6hf(x_{i-1},y_{i-1}) \\ 6hf(x_{i},y_{i}) \end{bmatrix} = \begin{bmatrix} 5 & -4 \\ 28 & -27 \end{bmatrix}\begin{bmatrix} y_{i-1} \\ y_{i} \end{bmatrix} + \begin{bmatrix} 1/3 & 2/3 \\ 2 & 3 \end{bmatrix}\begin{bmatrix} 6hf(x_{i-1},y_{i-1}) \\ 6hf(x_{i},y_{i}) \end{bmatrix} \tag{4}$$

$$\begin{bmatrix} y_{i+1}^{c} \\ y_{i+2}^{c} \end{bmatrix} = [C]\begin{bmatrix} y_{i-1} \\ y_{i} \end{bmatrix} + D\begin{bmatrix} 6hf(x_{i+1},y_{i+1}^{p}) \\ 6hf(x_{i+2},y_{i+2}^{p}) \end{bmatrix} = \begin{bmatrix} 5/23 & 23/28 \\ -4/23 & 7/23 \end{bmatrix}\begin{bmatrix} y_{i-1} \\ y_{i} \end{bmatrix} + \begin{bmatrix} 11/69 & -2/69 \\ 6/23 & 1/23 \end{bmatrix}\begin{bmatrix} 6hf(x_{i+1},y_{i+1}^{p}) \\ 6hf(x_{i+2},y_{i+2}^{p}) \end{bmatrix} \tag{5}$$

After the sending and receiving tasks have been assigned, then the scheduled parallel computation of the block predictor-corrector method can start as follows. Step 1: CPU0 sends $\begin{bmatrix} y_{i-1} \\ y_i \end{bmatrix}$ and t to CPU1. Step2: CPU0 calculates matrix multiplication between C and $\begin{bmatrix} y_{i-1} \\ y_i \end{bmatrix}$; and CPU1 computes Eq. 7. Step 3: Send the results from Step 2 to CPU0 and do the computation of Eq. 8. Step 4: Assign $\begin{bmatrix} y^c_{i+1} \\ y^c_{i+2} \end{bmatrix}$ to $\begin{bmatrix} y_{i-1} \\ y_i \end{bmatrix}$ and i=i+1, Go to step 1, the process continue until the end of time.

Table 1. One CPU used in computation

Num.	Elapsed time	Max error
Ex1	0.711834854788	1.10555832154e-18
Ex2	0.813653328789	3.40450334591e-10
Ex3	1.03526565147	0.000303762981558
Ex4	0.741259940251	4.3517644599e-08
Ex5	0.739818378763	7.76406716696e-13
Ex6	1.19902854207	2.68435584871e-09
Ex7	1.42250789867	1.45567252696e-08

Table 2. Two CPUs used in computation

Num.	Elapsed time	Max error
Ex 1	0.631384879381	1.10555832154e-18
Ex 2	0.805632562478	3.40450334591e-10
Ex 3	0.994352356787	0.000303762981558
Ex 4	0.631946758251	4.3517644599e-08
Ex 5	0.736624905512	7.76406716696e-13
Ex 6	0.933512719463	2.68435584871e-09
Ex 7	1.207442512834	1.45567252696e-08,

Results And Discussion

From Tables 1 and 2, we can see the elapsed time between using one CPU and 2 CPUs is decreasing, though it has only minor change, but it shows that the concurrency is confirmed by adding one extra CPU. Since the size of the problem is too small, the numbers of function evaluation caused by the block methods actually reduce the effect of obtaining good concurrency. But the results give us the momentum to continue the experiment by increasing number of CPUs to build a cluster parallel computation environment for scientific computation applications.

Conclusions And Future Work

The cluster parallel computation environment can be established through the use of simple personal computers with the help of programming of MPI, MPICH2, Python, and mpi4py, etc. The establishment of using the software to set up a parallel computation environment is the main purpose of this study. We hope that without spending too much budget, a good parallel computer environment can still be established and that may save much budget in hardware purchasing. In the future, we will try to expand the number of CPUs and test with large size of numerical examples to fully examine the possibility of this cluster parallel computation environment.

***Acknowledgement:** This work is partially supported by the National Science Council NSC (NSC100-2632-E-216-001-MY3)

References

[1] M.G Lee and R.W. Song, "A New Block Method for Stiff Differential equations", (2009) International Conference on Scientific Computation and Differential equations (SciCADE 2009), Beijing, China.

[2] E. Hairer and G. Wanner, "Solving Ordinary Differential Equations I Stiff and Differential Algebraic Problems", Springer Berlin, (1996).

[3] Chie-Chieh Cheng "Block predictor – corrector method on Python-based distributed computing environment", Department of Applied Statistics, Chung Hua University, Master Thesis, September, (2011).

[4] Lisandro Dalcin, "MPI for Python Release 1.2.2", September 13, (2010).