A parallel and distributed implementation of the Barnes–Hut quadtree algorithm for *n*-body simulations

Andrea Stedile
Department of Information Engineering and Computer Science
University of Trento
Email: andrea.stedile@studenti.unitn.it

Giuliano Andronic
Department of Information Engineering and Computer Science
University of Trento
Email: giuliano.andronic@studenti.unitn.it

***Abstract*—This document describes our work for the project required for the fulfillment of the High Performance Computing for Data Science course, taught by professor S. Fiore in the 2021/2022 AY. We implemented a parallel and distributed version of the Barnes–Hut quadtree algorithm for *n*-body simulations, using C++17 STL parallel functions, OpenMP and MPI. The data produced by our simulations shows that the algorithm can greatly benefit from the parallelization, and that the distribution can be of some help under some circumstances. Moreover, we implemented a tool for the visualization of the output produced by the simulations, including the motion of bodies and the quadtrees, which aids in assessing the correctness of the results.**

1. Introduction

1.1. The *n*-body problem and the Barnes–Hut approximation algorithm

The simulation of the motion of a set of bodies arises in a variety of domains such as astrophysics. A simulation proceeds in time-steps. In an exhaustive simulation, each time-step requires an all-to-all force interaction, yielding a complexity of $O(n^2)$ per time-step. The Barnes–Hut algorithm is an approximation algorithm that reduces the complexity to $O(n \log n)$ [1]. It works on the principle that a cluster of bodies sufficiently far from the observation point can be approximated by a single entity. In the case of 2D simulation scenarios, it decomposes the domain of a simulation recursively into a spatial tree-data structure called quadtree. Each node of the tree contains information about the center of mass of the bodies it contains and their aggregate mass. Figure 1 depicts an example of how a quadtree is constructed.

Once the quadtree has been constructed, the force on each body is computed by traversing the quadtree, starting from its root node:

- It is determined whether an interaction with the node can be computed by taking the ratio between the length of the node and the distance from the body.
- An interaction can be computed if the ratio is below a certain threshold θ. In this case, the computation of the gravitational force uses the information of the center of mass and aggregate mass contained in the node.
- An interaction cannot be computed if it is equal or above the threshold: the process is applied recursively for each of the four children of the node.

The choice of the θ threshold has an impact on the algorithm and the resulting forces:

- A high value produces the results faster, because the recursion traversing the quadtree stops early. Because nodes high in the tree are involved in the computation of the force on the bodies, the results are less accurate. A typical value of θ is 0.5.
- A low value produces more accurate results, at the expense of a slower execution. Setting $\theta = 0$ leads the algorithm to the degenerate execution where the quadtree is traversed entirely until all the leaves are reached.

1.2. Parallelization and distribution strategies of the Barnes–Hut algorithm

The construction of the quadtree is amebable to being distributed. Given a set of workers, each can compute the

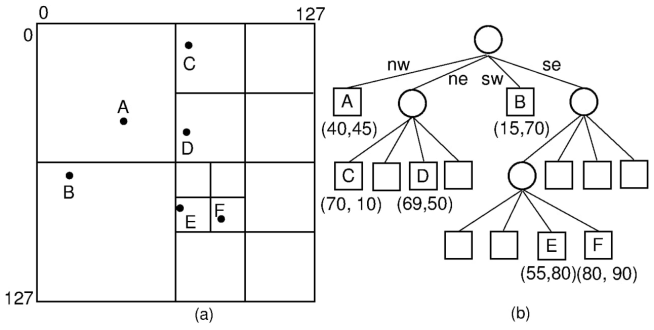

Figure 1. The quadtree data structure. It is a hierarchical subdivision of a spatial domain. Each leaf node contains exactly one body.

quadtree of a part of the domain of the simulation. Subsequently, the quadtrees are reuinited into a complete quadtree which is used in the computation of the forces acting on the set of bodies. The strategy with which the domain is subdivided is critical as it can either speed up or slow down the execution. The classic, naive schema establishes the domain be subdivided in subdomains of equal surface. This schema can become inefficient when the set of bodies in the domain has a non uniform spatial distribution: workers which are assigned subdomains with fewer bodies tend to compute a quadtree with fewer nodes. Some advanced load balancing schemas are discussed in [3] and [4], but are beyond the scope of our work.

The computation of the forces acting on the set of bodies is amenable to parallelization. Given a set of workers, each can compute the forces acting on a subset of the bodies by traversing the quadtree. This parallelization scales linearly with the number of workers.

Moreover, such computation can also be distributed: given a set of workers in possession of the complete quadtree, each can compute the forces acting on a subset of the bodies (each worker can possibly use the parallelization described above to produce a further speedup).

2. Contributions

The contributions of our work are the following:

- An *n*-body simulator for performing exhaustive $O(n^2)$ simulations. The simulator can be compiled to either use OpenMP or oneTBB as threading backends for computing the forces acting on the set bodies in parallel.
- An *n*-body simulator for performing simulations using the Barnes–Hut approximation algorithm. It can use OpenMP or oneTBB.
- An *n*-body simulator for performing distributed simulations using the Barnes–Hut approximation algorithm. In conjunction of OpenMP or oneTBB, it uses MPI to distribute the construction of the quadtree and the computation of the forces acting on the bodies.
- A tool for the visualization of small datasets produced by the Barnes–Hut simulators. The visualization shows how the bodies move in the domain due to the gravitational forces they exert on each other, and draws the quadtrees in the background. It allows to assess the goodness of the simulations.

The following list summarizes some choices we made prior to the implementation works:

- The simulators would accept a set of bodies on the two-dimensional plane but be easy to adapt for the three-dimensional case.
- The simulators would allow to pass the number of simulation steps to perform and the simulation timestep as an argument.
- The Barnes–Hut simulators would allow to pass the θ parameter as an argument.
- The Barnes–Hut simulators would use the classic, naive schema which we described in the previous section and work on square domains, and is easy enough to be implement as part of a course project.
- The simulators would time the various functions involved in a simulation step to identify the hot spots.
- The simulators would log the execution meaningfully to allow the debugging of distributed executions.
- A suite of tests must cover substantial part of the code, to identify the edge cases of the algorithms and regressions in the code.
- Use external libraries if they facilitate the implementation. For example, we decided to use the Eigen3 library to represent the position and velocity vectors of bodies and to represent the bounding box enclosing a set of bodies.

2.1. Implementation details

We report some excerpts of source code implementing the parallelization and distribution strategies discussed in section 1.1.

2.1.1. Parallel code. We have implemented the simulators using the C++17 programming language, making use of the STL library's parallel functions wherever possible, insisting on writing idiomatic code and using move semantics. The parallel constructs are `transform` and `transform_reduce`, which we use with the `execution::par_unseq` policy (standing for parallel unsequenced). The threading backend to which we linked the code is oneTBB.

We also implemented an OpenMP couterpart for all the code making use of the STL library's parallel functions. The counterpart of `transform` is the OpenMP parallel worksharing-loop construct `parallel for`; the counterpart of `transform_reduce` is the a `parallel for` with a reduction clause.

The exhaustive computation of the net force that a set of bodies exerts on a single body makes use of the `transform_reduce` function (or the OpenMP parallel loop counterpart):

```
Eigen::Vector2d
compute_exact_net_force_on_body_parallel(
 const std::vector<Body>& bodies,
 const Body& body, double G) {

 #ifdef WITH_TBB
 return std::transform_reduce(
  std::execution::par_unseq,
  bodies.begin(), bodies.end(),
  Eigen::Vector2d{0, 0},
  [](const Eigen::Vector2d& total,
   const Eigen::Vector2d& curr) {
    return (total + curr).eval();
   },
  [&](const Body& curr) {
```

```
   return compute_gravitational_force(
    curr, body, G);
  });

 #else
 Eigen::Vector2d net_force{0, 0};
 #pragma omp parallel for default(none) \
  shared(bodies, body, G) \
  reduction(+ : net_force)
 for (const Body &curr : bodies) {
  net_force += compute_gravitational_force(
   curr, body, G);
 }
 return net_force;
#endif
}
```

For completeness, we also report the OpennMP custom reduction used by the parallel loop:

```
#pragma omp declare reduction(+ : \
 Eigen::Vector2d : \
 omp_out += omp_in) \
 initializer(omp_priv = {0, 0})
```

The computation of the minimum bounding box enclosing the set of bodies (i.e., the domain of the simulation) uses `transform_reduce` too. The bodies move in the domain due to the gravitational force they exert on each other. If a body moves outwards (for example, due to a slingshot effect caused by another body passing nearby), the minimum bounding box containing all bodies can increase. Therefore, the minimum bounding box is recomputed at each simulation step (and possibly adjusted to be square).

The computation of the new position and velocity vectors of a body (represented by the `Body` struct) in a simulation step using the quadtree data structure (represented by the `Node` class) uses the `transform` function (or the OpenMP parallel loop counterpart). In the following excerpt, `last_step.bodies()` refers to the set of bodies produced by the previous simulation step, and `new_bodies` to the new set of bodies to compute as part of the current step.

```
#ifdef WITH_TBB
std::transform(std::execution::par_unseq,
 last_step.bodies().begin(),
 last_step.bodies().end(),
 new_bodies.begin(),
 [&](const Body& body) {
  return update_body(body, *quadtree,
   dt, G, theta);
 });

#else
#pragma omp parallel for default(none) \
 shared(last_step, new_bodies, quadtree, \
 dt, G, theta)
for (size_t i = 0;
 i < last_step.bodies().size(); i++) {
 new_bodies[i] = update_body(
  last_step.bodies()[i], *quadtree,
   dt, G, theta);
}
#endif
```

0	1	2	3
4	5	6	7
8	9	10	11
12	13	14	15

Figure 2. Naive assignment of the MPI processes to the domain subquadrants. The numbers in the grid are ranks of the MPI processes.

2.1.2. Distributed code. In section 1.1, we have discussed how the Barnes–Hut algorithm is amenable to distribution. We implemented the distributed Barnes–Hut simulator by means of the Message Passing Interface (MPI). The assignment of a specific MPI process to a specific subdomain (we call a subdomain *subquadrant*) of the simulation corresponds to the naive schema and is depicted in Figure 2. This entails that the number of MPI processes be a power of four.

An instance of the distributed Barnes–Hut simulator executing a simulation step does as follows:

- Determines the subquadrant for which it is going to compute the quadtree. This operation must be repeated at each step because the domain may increase or decrease depending on the motion of the bodies.
- Computes the set of bodies belonging to the subquadrant. As discussed in section 1.1, the number of bodies the subquadrant contains depends on the spatial distribution of the set of bodies.
- Computes the quadtree rooted in the subquadrant. The number of nodes contained in the quadtree depends on the spatial distribution of the set of bodies contained in the subquadrant.
- Produces a serialized version of the quadtree that is amenable to be transmitted using MPI. This entails a conversion of the C++ code implementing the quadtree (which uses modern C++ constructs) to a vector of plain C structs. Nodes that are forks in the quadtree have pointers to four child nodes. Since that such pointers would become meaningless when crossing the boundaries of the sending process, they are converted to indexes to other structs in the vector.
- Exchanges with all other simulators the quadtree that it has computed, and becomes in possession of the quadtree of the entire domain (this communication is all-to-all). This entails that the receiving processes deserialize the vectors of C structs back in a C++ quadtree. The exchange of quadtrees also requires that the processes first exchange the size of the vector of C structs, which is not known in advance.
- Determines the subset of bodies for which to compute the updates with a static assignment schema. The assignment schema is such that each process computes an equal amount of bodies (unless there is a X remainder which is assigned to the first X nodes).
- Computes in parallel the updates for the subset of bodies using the Barnes–Hut approximation algo-

rithm and the complete quadtree.

- Produces a serialized version of the bodies that is amenable to be transmitted using MPI. This entails a conversion of the C++ code implementing the body (which uses code from the Eigen3 C++ library) to a vector of plain C structs.
- Exchanges with all other simulators the subset of updated bodies that it has computed, and becomes in possession of the entire set of updates bodies (this communication is all-to-all). This entails that the receiving processes deserialize the vector of C structs back in a vector of C++ bodies.
- Computes in parallel the square bounding box containing all the bodies.
- Returns the set of updated bodies, the quadtree that has been computed as part of the process and the bounding box containing the bodies.

The exchange of quadtrees is implemented as follows:

```
std::unique_ptr<Node> gather_quadtree(
 int proc_id, int n_procs,
 const Node& my_quadtree) {
  // contains the number of nodes that are
  // to be received from each process
  std::vector<int> recv_n_nodes(n_procs);
  auto my_n_nodes = my_quadtree.n_nodes();
  MPI_Allgather(
   &my_n_nodes, 1, MPI_INT, &recv_n_nodes[0],
   1, MPI_INT, MPI_COMM_WORLD);

  int total_n_nodes = std::accumulate(
   recv_n_nodes.begin(),
   recv_n_nodes.end(),
   0, std::plus<>());

  // contains the number of bytes that are
  // to be received from each process
  std::vector<int> recv_n_bytes(recv_n_nodes);
  std::transform(recv_n_nodes.begin(),
   recv_n_nodes.end(),
   recv_n_bytes.begin(),
   [](const auto n_nodes) {
    return n_nodes * sizeof(mpi::Node);
   });

  // entry i specifies the displacement
  // (relative to recvbuf) at which to place
  // the incoming data from process i
  std::vector<int> displacements(n_procs);
  std::partial_sum(recv_n_bytes.begin(),
   recv_n_bytes.end(),
   displacements.begin() + 1,
   std::plus<>());

  std::vector<mpi::Node>
   my_serialized_quadtree =
   serialize_quadtree(my_quadtree);

  std::vector<mpi::Node>
   all_serialized_quadtrees(total_n_nodes);

  MPI_Allgatherv(
   &my_serialized_quadtree[0],
   recv_n_bytes[proc_id], MPI_BYTE,
   &all_serialized_quadtrees[0],
   &recv_n_bytes[0],
   &displacements[0], MPI_BYTE,
   MPI_COMM_WORLD);

  auto grid =
   deserialize_quadtrees(n_procs,
   all_serialized_quadtrees, recv_n_nodes);

  return reconstruct_quadtree(grid);
}
```

The exchange of bodies is implemented as follows:

```
std::vector<Body> gather_bodies(
 int proc_id, int n_procs,
 int total_n_bodies,
 const std::vector<Body>& my_bodies) {
  // contains the number of bodies that are
  // to be received from each process
  std::vector<int> recv_n_bodies(n_procs,
   total_n_bodies / n_procs);
  std::transform(recv_n_bodies.begin(),
   recv_n_bodies.begin() +
   (total_n_bodies % n_procs),
   recv_n_bodies.begin(),
   [](const int val) { return val + 1; });

  // contains the number of bytes that are
  // to be received from each process
  std::vector<int> recv_n_bytes(n_procs);
  std::transform(recv_n_bodies.begin(),
   recv_n_bodies.end(),
   recv_n_bytes.begin(),
   [](const auto n_nodes) {
    return n_nodes * sizeof(mpi::Body);
   });

  // entry i specifies the displacement
  // (relative to all_serialized_bodies)
  // at which to place the incoming data
  // from process i
  std::vector<int> displacements(n_procs);
  std::partial_sum(recv_n_bytes.begin(),
   recv_n_bytes.end(),
   displacements.begin() + 1,
   std::plus<>());

  std::vector<mpi::Body>
   my_serialized_bodies =
   serialize_bodies(my_bodies);

  std::vector<mpi::Body>
   all_serialized_bodies(total_n_bodies);

  MPI_Allgatherv(
   &my_serialized_bodies[0],
   recv_n_bytes[proc_id], MPI_BYTE,
   &all_serialized_bodies[0],
   &recv_n_bytes[0],
   &displacements[0], MPI_BYTE,
   MPI_COMM_WORLD);

  return
```

```
    deserialize_bodies(all_serialized_bodies);
}
```

3. Evaluation

Figure 3 depicts the execution times of the exhaustive $O(n^2)$ algorithm and the Barnes–Hut algorithm, averaged on four distinct runs of the non-distributed simulators, with an input of 10,000 bodies and 100 simulation steps. The input consists of 10,000 bodies generated randomly (their masses and distances are similar to the planets in the solar system). The simulators have been compiled with the oneTBB threading backend on a laptop running Manjaro Linux with Linux 5.4.201 and GCC 12.1.0. The laptop has an Intel i7-10750H with 12 processors operating at a clock speed of 2.60GHz, and 15.4 GiB of RAM. The data we obtained suggests that the implementation using the parallel function of the C++17 STL library and the oneTBB threading backend is slightly faster.

Figure 5 depicts the execution times of the Barnes–Hut algorithm with an input of 100,000 bodies utilizing a different number of threads of the laptop. It clearly indicates that the use of threads produces an great speedup. Intuitively, it would scale linearly with the number of threads to finally plateau upon saturating the processors of the host.

Figure 4 depicts the execution times of the Barnes–Hut algorithm, compiled with oneTBB as threading backend, run in the non-distributed setting with an input of 10,000 bodies and with different values of the θ threshold parameter. The data we gathered confirms that the increase of θ has the effect of lowering the execution time of the algorithm.

Finally, we put the Barnes–Hut simulator to work in the distributed setting of the University's HPC cluster. We compiled the simulator with GCC 9.1.0 (a more recent version was not available), linking the code to mpich-3.2. We organized the runs as follows:

- We requested the PBS scheduler to arrange 1, 4, 16 and 64 nodes with 16 cores each, and to run one single MPI process on each node. By doing so, each MPI process saturate the 16 cores available on the node when executing the parallel portion of the code.
- We prepared five different input files for the simulators: one with 1,000 bodies, one with 10,000, one with 100,000 and one with 1,000,000.
- We repeated the run with each input file for four different times and averaged their execution times.

Figures 3, 7, 8 and 3 depict the average execution time of the runs. The results let us appreciate the fact that increasing the number of processors does not necessarily produce a speedup. This is compatible with the observation of section 1.1 that the naive schema of subdividing the domain in subdomains of equal surface, which our simulator uses, entails some load balancing problems, but they are outside the scope of our work.

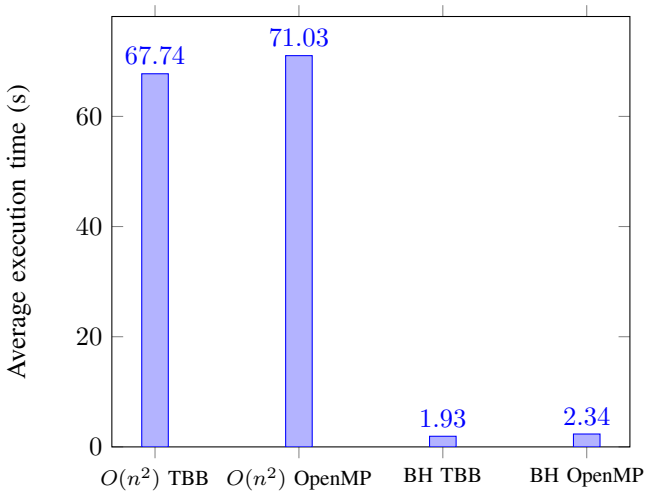

Figure 3. Average execution times of the exhaustive $O(n^2)$ algorithm and the Barnes–Hut algorithm, run in the non-distributed setting with an input of 10.000 bodies and for 100 simulation steps.

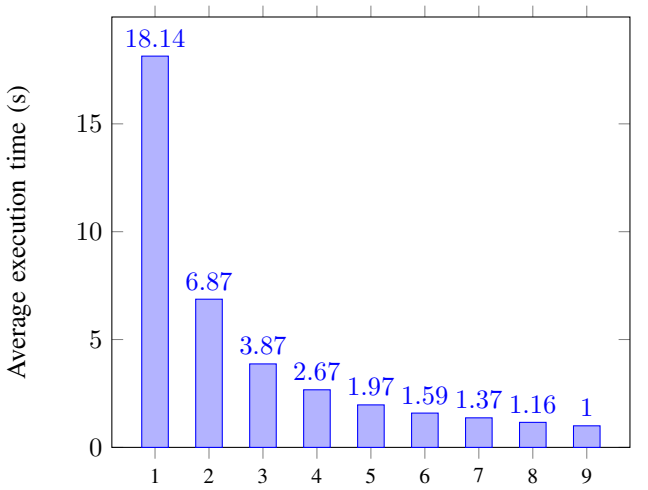

Figure 4. Average execution times of the Barnes–Hut algorithm with different values of the θ threshold parameter, run in the non-distributed setting with an input of 1,000 bodies and for 100 simulation steps. The threading backend is oneTBB.

References

[1] J. Barnes e P. Hut, A hierarchical O(N log N) force-calculation algorithm, in Nature, vol. 324, n. 4, December 1986, pp. 446-449

[2] H. Kopka and P. W. Daly, *A Guide to LaTeX*, 3rd ed. Harlow, England: Addison-Wesley, 1999.

[3] Grama, A., Kumar, V. & Sameh, A. Scalable Parallel Formulations of the Barnes-Hut Method for n-Body Simulations. *Proceedings Of The 1994 ACM/IEEE Conference On Supercomputing*. pp. 439-448 (1994)

[4] Winkel, M., Speck, R., Hübner, H., Arnold, L., Krause, R. & Gibbon, P. A massively parallel, multi-disciplinary Barnes–Hut tree code for extreme-scale N-body simulations. *Computer Physics Communications*. **183**, 880-889 (2012), https://www.sciencedirect.com/science/article/pii/S0010465511004012

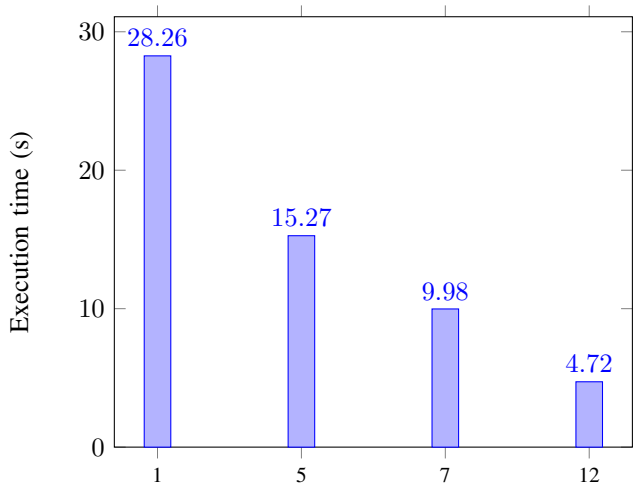

Figure 5. Execution times of the Barnes–Hut algorithm run with an input of 1,000,000 bodies, for 1 simulation step and utilizing different number of threads.

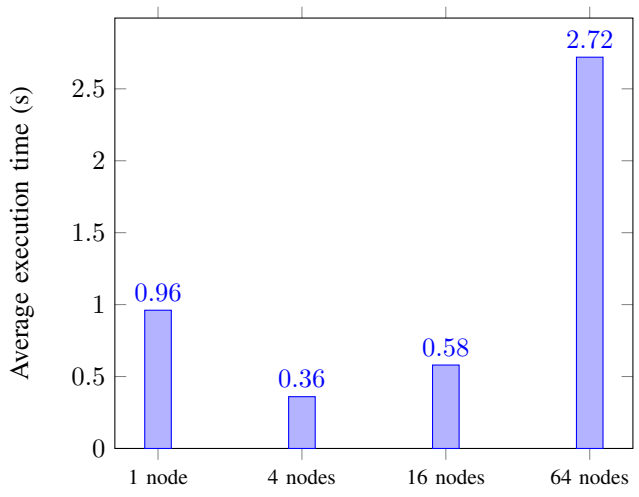

Figure 6. Average execution times of the Barnes–Hut algorithm run in the University's HPC cluster, with an input of 1,000 bodies and for 100 simulation steps, utilizing different numbers of nodes with 16 processors each.

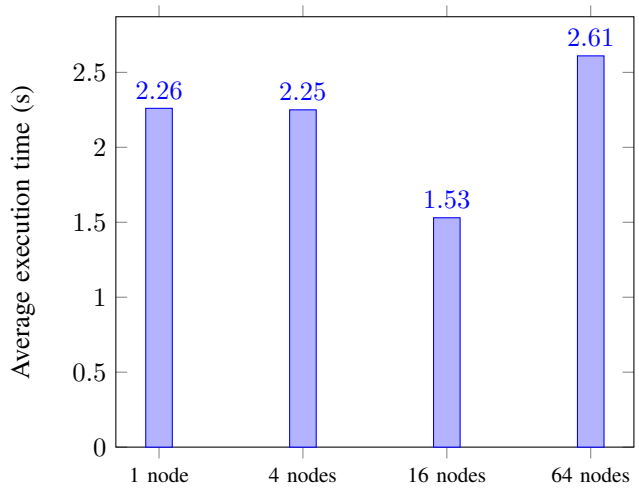

Figure 7. Average execution times of the Barnes–Hut algorithm run in the University's HPC cluster, with an input of 10,000 bodies and for 100 simulation steps, utilizing different numbers of nodes with 16 processors each.

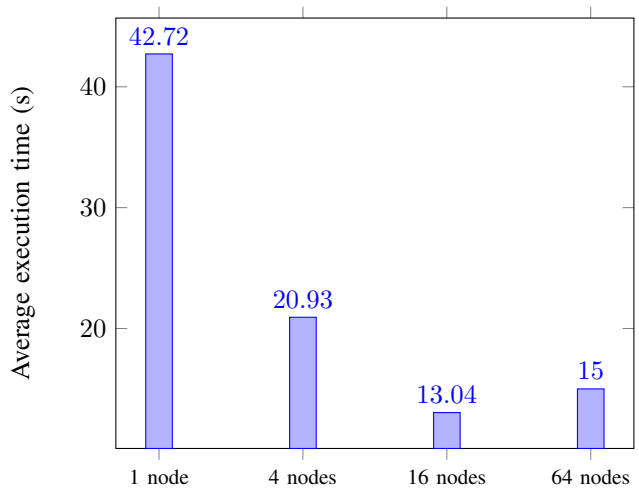

Figure 8. Average execution times of the Barnes–Hut algorithm run in the University's HPC cluster, with an input of 100,000 bodies and for 100 simulation steps, utilizing different numbers of nodes with 16 processors each.

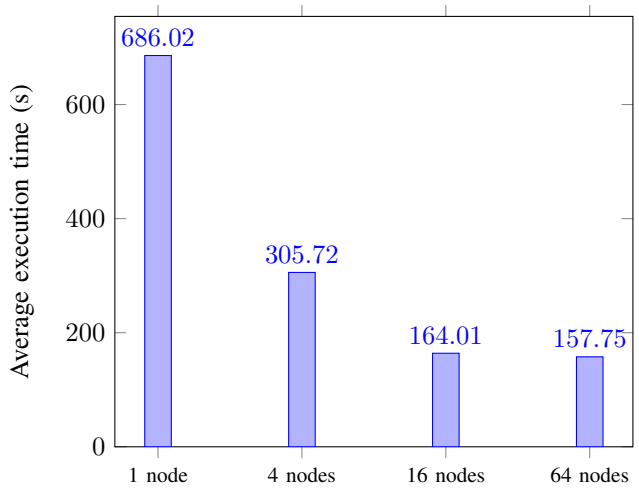

Figure 9. Average execution times of the Barnes–Hut algorithm run in the University's HPC cluster, with an input of 1,000,000 bodies and for 100 simulation steps, utilizing different numbers of nodes, with 16 processors each.