On the parallelization and performance analysis of Barnes–Hut algorithm using Java parallel platforms

Badri Munier¹ · Muhammad Aleem² · Majid Khan³ · Muhammad Arshad Islam² · Muhammad Azhar Iqbal⁴ · Muhammad Kamran Khattak⁵

Received: 27 September 2019 / Accepted: 2 March 2020 / Published online: 10 March 2020
© Springer Nature Switzerland AG 2020

Abstract

Multi-core processors provide time-efficient and cost-effective solutions to execute the algorithms for complex physical systems. However, to efficiently exploit the processing capabilities of the underlying architectures, the applications executing on multi-core processors must be parallelized. Conventionally, the applications for high-performance computing (HPC) are written in *native* (programming) languages. However, several researchers have argued that Java can be an excellent alternative for writing HPC applications. To gauge the comparative performance of Java parallel platforms, a benchmark study is required considering some real HPC applications. In this study, as a pioneer application, Java-based parallel platforms, namely Java thread API, JavaSymphony, and MPJ Express, are used to parallelize and benchmark the Barnes–Hut algorithm to simulate the N-body physical system of celestial objects. The parallel implementations of the Barnes–Hut algorithm are tested for performance on shared memory multi-core architectures. The hardware-level performance analysis involving the parameters like the number of instructions executed by the CPU, level-1 and level-3 cache misses, and the number of main memory accesses is conducted to investigate the insights of the attained performances. The celestial objects up to one million are simulated, and the results revealed that the JavaSymphony-based implementation of the Barnes–Hut algorithm produces better results as compared to the other employed parallel frameworks.

Keywords Barnes–Hut algorithm · High-performance computing · Java parallel frameworks · Locality optimization

1 Introduction

In the modern era, multi-core processors render the feasibility in attaining the higher performance in terms of execution times through efficient distribution of workload [1]. However, if the applications running on multi-core processors are not parallelized, they might produce unsatisfactory results adhering to the huge time complexity than the applications running on a single-core processor. Parallelism can be added at both hardware and software levels [2] to fulfill the demands of high-performance computing (HPC) applications. Due to ever-increasing demands

of HPC in different applications like in space exploration, simulation of physical systems (terrestrial, stellar, or interstellar), defense technologies, financial and economic modeling, web search engines, networked video and multimedia technologies, web-based business services, collaborative work environments, etc.), parallel architectures have become the dominant paradigm in the modern computer industry [3]. Although, HPC-based applications are mostly developed in native programming languages like C, C++, UPC, and Fortran. However, various researchers have argued that Java could be deemed as an alternative for developing HPC applications [4]. JavaSymphony [5]

✉ Muhammad Aleem, m.aleem@nu.edu.pk | ¹Capital University of Science and Technology, Islamabad, Pakistan. ²Department of Computer Science, National University of Computer and Emerging Sciences, Islamabad 44000, Pakistan. ³Department of Physics, Quaid-i-Azam University, Islamabad 44000, Pakistan. ⁴SWJTU-Leeds Joint School, Southwest Jiaotong University, Chengdu, China. ⁵Department of Physics, COMSATS University Islamabad, Park Road, Chak Shahzad, Islamabad 44000, Pakistan.

and MPJ Express [6] are two Java-based parallel programming platforms that have been developed to assist programmers in parallelizing the applications on multi-core processors.

In this paper, a real HPC application i.e., *Barnes–Hut algorithm* [7] is parallelized using the Java-based parallel platform to analyze the potential of Java parallel platforms in terms of execution time and low-level performance analysis (considering several hardware-level parameters). To the best of our knowledge, as a first application, it is benchmarked for the simulation of a complex physical system involving a large number of interacting bodies. The significance of such a system is evident from the fact that it has a wide range of applications. To implement the proposed idea, we have harnessed the Java language's threads API [8], JavaSymphony [5], and MPJ Express [6]. A detailed hardware-level performance analysis is conducted to evaluate the performance of these Java-based parallel platforms, using performance counters for Linux (PERF) [9] and OProfile [10] tools. The obtained results are comprehensively analyzed based on a comparative analysis of the above-stated three Java-based parallel programming frameworks.

The rest of the paper is organized as follows: Sect. 2 provides a brief overview of the Barnes–Hut Algorithm. Section 3 explains the literature about previous HPC implementations of the Barnes–Hut algorithm, details about parallelism in Java, and discussion about Java parallel platforms. Section 4 describes the overall methodology. Section 5 presents results and discussion, and Sect. 6 provides the overall conclusion of this study.

2 Barnes–Hut algorithm

In 1986, Joshua Barnes and Piet Hut [7] proposed *A hierarchical $O(N \log N)$ force calculation algorithm* to reduce the number of force calculations in the N-body problem. Joshua Barnes and Piet Hut observed that if a group of bodies is far enough from a specific body in the system, the force exerted on this body by the bodies in the group can be approximated by assuming that all the bodies in the group are located at the center of mass of the group. Hence, a single force from the center of mass of the group is sufficient instead of calculating individual forces for the whole group. This proposed technique was based on a tree-structured hierarchical subdivision of space into cubic cells, if any of the cells contain more than one object, then each of them is recursively divided into eight sub-cells (Fig. 1). The algorithm comprises the following three main steps:

Step I Division of gravitational space into virtual cubic sub-cells (exactly half the length, width, and breadth of their parent cells);

Fig. 1 Barnes–Hut force calculation algorithm (grouping objects in a cell)

Step II Tree construction from the virtual cubes by (a) discarding all the empty cells, (b) accepting daughter cells having only one object, and (c) Recursively dividing daughter cells having more than one objects;

Step III Tree construction at every time step from the beginning because of the constant movement of the objects.

The force of attraction is calculated for every non-empty cell and higher-order cells that contain more than one object. A pseudo-object that contains a total mass in the cell and is present at the center of mass of the cell is defined for each cell. A particular real object experiences the force of attraction from all the pseudo-objects in the system that represents a cell small enough and far enough to forego further divisions. The force on a particle in the system can be evaluated by “walking” down the tree level by level beginning with the top cell [11]. An approximated force for the cell is calculated if the length l of any side of the cell divided by the distance d between the object and center of mass of the cell is less than θ . Otherwise, the cell is divided into sub-cells, wherein θ is a fixed value ≈ 1 .

3 Related work

The Barnes–Hut algorithm is a versatile application for simulating astronomical objects. The algorithm has previously been parallelized by several researchers. The Barnes–Hut algorithm has been optimized in UPC [12] by Zhang et al by (1) replicating shared scalar variables (global parameters) which are accessed iteratively by parallel threads in tree-building step, (2) body redistribution (local access of

using shared memory), (3) cache remote nodes (to reduce remote access penalties), (4) octree building (using local octrees that are merged to form a global tree) using Singh's tree-building algorithm [13], and (5) non-blocking communication and message aggregation (to overlap communication with computation).

A highly scalable implementation of the Barnes–Hut algorithm for pure particle-based experimentations [14] simulates up to 2 billion particles. In [14], a novel approach has been presented that combines MPI and POSIX threads for tree traversal algorithm. In the study, the load-balancing strategy ensures the evenly distributed workloads even for clustered particle sets. The algorithm has been optimized to diminish the overhead produced by the parallel tree data structure. The C language-based code has a wide range of applications including creation and transport of laser-accelerated ion beams, plasma-wall interactions in tokamaks (strongly coupled plasmas and vortex fluids). Java is one of the most popular programming languages in the market today. However, the use of the Java language to write applications for HPC is not up to its full potential. Many researchers have favored the notion that Java could be an excellent alternative for developing HPC applications [15].

Taboada et al. [15] have analyzed the current state of Java for HPC, both for shared and distributed memory programming. Recently, researchers have observed that Java can achieve almost similar performance as of other native languages for sequential as well as for parallel applications. The parallelization and benchmarking of several scientific applications indicate that Java is an appropriate high-level language to program HPC applications. Another study [4] has given an overview of the current state of Java for HPC. A comparison in the performance of Java implementation of the NAS Parallel Benchmarks (NPBs) [16] to a Fortran MPI has been performed. Overall, the study has attained good results.

The results have further suggested that the overhead of Java has become acceptable for computationally intensive tasks. The Java bytecode is portable on any hardware with a compliant JVM. The lower performance of Java can be compensated using optimized network layers. The issue has already been addressed by the HPC community showing interesting results. The interest in Java to program parallel architectures is growing rapidly in the software industry. Numerous Java-based parallel programming platforms have been developed to assist programmers in writing Java-based parallel applications ranging from desktop applications to distributed memory cluster applications. Aleem et al. [5] have proposed JavaSymphony, Java-based programming, and execution environment for programming and scheduling performance-based applications for multi-core parallel architectures. JavaSymphony was originally designed to

program distributed memory heterogeneous clusters and computational grids. The idea of JavaSymphony pertains to the provisioning of distributed virtual architecture for modeling the hierarchical resources in distributed architectures. These resources may range from individual shared memory multi-core machines to more complex distributed memory clusters. The architecture allows distributing, migrating, and explicitly invoking objects. JavaSymphony enables the end-users to explicitly map objects and tasks to the computing nodes. The user-controlled locality of tasks, objects, and parallel applications often results in improved performance as compared to other alternate Java parallel programming frameworks.

Javed et al. [6] have proposed a thread-safe Java messaging library called MPJ Express (MPJE) based on the specifications of *mpiJava* 1.2 API. The MPJE implements MPI-like binding for applications written in Java language [17]. The MPJE was designed to ensure thread safety during communications for applications developed in Java for multi-core clusters. The MPJE uses direct byte buffer to store and retrieve messages at the sending and receiving ends, avoiding data copying during communications. Initially, two thread-safe communication devices *niodev* and *mxdev* were introduced in the MPJE. Both of these devices are based on pure Java. A Java-based multi-core device was added to the API to program shared memory multi-core devices. The MPJE provides both pure Java-based and native MPI-based communication techniques. To the best of our knowledge, none of the existing studies have performed the parallelization and benchmarking of the Barnes–Hut algorithm using Java-based parallel platforms and hardware-level performance analysis. This study is a pioneer in evaluating the suitability of Java parallel platforms for HPC applications like the *Barnes–Hut algorithm*. Furthermore, the study also encourages the parallelization and benchmarking of other HPC applications similar in computing and communication requirements as the Barnes–Hut algorithm. Table 1 recapitulates the contemporary state-of-the-art studies along with the parallel implementations of the Barnes–Hut algorithm. The previous studies have implemented the parallel algorithm on different compute clusters. The programming platforms used to develop the parallel algorithm are mostly HPC native platforms. Particles ranging from 12,000 to 2 billion have been simulated in the reviewed studies. Only one of these studies has used a Java-based parallel platform to parallelize the Barnes–Hut algorithm. However, the benchmarking of this Java-based parallel implementation has been performed using the C language version of the Barnes–Hut algorithm. Furthermore, none of them have conducted a hardware-level performance analysis to investigate the hardware performance counters for these implementations of the algorithm.

Table 1 Summary of related work (Barnes–Hut parallel implementations)

References	Hardware specifications	Programming platform	No. of particles simulated	Hardware-level performance evaluation
Zhang et al. [12]	IBM Power5 cluster with 118 nodes (16 cores and 64 GB memory for each node)	UPC	2 M	X
Zhang et al. [18]	x86 Linux Infiniband cluster. Each node has two hex-core Intel Xeon 5650 CPUs running at 2.67 GHz	UPC, C++	1–4 M	X
Dinan et al. [19]	887-node IBM 1350 Cluster	UPC, MPI	150,000	X
Hamada et al. [20]	GPU cluster using 128 NVIDIA GeForce 8800GTS GPUs	C++, OpenMP	562 M	X
Javed et al. [6]	StarBug cluster consisting of 8 dual Intel Xeon 2.8 GHz processors with 2 GB RAM	C, Java	12,000	X
Winkel et al. [14]	288 K cores of IBM Blue Gene/P system JUGENE	Fortran, C	2 billion	X

4 Methodology

Over the last decade, various studies have parallelized the *Barnes–Hut algorithm* for shared memory as well as for distributed memory architectures. However, the algorithm has mostly been parallelized using the native languages of HPC such as UPC [21] and Fortran [22]. Various researchers have contended that Java could be deemed as a potential language for high-performance computing.

In this paper, we propose parallelization and benchmarking of the Barnes–Hut algorithm using the Java-based

parallel platforms namely Java threads API [8], JavaSymphony [5], and MPJ Express [6]. Each of these Java-based parallel programming platforms has its own predefined mechanism of developing parallel applications for shared and distributed memory architectures. However, the generic parallelization strategy adopted in this research is similar (with implementation constrained changes) for all these Java-based parallel frameworks.

The overall structure of the proposed methodology is shown in Fig. 2. At the initial phase, the Java-based serial Barnes–Hut algorithm is tested for the most

Fig. 2 Proposed methodology

time-consuming parts (hotspots) of the algorithm using the OProfile tool.

The *OProfile tool* has provided the profiling that shows that the tree construction and the force calculation parts of the algorithm are the most time-consuming. A detailed data dependence analysis of these two hotspots is then conducted to confirm the parallelizability of these parts of the algorithm. Once the parallelizability has been ensured, these parts are converted into parallel tasks. Thereafter, the parallel algorithm is implemented using Java language's thread API, JavaSymphony, and MPJ Express. The JavaSymphony-based implementation is further extended into two categories. One of the implementations implies JavaSymphony-based parallel implementation for shared memory multi-core systems, while the second implementation implies a distinctive feature of JavaSymphony called the *user-controlled locality*. The user-controlled locality feature of JavaSymphony enables the programmers to control the locality of the parallel tasks, objects, and applications, which are currently being executed. Afterward, a low-level hardware performance analysis is carried out,

involving the *number of instructions* executed by the CPU, *level-1 cache misses*, *level-3 cache misses*, and the *number of main memory accesses* for each parallel implementation of the algorithm using the PERF tool. In the end, benchmarking and performance analysis are carried out for all the four parallel implementations of the algorithm. The parallelization strategy and details regarding four algorithm implementations are delineated in the following subsections. Moreover, we develop and employ the distributed memory versions of the application and experiment using a cluster of 10 multi-core machines.

4.1 Parallelization strategy

In the proposed study and in several other studies [7, 11, 12, 20], the OProfile tests conducted on the serial Barnes–Hut algorithm for hotspots confirm that the tree construction and the force computation parts are the most time-consuming code sections (i.e., target hotspots for parallelization) in the Barnes–Hut algorithm. Considering the compute-intensive code parts i.e., hotspots, the following parallel version of the algorithm is designed:

Algorithm 1: Barnes-Hut algorithm

Input : Particles /* read input */

```

1 foreach core do
2   Particles_per_core =  $\frac{\text{TotalNumberofParticles}}{\text{TotalNumberofCores}}$ 
3   Total Number of Particles -= Particles_per_core
4   Total Number of Cores -= 1
5   for iteration=0 to max.iteration do
6     octree tree = new octree()
7     foreach particles p that  $\in$  bodies do
8       | tree.insert (p)
9     end
10    foreach Sub_tree st that  $\in$  tree do
11      | st.CalculateTotalMass()
12      | st.CalculateCenterofMass()
13    end
14    foreach particle p that  $\in$  bodies do
15      | p.ComputeForce(tree)
16    end
17    foreach particle p that  $\in$  bodies do
18      | p.Update() /* update velocity and position of p */
19    end
20    Increment iteration
21  end
22 end

```

To ensure a balanced work-distribution, we partition the particles into equal parts to map on different computing cores. Algorithm 1 executes for each computing core assigning an equal workload to each core (lines 2–3, Algorithm 1). The parallel code parts are mentioned for each loop constructs in Algorithm 1. After assigning the load, the current compute core is excluded from the available cores (line 4, Algorithm 1). Lines 5–21 iterate for the total iterations to calculate the force values of each assigned particle. First, an *octree* object is constructed (line 6, Algorithm 1) and each body particle is added to that tree (lines 7–9, Algorithm 1). For each octree, all of its subtrees are employed and a *total mass* and the *center of mass* are calculated for each subtree (lines 10–13, Algorithm 1). Afterward, for each particle in the tree, a force value is calculated and the specific particle is updated (lines 14–19, Algorithm 1). These local trees are then merged to form a single application-level global tree by the main computing thread (main program). The global tree accommodates all the particles from all the parallel tasks. Then, each parallel task computes forces of attractions for the assigned set of particles. This scheme of parallel computing ensures an excellent load-balancing and least synchronization overhead. These parallel computations are carried out until all the participating objects-related gravitation forces are calculated.

4.2 Parallel implementations

This section encompasses the details regarding parallel implementations of the Barnes–Hut algorithm using Java thread API, JavaSymphony, and MPJ Express.

4.2.1 Java-based parallel implementation

The first version of the parallel Barnes–Hut algorithm is implemented using the Java-based parallel constructs, the Java threads API [8]. There are two parallelization mechanisms of serial applications in Java. One of them is based on extending the *Runnable* interface, and the second method is based on extending the thread class *Java* API. In this study, the parallel tasks of the Barnes–Hut application are implemented by extending the *Runnable* interface of the Java language. The attributes such as mass, velocity, and position of the gravitational object in three-dimensional space are provided using an input-related implementation class. The above-stated attributes of each object are passed to the constructor of the main class (main program). The input class contains the data for only one thousand objects. These attributes (i.e., mass, velocity, and position) of the celestial objects are

reused for the next one thousand objects with minor amendments in the parameters.

These updates in parameters are rendered dynamically by the algorithm. The updated parameters are used to differentiate the new objects from the previously instantiated celestial objects. We employ this method of instantiating the objects to reduce memory usage (especially of the input data files) by the implemented programs. These object attributes are stored in an array by the main program. Each thread accesses this array to retrieve attributes for the number of objects assigned to it. The main loop of the program repeats the tree construction, inserting objects in the tree, and force calculation in each time step. Each thread creates a local tree and inserts the objects in it. Thereafter, these local trees from all the parallel tasks are combined with the main program to form a single application-level global tree. Each parallel task then calculates the forces of attraction for each object within the assigned subtree. Finally, the calculations performed by the individual threads are aggregated into the final result.

4.2.2 JavaSymphony-based parallel implementations

The JavaSymphony-based parallel implementation differs from the Java-based implementation in the way that JavaSymphony uses the *SJSObjects* (of the JavaSymphony API) for parallelizing applications on shared memory multi-core machines. The *SJSObjects* employ a worker class that implements the parallelization logic of the *SJSObject*. An *SJSObject* can be based on a single- or multi-threaded execution models. A single-threaded *SJSObject* represents a parallel task (encapsulated by the *SJSObject*) that is executed at most by a single executor thread of the JavaSymphony runtime system. A parallel JS program based on single-threaded *SJSObjects* represents a multi-object program, wherein each object's worker task is being executed by a single executor thread of the JavaSymphony runtime system. A multi-threaded *SJSObject*-based parallel task is executed by the multiple concurrent executor threads of JavaSymphony runtime. As the algorithm is implemented on shared memory multi-core systems; therefore, for the parallel implementation of the Barnes–Hut algorithm, we employ multi-threaded *SJSObjects*. The *SJSObjects* can be invoked using three types of method invocations: (1) synchronous, (2) asynchronous, and (3) one-sided method invocations. In this implementation, we employ asynchronous method invocations to enable parallel execution of the invoked methods. A one-sided method invocation may also be employed for parallel invocation; however, it does not return any resultant value. The tree construction, insertion of objects in the tree, and force

calculations by retrieving objects from the tree are implemented in the worker class (implemented as SJS based). The attributes of the celestial objects are passed to the worker class by reference to enable the sharing of the instantiated data instead of data copying. The worker class constructs the local tree and computes the forces of attractions for all the objects assigned to a single thread. The local trees constructed for each thread are merged to build a global tree by the main program prior to the force computations for each object in all the threads. The computed forces of attractions by each thread are summed up by the main program. To further improve the performance of the JS-based parallel implementation of the Barnes–Hut algorithm, we developed a second version of the JS-based implementation by utilizing the user-controlled locality feature of the JavaSymphony platform. This feature enables the programmer to control locality at tasks, objects, and application levels. We employed the user-controlled locality feature to bind each parallel task to a specific core so that the CPU migrations overhead can be reduced to a minimum.

4.2.3 MPJ Express-based parallel implementation

To develop applications for shared memory multi-core architectures, the MPJ Express introduces the multi-core device programming constructs. This multi-core-specific device object is added to the *xdev* layer of the MPJE architecture [23]. The multi-core device of the MPJE employs Java threads to exploit parallelism in the proposed application. The MPJ Express threads are created in a single JVM and can communicate with each other through the global shared memory. The input containing mass, velocity, and position in 3-dimensional space, etc. of the celestial objects is provided to the MPJE-based parallel implementation of the Barnes–Hut algorithm using external binary files. The input is read into the constructor of the main class and is stored in an array. The algorithm uses these attributes to calculate the forces of attraction for a celestial object. Each thread (i.e., MPJE process) receives an equal number of celestial objects to compute the forces of attraction. Each thread creates a local tree and inserts the

objects in the tree for the force computation step. Local trees constructed by all the MPJE processes are joined together to form a global tree by the main program. Each parallel task (MPJE process) computes the forces of attractions for the assigned set of celestial objects. All the forces computed by the individual processes are then summed up by the main program.

4.3 Benchmarking and low-level performance analysis

This section delineates the benchmarking and performance analysis of the Java-based parallel implementations of the Barnes–Hut algorithm. The main purpose of the benchmarking is to investigate the comparative performance of all the three employed Java-based parallel frameworks and analysis of the obtained performances. For detailed performance analysis, the low-level hardware performance counters are measured along with the execution times of the parallel executions. The CPU hardware register maintains the record of hardware events, which are utilized by performance tools to profile an application executing on a system. Performance tools can provide per task, per CPU, and per-workload hardware counters measurements [9] that reveals the true execution behavior of the applications. Performance counters such as a number of instructions processed, level-1 cache misses, and last-level cache misses, etc. provide the information pertaining to the hardware utilization of these Java-based implementations. These performance counters also enable us to analyze the bottlenecks in an application, which could benefit the possible optimizations or the process of decision making for picking a particular parallel framework for parallelization of scientific work such as the Barnes–Hut algorithms. In this study, we employ PERF [9] and OProfile [10] tools to analyze the hardware-level performance counters.

Numerous performance counters can be obtained using tools like PERF and OProfile. However, this study contemplates only five types of performance counters which could be interesting for the analysis of scientific

Table 2 Hardware performance counters

S. no.	CPU performance counters	Explanation
1	Time elapsed	The total time spent while executing a task on a computer system (wall clock time)
2	No. of instructions	A number of instructions executed by the CPU while executing a program
3	L1-cache misses	A number of misses occurred through the level-1 cache
4	L3-cache misses	A number of misses occurred while loading instructions or data from level-3 cache (last-level cache)
5	Main memory accesses	A number of times the main memory is accessed for data fetching

Table 3 Experimental setup (hardware)

Hardware/software environment	Specification
CPU model	Intel Core i7-6700 CPU
CPU clock speed	3.4 GHz
Number of cores	4
Number of threads	8
Main memory	4 GB
Level-1 data cache	128 KB
Level-1 instruction cache	128 KB
Last-level cache	8 MB
Operating system	Ubuntu 16.04
Linux Kernel version	4.4.0-75-generic
Java standard edition version	1.8.0_101
PERF version	4.4.62
OProfile version	1.0.0

applications. Table 2 enlists these performance counters and their details.

5 Results and discussion

This section sheds a light on the results obtained by applying a methodology and the concrete analysis of the outcomes. The details of hardware and software specifications regarding the experimental phase are illustrated in Table 3 (environment for shared memory experiments).

The experimental results are based on the four parallel implementations, the pure *Java*-based parallel implementation (JMT), the *JavaSymphony* (JS)-based parallel implementation, the *JavaSymphony* + *Locality*-optimized parallel implementation, and the *MPJ Express*-based parallel implementation of the Barnes–Hut algorithm. Two versions of the *JavaSymphony*-based implementations have been developed and tested for the optimized performance. The first implementation is based on the pure JS-based parallel implementation for the shared memory multi-core architecture. In the second version, the execution performance of the JS-based parallel implementation has been further enhanced by applying the *locality optimization*. For the locality optimization, we have employed the user-controlled locality feature of the JS programming framework to enable a better placement or mapping of the parallel tasks. A locality-aware mapped application often attains the improved performance on multi-core architectures. Our results validate the foreseen performance improvement of the used locality optimization.

For the first set of experiments, 20 different data sizes of simulating celestial objects have been employed. These data sizes contain attributes for celestial objects ranging from 50,000 to 1,000,000 objects (for JS and Java-based

Fig. 3 Barnes–Hut executions (2 parallel tasks)

experiments). The two JS-based (*JS with* and *without* locality optimization) and Java-based parallel implementations have been benchmarked using 2, 4, 6, and 8 parallel tasks. In the second set of experiments, the Java, JS, and MPJ Express-based benchmarks have been conducted using the number of simulated objects ranging from 6000 to 276,498. All these four parallel versions of the algorithm have been benchmarked using the 2, 4, 6, and 8 parallel tasks. The hardware-level performance analysis based on measuring the hardware performance counters has been conducted for the largest size object configuration (i.e., 276,498 objects) for all four parallel implementations (i.e., Java, JS, JS + Locality, MPJ Express-based implementations).

The execution time of the Barnes–Hut algorithm for a number of celestial objects ranging from 50,000 to 1,000,000 for *Java-based multi-threaded version* (JMT), *JavaSymphony* (JS), and *JavaSymphony with locality optimization* (JS (Affinity)) based on 2 parallel tasks is shown in Fig. 3. The results depicted in Fig. 3 indicate that the JS (Affinity) has the lowest execution time while JMT has the highest execution time. The JS (Affinity) has outperformed

Fig. 4 Barnes–Hut executions (4 parallel tasks)

both the JMT and the default JS-based parallel versions for all the employed data sizes (i.e., all numbers of the simulated particles). As compared to the JS, *JS (Affinity)* version of the application has consumed the lower execution time (i.e., on average of 11.98%). The JS (Affinity) has consumed on average 12.5% lower execution time as compared to the JMT-based parallel executions.

The execution time of the Barnes–Hut parallel implementation based on different problem sizes (50,000 to 1,000,000 objects) is shown in Fig. 4. The parallel execution shown in the figure presents the *Java multithreaded* (JMT), *JavaSymphony* (JS), and *JavaSymphony with locality optimization* (JS (Affinity))-based executions. These results represent the parallel executions using 4 parallel tasks. The results depicted in Fig. 4 show that JS (Affinity)-based version has attained lower execution time while the JMT-based execution has the highest execution time. The JS (Affinity) has outperformed both

the JMT and the JS for all executions based on a different number of the simulated particles. In comparison with the JS-based parallel version, the *JS (Affinity)* execution has consumed lower execution time (on average 13.6%) because of the locality-based optimization. The locality-based optimization provided by the JS framework results in a smaller number of remote and more local data accesses. As compared to the JMT-based execution, the JS (Affinity) has achieved on average 14% lower execution time. The low execution time for the JS (Affinity) validates the effectiveness of the employed locality optimizations that ensure low-cost memory accesses and useful cache reuse effect.

Figure 5 shows the execution time of the Barnes–Hut parallel implementation experimented using different problem sizes (i.e., 50,000 to 1,000,000 objects). The experimental results shown in Fig. 5 exhibits that the JS (Affinity) parallel version has consumed lower execution time

Fig. 5 Barnes–Hut executions (6 parallel tasks)

Fig. 6 Barnes–Hut executions (8 parallel tasks)

Fig. 7 Java, JS, and MPJE-based comparison (2 parallel tasks)

as compared to the JMT-based parallel execution. The JS (Affinity) version has outperformed both the JMT and the JS versions for all data sizes. As compared to the JS, the JS (Affinity) attains lower execution time (on average 5.7%) because of the locality-based optimization. Moreover, compared to the JMT-based execution, the JS (Affinity) has achieved on average 5.3% lower execution time.

Figure 6 shows the execution time of the Barnes–Hut parallel implementations based on different problem sizes (i.e., 50,000 to 1,000,000 objects). These results indicate that JS (Affinity)-based version of the application has consumed the lower execution time as compared to the JMT-based execution. The JS (Affinity)-based version has outperformed both the JMT and the JS versions for all the employed data sizes. As compared to the JS, the JS (Affinity) version has consumed the lower execution time (on average 4.1%) because of the locality-based optimization. Similarly, as compared to the JMT-based execution, the JS (Affinity) version has attained on average 5.6% lower execution time. The lower execution time for the JS (Affinity) version validates the effectiveness of the employed locality that ensures low-cost memory access and the performance benefits in terms of spatial cache locality. Moreover, the results are shown for 2, 4, 6, and 8 parallel tasks (Figs. 3, 4, 5, and 6) highlight the scalability of the JS and JS (Affinity) versions that are comparable with the pure Java-based parallel implementation. The improved execution performance by the JS (Affinity) versions asserts the effectiveness of the locality optimization features, which are unavailable in the other Java frameworks.

Figure 7 shows the execution performance comparison of Barnes–Hut (data size ranging from 6,000 to 276,498 particles) parallel implementations i.e., *Java multi-threaded* (JMT), *JavaSymphony* (JS), and *JavaSymphony with locality optimization* (JS (Affinity)) and *MPJ Express* (MPJE). The results depicted in Fig. 7 represent the parallel executions based on 2 parallel tasks. These results determine that JS (Affinity) has outperformed the JMT, the default JS, and

Fig. 8 Java, JS, and MPJE-based comparison (4 parallel tasks)

Fig. 9 Java, JS, and MPJE-based comparison (6 parallel tasks)

Fig. 10 Java, JS, and MPJE-based comparison (8 parallel tasks)

the MPJE-based executions for all number of the simulated particles. As compared to the JS and the JMT, JS (Affinity) has attained an average of 11.74% and 11.92% lower execution time, respectively. As compared to the MPJE-based parallel execution, the JS (Affinity)-based execution has achieved commendable performance i.e., up to 40.36% lower execution time. The results of the cache-friendly locality optimization have enabled the JS (Affinity) version to attain commendable performance.

Figure 8 shows execution time (using 4 parallel tasks) of a Barnes–Hut algorithm for a number of particles ranging from 6000 to 276,498 employed for JMT, JS, JS (Affinity), and MPJE-based parallel implementations. These outcomes indicate that JS (Affinity) has outperformed the JMT, the default JS, and the MPJE-based executions for all number of the simulated particles. As compared to the JS and the JMT, the JS (Affinity) has attained lower execution time on average of 8.4% and 7.6%, respectively. As compared to the MPJE-based parallel executions, the JS (Affinity) parallel execution has achieved commendable performance i.e., up to 44.4% low execution time.

Correspondingly, Figs. 9 and 10 show the execution performance of the four parallel implementations (i.e., JMT, JS, JS (Affinity), and MPJE) using 6 and 8 parallel tasks, respectively. For 6 parallel tasks, the JS (Affinity) parallel implementation has consumed 5.2% and 6.2% lower execution time as compared to JS and JMT, respectively. As compared to MPJE-based parallel execution time (for 6 threads), the JS (Affinity) has achieved commendable performance results (up to 37.3% low execution time). The improved execution performance (as compared to the other parallel implementations) attained by JS (Affinity) version shows the positive impact of the tightly mapped parallel tasks (resulting in cache-level data sharing among the parallel tasks) in contrast to MPJE that uses a separate process for each parallel task which are loosely mapped on a multi-core machine. Similarly, for 8 parallel thread, the performance depicted in Fig. 10 shows that the JS (Affinity) has outperformed the other parallel executions and has consumed on average 3.6%, 5.4%, and 28.2% lower execution time as compared to JS, JMT, and MPJE-based parallel executions, respectively.

To analyze the performance of the parallel executions, low-level hardware performance counters have been employed to measure the processor's execution events responsible for the attained performances. Figure 11 shows the count (in billions) of the four low-level hardware

performance counters measured for the two task-based parallel executions (simulating 276,498 celestial objects) of Java multi-threaded (JMT), JavaSymphony (JS), JavaSymphony with Affinity optimization (JS Affinity), and MPJ Express-based executions. Four low-level performance counters have been measured for the parallel executions, which are a total number of hardware instructions executed (Inst. Executed), L1 cache misses, last-level cache misses (LLC misses), and total memory accesses.

Figure 11 shows the hardware-level total instructions executed for each parallel implementation (i.e., JMT, JS, JS (Affinity), and MPJE) of the program. A smaller number of instructions is an indication of improved performance showing the least overhead by the runtime systems of the employed parallel frameworks. The results suggest that JS (Affinity) parallel execution has performed a smaller number of instructions as compared to default JS, JMT, and MPJE-based executions. As compared to the JS and the JMT, the JS (Affinity) has executed 2.8–4.2% smaller number of instructions, respectively. As compared to the MPJE, the number of instructions executed by JS (Affinity) has been receded to 41.1% showing a significantly low overhead of the JS runtime system and a beneficial impact of affinity-based optimizations.

Figure 11 shows the first-level (level-1) cache miss profile for parallel executions of the algorithm. A higher number of level-1 misses indicate poor performance (due to cache miss penalties). As compared to the MPJE and the JS, the JS (Affinity) parallel execution has endured 17.4% and 10.7% fewer cache misses, respectively. The low number of cache misses directs the positive performance impact because of the locality optimizations employed by the JS (Affinity) parallel implementation. A smaller number of cache misses results in reduced execution time because of the data reuse by the other tightly mapped threads (as validated in our initial execution time-based performance results). The level-1 cache misses of the JS (Affinity) version as compared to the JMT execution is 22% less, showing a significantly improved performance of the locality-optimized JS version as compared to the JMT.

Figure 11 also shows the last-level cache (LLC) misses for the parallel executions of the Barnes–Hut algorithm. The JS (Affinity)-based execution suffers 3.7%, 10.2%, and 23% less last-level cache misses as compared to JS, JMT, and MPJE-based executions, respectively. The smaller number of LLC misses specifies the reason for the improved performance of the JS-based executions. The lower LLC misses highlight reduced data access latencies for the executing applications because of a low number of memory access operations performed during execution. Whenever a cache miss occurs, the main memory is accessed to load the required *data* or *instructions* into cache incurring a higher access latency resulting in longer execution time.

Fig. 11 Performance analysis using hardware performance counters

Fig. 12 Barnes–Hut executions on the compute cluster

The results depicted in Fig. 11 show that the JS (Affinity) parallel execution attempts the lowest number of main memory accesses as compared to the other parallel executions. As compared to JS, JMT, and MPJE, the JS (Affinity) execution has attained 5%, 12.2%, and 24% lower main memory accesses, respectively. The low number of memory accesses is due to the higher rate of cache hits (a positive impact of the tightly mapped threads).

In this paper, we have evaluated the performance of Java parallel frameworks harnessing a real scientific application. To unearth the performance bottlenecks, a low-level performance analysis has been performed using hardware performance counters. The experimental results revealed that the JS-based parallel implementations have outperformed the other implementations. A prominent feature of the task, object, and application-based locality features enables JS-based executions to outperform the other parallel implementations. The locality optimizations resulted in improved performance because of the low cache miss rate and low-level performance analysis.

For distributed memory experiments, we employ a cluster of 10 machines (each based on two Quad-core 8th Generation Intel i7 Processors i7-8650U). The distributed memory versions of the application are developed using MPJ Express, JS (without any locality optimization), and JS Affinity (with locality optimization) versions. The machine size used is 4 to 80 processor cores of the employed cluster. Figure 12 shows the execution time performance for the three executions i.e., MPJ Express, JS, and JS Affinity.

As seen in Fig. 12, JS Affinity-based executions consume overall less execution time as compared to JS (with Affinity optimization) and MPJ Express implementations of the applications. As compared to JS (without optimization), JS Affinity consumes on average 21.29% less execution time for the machine size 2–80 processor cores. For 32 core-based execution, JS (Affinity) attains up to 39.82% lower

execution time (due to the locality optimizations) as compared to simple JS-based distributed memory version of the program.

As compared to MPJ Express (distributed memory version), the JS Affinity-based execution consumes on average 31.54% low execution time. For the 32 core-based execution, up to 49.83%, reduced execution time was observed. The commendable performance of JS (Affinity) as compared to the distributed memory implementations of JS and MPJ Express highlights the fact the reduced execution overhead (data access latency) employed by JS (Affinity) version of the application.

6 Conclusions

In this study, the Barnes–Hut algorithm (an N-body simulation of celestial objects) has been parallelized harnessing renowned Java parallel frameworks on multi-core architectures. The comparative performance analysis has been performed using several machine sizes and a large number (i.e., up to a million) of celestial objects. For low-level performance analysis, *PERF* and *OProfile* tools have been employed. The comparison of execution time is performed using several parallel Java platforms and further investigated using low-level hardware profilers that indicate the commendable performance by the JavaSymphony (with locality optimization) as compared to the other employed Java parallel frameworks (for both the shared and distributed memory versions). The user-controlled locality control of the parallel tasks, application objects, and whole application are one of the prominent features provided by the JavaSymphony framework that enables a user-directed mapping of tasks. With user-defined mappings, the data access latencies could be reduced benefiting in the reduced execution time. Moreover, the low-level performance analysis has revealed that the JavaSymphony's locality optimization and its parallel framework also has a lower execution overhead as compared to the MPJ Express framework. In the future, we intend to benchmark this scientific application on distributed memory architectures to study the potential benefit of locality optimization and the performance of remote communication employed by the Java parallel frameworks.

Compliance with ethical standards

Conflict of interest The authors declare that they have no conflict of interest.

Informed consent Informed consent was obtained from all individual participants included in the study.

Human and animal rights This article does not contain any studies with human participants or animals performed by any of the authors.

References

- Dad C, Vialle S, Caujolle M, Tavella JP, Ianotto M (2016) Scaling of distributed multi-simulations on multi-core clusters. In: 2016 IEEE 25th international conference on enabling technologies: infrastructure for collaborative enterprises (WETICE). IEEE, pp 142–147
- Aleem M, Prodan R, Fahringer T (2012) The javasymphony extensions for parallel gpu computing. In: 2012 41st international conference on parallel processing (ICPP). IEEE, pp 30–39
- Rodríguez A, Valverde J, Portilla J, Otero A, Riesgo T, De la Torre E (2018) FPGA-based high-performance embedded systems for adaptive edge computing in cyber-physical systems: the artico3 framework. *Sensors* 18(6):1877
- Taboada GL, Ramos S, Expósito RR, Touriño J, Doallo R (2013) Java in the high-performance computing arena: research, practice and experience. *Sci Comput Program* 78(5):425–444
- Aleem M, Prodan R, Fahringer T (2010) JavaSymphony: a programming and execution environment for parallel and distributed many-core architectures. In: European conference on parallel processing. Springer, Berlin, pp 139–150
- Javed A, Qamar B, Jameel M, Shafi A, Carpenter B (2016) Towards scalable java HPC with hybrid and native communication devices in MPJ express. *Int J Parallel Prog* 44(6):1142–1172
- Barnes J, Hut P (1986) A hierarchical O(N log N) force-calculation algorithm. *Nature* 324(6096):446
- Wellings AJ (2004) Concurrent and real-time programming in Java. Wiley, Chichester, pp I–XIV
- De Melo AC (2010) The new linux'perf' tools. In: Slides from Linux Kongress, vol 18, pp 1–42
- Prada-Rojas C, Riss F, Raynaud X, De Paoli S, Santana M (2009) Observation tools for debugging and performance analysis of embedded linux applications. In: Conference on system software, SoC and silicon debug-S4D
- Xu TC, Liljeberg P, Plosila J, Tenhunen H (2013) Evaluate and optimize parallel Barnes–Hut algorithm for emerging many-core architectures. In: 2013 international conference on high performance computing and simulation (HPCS). IEEE, pp 421–428
- Zhang J, Behzad B, Snir M (2011) Optimizing the Barnes–Hut algorithm in UPC. In: 2011 international conference for high performance computing, networking, storage and analysis (SC). IEEE, pp 1–11
- Singh JP (1993) Parallel Hierarchical N-Body Methods and their Implications for Multiprocessors. PhD thesis, Stanford Univ., Dept. of Electrical Engineering
- Winkel M, Speck R, Hübner H, Arnold L, Krause R, Gibbon P (2012) A massively parallel, multi-disciplinary Barnes–Hut tree code for extreme-scale N-body simulations. *Comput Phys Commun* 183(4):880–889
- Taboada GL, Touriño J, Doallo R (2009) Java for high performance computing: assessment of current research and practice. In: Proceedings of the 7th international conference on principles and practice of programming in Java. ACM, pp 30–39
- Feng H, Van der Wijngaart RF, Biswas R, Mavriplis C (2004) Unstructured adaptive (UA) NAS parallel benchmark. Version 1.0. Technical report, NASA technical report NAS-04-006
- Shafi A, Manzoor J (2009) Towards efficient shared memory communications in MPJ express. In: 2009 IEEE International Symposium on Parallel & Distributed Processing. IEEE, pp 1–7
- Zhang J, Behzad B, Snir M (2015) Design of a multithreaded Barnes–Hut algorithm for multicore clusters. *IEEE Trans Parallel Distrib Syst* 26(7):1861–1873
- Dinan J, Balaji P, Lusk E, Sadayappan P, Thakur R (2010) Hybrid parallel programming with MPI and unified parallel C. In: Proceedings of the 7th ACM international conference on computing frontiers. ACM, pp 177–186
- Hamada T, Nitadori K, Benkrid K, Ohno Y, Morimoto G, Masada T, Shibata Y, Oguri K, Taiji M (2009) A novel multiple-walk parallel algorithm for the Barnes–Hut treecode on GPUs—towards cost effective, high performance N-body simulation. *Comput Sci Res Dev* 24(1–2):21–31
- Griebler D, Loff J, Mencagli G, Danelutto M, Fernandes LG (2018) Efficient nas benchmark kernels with C++ parallel programming. In: 2018 26th Euromicro international conference on parallel, distributed and network-based processing (PDP). IEEE, pp 733–740
- Conn AR, Gould GIM, Toint PL (2013) LANCELOT: a Fortran package for large-scale nonlinear optimization (Release A), vol 17. Springer, Berlin
- Shafi A, Manzoor J, Hameed K, Carpenter B, Baker M (2010) Multicore-enabling the MPJ Express messaging library. In: Proceedings of the 8th international conference on the principles and practice of programming in Java. ACM, pp 49–58

Publisher's Note Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.