NÖHÜ Müh. Bilim. Derg. / NOHU J. Eng. Sci., 2022; 11(4), 871-878

Niğde Ömer Halisdemir Üniversitesi Mühendislik Bilimleri Dergisi
Niğde Ömer Halisdemir University Journal of Engineering Sciences

Araştırma makalesi / Research article

www.dergipark.org.tr/tr/pub/ngumuh / www.dergipark.org.tr/en/pub/ngumuh

Parallel solution of Lambert's problem using modified Chebyshev-Picard iteration method

Lambert probleminin modifiye Chebyshev-Picard yineleme yöntemini kullanarak paralel çözümü

Majd Ajroudi[1], F. Şükrü Torun[2,*]

[1,2] *Ankara Yıldırım Beyazıt University, Department of Computer Engineering, 06020, Ankara, Turkey,*

Abstract

Lambert's problem is one of the classical methods for solving the multiple revolution problem in orbit determination. With the increasing interest in space exploration programs and using satellite networks, it is important to provide an accurate and rapid method that will provide the network control center with information regarding the orbit of each satellite in the network and help the satellites improve routing decisions in onboard processing satellites. Lambert's problem is one of the methods that solve the problem iteratively and this iteration was originally done using Newton's iteration method. In recent studies, it is recommended to use the Chebyshev-Picard iteration method to solve this problem. Since the aim here is to provide a method that solves the problem rapidly, the Chebyshev-Picard iteration method serves our objective since it is highly parallelizable. In this work, we have developed a parallel algorithm that solves Lambert's problem in a parallel environment. We have conducted experiments to demonstrate the parallel scalability of the algorithm on both shared and distributed memory architectures. The experimental results show that the parallel algorithm achieves 8.26- and 3.94-times faster execution time on distributed memory and shared memory architectures, respectively.

Keywords: High performance computing, Lambert's problem, Modified Chebyshev-Picard iteration, Orbit determination, Parallel computing.

Öz

Lambert problemi, yörünge belirlemede çoklu devir problemini çözmek için kullanılan klasik yöntemlerden biridir. Uzay araştırma programlarına ve uydu ağlarının kullanımına olan ilginin artmasıyla, ağ kontrol merkezine ağdaki her bir uydunun yörüngesine ilişkin bilgileri sağlayacak ve uyduların yönlendirme kararlarını iyileştirmesine yardımcı olacak doğru ve hızlı bir yöntemin sağlanması önemlidir. Lambert problemi, bu problemi yinelemeli olarak çözen yöntemlerden biridir ve bu yineleme önceki yıllarda Newton'un yineleme yöntemi kullanılarak yapılmaktaydı. Daha güncel araştırmalarda bu problemi çözmek için Chebyshev-Picard yineleme yöntemi kullanılması önerilmektedir. Önerilen metot çözüm süresinde iyileştirmeler sunmasına rağmen büyük problemlerde çözüm çok uzun süreler alabilmektedir. Bu çalışmada, Lambert problemini paralel programlama teknikleri kullanarak daha hızlı çözen yeni bir paralel algoritma önerilmiştir. Ayrıca algoritmanın paralel ölçeklenebilirliğini göstermek için 2 farklı paralel sistemde; paylaşımlı ve dağıtık bellek mimarilerinde deneyler yapılmıştır. Deneysel sonuçlar, paralel algoritmanın dağıtık bellek ve paylaşımlı bellek mimarilerinde sırasıyla 8.26 ve 3.94 kat daha hızlı çözüm süresine ulaştığını göstermektedir.

Anahtar kelimeler: Yüksek performanslı hesaplama, Lambert problemi, Modifiye Chebyshev-Picard yinelemesi, Yörünge belirleme, Paralel hesaplama.

1 Introduction

Today, astronomy, astronautics, artificial satellites, satellite navigation systems [1], and orbital calculations are fields of increasing importance. When sending a satellite into orbit, it is essential to have a method that can predict the state vector of the satellite at any given time. Once a state vector of the satellite is determined, the six classical orbital elements that define an orbit can be calculated [2]. This procedure is called orbit determination. Lambert's problem is one of the classical methods of determining a preliminary orbit of celestial objects from two position vectors and the time of flight between the two points, which makes it a Boundary Value Problem (BVP) [3] and requires solving the differential as shown in Equation (1). The basic idea of Lambert's problem is to calculate the trajectory that connects two points where the initial and final time is given, as shown in Figure 1.

$$\ddot{r} = -\mu . \frac{\hat{r}}{r^2} \tag{1}$$

Some recent studies [3, 4] have shown a significant improvement in the performance of the Chebyshev-Picard iterative method after developing an approach to run the method in a parallel environment. Chebyshev-Picard iterative method [3] is a method that uses Chebyshev polynomials to approximate the state trajectory in a Picard iteration, where the boundary conditions are preserved by constraining the Chebyshev polynomials coefficients. This new approach is called Modified Chebyshev-Picard Iteration (MCPI). Solving Lambert's Problem using MCPI is

* Sorumlu yazar / Corresponding author, e-posta / e-mail: fstorun@aybu.edu.tr (F. S. Torun)
Geliş / Recieved: 07.02.2022 Kabul / Accepted: 03.10.2022 Yayımlanma / Published: 14.10.2022
doi: 10.28948/ngmuh.1069509

proposed in [5], where a solution for multiple revolutions perturbed Lambert's problem was proposed, as will be explained in the next section.

The original MCPI is a combination of the works of Emile Picard (Picard iteration) [6] and Rafnuty Chebyshev (Chebyshev polynomials) [7]. Since Chebyshev function approximation is orthogonal, Clenshaw and Norton found it beneficial to combine it with Picard iteration in a simultaneous manner to provide a solution to non-linear ordinary differential equations [8].

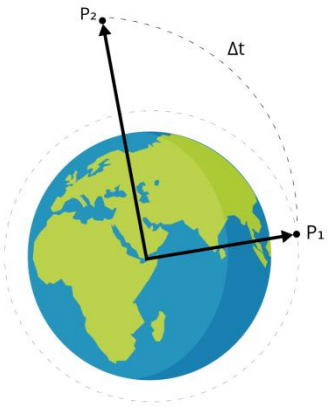

Figure 1. Trajectory calculation using Lambert's problem.

The MCPI algorithm is quite prone to parallel processing, and several early studies proposed different approaches [3, 9, 10, 11] to the method. Additionally, [3] demonstrates the efficiency of MCPI in non-linear IVP and BVP when compared to other solvers, as will be discussed in the next section.

The time taken to calculate a highly-accurate position of a single satellite using this algorithm is often acceptable depending on the case. However, when working with a satellite constellation such as Starlink [12], which forms a network of more than 3000 thousand satellites constantly communicating with each other, such calculations can be quite exhaustive. Parallel computing techniques should be considered to accomplish these tasks in a reasonable time.

This work aims to use the approach of MCPI to solve the second-order differential equation of Lambert's problem, shown in Equation (1), in a parallel environment provided by the Message Passing Interface (MPI) [13]. Similar applications of MCPI exist in the literature [3, 4, 5]; however, these studies lack the discussion of the problem from the computer science perspective, thus the work presented in this paper emphasizes this by discussing the detailed parallelization of the problem along with comparing the performance in two different computer architectures.

In the rest of this paper, we begin with discussing related works. Next, the proposed algorithm and the implementation details are presented. Finally, we discuss the results and conclude the work.

2 Related works

Solving Initial Value Problems (IVP) and Boundary Value Problems (BVP) using a combination of Picard iteration and Chebyshev polynomials is first proposed in [8], where a method to approximate the trajectory and the integrand by the same set of discrete Chebyshev polynomials is proposed. The Chebyshev polynomials in approximating the integrand of Picard iteration along the i^{th} trajectory gives an efficient and accurate approximation [8].

The parallelization of the Chebyshev-Picard iteration is mentioned in many studies [10, 14, 15, 17]. One of the recent works is Bai's Ph.D. dissertation [4] which proves the great capability of the method by extending the earlier works to show an outperforming ODEs' numerical integration in the sequential computing environment. The improvement of the Chebyshev-Picard iterations method encouraged its application to classical methods and problems to improve their performance, such as Lambert's Problem as we will see later in the rest of this section.

Junkin and Bai extend their previous work on developing parallel structured MCPI in [5]. The authors of the paper have compared the results of solving single orbit propagation between Runge-Kutta12(10) versus MCPI for a different number of nodes and various spherical harmonic orders. It is explicitly mentioned that, although the results of the experiments show a positive impact of MCPI, the experiments were limited and should be generalized for other cases.

The method of particular solutions and MCPI are combined in [5] to solve the multiple revolutions perturbed Lambert's problem for orbit transfer of a satellite. According to [5], solving the two-point boundary problems with the method of particular solutions can be done using any numerical integrator, however, MCPI increased the efficiency that cannot be provided with step-by-step integrators.

An implementation of MCPI in a parallel environment to solve a perturbed orbital trajectory is presented in [16]. The framework of this research is divided into three modules: a control module, a set of worker modules, and a renderer module. The control module ensures the coordination using the database and sends jobs to MCPI working processes, which propagate an orbit trajectory and report the propagated data to the renderer module and the control module for catalog update. The main module and MCPI workers can run on any multi-core CPU machine.

Recent studies used MCPI in a parallel environment to improve Space Situational Awareness by applying it to the process of Conjunction Assessment [17-18]. The work provided in [17] compared the MCPI method with the single satellite method using SGP4 and has shown that MCPI-aided conjunction analysis provided approximately a 50% increase in the speed. These findings can potentially protect space assets by providing timely warnings of potential collisions.

3 Proposed algorithm and implementation details

3.1.1 Introduction to MCPI

The first form of the method introduced by Emile Picard for path approximation is presented as

$$\dot{x}(t) = f\big(t, x(t)\big),$$

with the initial condition $x(t_0)$. Then the form can be rearranged to:

$$x(t) = x(t_0) + \int_{t_0}^{t} f\big(\tau, x(\tau)\big)\, d\tau. \tag{2}$$

It has been stated that the convergence of Picard iteration was bounded to a time interval $t - t_0$ less than δ, this time interval for computing the satellite trajectories in Earth orbit can approach 20,000 seconds, which is more than three periods of a typical low Earth orbit satellite [4].

Chebyshev polynomials can be obtained by the following recurrence relation:

$$\begin{aligned} T_0(x) &= 1 \\ T_1(x) &= x \\ T_n(x) &= 2xT_{n-1} - T_{n-2.} \end{aligned} \tag{3}$$

Note that the first Chebyshev polynomials T are defined by the first two relations and the rest of the polynomials can be calculated using the recurrence relation.

In the MCPI algorithm, Chebyshev polynomials are used to approximate unknown trajectory and the integrand of Picard iteration. The discrete nodes τ used for the approximation of the state are the Chebyshev-Gauss-Lobatto nodes and are given by

$$\tau_n = -cos\left(\frac{n\pi}{N}\right), \qquad n = 0, 1, 2, \dots, N \tag{4}$$

$$\begin{bmatrix} T_0(\tau_0) & T_1(\tau_0) & \dots & T_N(\tau_0) \\ T_0(\tau_1) & T_1(\tau_1) & T_1(\tau_1) & T_N(\tau_1) \\ \vdots & \vdots & \ddots & \vdots \\ T_0(\tau_N) & T_1(\tau_N) & \dots & T_N(\tau_N) \end{bmatrix}, \tag{5}$$

where N is the total number of nodes throughout the trajectory. Notice that each row in the previous matrix is separate from the other rows, thus the calculation of this matrix can be done in parallel, where a row-wise partitioning method is performed, and each process takes an equal number of rows. The second and the third functions that are computed in parallel are the coefficient vector and the

$$g\left(\tau, x^{i-1}(\tau)\right) \sim \tfrac{1}{2} F_0^{i-1} T_0(\tau) + \textstyle\sum_{k=1}^{k=N} F_k^{i-1} T_k(\tau)\,, \tag{6}$$

where

$$\textstyle\sum_{k=1}^{k=N} F_k^{i-1} T_k(\tau) = F_1^{i-1} T_1(\tau) + \dots + F_N^{i-1} T_N(\tau). \tag{7}$$

integrand approximation, as shown in Equations (6) and (8).

$$\begin{aligned} F_k^{i-1} = {} & \frac{1}{2}\, g\left(\tau_0, x^{i-1}(\tau)\right) T_k(\tau_0) \\ & + \sum_{j=0}^{j=N-1} g\left(\tau_j, x^{i-1}(\tau)\right) T_k(\tau_j) \\ & + \frac{1}{2}\, g\left(\tau_N, x^{i-1}(\tau)\right) T_k(\tau_N), \end{aligned} \tag{8}$$

where

$$\begin{aligned} \sum_{j=0}^{j=N-1} & g\left(\tau_j, x^{i-1}(\tau)\right) T_k(\tau_j) \\ & = g\left(\tau_1, x^{i-1}(\tau)\right) T_k(\tau_1) \\ & + g\left(\tau_{N-1}, x^{i-1}(\tau)\right) T_k(\tau_{N-1}). \end{aligned} \tag{9}$$

The function that calculates the coefficient vector F takes two inputs, vector G, and Chebyshev polynomials matrix T, each row for these two inputs independently results in one entry of F. Thus, row-wise partitioning is done to those two inputs and then distributed to the processes where each worker process performs its tasks before the master process collects the results and returns them as one vector. We note that the overall time of the execution is mostly dominated by the function which is responsible for the calculation of F. Then the trajectory vector for i^{th} iteration is calculated using

$$\begin{aligned} x^i(\tau) = {} & x_0 + \sum_{j=0}^{N-1} F_j^{i-1} \int_{-1}^{\tau} T(s) ds\,, \\ & = \frac{\gamma_0^i}{2} T_0(\tau) + \sum_{k=1}^{N} \gamma_k^i T_k(\tau)\,, \end{aligned} \tag{10}$$

where x_0 is shown in Equations (11). The first and last entries of the trajectory vector represent the boundary conditions stated for the problem. These two entries are found by applying the value of τ in the boundary conditions, namely $\tau = -1$ and $\tau = 1$.

$$x_0 = x(-1) = \frac{\gamma_0^i}{2} T_0(-1) + \sum_{k=1}^{N} \gamma_k^i T_k(-1) \tag{11}$$

$$x_f = x(1) = \frac{\gamma_0^i}{2} T_0(1) + \sum_{k=1}^{N} \gamma_k^i T_k(1) \tag{12}$$

In Equations (11) and (12), γ is a coefficient vector that is updated in each iteration. This coefficient can be calculated using the following formulae, which are derived in [3]:

$$\gamma_N^i = \frac{F_{N-1}^{i-1}}{2N} \tag{13}$$

$$\gamma_k^i = \frac{1}{2k}\left(F_{k-1}^{i-1} - F_{k+1}^{i-1}\right), \text{for } k = 1,2,\dots,N-1 \quad (14)$$

$$\gamma_0^i = x_f + x_0 - 2(\gamma_2 + \gamma_4 + \gamma_6 + \cdots) \quad (15)$$

$$\gamma_1^i = \frac{x_f - x_0}{2} - (\gamma_3 + \gamma_5 + \gamma_7 + \cdots). \quad (16)$$

3.1.2 MCPI for second-order ODE

The challenging part of Lambert's problem is that the differential equation is a second-order ODE, and thus, the position vector x along with the velocity vector v will be updated in each iteration.

The second-order ODE has the form of Equation (17) and can be solved using a cascaded MCPI formulation. In this formulation, the velocity calculation is done following the approach mentioned in the previous subsection, and the position is integrated directly from the approximated velocity.

In the case of the first-order ODE, x is updated using the formula

$$\frac{d^2x}{dt^2} = f(t,x,\dot{x}). \quad (17)$$

To perform the second integration, we use the fact that the velocity is the time derivative of the position to obtain the second differential equation. Thus, our main differential equations become:

$$\frac{dv}{d\tau} = g(\tau,x,v) \quad (18)$$

and,

$$\frac{dx}{d\tau} = v. \quad (19)$$

Then using the fundamentals of Picard iteration, these two equations can be transformed to the following:

$$v^i(\tau) = v_0 + \int_{-1}^{\tau} g\left(s, x^{i-1}(s), v^{i-1}(s)\right) ds, \quad \text{for } i = 1,2,\dots, \quad (20)$$

$$x^i(\tau) = x_0 + \int_{-1}^{\tau} v(s) ds \frac{dx}{d\tau}. \quad (21)$$

3.1.3 Matrix-Vector form

Since our calculations are done for each node along the trajectory, we can organize our results regarding each function in a matrix form. This matrix-vector form provides the possibility of solving the problem in parallel by dividing the tasks on multiple processors.

The solution update formulas for the velocity and position are shown below:

$$V^i = C_x C_\alpha g_{prev.} + C_x \Theta_{v0}, \quad (22)$$

$$X^i = \frac{t_f - t_0}{2} C_x C_\alpha (V^i) + C_x \Theta_{x0}, \quad (23)$$

where V^i and X^i are the i^{th} solutions for the velocity and position, respectively. C_x and C_α are constant matrices that are determined by the number of nodes n only, $g_{prev.}$ is the approximation of the integrand and Θ_0 is a vector of the boundary conditions for V and X. The flowchart for the second-order matrix-vector approach is shown in Figure 2.

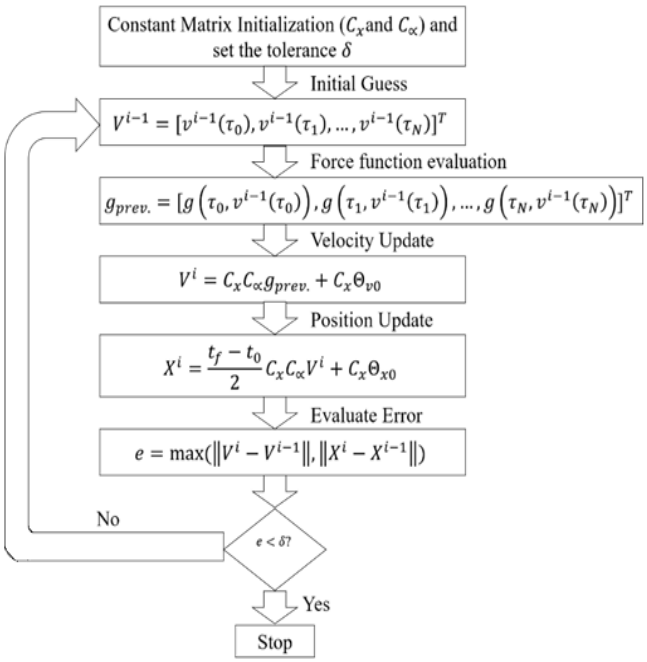

Figure 2. Matrix-vector approach of MCPI algorithm for second-order differential equations

3.1.4 Proposed solution

As mentioned in the previous sections, the MCPI algorithm is implemented through several functions, some of which are iterated several times. In this section, we discuss those that were written in parallel and the data that was transferred between processors. This exchange of data was done using Message Passing Interface (MPI) [13] library. Parallel algorithms that are implemented with the MPI library can work on both shared memory and distributed memory architectures efficiently. Applying MPI to the algorithm allows the implementation of data parallelism by distributing the matrices' rows or columns among several processors and similar instructions are executed over the distributed data.

Chebyshev polynomials matrix: This function depends on the range of nodes distributed over the interval [-1, 1], which are stored in the array τ. Chebyshev polynomials of order M are generated for each node as mentioned in Equation (3). Since the polynomials for each node are independent of the other rows, we can divide the values of

the array τ among several processors, of which each processor would be responsible for generating Chebyshev polynomials for the nodes that were sent to it. The data distribution of this function is described in Figure 3.

Force function: The parallelization of this function is the most fundamental process in the proposed solution since it defines the ODE that is being solved. In our case, we are working with a 3D problem, meaning that this function is called three times in each iteration. Each entry of the resulting column-vector is the result of the sum of multiplications of the coefficient's vector with its corresponding Chebyshev polynomials entry. Thus, the function is parallelized by sending the vector of the coefficient to each process and the rows of the Chebyshev polynomials matrix to the process responsible for them.

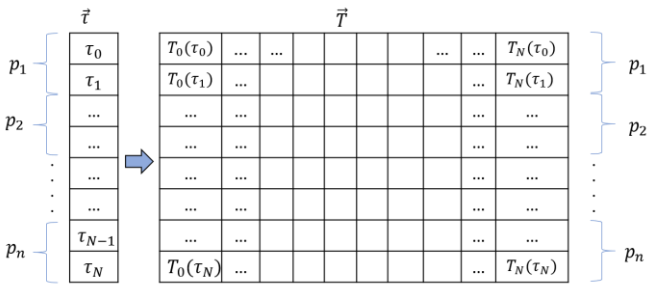

Figure 3. Description of the parallel Chebyshev polynomials function

Chebyshev coefficients: The result of this function is a column vector containing coefficients for each order of the Chebyshev polynomials. The computation of the result of this function is similar to the force function; thus, the parallelization is implemented similarly.

The matrix partitioning, data distribution, and parallelization of the Chebyshev coefficient's function and the force function are described in Figure 4. Here, each processor p_i is responsible for computations of subsequent rows of the matrix and consecutive entries of the vector according to the owner-compute rule.

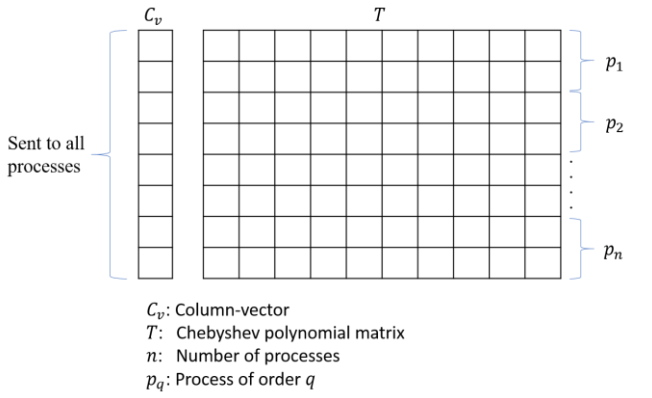

Figure 4. Data distribution of the parallel force function and coefficient function

3.1.5 Implementation details

The parallel algorithm is implemented with the Python programming language. We have used the MPI library to distribute, synchronize, and gather data between distributed processes in our parallel algorithm. We have exploited the MPI4Py package which provides bindings of the MPI routines for Python, allowing it to exploit multiple distributed processors. In the experiments, Python version 3.0 and OpenMPI version 4.0.0 MPI implementations were used.

4 Experimental results

The results that are presented in this section are discussed and analyzed in terms of two main concepts: speed-up and efficiency. Speed-up is the factor of reduction in the execution time of the parallel implementation against the sequential implementation. According to the values of speed-up, it can be categorized into three types; linear: if the value of speed-up is equal to the number of processors, sub-linear: if the value of the speed-up is less than the number of processors, and super-linear: if the value of speed-up is larger than the number of processors. Speed-up values can be obtained using the following equation:

$$Speed\text{-}up = \frac{Time\ of\ the\ sequential\ algorithm}{Time\ of\ the\ parallel\ algorithm}. \qquad (24)$$

The parallel efficiency measures the effectiveness of the algorithm in a parallel environment. In other words, the efficiency can be used to measure the computing power deficiency in parallel execution. This ratio is obtained by the following formula:

$$Efficiency = \frac{Speed\text{-}up}{Number\ of\ processors}. \qquad (25)$$

The experiments were conducted on two different architectures. The first experiment was conducted on a shared memory architecture with Intel i7-5930K 6-core CPU and 32GB memory and for the second we used a distributed memory HPC cluster named Barbun HPC Supercomputer in TUBITAK ULAKBIM High Performance and Grid Computing Center (TRUBA), where each computation node has the following configuration: 2 sockets, 20 cores per socket, Intel Xeon Gold 6148 CPU and 384 GBs of main memory. In both experiments, the number of trajectory nodes was set to N = 5000 and the tolerance value to 0.0001. In operational cases, the number of trajectory nodes decided depends on the accuracy of the trajectory intended to be found.

Tables 1 and 2 show the execution time alongside the speed-up and the efficiency for the experiment on the shared memory machine and the HPC cluster, respectively. As can be seen in the tables, the speed-up values increase when the number of cores used in the parallel run increases. The proposed parallel algorithm gets sub-linear speed-up values for all test cases since the obtained speed-up values are less

than the number of processors used, and the efficiency is always less than one.

The proposed algorithm achieves the highest speed-up on the shared memory machine with 3.94 when it uses 6 cores, and the efficiency is 0.66 for this test. The proposed algorithm achieves the highest speed-up on the HPC cluster with 8.26 speed-up on 80 processors with an efficiency of 0.10. According to Amdahl's law [19], this experimental finding is expected since the sequential portion of the algorithm cannot be parallelized, and the portion of the sequential part increases proportionally for large number of processes. Furthermore, for large number of processors, the communication overhead increases as well. These two factors limit the speed-up for the larger number of cores, and it causes lower efficiency values in the parallel system.

Table 1. Speed-up and efficiency results for the shared memory system

#Cores	Execution Time (s)	Speed-up	Efficiency
1	147.50	1.00	1.00
2	82.52	1.79	0.89
3	59.37	2.48	0.83
4	52.06	2.83	0.71
5	42.05	3.51	0.70
6	37.48	3.94	0.66

Table 2. Speed-up and efficiency results for the HPC cluster

#Cores	Execution Time (s)	Speed-up	Efficiency
1	163.66	1.00	1.00
4	57.20	2.86	0.72
8	42.90	3.82	0.48
16	30.75	5.32	0.33
32	26.49	6.18	0.19
48	20.93	7.82	0.16
64	20.12	8.13	0.13
80	19.82	8.26	0.10

Figures 5 and 7 depict the obtained speed-up values for each number of cores respectively for the shared memory and HPC platforms. Additionally, Figures 6 and 8 show the respective efficiency values for each number of cores. As seen in the figures, when the number of processors increases the speed-up increases for both cases. However, the speed-up starts to stabilize in the HPC cluster with an increasing number of processors. In addition to these experiments, we have conducted extra experiments with larger problems where the number of trajectory nodes was set to N = 10.000 and N = 20.000 in the HPC cluster by using 80 cores. Since we increased the granularity of the problem in these tests, we got higher efficiency values of 0.11 and 0.15 respectively for N = 10000 and N = 20000 as expected.

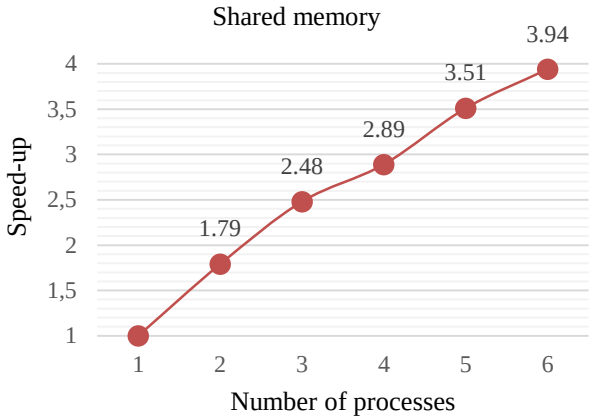

Figure 5. Speed-up versus the number of processes for shared memory machine

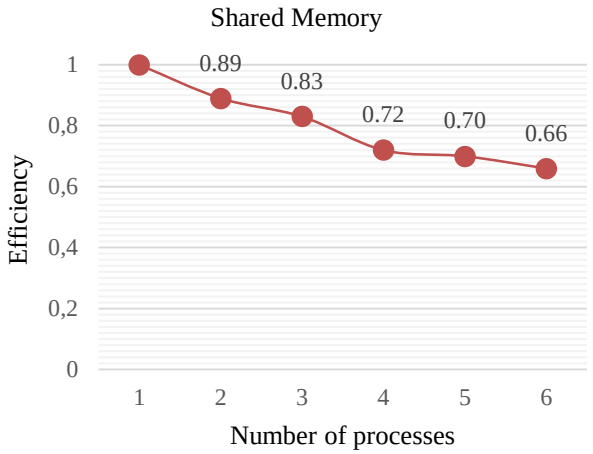

Figure 6. Efficiency versus the number of processes for shared-memory machine

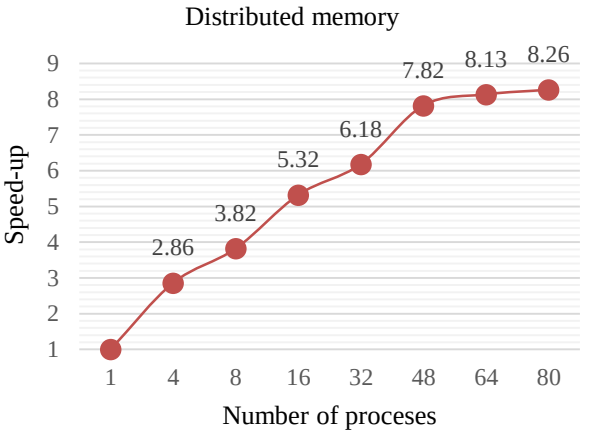

Figure 7. Speed-up versus the number of processes for HPC cluster

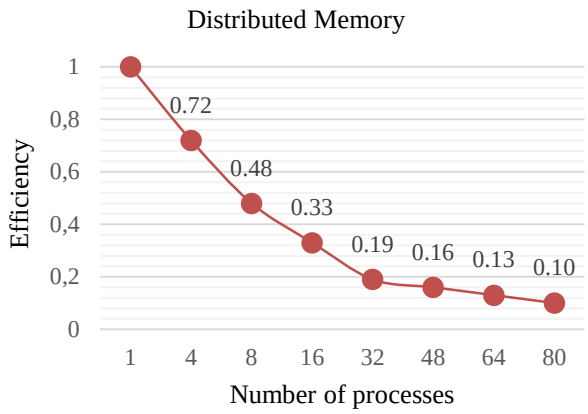

Figure 6. Efficiency versus Number of processes for HPC cluster

5 Discussion

The experimental results obtained in this work show a considerable improvement in the execution time of the MCPI algorithm on two different parallel architectures. These findings also confirm the results obtained in previous studies [4,5]. Additionally, in [5], the performance of MCPI is compared with the Runge-Kutta12 (10) algorithm and it is reported that MCPI is 11 times faster than the other algorithm. GPU-based parallelism is also proposed in [3] for the MCPI algorithm. In [3], for performance analysis, the execution time of a sequential MATLAB program is compared with a CUDA-based parallel program written in C/C++ programming language. When N is set to 511, it is reported that the CUDA-based algorithm is 34 times faster than the corresponding MATLAB program by using Nvidia 9400 GT (includes 16 CUDA cores) GPU card. Although it seems a high speed-up of 34 is achieved, we note that MATLAB programs are known to be much slower than similar C programs. For instance, one recent study [20] reports that their MATLAB program is approximately 30 times slower than the corresponding C++ program.

6 Conclusions

This study presents the parallelization of the solution of Lambert's problem using the Modified Chebyshev-Picard Iteration (MCPI) algorithm. It provides a parallel algorithm and its implementation details for astrodynamics methods by applying the parallel version of the MCPI solver to the famous Lambert's problem. Since the problem is three-dimensional, the biggest challenge is reorganizing the data in each iteration to avoid the corruption of data in the communication step between the parallel processes. We have conducted experimental results and presented the performance of the algorithm on two different parallel architectures. The experimental results on an HPC cluster show that the proposed parallel algorithm achieves 8.26 times faster execution time compared to the sequential algorithm. The proposed algorithm also achieves 3.83 times faster execution time compared to the sequential algorithm on a 6-core shared memory system. As future work, we intend to improve the parallel solution time further by parallelizing the integrand approximation function.

Acknowledgment

The numerical calculations reported in this paper were fully/partially performed at TUBITAK ULAKBIM, High Performance and Grid Computing Center (TRUBA resources).

Conflict of interest

The authors declare that there is no conflict of interest.

Similarity rate (iThenticate): 11%

References

[1] C. İnal, B. Bilgen, S. Bülbül and M. Başbük, Farklı uydu sistemi kombinasyonlarının gerçek zamanlı hassas nokta konumlamaya etkisi. Niğde Ömer Halisdemir Üniversitesi Mühendislik Bilimleri Dergisi, 11(1), 109-115, 2022. https://doi.org/10.28948/ngumuh.996018

[2] H. D. Curtis, Orbital mechanics for engineering students, Elsevier, Florida, 2005. https://doi.org/10.1016/B978-0-08-097747-8.00003-7

[3] X. Bai, Modified Chebyshev-Picard iteration method for solution of boundary value problems. Ph.D dissertation, Texas A&M University, Texas, 2010.

[4] J. L. Junkins, A. B. Younes, R. M. Woollands, and X. Bai, Picard iteration, Chebyshev polynomials and Chebyshev-Picard methods: Application in astrodynamics. The Journal of Astronautical Sciences, vol. 60, no. 3–4, pp. 623–653, 2013. https://doi.org/10.1007/s40295-015-0061-1

[5] R. M. Woollands, J. L. Read, A. B. Probe, and J. L. Junkins, Multiple revolution solutions for the perturbed lambert problem using the method of particular solutions and Picard iteration. The Journal of Astronautical Sciences, vol. 64, no. 4, pp. 361–378, 2017. https://doi.org/10.1007/s40295-017-0116-6

[6] P. B. Bailey, Nonlinear two point boundary value problems, 1st ed., vol. 44, NX Amsterdam, The Netherlands: Elsevier B.V., pp. 21–49, 1968. https://doi.org/10.1090/S0002-9904-1969-12263-9

[7] J. C. Mason and D. Handscomb, Chebyshev polynomials. Boca Raton: Chapman & Hall/CRC, 2003. https://doi.org/10.1201/9781420036114

[8] C. W. Clenshaw and H. J. Norton, The solution of nonlinear ordinary differential equations in Chebyshev series. The Computer Journal, vol. 6, no. 1, pp. 88–92, 1963. https://doi.org/10.1093/comjnl/6.1.88

[9] T. Feagin and P. Nacozy, Matrix formulation of the Picard method for parallel computation. Celestial Mechanics and Dynamical Astronomy, vol. 29, no. 2, pp. 107–115, 1983. https://doi.org/10.1007/BF01232802

[10] J. Shaver, Formulation and evaluation of parallel algorithms for the orbit determination problem. Ph.D dissertation, United States Airforce, 1980.

[11] T. Fukushima, Vector integration of dynamical motions by the Picard-Chebyshev method. The Astronomical

Journal, vol. 113, p. 2325, 1997. https://doi.org/10.1086/118443

[12] J. C. McDowell, The low earth orbit satellite population and impacts of the SpaceX Starlink constellation. The Astrophysical Journal Letters 892.2 (2020): L36. https://doi.org/10.3847/2041-8213/ab8016

[13] W. Gropp, E. Lusk, N. Doss and A. Skjellum. A high-performance, portable implementation of the MPI message passing interface standard. Parallel Computing, vol. 22, no. 6, pp. 789–828, 1996. https://doi.org/10.1016/0167-8191(96)00024-5

[14] T. Fukushima, Picard iteration method, Chebyshev polynomial approximation, and global numerical integration of dynamical motions. The Astronomical Journal, vol. 113, pp. 1909–1914, 1997. https://doi.org/10.1086/118404

[15] G. Miel, Numerical solution on parallel processors of two-point boundary-value problems of astrodynamics. Numerical Solution of Integral Equations. Springer, Boston, MA, 1990. pp, 131-182. https://doi.org/10.1007/978-1-4899-2593-0_4

[16] B. Macomber, A. Probe, R. Woollands, and J. L. Junkins, Parallel Modified-Chebyshev Picard iteration for orbit catalog propagation and Monte Carlo analysis. 38th Annual AAS/AIAA Guidance and Control Conference, Breckenridge, USA, Jan 2015.

[17] A. Probe, B. Macomber, J. Read, R. Woollands, A. Masher, and J. Junkins, Efficient conjunction assessment using modified Chebyshev picard iteration. Proceedings of the Advanced Maui Optical and Space Surveillance Technologies Conference, Maui, Hawaii, 2015.

[18] C. T. Shelton, Adaptive and orbital element methods for conjunction analysis. Ph.D dissertation, Texas A&M University, Texas, 2020.

[19] G. M. Amdahl, Computer architecture and Amdahl's law. Computer 46.12, 2013. https://doi.org/10.1109/MC.2013.418

[20] Q. Do, S. Acuña, J. I. Kristiansen, K. Agarwal and P. H. Ha, Highly efficient and scalable framework for high-speed super-resolution microscopy. IEEE Access, vol. 9, pp. 97053-97067, 2021. https://doi.org/10.1109/ACCESS.2021.3094840